IW 7696177

THE UNITED STATES OF AMERICA

TO ALL TO WHOM THESE PRESENTS SHALL COME:

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office

October 17, 2018

THIS IS TO CERTIFY THAT ANNEXED IS A TRUE COPY FROM THE RECORDS OF THIS OFFICE OF THE FILE WRAPPER AND CONTENTS OF:

APPLICATION NUMBER: *09/608,237*
FILING DATE: *June 30, 2000*
PATENT NUMBER: *6,651,099*
ISSUE DATE: *November 18, 2003*

By Authority of the
Under Secretary of Commerce for Intellectual Property
and Director of the United States Patent and Trademark Office

P. R. Grant

P. R. GRANT
Certifying Officer

PART (1) OF (2) PART(S)

09/608237 06/30/00

709 | 224 | Class | Subclass | ISSUE CLASSIFICATION

PATENT NUMBER 6651099

6651099

U.S. UTILITY Patent Application

O.I.P.E.	PATENT DATE
SCANNED ___ Q.A. ___	NOV 18 2003

APPLICATION NO.	CONT/PRIOR	CLASS	SUBCLASS	ART UNIT	EXAMINER
09/608237	D	709	224	~~2157~~ 2157	Burgess

APPLICANTS

Russell Dietz
Joseph Maixner
Andrew Koppenhaver
William Bares

TITLE

Method and apparatus for monitoring traffic in a network

Certificate of Correction MAY 04 2004

PTO-2040
12/99

ISSUING CLASSIFICATION

ORIGINAL		CROSS REFERENCE(S)	
CLASS	SUBCLASS	CLASS	SUBCLASS (ONE SUBCLASS PER BLOCK)
709	224	370	389
INTERNATIONAL CLASSIFICATION			
G06F 13/00			

☐ Continued on Issue Slip Inside File Jacket

10-6-03 Formal Drawings (18 shts) set 1 6-30-00

☐ TERMINAL DISCLAIMER	DRAWINGS			CLAIMS ALLOWED	
	Sheets Drwg.	Figs. Drwg.	Print Fig.	Total Claims	Print Claim for O.G.
	18	20	10	10	1

☐ The term of this patent subsequent to ____ (date) has been disclaimed.

(Assistant Examiner) (Date)

NOTICE OF ALLOWANCE MAILED

7-17-03

☐ The term of this patent shall not extend beyond the expiration date of U.S Patent. No. ____

Moustafa M. Meky
MOUSTAFA M. MEKY
PRIMARY EXAMINER
(Primary Examiner) 7/16/2003 (Date)

ISSUE FEE

Amount Due	Date Paid
$1,300.00	9-24-03

☐ The terminal ____ months of this patent have been disclaimed.

R Johnson 7-23-03
(Legal Instruments Examiner) (Date)

ISSUE BATCH NUMBER

WARNING:
The information disclosed herein may be restricted. Unauthorized disclosure may be prohibited by the United States Code Title 35, Sections 122, 181 and 368. Possession outside the U.S. Patent & Trademark Office is restricted to authorized employees and contractors only.

Form PTO-436A
(Rev. 6/99)

FILED WITH: ☐ DISK (CRF) ☐ FICHE ☐ CD-ROM
(Attached in pocket on right inside flap)

ISSUE FEE IN FILE

(FACE)

UNITED STATES PATENT AND TRADEMARK OFFICE

COMMISSIONER FOR PATENTS
UNITED STATES PATENT AND TRADEMARK OFFICE
WASHINGTON, D.C. 20231
www.uspto.gov

Bib Data Sheet

CONFIRMATION NO. 9993

SERIAL NUMBER	FILING DATE	CLASS	GROUP ART UNIT	ATTORNEY DOCKET NO.
09/608,237	06/30/2000 RULE	709	2755	APPT-001-1

APPLICANTS

Russell S. Dietz, San Jose, CA;
Joseph R. Maixner, Aptos, CA;
Andrew A. Koppenhaver, Littleton, CO;
William H. Bares, Germantown, TN;
Haig A. Sarkissian, San Antonio, TX;
James F. Torgerson, Andover, MN;

** CONTINUING DATA **********

THIS APPLN CLAIMS BENEFIT OF 60/141,903 06/30/1999

Yes, MMM

** FOREIGN APPLICATIONS **********

Non, MMM

IF REQUIRED, FOREIGN FILING LICENSE GRANTED ** 08/21/2000

Foreign Priority claimed	STATE OR COUNTRY	SHEETS DRAWING	TOTAL CLAIMS	INDEPENDENT CLAIMS
☐ yes ☒ no	CA	18	59	4

35 USC 119 (a-d) conditions met: ☐ yes ☒ no ☐ Met after Allowance

Verified and Acknowledged: Examiner's Signature _____ Initials MMM

ADDRESS

Dov Rosenfeld
Suite 2
5507 College Avenue
Oakland ,CA 94618

TITLE

Method and apparatus for monitoring traffic in a network

FILING FEE RECEIVED	FEES: Authority has been given in Paper
1622	No. _____ to charge/credit DEPOSIT ACCOUNT No. _____ for following:

☐ All Fees
☐ 1.16 Fees (Filing)
☐ 1.17 Fees (Processing Ext. of time)
☐ 1.18 Fees (Issue)
☐ Other _____
☐ Credit

07-03-00

IN THE U.S. PATENT AND TRADEMARK OFFICE

Application Transmittal Sheet

Our Ref./Docket No.: APPT-001-1

Box Patent Application
ASSISTANT COMMISSIONER FOR PATENTS
Washington, D.C. 20231

Dear Assistant Commissioner:

Transmitted herewith is the patent application of

INVENTOR(s)/APPLICANT(s)

Last Name	First Name, MI	Residence (City and State or Country)
Dietz	Russell S.	San Jose, CA
Maixner	Joseph R.	Aptos, CA
Koppenhaver	Andrew A.	Fairfax, VA

Additional inventors are being named on separately numbered sheets attached hereto.

TITLE OF THE INVENTION

METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK

CORRESPONDENCE ADDRESS AND AGENT FOR APPLICANT(S)

Dov Rosenfeld, Reg. No. 38,387
5507 College Avenue, Suite 2
Oakland, California, 94618
Telephone: (510) 547-3378; Fax: (510) 653-7992

ENCLOSED APPLICATION PARTS (check all that apply)

Included are:
- X 66 sheet(s) of specification, claims, and abstract
- X 18 sheet(s) of formal Drawing(s) with a submission letter to the Official Draftsperson
- _____ Information Disclosure Statement.
- _____ Form PTO-1449: INFORMATION DISCLOSURE CITATION IN ANAPPLICATION, together with a copy of each references included in PTO-1449.
- _____ Declaration and Power of Attorney
- _____ An assignment of the invention to Apptitude, Inc.
- _____ A letter requesting recordation of the assignment.
- _____ An assignment Cover Sheet.
- X Additional inventors are being named on separately numbered sheets attached hereto.
- X Return postcard.

This application has:
- _____ a small entity status. A verified statement:
  - _____ is enclosed
  - _____ was already filed.

The fee has been calculated as shown in the following page.

Certificate of Mailing under 37 CFR 1.10

I hereby certify that this application and all attachments are being deposited with the United States Postal Service as Express Mail (Express Mail Label: EI417961944US in an envelope addressed to Box Patent Application, Assistant Commissioner for Patents, Washington, D.C. 20231 on.

Date: June 30, 2000 Signed: _____

Name: Dov Rosenfeld, Reg. No. 38687

NOAC Ex. 1015 Page 4

	TOTAL CLAIMS	NO. OF EXTRA CLAIMS	RATE	EXTRA CLAIM FEE
TOTAL CLAIMS	59	39	$18	$ 702.00
INDEP. CLAIMS	4	1	$78	$ 78.00
			BASIC APPLICATION FEE:	$ 690.00
			TOTAL FEES PAYABLE:	$1,470.00

METHOD OF PAYMENT

_____ A check in the amount of _____ is attached for application fee and presentation of claims.
_____ A check in the amount of $ 40.00 is attached for recordation of the Assignment.
_____ The Commissioner is hereby authorized to charge payment of the any missing filing or other fees required for this filing or credit any overpayment to Deposit Account No. 50-0292 (A DUPLICATE OF THIS TRANSMITTAL IS ATTACHED):

Respectfully Submitted,

June 30, 2000
Date

Dov Rosenfeld , Reg. No. 38687

Correspondence Address:
Dov Rosenfeld
5507 College Avenue, Suite 2
Oakland, California, 94618
Telephone: (510) 547-3378; Fax: (510) 653-7992

ATTORNEY DOCKET NO. APPT-001-1

Application Cover Sheet (cont.)

INVENTOR(s)/APPLICANT(s)

Last Name	First Name, MI	Residence (City and Either State or Foreign Country)
Bares	William H.	Germantown, TN
Sarkissian	Haig A.	San Antonio, Texas
Torgerson	James F.	Andover, MN

Our Ref./Docket No: APPT-001-1 Patent

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant(s): Dietz, *et al.* Title: METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK	Group Art Unit: unassigned Examiner: unassigned

LETTER TO OFFICIAL DRAFTSPERSON SUBMISSION OF FORMAL DRAWINGS

The Assistant Commissioner for Patents
Washington, DC 20231
ATTN: Official Draftsperson

Dear Sir or Madam:

Attached please find 18 sheets of formal drawings to be made of record for the above identified patent application submitted herewith.

Respectfully Submitted,

June 30, 2000
Date

Dov Rosenfeld, Reg. No. 38687

Address for correspondence and attorney for applicant(s):
Dov Rosenfeld, Reg. No. 38,687
5507 College Avenue, Suite 2
Oakland, CA 94618
Telephone: (510) 547-3378; Fax: (510) 653-7992

Certificate of Mailing under 37 CFR 1.10

I hereby certify that this application and all attachments are being deposited with the United States Postal Service as Express Mail (Express Mail Label: EI417961944US in an envelope addressed to Box Patent Application, Assistant Commissioner for Patents, Washington, D.C. 20231 ~~on~~.

Date: June 30, 2000

Signed:
Name: Dov Rosenfeld, Reg. No. 38687

NOAC Ex. 1015 Page 7

Our Ref./Docket No.: APPT-001-1

METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK

Inventor(s):

DIETZ, Russell S.
San Jose, CA

MAIXNER, Joseph R.
Aptos, CA

KOPPENHAVER, Andrew A.
Fairfax, VA

BARES, William H.
Germantown, TN

SARKISSIAN, Haig A.
San Antonio, Texas

TORGERSON, James F.
Andover, MN

Certificate of Mailing under 37 CFR 1.10

I hereby certify that this application and all attachments are being deposited with the United States Postal Service as Express Mail (Express Mail Label: EI417961944US in an envelope addressed to Box Patent Application, Assistant Commissioner for Patents, Washington, D.C. 20231 on.

Date: June 30, 2000

Signed: [signature]

Name: Dov Rosenfeld, Reg. No. 38687

NOAC Ex. 1015 Page 8

METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK

CROSS-REFERENCE TO RELATED APPLICATION

This application claims the benefit of U.S. Provisional Patent Application Serial No.: 60/141,903 for METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK to inventors Dietz, et al., filed June 30, 1999, the contents of which are incorporated herein by reference.

This application is related to the following U.S. patent applications, each filed concurrently with the present application, and each assigned to Apptitude, Inc., the assignee of the present invention:

U.S. Patent Application Serial No. 09/609,179 for PROCESSING PROTOCOL SPECIFIC INFORMATION IN PACKETS SPECIFIED BY A PROTOCOL DESCRIPTION LANGUAGE, to inventors Koppenhaver, et al., filed June 30, 2000, ~~Attorney/Agent Reference Number APPT-001-2~~ still pending, and incorporated herein by reference.

U.S. Patent Application Serial No. 09/608,126 for RE-USING INFORMATION FROM DATA TRANSACTIONS FOR MAINTAINING STATISTICS IN NETWORK MONITORING, to inventors Dietz, et al., filed June 30, 2000, ~~Attorney/Agent Reference Number APPT-001-3~~ still pending, and incorporated herein by reference.

U.S. Patent Application Serial No. 09/608,266 for ASSOCIATIVE CACHE STRUCTURE FOR LOOKUPS AND UPDATES OF FLOW RECORDS IN A NETWORK MONITOR, to inventors Sarkissian, et al., filed June 30, 2000, ~~Attorney/Agent Reference Number APPT-001-4~~ still pending, and incorporated herein by reference.

U.S. Patent Application Serial No. 09/608,267 for STATE PROCESSOR FOR PATTERN MATCHING IN A NETWORK MONITOR DEVICE, to inventors Sarkissian, et al., filed June 30, 2000, ~~Attorney/Agent Reference Number APPT-001-5,~~ still pending, and incorporated herein by reference.

FIELD OF INVENTION

The present invention relates to computer networks, specifically to the real-time elucidation of packets communicated within a data network, including classification according to protocol and application program.

BACKGROUND TO THE PRESENT INVENTION

There has long been a need for network activity monitors. This need has become especially acute, however, given the recent popularity of the Internet and other internets—an "internet" being any plurality of interconnected networks which forms a larger, single network. With the growth of networks used as a collection of clients obtaining services from one or more servers on the network, it is increasingly important to be able to monitor the use of those services and to rate them accordingly. Such objective information, for example, as which services (*i.e.,* application programs) are being used, who is using them, how often they have been accessed, and for how long, is very useful in the maintenance and continued operation of these networks. It is especially important that selected users be able to access a network remotely in order to generate reports on network use in real time. Similarly, a need exists for a real-time network monitor that can provide alarms notifying selected users of problems that may occur with the network or site.

One prior art monitoring method uses log files. In this method, selected network activities may be analyzed retrospectively by reviewing log files, which are maintained by network servers and gateways. Log file monitors must access this data and analyze ("mine") its contents to determine statistics about the server or gateway. Several problems exist with this method, however. First, log file information does not provide a map of real-time usage; and secondly, log file mining does not supply complete information. This method relies on logs maintained by numerous network devices and servers, which requires that the information be subjected to refining and correlation. Also, sometimes information is simply not available to any gateway or server in order to make a log file entry.

One such case, for example, would be information concerning NetMeeting® (Microsoft Corporation, Redmond, Washington) sessions in which two computers connect directly on the network and the data is never seen by a server or a gateway.

Another disadvantage of creating log files is that the process requires data logging features of network elements to be enabled, placing a substantial load on the device , which results in a subsequent decline in network performance. Additionally, log files can grow rapidly, there is no standard means of storage for them, and they require a

significant amount of maintenance.

Though Netflow® (Cisco Systems, Inc., San Jose, California), RMON2, and other network monitors are available for the real-time monitoring of networks, they lack visibility into application content and are typically limited to providing network layer level information.

Pattern-matching parser techniques wherein a packet is parsed and pattern filters are applied are also known, but these too are limited in how deep into the protocol stack they can examine packets.

Some prior art packet monitors classify packets into connection flows. The term "connection flow" is commonly used to describe all the packets involved with a single connection. A conversational flow, on the other hand, is the sequence of packets that are exchanged in any direction as a result of an activity—for instance, the running of an application on a server as requested by a client. It is desirable to be able to identify and classify conversational flows rather than only connection flows. The reason for this is that some conversational flows involve more than one connection, and some even involve more than one exchange of packets between a client and server. This is particularly true when using client/server protocols such as RPC, DCOMP, and SAP, which enable a service to be set up or defined prior to any use of that service.

An example of such a case is the SAP (Service Advertising Protocol), a NetWare (Novell Systems, Provo, Utah) protocol used to identify the services and addresses of servers attached to a network. In the initial exchange, a client might send a SAP request to a server for print service. The server would then send a SAP reply that identifies a particular address—for example, SAP#5—as the print service on that server. Such responses might be used to update a table in a router, for instance, known as a Server Information Table. A client who has inadvertently seen this reply or who has access to the table (via the router that has the Service Information Table) would know that SAP#5 for this particular server is a print service. Therefore, in order to print data on the server, such a client would not need to make a request for a print service, but would simply send data to be printed specifying SAP#5. Like the previous exchange, the transmission of data to be printed also involves an exchange between a client and a server, but requires a second connection and is therefore independent of the initial exchange. In order to eliminate the

possibility of disjointed conversational exchanges, it is desirable for a network packet monitor to be able to "virtually concatenate"—that is, to link—the first exchange with the second. If the clients were the same, the two packet exchanges would then be correctly identified as being part of the same conversational flow.

Other protocols that may lead to disjointed flows, include RPC (Remote Procedure Call); DCOM (Distributed Component Object Model), formerly called Network OLE (Microsoft Corporation, Redmond, Washington); and CORBA (Common Object Request Broker Architecture). RPC is a programming interface from Sun Microsystems (Palo Alto, California) that allows one program to use the services of another program in a remote machine. DCOM, Microsoft's counterpart to CORBA, defines the remote procedure call that allows those objects—objects are self-contained software modules—to be run remotely over the network. And CORBA, a standard from the Object Management Group (OMG) for communicating between distributed objects, provides a way to execute programs (objects) written in different programming languages running on different platforms regardless of where they reside in a network.

What is needed, therefore, is a network monitor that makes it possible to continuously analyze all user sessions on a heavily trafficked network. Such a monitor should enable non-intrusive, remote detection, characterization, analysis, and capture of all information passing through any point on the network (*i.e.,* of all packets and packet streams passing through any location in the network). Not only should all the packets be detected and analyzed, but for each of these packets the network monitor should determine the protocol (*e.g.,* http, ftp, H.323, VPN, etc.), the application/use within the protocol (*e.g.,* voice, video, data, real-time data, etc.), and an end user's pattern of use within each application or the application context (*e.g.,* options selected, service delivered, duration, time of day, data requested, etc.). Also, the network monitor should not be reliant upon server resident information such as log files. Rather, it should allow a user such as a network administrator or an Internet service provider (ISP) the means to measure and analyze network activity objectively; to customize the type of data that is collected and analyzed; to undertake real time analysis; and to receive timely notification of network problems.

Considering the previous SAP example again, because one features of the invention is to correctly identify the second exchange as being associated with a print

service on that server, such exchange would even be recognized if the clients were not the same. What distinguishes this invention from prior art network monitors is that it has the ability to recognize disjointed flows as belonging to the same conversational flow.

The data value in monitoring network communications has been recognized by many inventors. Chiu, *et al.*, describe a method for collecting information at the session level in a computer network in United States Patent 5,101,402, titled "APPARATUS AND METHOD FOR REAL-TIME MONITORING OF NETWORK SESSIONS AND A LOCAL AREA NETWORK" (the "402 patent"). The 402 patent specifies fixed locations for particular types of packets to extract information to identify session of a packet. For example, if a DECnet packet appears, the 402 patent looks at six specific fields (at 6 locations) in the packet in order to identify the session of the packet. If, on the other hand, an IP packet appears, a different set of six different locations is specified for an IP packet. With the proliferation of protocols, clearly the specifying of all the possible places to look to determine the session becomes more and more difficult. Likewise, adding a new protocol or application is difficult. In the present invention, the locations examined and the information extracted from any packet are adaptively determined from information in the packet for the particular type of packet. There is no fixed definition of what to look for and where to look in order to form an identifying signature. A monitor implementation of the present invention, for example, adapts to handle differently IEEE 802.3 packet from the older Ethernet Type 2 (or Version 2) DIX (Digital-Intel-Xerox) packet.

The 402 patent system is able to recognize up to the session layer. In the present invention, the number of levels examined varies for any particular protocol. Furthermore, the present invention is capable of examining up to whatever level is sufficient to uniquely identify to a required level, even all the way to the application level (in the OSI model).

Other prior art systems also are known. Phael describes a network activity monitor that processes only randomly selected packets in United States Patent 5,315,580, titled "NETWORK MONITORING DEVICE AND SYSTEM." Nakamura teaches a network monitoring system in United States Patent 4,891,639, titled "MONITORING SYSTEM OF NETWORK." Ross, *et al.*, teach a method and apparatus for analyzing and monitoring network activity in United States Patent 5,247,517, titled "METHOD AND

APPARATUS FOR ANALYSIS NETWORKS," McCreery, *et al.*, describe an Internet activity monitor that decodes packet data at the Internet protocol level layer in United States Patent 5,787,253, titled "APPARATUS AND METHOD OF ANALYZING INTERNET ACTIVITY." The McCreery method decodes IP-packets. It goes through the decoding operations for each packet, and therefore uses the processing overhead for both recognized and unrecognized flows. In a monitor implementation of the present invention, a signature is built for every flow such that future packets of the flow are easily recognized. When a new packet in the flow arrives, the recognition process can commence from where it last left off, and a new signature built to recognize new packets of the flow.

SUMMARY

In its various embodiments the present invention provides a network monitor that can accomplish one or more of the following objects and advantages:

- Recognize and classify all packets that are exchanges between a client and server into respective client/server applications.
- Recognize and classify at all protocol layer levels conversational flows that pass in either direction at a point in a network.
- Determine the connection and flow progress between clients and servers according to the individual packets exchanged over a network.
- Be used to help tune the performance of a network according to the current mix of client/server applications requiring network resources.
- Maintain statistics relevant to the mix of client/server applications using network resources.
- Report on the occurrences of specific sequences of packets used by particular applications for client/server network conversational flows.

Other aspects of embodiments of the invention are:

- Properly analyzing each of the packets exchanged between a client and a server and maintaining information relevant to the current state of each of these conversational flows.

- Providing a flexible processing system that can be tailored or adapted as new applications enter the client/server market.
- Maintaining statistics relevant to the conversational flows in a client/sever network as classified by an individual application.
- Reporting a specific identifier, which may be used by other network-oriented devices to identify the series of packets with a specific application for a specific client/server network conversational flow.

In general, the embodiments of the present invention overcome the problems and disadvantages of the art.

As described herein, one embodiment analyzes each of the packets passing through any point in the network in either direction, in order to derive the actual application used to communicate between a client and a server. Note that there could be several simultaneous and overlapping applications executing over the network that are independent and asynchronous.

A monitor embodiment of the invention successfully classifies each of the individual packets as they are seen on the network. The contents of the packets are parsed and selected parts are assembled into a signature (also called a key) that may then be used identify further packets of the same conversational flow, for example to further analyze the flow and ultimately to recognize the application program. Thus the key is a function of the selected parts, and in the preferred embodiment, the function is a concatenation of the selected parts. The preferred embodiment forms and remembers the state of any conversational flow, which is determined by the relationship between individual packets and the entire conversational flow over the network. By remembering the state of a flow in this way, the embodiment determines the context of the conversational flow, including the application program it relates to and parameters such as the time, length of the conversational flow, data rate, etc.

The monitor is flexible to adapt to future applications developed for client/server networks. New protocols and protocol combinations may be incorporated by compiling files written in a high-level protocol description language.

The monitor embodiment of the present invention is preferably implemented in application-specific integrated circuits (ASIC) or field programmable gate arrays (FPGA). In one embodiment, the monitor comprises a parser subsystem that forms a signature from a packet. The monitor further comprises an analyzer subsystem that receives the signature from the parser subsystem.

A packet acquisition device such as a media access controller (MAC) or a segmentation and reassemble module is used to provide packets to the parser subsystem of the monitor.

In a hardware implementation, the parsing subsystem comprises two sub-parts, the pattern analysis and recognition engine (PRE), and an extraction engine (slicer). The PRE interprets each packet, and in particular, interprets individual fields in each packet according to a pattern database.

The different protocols that can exist in different layers may be thought of as nodes of one or more trees of linked nodes. The packet type is the root of a tree. Each protocol is either a parent node or a terminal node. A parent node links a protocol to other protocols (child protocols) that can be at higher layer levels. For example, An Ethernet packet (the root node) may be an Ethertype packet—also called an Ethernet Type/Version 2 and a DIX (DIGITAL-Intel-Xerox packet)—or an IEEE 802.3 packet. Continuing with the IEEE 802.3-type packet, one of the children nodes may be the IP protocol, and one of the children of the IP protocol may be the TCP protocol.

The pattern database includes a description of the different headers of packets and their contents, and how these relate to the different nodes in a tree. The PRE traverses the tree as far as it can. If a node does not include a link to a deeper level, pattern matching is declared complete. Note that protocols can be the children of several parents. If a unique node was generated for each of the possible parent/child trees, the pattern database might become excessively large. Instead, child nodes are shared among multiple parents, thus compacting the pattern database.

Finally the PRE can be used on its own when only protocol recognition is required.

For each protocol recognized, the slicer extracts important packet elements from the packet. These form a signature (*i.e.*, key) for the packet. The slicer also preferably

generates a hash for rapidly identifying a flow that may have this signature from a database of known flows.

The flow signature of the packet, the hash and at least some of the payload are passed to an analyzer subsystem. In a hardware embodiment, the analyzer subsystem includes a unified flow key buffer (UFKB) for receiving parts of packets from the parser subsystem and for storing signatures in process, a lookup/update engine (LUE) to lookup a database of flow records for previously encountered conversational flows to determine whether a signature is from an existing flow, a state processor (SP) for performing state processing, a flow insertion and deletion engine (FIDE) for inserting new flows into the database of flows, a memory for storing the database of flows, and a cache for speeding up access to the memory containing the flow database. The LUE, SP, and FIDE are all coupled to the UFKB, and to the cache.

The unified flow key buffer thus contains the flow signature of the packet, the hash and at least some of the payload for analysis in the analyzer subsystem. Many operations can be performed to further elucidate the identity of the application program content of the packet involved in the client/server conversational flow while a packet signature exists in the unified flow signature buffer. In the particular hardware embodiment of the analyzer subsystem several flows may be processed in parallel, and multiple flow signatures from all the packets being analyzed in parallel may be held in the one UFKB.

The first step in the packet analysis process of a packet from the parser subsystem is to lookup the instance in the current database of known packet flow signatures. A lookup/update engine (LUE) accomplishes this task using first the hash, and then the flow signature. The search is carried out in the cache and if there is no flow with a matching signature in the cache, the lookup engine attempts to retrieve the flow from the flow database in the memory. The flow-entry for previously encountered flows preferably includes state information, which is used in the state processor to execute any operations defined for the state, and to determine the next state. A typical state operation may be to search for one or more known reference strings in the payload of the packet stored in the UFKB.

Once the lookup processing by the LUE has been completed a flag stating whether

it is found or is new is set within the unified flow signature buffer structure for this packet flow signature. For an existing flow, the flow-entry is updated by a calculator component of the LUE that adds values to counters in the flow-entry database used to store one or more statistical measures of the flow. The counters are used for determining network usage metrics on the flow.

After the packet flow signature has been looked up and contents of the current flow signature are in the database, a state processor can begin analyzing the packet payload to further elucidate the identity of the application program component of this packet. The exact operation of the state processor and functions performed by it will vary depending on the current packet sequence in the stream of a conversational flow. The state processor moves to the next logical operation stored from the previous packet seen with this same flow signature. If any processing is required on this packet, the state processor will execute instructions from a database of state instruction for this state until there are either no more left or the instruction signifies processing.

In the preferred embodiment, the state processor functions are programmable to provide for analyzing new application programs, and new sequences of packets and states that can arise from using such application.

If during the lookup process for this particular packet flow signature, the flow is required to be inserted into the active database, a flow insertion and deletion engine (FIDE) is initiated. The state processor also may create new flow signatures and thus may instruct the flow insertion and deletion engine to add a new flow to the database as a new item.

In the preferred hardware embodiment, each of the LUE, state processor, and FIDE operate independently from the other two engines.

BRIEF DESCRIPTION OF THE DRAWINGS

Although the present invention is better understood by referring to the detailed preferred embodiments, these should not be taken to limit the present invention to any specific embodiment because such embodiments are provided only for the purposes of explanation. The embodiments, in turn, are explained with the aid of the following figures.

FIG. 1 is a functional block diagram of a network embodiment of the present invention in which a monitor is connected to analyze packets passing at a connection point.

FIG. 2 is a diagram representing an example of some of the packets and their formats that might be exchanged in starting, as an illustrative example, a conversational flow between a client and server on a network being monitored and analyzed. A pair of flow signatures particular to this example and to embodiments of the present invention is also illustrated. This represents some of the possible flow signatures that can be generated and used in the process of analyzing packets and of recognizing the particular server applications that produce the discrete application packet exchanges.

FIG. 3 is a functional block diagram of a process embodiment of the present invention that can operate as the packet monitor shown in FIG. 1. This process may be implemented in software or hardware.

FIG. 4 is a flowchart of a high-level protocol language compiling and optimization process, which in one embodiment may be used to generate data for monitoring packets according to versions of the present invention.

FIG. 5 is a flowchart of a packet parsing process used as part of the parser in an embodiment of the inventive packet monitor.

FIG. 6 is a flowchart of a packet element extraction process that is used as part of the parser in an embodiment of the inventive packet monitor.

FIG. 7 is a flowchart of a flow-signature building process that is used as part of the parser in the inventive packet monitor.

FIG. 8 is a flowchart of a monitor lookup and update process that is used as part of the analyzer in an embodiment of the inventive packet monitor.

FIG. 9 is a flowchart of an exemplary Sun Microsystems Remote Procedure Call application than may be recognized by the inventive packet monitor.

FIG. 10 is a functional block diagram of a hardware parser subsystem including the pattern recognizer and extractor that can form part of the parser module in an embodiment of the inventive packet monitor.

FIG. 11 is a functional block diagram of a hardware analyzer including a state processor that can form part of an embodiment of the inventive packet monitor.

FIG. 12 is a functional block diagram of a flow insertion and deletion engine process that can form part of the analyzer in an embodiment of the inventive packet monitor.

FIG. 13 is a flowchart of a state processing process that can form part of the analyzer in an embodiment of the inventive packet monitor.

FIG. 14 is a simple functional block diagram of a process embodiment of the present invention that can operate as the packet monitor shown in FIG. 1. This process may be implemented in software.

FIG. 15 is a functional block diagram of how the packet monitor of FIG. 3 (and FIGS. 10 and 11) may operate on a network with a processor such as a microprocessor.

FIG. 16 is an example of the top (MAC) layer of an Ethernet packet and some of the elements that may be extracted to form a signature according to one aspect of the invention.

FIG. 17A is an example of the header of an Ethertype type of Ethernet packet of FIG. 16 and some of the elements that may be extracted to form a signature according to one aspect of the invention.

FIG. 17B is an example of an IP packet, for example, of the Ethertype packet shown in FIGs. 16 and 17A, and some of the elements that may be extracted to form a signature according to one aspect of the invention.

FIG. 18A is a three dimensional structure that can be used to store elements of the pattern, parse and extraction database used by the parser subsystem in accordance to one embodiment of the invention.

FIG. 18B is an alternate form of storing elements of the pattern, parse and extraction database used by the parser subsystem in accordance to another embodiment of the invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Note that this document includes hardware diagrams and descriptions that may include signal names. In most cases, the names are sufficiently descriptive, in other cases however the signal names are not needed to understand the operation and practice of the invention.

Operation in a Network

FIG. 1 represents a system embodiment of the present invention that is referred to herein by the general reference numeral 100. The system 100 has a computer network 102 that communicates packets (*e.g.,* IP datagrams) between various computers, for example between the clients 104–107 and servers 110 and 112. The network is shown schematically as a cloud with several network nodes and links shown in the interior of the cloud. A monitor 108 examines the packets passing in either direction past its connection point 121 and, according to one aspect of the invention, can elucidate what application programs are associated with each packet. The monitor 108 is shown examining packets (*i.e.,* datagrams) between the network interface 116 of the server 110 and the network. The monitor can also be placed at other points in the network, such as connection point 123 between the network 102 and the interface 118 of the client 104, or some other location, as indicated schematically by connection point 125 somewhere in network 102. Not shown is a network packet acquisition device at the location 123 on the network for converting the physical information on the network into packets for input into monitor 108. Such packet acquisition devices are common.

Various protocols may be employed by the network to establish and maintain the required communication, *e.g.,* TCP/IP, etc. Any network activity—for example an application program run by the client 104 (CLIENT 1) communicating with another running on the server 110 (SERVER 2)—will produce an exchange of a sequence of packets over network 102 that is characteristic of the respective programs and of the network protocols. Such characteristics may not be completely revealing at the individual packet level. It may require the analyzing of many packets by the monitor 108 to have enough information needed to recognize particular application programs. The packets may need to be parsed then analyzed in the context of various protocols, for example, the

transport through the application session layer protocols for packets of a type conforming to the ISO layered network model.

Communication protocols are layered, which is also referred to as a protocol stack. The ISO (International Standardization Organization) has defined a general model that provides a framework for design of communication protocol layers. This model, shown in table form below, serves as a basic reference for understanding the functionality of existing communication protocols.

ISO MODEL

Layer	Functionality	Example
7	Application	Telnet, NFS, Novell NCP, HTTP, H.323
6	Presentation	XDR
5	Session	RPC, NETBIOS, SNMP, *etc.*
4	Transport	TCP, Novel SPX, UDP, *etc.*
3	Network	IP, Novell IPX, VIP, AppleTalk, *etc.*
2	Data Link	Network Interface Card (Hardware Interface). MAC layer
1	Physical	Ethernet, Token Ring, Frame Relay, ATM, T1 (Hardware Connection)

Different communication protocols employ different levels of the ISO model or may use a layered model that is similar to but which does not exactly conform to the ISO model. A protocol in a certain layer may not be visible to protocols employed at other layers. For example, an application (Level 7) may not be able to identify the source computer for a communication attempt (Levels 2–3).

In some communication arts, the term "frame" generally refers to encapsulated data at OSI layer 2, including a destination address, control bits for flow control, the data or payload, and CRC (cyclic redundancy check) data for error checking. The term

"packet" generally refers to encapsulated data at OSI layer 3. In the TCP/IP world, the term "datagram" is also used. In this specification, the term "packet" is intended to encompass packets, datagrams, frames, and cells. In general, a packet format or frame format refers to how data is encapsulated with various fields and headers for transmission across a network. For example, a data packet typically includes an address destination field, a length field, an error correcting code (ECC) field, or cyclic redundancy check (CRC) field, as well as headers and footers to identify the beginning and end of the packet. The terms "packet format" and "frame format," also referred to as "cell format," are generally synonymous.

Monitor 108 looks at every packet passing the connection point 121 for analysis. However, not every packet carries the same information useful for recognizing all levels of the protocol. For example, in a conversational flow associated with a particular application, the application will cause the server to send a type-A packet, but so will another. If, though, the particular application program always follows a type-A packet with the sending of a type-B packet, and the other application program does not, then in order to recognize packets of that application's conversational flow, the monitor can be available to recognize packets that match the type-B packet to associate with the type-A packet. If such is recognized after a type-A packet, then the particular application program's conversational flow has started to reveal itself to the monitor 108.

Further packets may need to be examined before the conversational flow can be identified as being associated with the application program. Typically, monitor 108 is simultaneously also in partial completion of identifying other packet exchanges that are parts of conversational flows associated with other applications. One aspect of monitor 108 is its ability to maintain the state of a flow. The state of a flow is an indication of all previous events in the flow that lead to recognition of the content of all the protocol levels, *e.g.*, the ISO model protocol levels. Another aspect of the invention is forming a signature of extracted characteristic portions of the packet that can be used to rapidly identify packets belonging to the same flow.

In real-world uses of the monitor 108, the number of packets on the network 102 passing by the monitor 108's connection point can exceed a million per second. Consequently, the monitor has very little time available to analyze and type each packet and identify and maintain the state of the flows passing through the connection point. The

monitor 108 therefore masks out all the unimportant parts of each packet that will not contribute to its classification. However, the parts to mask-out will change with each packet depending on which flow it belongs to and depending on the state of the flow.

The recognition of the packet type, and ultimately of the associated application programs according to the packets that their executions produce, is a multi-step process within the monitor 108. At a first level, for example, several application programs will all produce a first kind of packet. A first "signature" is produced from selected parts of a packet that will allow monitor 108 to identify efficiently any packets that belong to the same flow. In some cases, that packet type may be sufficiently unique to enable the monitor to identify the application that generated such a packet in the conversational flow. The signature can then be used to efficiently identify all future packets generated in traffic related to that application.

In other cases, that first packet only starts the process of analyzing the conversational flow, and more packets are necessary to identify the associated application program. In such a case, a subsequent packet of a second type—but that potentially belongs to the same conversational flow—is recognized by using the signature. At such a second level, then, only a few of those application programs will have conversational flows that can produce such a second packet type. At this level in the process of classification, all application programs that are not in the set of those that lead to such a sequence of packet types may be excluded in the process of classifying the conversational flow that includes these two packets. Based on the known patterns for the protocol and for the possible applications, a signature is produced that allows recognition of any future packets that may follow in the conversational flow.

It may be that the application is now recognized, or recognition may need to proceed to a third level of analysis using the second level signature. For each packet, therefore, the monitor parses the packet and generates a signature to determine if this signature identified a previously encountered flow, or shall be used to recognize future packets belonging to the same conversational flow. In real time, the packet is further analyzed in the context of the sequence of previously encountered packets (the state), and of the possible future sequences such a past sequence may generate in conversational flows associated with different applications. A new signature for recognizing future packets may also be generated. This process of analysis continues until the applications

are identified. The last generated signature may then be used to efficiently recognize future packets associated with the same conversational flow. Such an arrangement makes it possible for the monitor 108 to cope with millions of packets per second that must be inspected.

Another aspect of the invention is adding Eavesdropping. In alternative embodiments of the present invention capable of eavesdropping, once the monitor 108 has recognized the executing application programs passing through some point in the network 102 (for example, because of execution of the applications by the client 105 or server 110), the monitor sends a message to some general purpose processor on the network that can input the same packets from the same location on the network, and the processor then loads its own executable copy of the application program and uses it to read the content being exchanged over the network. In other words, once the monitor 108 has accomplished recognition of the application program, eavesdropping can commence.

The Network Monitor

FIG. 3 shows a network packet monitor 300, in an embodiment of the present invention that can be implemented with computer hardware and/or software. The system 300 is similar to monitor 108 in FIG. 1. A packet 302 is examined, *e.g.*, from a packet acquisition device at the location 121 in network 102 (FIG. 1), and the packet evaluated, for example in an attempt to determine its characteristics, *e.g.*, all the protocol information in a multilevel model, including what server application produced the packet.

The packet acquisition device is a common interface that converts the physical signals and then decodes them into bits, and into packets, in accordance with the particular network (Ethernet, frame relay, ATM, *etc.*). The acquisition device indicates to the monitor 108 the type of network of the acquired packet or packets.

Aspects shown here include: (1) the initialization of the monitor to generate what operations need to occur on packets of different types—accomplished by compiler and optimizer 310, (2) the processing—parsing and extraction of selected portions—of packets to generate an identifying signature—accomplished by parser subsystem 301, and (3) the analysis of the packets—accomplished by analyzer 303.

The purpose of compiler and optimizer 310 is to provide protocol specific information to parser subsystem 301 and to analyzer subsystem 303. The initialization

occurs prior to operation of the monitor, and only needs to re-occur when new protocols are to be added.

A flow is a stream of packets being exchanged between any two addresses in the network. For each protocol there are known to be several fields, such as the destination (recipient), the source (the sender), and so forth, and these and other fields are used in monitor 300 to identify the flow. There are other fields not important for identifying the flow, such as checksums, and those parts are not used for identification.

Parser subsystem 301 examines the packets using pattern recognition process 304 that parses the packet and determines the protocol types and associated headers for each protocol layer that exists in the packet 302. An extraction process 306 in parser subsystem 301 extracts characteristic portions (signature information) from the packet 302. Both the pattern information for parsing and the related extraction operations, *e.g.*, extraction masks, are supplied from a parsing-pattern-structures and extraction-operations database (parsing/extractions database) 308 filled by the compiler and optimizer 310.

The protocol description language (PDL) files 336 describes both patterns and states of all protocols that an occur at any layer, including how to interpret header information, how to determine from the packet header information the protocols at the next layer, and what information to extract for the purpose of identifying a flow, and ultimately, applications and services. The layer selections database 338 describes the particular layering handled by the monitor. That is, what protocols run on top of what protocols at any layer level. Thus 336 and 338 combined describe how one would decode, analyze, and understand the information in packets, and, furthermore, how the information is layered. This information is input into compiler and optimizer 310.

When compiler and optimizer 310 executes, it generates two sets of internal data structures. The first is the set of parsing/extraction operations 308. The pattern structures include parsing information and describe what will be recognized in the headers of packets; the extraction operations are what elements of a packet are to be extracted from the packets based on the patterns that get matched. Thus, database 308 of parsing/extraction operations includes information describing how to determine a set of one or more protocol dependent extraction operations from data in the packet that indicate a protocol used in the packet.

The other internal data structure that is built by compiler 310 is the set of state patterns and processes 326. These are the different states and state transitions that occur in different conversational flows, and the state operations that need to be performed (*e.g.*, patterns that need to be examined and new signatures that need to be built) during any state of a conversational flow to further the task of analyzing the conversational flow.

Thus, compiling the PDL files and layer selections provides monitor 300 with the information it needs to begin processing packets. In an alternate embodiment, the contents of one or more of databases 308 and 326 may be manually or otherwise generated. Note that in some embodiments the layering selections information is inherent rather than explicitly described. For example, since a PDL file for a protocol includes the child protocols, the parent protocols also may be determined.

In the preferred embodiment, the packet 302 from the acquisition device is input into a packet buffer. The pattern recognition process 304 is carried out by a pattern analysis and recognition (PAR) engine that analyzes and recognizes patterns in the packets. In particular, the PAR locates the next protocol field in the header and determines the length of the header, and may perform certain other tasks for certain types of protocol headers. An example of this is type and length comparison to distinguish an IEEE 802.3 (Ethernet) packet from the older type 2 (or Version 2) Ethernet packet, also called a DIGITAL-Intel-Xerox (DIX) packet. The PAR also uses the pattern structures and extraction operations database 308 to identify the next protocol and parameters associated with that protocol that enables analysis of the next protocol layer. Once a pattern or a set of patterns has been identified, it/they will be associated with a set of none or more extraction operations. These extraction operations (in the form of commands and associated parameters) are passed to the extraction process 306 implemented by an extracting and information identifying (EII) engine that extracts selected parts of the packet, including identifying information from the packet as required for recognizing this packet as part of a flow. The extracted information is put in sequence and then processed in block 312 to build a unique flow signature (also called a "key") for this flow. A flow signature depends on the protocols used in the packet. For some protocols, the extracted components may include source and destination addresses. For example, Ethernet frames have end-point addresses that are useful in building a better flow signature. Thus, the signature typically includes the client and server address pairs. The signature is used to

recognize further packets that are or may be part of this flow.

In the preferred embodiment, the building of the flow key includes generating a hash of the signature using a hash function. The purpose if using such a hash is conventional—to spread flow-entries identified by the signature across a database for efficient searching. The hash generated is preferably based on a hashing algorithm and such hash generation is known to those in the art.

In one embodiment, the parser passes data from the packet—a parser record—that includes the signature (i.e., selected portions of the packet), the hash, and the packet itself to allow for any state processing that requires further data from the packet. An improved embodiment of the parser subsystem might generate a parser record that has some predefined structure and that includes the signature, the hash, some flags related to some of the fields in the parser record, and parts of the packet's payload that the parser subsystem has determined might be required for further processing, e.g., for state processing.

Note that alternate embodiments may use some function other than concatenation of the selected portions of the packet to make the identifying signature. For example, some "digest function" of the concatenated selected portions may be used.

The parser record is passed onto lookup process 314 which looks in an internal data store of records of known flows that the system has already encountered, and decides (in 316) whether or not this particular packet belongs to a known flow as indicated by the presence of a flow-entry matching this flow in a database of known flows 324. A record in database 324 is associated with each encountered flow.

The parser record enters a buffer called the unified flow key buffer (UFKB). The UFKB stores the data on flows in a data structure that is similar to the parser record, but that includes a field that can be modified. In particular, one or the UFKB record fields stores the packet sequence number, and another is filled with state information in the form of a program counter for a state processor that implements state processing 328.

The determination (316) of whether a record with the same signature already exists is carried out by a lookup engine (LUE) that obtains new UFKB records and uses the hash in the UFKB record to lookup if there is a matching known flow. In the particular embodiment, the database of known flows 324 is in an external memory. A

cache is associated with the database 324. A lookup by the LUE for a known record is carried out by accessing the cache using the hash, and if the entry is not already present in the cache, the entry is looked up (again using the hash) in the external memory.

The flow-entry database 324 stores flow-entries that include the unique flow-signature, state information, and extracted information from the packet for updating flows, and one or more statistical about the flow. Each entry completely describes a flow. Database 324 is organized into bins that contain a number, denoted N, of flow-entries (also called flow-entries, each a bucket), with N being 4 in the preferred embodiment. Buckets (i.e., flow-entries) are accessed via the hash of the packet from the parser subsystem 301 (i.e., the hash in the UFKB record). The hash spreads the flows across the database to allow for fast lookups of entries, allowing shallower buckets. The designer selects the bucket depth N based on the amount of memory attached to the monitor, and the number of bits of the hash data value used. For example, in one embodiment, each flow-entry is 128 bytes long, so for 128K flow-entries, 16 Mbytes are required. Using a 16-bit hash gives two flow-entries per bucket. Empirically, this has been shown to be more than adequate for the vast majority of cases. Note that another embodiment uses flow-entries that are 256 bytes long.

Herein, whenever an access to database 324 is described, it is to be understood that the access is via the cache, unless otherwise stated or clear from the context.

If there is no flow-entry found matching the signature, i.e., the signature is for a new flow, then a protocol and state identification process 318 further determines the state and protocol. That is, process 318 determines the protocols and where in the state sequence for a flow for this protocol's this packet belongs. Identification process 318 uses the extracted information and makes reference to the database 326 of state patterns and processes. Process 318 is then followed by any state operations that need to be executed on this packet by a state processor 328.

If the packet is found to have a matching flow-entry in the database 324 (e.g., in the cache), then a process 320 determines, from the looked-up flow-entry, if more classification by state processing of the flow signature is necessary. If not, a process 322 updates the flow-entry in the flow-entry database 324 (e.g., via the cache). Updating includes updating one or more statistical measures stored in the flow-entry. In our

embodiment, the statistical measures are stored in counters in the flow-entry.

If state processing is required, state process 328 is commenced. State processor 328 carries out any state operations specified for the state of the flow and updates the state to the next state according to a set of state instructions obtained form the state pattern and processes database 326.

The state processor 328 analyzes both new and existing flows in order to analyze all levels of the protocol stack, ultimately classifying the flows by application (level 7 in the ISO model). It does this by proceeding from state-to-state based on predefined state transition rules and state operations as specified in state processor instruction database 326. A state transition rule is a rule typically containing a test followed by the next-state to proceed to if the test result is true. An operation is an operation to be performed while the state processor is in a particular state—for example, in order to evaluate a quantity needed to apply the state transition rule. The state processor goes through each rule and each state process until the test is true, or there are no more tests to perform.

In general, the set of state operations may be none or more operations on a packet, and carrying out the operation or operations may leave one in a state that causes exiting the system prior to completing the identification, but possibly knowing more about what state and state processes are needed to execute next, *i.e.*, when a next packet of this flow is encountered. As an example, a state process (set of state operations) at a particular state may build a new signature for future recognition packets of the next state.

By maintaining the state of the flows and knowing that new flows may be set up using the information from previously encountered flows, the network traffic monitor 300 provides for (a) single-packet protocol recognition of flows, and (b) multiple-packet protocol recognition of flows. Monitor 300 can even recognize the application program from one or more disjointed sub-flows that occur in server announcement type flows. What may seem to prior art monitors to be some unassociated flow, may be recognized by the inventive monitor using the flow signature to be a sub-flow associated with a previously encountered sub-flow.

Thus, state processor 328 applies the first state operation to the packet for this particular flow-entry. A process 330 decides if more operations need to be performed for this state. If so, the analyzer continues looping between block 330 and 328 applying

additional state operations to this particular packet until all those operations are completed—that is, there are no more operations for this packet in this state. A process 332 decides if there are further states to be analyzed for this type of flow according to the state of the flow and the protocol, in order to fully characterize the flow. If not, the conversational flow has now been fully characterized and a process 334 finalizes the classification of the conversational flow for the flow.

In the particular embodiment, the state processor 328 starts the state processing by using the last protocol recognized by the parser as an offset into a jump table (jump vector). The jump table finds the state processor instructions to use for that protocol in the state patterns and processes database 326. Most instructions test something in the unified flow key buffer, or the flow-entry in the database of known flows 324, if the entry exists. The state processor may have to test bits, do comparisons, add, or subtract to perform the test. For example, a common operation carried out by the state processor is searching for one or more patterns in the payload part of the UFKB.

Thus, in 332 in the classification, the analyzer decides whether the flow is at an end state. If not at an end state, the flow-entry is updated (or created if a new flow) for this flow-entry in process 322.

Furthermore, if the flow is known and if in 332 it is determined that there are further states to be processed using later packets, the flow-entry is updated in process 322.

The flow-entry also is updated after classification finalization so that any further packets belonging to this flow will be readily identified from their signature as belonging to this fully analyzed conversational flow.

After updating, database 324 therefore includes the set of all the conversational flows that have occurred.

Thus, the embodiment of present invention shown in FIG. 3 automatically maintains flow-entries, which in one aspect includes storing states. The monitor of FIG. 3 also generates characteristic parts of packets—the signatures—that can be used to recognize flows. The flow-entries may be identified and accessed by their signatures. Once a packet is identified to be from a known flow, the state of the flow is known and this knowledge enables state transition analysis to be performed in real time for each different protocol and application. In a complex analysis, state transitions are traversed as

more and more packets are examined. Future packets that are part of the same conversational flow have their state analysis continued from a previously achieved state. When enough packets related to an application of interest have been processed, a final recognition state is ultimately reached, *i.e.*, a set of states has been traversed by state analysis to completely characterize the conversational flow. The signature for that final state enables each new incoming packet of the same conversational flow to be individually recognized in real time.

In this manner, one of the great advantages of the present invention is realized. Once a particular set of state transitions has been traversed for the first time and ends in a final state, a short-cut recognition pattern—a signature—can be generated that will key on every new incoming packet that relates to the conversational flow. Checking a signature involves a simple operation, allowing high packet rates to be successfully monitored on the network.

In improved embodiments, several state analyzers are run in parallel so that a large number of protocols and applications may be checked for. Every known protocol and application will have at least one unique set of state transitions, and can therefore be uniquely identified by watching such transitions.

When each new conversational flow starts, signatures that recognize the flow are automatically generated on-the-fly, and as further packets in the conversational flow are encountered, signatures are updated and the states of the set of state transitions for any potential application are further traversed according to the state transition rules for the flow. The new states for the flow—those associated with a set of state transitions for one or more potential applications—are added to the records of previously encountered states for easy recognition and retrieval when a new packet in the flow is encountered.

Detailed operation

FIG. 4 diagrams an initialization system 400 that includes the compilation process. That is, part of the initialization generates the pattern structures and extraction operations database 308 and the state instruction database 328. Such initialization can occur off-line or from a central location.

The different protocols that can exist in different layers may be thought of as nodes of one or more trees of linked nodes. The packet type is the root of a tree (called

level 0). Each protocol is either a parent node or a terminal node. A parent node links a protocol to other protocols (child protocols) that can be at higher layer levels. Thus a protocol may have zero or more children. Ethernet packets, for example, have several variants, each having a basic format that remains substantially the same. An Ethernet packet (the root or level 0 node) may be an Ethertype packet—also called an Ethernet Type/Version 2 and a DIX (DIGITAL-Intel-Xerox packet)—or an IEEE 803.2 packet. Continuing with the IEEE 802.3 packet, one of the children nodes may be the IP protocol, and one of the children of the IP protocol may be the TCP protocol.

FIG. 16 shows the header 1600 (base level 1) of a complete Ethernet frame (*i.e.*, packet) of information and includes information on the destination media access control address (Dst MAC 1602) and the source media access control address (Src MAC 1604). Also shown in FIG. 16 is some (but not all) of the information specified in the PDL files for extraction the signature.

FIG. 17A now shows the header information for the next level (level-2) for an Ethertype packet 1700. For an Ethertype packet 1700, the relevant information from the packet that indicates the next layer level is a two-byte type field 1702 containing the child recognition pattern for the next level. The remaining information 1704 is shown hatched because it not relevant for this level. The list 1712 shows the possible children for an Ethertype packet as indicated by what child recognition pattern is found offset 12. FIG. 17B shows the structure of the header of one of the possible next levels, that of the IP protocol. The possible children of the IP protocol are shown in table 1752.

The pattern, parse, and extraction database (pattern recognition database, or PRD) 308 generated by compilation process 310, in one embodiment, is in the form of a three dimensional structure that provides for rapidly searching packet headers for the next protocol. FIG. 18A shows such a 3-D representation 1800 (which may be considered as an indexed set of 2-D representations). A compressed form of the 3-D structure is preferred.

An alternate embodiment of the data structure used in database 308 is illustrated in FIG. 18B. Thus, like the 3-D structure of FIG. 18A, the data structure permits rapid searches to be performed by the pattern recognition process 304 by indexing locations in a memory rather than performing address link computations. In this alternate embodiment,

the PRD 308 includes two parts, a single protocol table 1850 (PT) which has an entry for each protocol known for the monitor, and a series of Look Up Tables 1870 (LUT's) that are used to identify known protocols and their children. The protocol table includes the parameters needed by the pattern analysis and recognition process 304 (implemented by PRE 1006) to evaluate the header information in the packet that is associated with that protocol, and parameters needed by extraction process 306 (implemented by slicer 1007) to process the packet header. When there are children, the PT describes which bytes in the header to evaluate to determine the child protocol. In particular, each PT entry contains the header length, an offset to the child, a slicer command, and some flags.

The pattern matching is carried out by finding particular "child recognition codes" in the header fields, and using these codes to index one or more of the LUT's. Each LUT entry has a node code that can have one of four values, indicating the protocol that has been recognized, a code to indicate that the protocol has been partially recognized (more LUT lookups are needed), a code to indicate that this is a terminal node, and a null node to indicate a null entry. The next LUT to lookup is also returned from a LUT lookup.

Compilation process is described in FIG. 4. The source-code information in the form of protocol description files is shown as 402. In the particular embodiment, the high level decoding descriptions includes a set of protocol description files 336, one for each protocol, and a set of packet layer selections 338, which describes the particular layering (sets of trees of protocols) that the monitor is to be able to handle.

A compiler 403 compiles the descriptions. The set of packet parse-and-extract operations 406 is generated (404), and a set of packet state instructions and operations 407 is generated (405) in the form of instructions for the state processor that implements state processing process 328. Data files for each type of application and protocol to be recognized by the analyzer are downloaded from the pattern, parse, and extraction database 406 into the memory systems of the parser and extraction engines. (See the parsing process 500 description and FIG. 5; the extraction process 600 description and FIG. 6; and the parsing subsystem hardware description and FIG. 10). Data files for each type of application and protocol to be recognized by the analyzer are also downloaded from the state-processor instruction database 407 into the state processor. (see the state processor 1108 description and FIG. 11.).

Note that generating the packet parse and extraction operations builds and links the three dimensional structure (one embodiment) or the or all the lookup tables for the PRD.

Because of the large number of possible protocol trees and subtrees, the compiler process 400 includes optimization that compares the trees and subtrees to see which children share common parents. When implemented in the form of the LUT's, this process can generate a single LUT from a plurality of LUT's. The optimization process further includes a compaction process that reduces the space needed to store the data of the PRD.

As an example of compaction, consider the 3-D structure of FIG. 18A that can be thought of as a set of 2-D structures each representing a protocol. To enable saving space by using only one array per protocol which may have several parents, in one embodiment, the pattern analysis subprocess keeps a "current header" pointer. Each location (offset) index for each protocol 2-D array in the 3-D structure is a relative location starting with the start of header for the particular protocol. Furthermore, each of the two-dimensional arrays is sparse. The next step of the optimization, is checking all the 2-D arrays against all the other 2-D arrays to find out which ones can share memory. Many of these 2-D arrays are often sparsely populated in that they each have only a small number of valid entries. So, a process of "folding" is next used to combine two or more 2-D arrays together into one physical 2-D array without losing the identity of any of the original 2-D arrays (i.e., all the 2-D arrays continue to exist logically). Folding can occur between any 2-D arrays irrespective of their location in the tree as long as certain conditions are met. Multiple arrays may be combined into a single array as long as the individual entries do not conflict with each other. A fold number is then used to associate each element with its original array. A similar folding process is used for the set of LUTs 1850 in the alternate embodiment of FIG. 18B.

In 410, the analyzer has been initialized and is ready to perform recognition.

FIG. 5 shows a flowchart of how actual parser subsystem 301 functions. Starting at 501, the packet 302 is input to the packet buffer in step 502. Step 503 loads the next (initially the first) packet component from the packet 302. The packet components are extracted from each packet 302 one element at a time. A check is made (504) to determine

if the load-packet-component operation 503 succeeded, indicating that there was more in the packet to process. If not, indicating all components have been loaded, the parser subsystem 301 builds the packet signature (512)—the next stage (FIG 6).

If a component is successfully loaded in 503, the node and processes are fetched (505) from the pattern, parse and extraction database 308 to provide a set of patterns and processes for that node to apply to the loaded packet component. The parser subsystem 301 checks (506) to determine if the fetch pattern node operation 505 completed successfully, indicating there was a pattern node that loaded in 505. If not, step 511 moves to the next packet component. If yes, then the node and pattern matching process are applied in 507 to the component extracted in 503. A pattern match obtained in 507 (as indicated by test 508) means the parser subsystem 301 has found a node in the parsing elements; the parser subsystem 301 proceeds to step 509 to extract the elements.

If applying the node process to the component does not produce a match (test 508), the parser subsystem 301 moves (510) to the next pattern node from the pattern database 308 and to step 505 to fetch the next node and process. Thus, there is an "applying patterns" loop between 508 and 505. Once the parser subsystem 301 completes all the patterns and has either matched or not, the parser subsystem 301 moves to the next packet component (511).

Once all the packet components have been the loaded and processed from the input packet 302, then the load packet will fail (indicated by test 504), and the parser subsystem 301 moves to build a packet signature which is described in FIG. 6

FIG. 6 is a flow chart for extracting the information from which to build the packet signature. The flow starts at 601, which is the exit point 513 of FIG. 5. At this point parser subsystem 301 has a completed packet component and a pattern node available in a buffer (602). Step 603 loads the packet component available from the pattern analysis process of FIG. 5. If the load completed (test 604), indicating that there was indeed another packet component, the parser subsystem 301 fetches in 605 the extraction and process elements received from the pattern node component in 602. If the fetch was successful (test 606), indicating that there are extraction elements to apply, the parser subsystem 301 in step 607 applies that extraction process to the packet component based on an extraction instruction received from that pattern node. This removes and

saves an element from the packet component.

In step 608, the parser subsystem 301 checks if there is more to extract from this component, and if not, the parser subsystem 301 moves back to 603 to load the next packet component at hand and repeats the process. If the answer is yes, then the parser subsystem 301 moves to the next packet component ratchet. That new packet component is then loaded in step 603. As the parser subsystem 301 moved through the loop between 608 and 603, extra extraction processes are applied either to the same packet component if there is more to extract, or to a different packet component if there is no more to extract.

The extraction process thus builds the signature, extracting more and more components according to the information in the patterns and extraction database 308 for the particular packet. Once loading the next packet component operation 603 fails (test 604), all the components have been extracted. The built signature is loaded into the signature buffer (610) and the parser subsystem 301 proceeds to FIG. 7 to complete the signature generation process.

Referring now to FIG. 7, the process continues at 701. The signature buffer and the pattern node elements are available (702). The parser subsystem 301 loads the next pattern node element. If the load was successful (test 704) indicating there are more nodes, the parser subsystem 301 in 705 hashes the signature buffer element based on the hash elements that are found in the pattern node that is in the element database. In 706 the resulting signature and the hash are packed. In 707 the parser subsystem 301 moves on to the next packet component which is loaded in 703.

The 703 to 707 loop continues until there are no more patterns of elements left (test 704). Once all the patterns of elements have been hashed, processes 304, 306 and 312 of parser subsystem 301 are complete. Parser subsystem 301 has generated the signature used by the analyzer subsystem 303.

A parser record is loaded into the analyzer, in particular, into the UFKB in the form of a UFKB record which is similar to a parser record, but with one or more different fields.

FIG. 8 is a flow diagram describing the operation of the lookup/update engine (LUE) that implements lookup operation 314. The process starts at 801 from FIG. 7 with the parser record that includes a signature, the hash and at least parts of the payload. In

802 those elements are shown in the form of a UFKB-entry in the buffer. The LUE, the lookup engine 314 computes a "record bin number" from the hash for a flow-entry. A bin herein may have one or more "buckets" each containing a flow-entry. The preferred embodiment has four buckets per bin.

Since preferred hardware embodiment includes the cache, all data accesses to records in the flowchart of FIG. 8 are stated as being to or from the cache.

Thus, in 804, the system looks up the cache for a bucket from that bin using the hash. If the cache successfully returns with a bucket from the bin number, indicating there are more buckets in the bin, the lookup/update engine compares (807) the current signature (the UFKB-entry's signature) from that in the bucket (i.e., the flow-entry signature). If the signatures match (test 808), that record (in the cache) is marked in step 810 as "in process" and a timestamp added. Step 811 indicates to the UFKB that the UFKB-entry in 802 has a status of "found." The "found" indication allows the state processing 328 to begin processing this UFKB element. The preferred hardware embodiment includes one or more state processors, and these can operate in parallel with the lookup/update engine.

In the preferred embodiment, a set of statistical operations is performed by a calculator for every packet analyzed. The statistical operations may include one or more of counting the packets associated with the flow; determining statistics related to the size of packets of the flow; compiling statistics on differences between packets in each direction, for example using timestamps; and determining statistical relationships of timestamps of packets in the same direction. The statistical measures are kept in the flow-entries. Other statistical measures also may be compiled. These statistics may be used singly or in combination by a statistical processor component to analyze many different aspects of the flow. This may include determining network usage metrics from the statistical measures, for example to ascertain the network's ability to transfer information for this application. Such analysis provides for measuring the quality of service of a conversation, measuring how well an application is performing in the network, measuring network resources consumed by an application, and so forth.

To provide for such analyses, the lookup/update engine updates one or more counters that are part of the flow-entry (in the cache) in step 812. The process exits at 813.

In our embodiment, the counters include the total packets of the flow, the time, and a differential time from the last timestamp to the present timestamp.

It may be that the bucket of the bin did not lead to a signature match (test 808). In such a case, the analyzer in 809 moves to the next bucket for this bin. Step 804 again looks up the cache for another bucket from that bin. The lookup/update engine thus continues lookup up buckets of the bin until there is either a match in 808 or operation 804 is not successful (test 805), indicating that there are no more buckets in the bin and no match was found.

If no match was found, the packet belongs to a new (not previously encountered) flow. In 806 the system indicates that the record in the unified flow key buffer for this packet is new, and in 812, any statistical updating operations are performed for this packet by updating the flow-entry in the cache. The update operation exits at 813. A flow insertion/deletion engine (FIDE) creates a new record for this flow (again via the cache).

Thus, the update/lookup engine ends with a UFKB-entry for the packet with a "new" status or a "found" status.

Note that the above system uses a hash to which more than one flow-entry can match. A longer hash may be used that corresponds to a single flow-entry. In such an embodiment, the flow chart of FIG. 8 is simplified as would be clear to those in the art.

The hardware system

Each of the individual hardware elements through which the data flows in the system are now described with reference to FIGS. 10 and 11. Note that while we are describing a particular hardware implementation of the invention embodiment of FIG. 3, it would be clear to one skilled in the art that the flow of FIG. 3 may alternatively be implemented in software running on one or more general-purpose processors, or only partly implemented in hardware. An implementation of the invention that can operate in software is shown in FIG. 14. The hardware embodiment (FIGS. 10 and 11) can operate at over a million packets per second, while the software system of FIG. 14 may be suitable for slower networks. To one skilled in the art it would be clear that more and more of the system may be implemented in software as processors become faster.

FIG. 10 is a description of the parsing subsystem (301, shown here as subsystem

1000) as implemented in hardware. Memory 1001 is the pattern recognition database memory, in which the patterns that are going to be analyzed are stored. Memory 1002 is the extraction-operation database memory, in which the extraction instructions are stored. Both 1001 and 1002 correspond to internal data structure 308 of FIG. 3. Typically, the system is initialized from a microprocessor (not shown) at which time these memories are loaded through a host interface multiplexor and control register 1005 via the internal buses 1003 and 1004. Note that the contents of 1001 and 1002 are preferably obtained by compiling process 310 of FIG. 3.

A packet enters the parsing system via 1012 into a parser input buffer memory 1008 using control signals 1021 and 1023, which control an input buffer interface controller 1022. The buffer 1008 and interface control 1022 connect to a packet acquisition device (not shown). The buffer acquisition device generates a packet start signal 1021 and the interface control 1022 generates a next packet (i.e., ready to receive data) signal 1023 to control the data flow into parser input buffer memory 1008. Once a packet starts loading into the buffer memory 1008, pattern recognition engine (PRE) 1006 carries out the operations on the input buffer memory described in block 304 of FIG. 3. That is, protocol types and associated headers for each protocol layer that exist in the packet are determined.

The PRE searches database 1001 and the packet in buffer 1008 in order to recognize the protocols the packet contains. In one implementation, the database 1001 includes a series of linked lookup tables. Each lookup table uses eight bits of addressing. The first lookup table is always at address zero. The Pattern Recognition Engine uses a base packet offset from a control register to start the comparison. It loads this value into a current offset pointer (COP). It then reads the byte at base packet offset from the parser input buffer and uses it as an address into the first lookup table.

Each lookup table returns a word that links to another lookup table or it returns a terminal flag. If the lookup produces a recognition event the database also returns a command for the slicer. Finally it returns the value to add to the COP.

The PRE 1006 includes of a comparison engine. The comparison engine has a first stage that checks the protocol type field to determine if it is an 802.3 packet and the field should be treated as a length. If it is not a length, the protocol is checked in a second

stage. The first stage is the only protocol level that is not programmable. The second stage has two full sixteen bit content addressable memories (CAMs) defined for future protocol additions.

Thus, whenever the PRE recognizes a pattern, it also generates a command for the extraction engine (also called a "slicer") 1007. The recognized patterns and the commands are sent to the extraction engine 1007 that extracts information from the packet to build the parser record. Thus, the operations of the extraction engine are those carried out in blocks 306 and 312 of FIG. 3. The commands are sent from PRE 1006 to slicer 1007 in the form of extraction instruction pointers which tell the extraction engine 1007 where to a find the instructions in the extraction operations database memory (i.e., slicer instruction database) 1002.

Thus, when the PRE 1006 recognizes a protocol it outputs both the protocol identifier and a process code to the extractor. The protocol identifier is added to the flow signature and the process code is used to fetch the first instruction from the instruction database 1002. Instructions include an operation code and usually source and destination offsets as well as a length. The offsets and length are in bytes. A typical operation is the MOVE instruction. This instruction tells the slicer 1007 to copy n bytes of data unmodified from the input buffer 1008 to the output buffer 1010. The extractor contains a byte-wise barrel shifter so that the bytes moved can be packed into the flow signature. The extractor contains another instruction called HASH. This instruction tells the extractor to copy from the input buffer 1008 to the HASH generator.

Thus these instructions are for extracting selected element(s) of the packet in the input buffer memory and transferring the data to a parser output buffer memory 1010. Some instructions also generate a hash.

The extraction engine 1007 and the PRE operate as a pipeline. That is, extraction engine 1007 performs extraction operations on data in input buffer 1008 already processed by PRE 1006 while more (i.e., later arriving) packet information is being simultaneously parsed by PRE 1006. This provides high processing speed sufficient to accommodate the high arrival rate speed of packets.

Once all the selected parts of the packet used to form the signature are extracted, the hash is loaded into parser output buffer memory 1010. Any additional payload from

the packet that is required for further analysis is also included. The parser output memory 1010 is interfaced with the analyzer subsystem by analyzer interface control 1011. Once all the information of a packet is in the parser output buffer memory 1010, a data ready signal 1025 is asserted by analyzer interface control. The data from the parser subsystem 1000 is moved to the analyzer subsystem via 1013 when an analyzer ready signal 1027 is asserted.

FIG. 11 shows the hardware components and dataflow for the analyzer subsystem that performs the functions of the analyzer subsystem 303 of FIG. 3. The analyzer is initialized prior to operation, and initialization includes loading the state processing information generated by the compilation process 310 into a database memory for the state processing, called state processor instruction database (SPID) memory 1109.

The analyzer subsystem 1100 includes a host bus interface 1122 using an analyzer host interface controller 1118, which in turn has access to a cache system 1115. The cache system has bi-directional access to and from the state processor of the system 1108. State processor 1108 is responsible for initializing the state processor instruction database memory 1109 from information given over the host bus interface 1122.

With the SPID 1109 loaded, the analyzer subsystem 1100 receives parser records comprising packet signatures and payloads that come from the parser into the unified flow key buffer (UFKB) 1103. UFKB is comprised of memory set up to maintain UFKB records. A UFKB record is essentially a parser record; the UFKB holds records of packets that are to be processed or that are in process. Furthermore, the UFKB provides for one or more fields to act as modifiable status flags to allow different processes to run concurrently.

Three processing engines run concurrently and access records in the UFKB 1103: the lookup/update engine (LUE) 1107, the state processor (SP) 1108, and the flow insertion and deletion engine (FIDE) 1110. Each of these is implemented by one or more finite state machines (FSM's). There is bi-directional access between each of the finite state machines and the unified flow key buffer 1103. The UFKB record includes a field that stores the packet sequence number, and another that is filled with state information in the form of a program counter for the state processor 1108 that implements state processing 328. The status flags of the UFKB for any entry includes that the LUE is done

and that the LUE is transferring processing of the entry to the state processor. The LUE done indicator is also used to indicate what the next entry is for the LUE. There also is provided a flag to indicate that the state processor is done with the current flow and to indicate what the next entry is for the state processor. There also is provided a flag to indicate the state processor is transferring processing of the UFKB-entry to the flow insertion and deletion engine.

A new UFKB record is first processed by the LUE 1107. A record that has been processed by the LUE 1107 may be processed by the state processor 1108, and a UFKB record data may be processed by the flow insertion/deletion engine 1110 after being processed by the state processor 1108 or only by the LUE. Whether or not a particular engine has been applied to any unified flow key buffer entry is determined by status fields set by the engines upon completion. In one embodiment, a status flag in the UFKB-entry indicates whether an entry is new or found. In other embodiments, the LUE issues a flag to pass the entry to the state processor for processing, and the required operations for a new record are included in the SP instructions.

Note that each UFKB-entry may not need to be processed by all three engines. Furthermore, some UFKB entries may need to be processed more than once by a particular engine.

Each of these three engines also has bi-directional access to a cache subsystem 1115 that includes a caching engine. Cache 1115 is designed to have information flowing in and out of it from five different points within the system: the three engines, external memory via a unified memory controller (UMC) 1119 and a memory interface 1123, and a microprocessor via analyzer host interface and control unit (ACIC) 1118 and host interface bus (HIB) 1122. The analyzer microprocessor (or dedicated logic processor) can thus directly insert or modify data in the cache.

The cache subsystem 1115 is an associative cache that includes a set of content addressable memory cells (CAMs) each including an address portion and a pointer portion pointing to the cache memory (e.g., RAM) containing the cached flow-entries. The CAMs are arranged as a stack ordered from a top CAM to a bottom CAM. The bottom CAM's pointer points to the least recently used (LRU) cache memory entry. Whenever there is a cache miss, the contents of cache memory pointed to by the bottom

CAM are replaced by the flow-entry from the flow-entry database 324. This now becomes the most recently used entry, so the contents of the bottom CAM are moved to the top CAM and all CAM contents are shifted down. Thus, the cache is an associative cache with a true LRU replacement policy.

The LUE 1107 first processes a UFKB-entry, and basically performs the operation of blocks 314 and 316 in FIG. 3. A signal is provided to the LUE to indicate that a "new" UFKB-entry is available. The LUE uses the hash in the UFKB-entry to read a matching bin of up to four buckets from the cache. The cache system attempts to obtain the matching bin. If a matching bin is not in the cache, the cache 1115 makes the request to the UMC 1119 to bring in a matching bin from the external memory.

When a flow-entry is found using the hash, the LUE 1107 looks at each bucket and compares it using the signature to the signature of the UFKB-entry until there is a match or there are no more buckets.

If there is no match, or if the cache failed to provide a bin of flow-entries from the cache, a time stamp in set in the flow key of the UFKB record, a protocol identification and state determination is made using a table that was loaded by compilation process 310 during initialization, the status for the record is set to indicate the LUE has processed the record, and an indication is made that the UFKB-entry is ready to start state processing. The identification and state determination generates a protocol identifier which in the preferred embodiment is a "jump vector" for the state processor which is kept by the UFKB for this UFKB-entry and used by the state processor to start state processing for the particular protocol. For example, the jump vector jumps to the subroutine for processing the state.

If there was a match, indicating that the packet of the UFKB-entry is for a previously encountered flow, then a calculator component enters one or more statistical measures stored in the flow-entry, including the timestamp. In addition, a time difference from the last stored timestamp may be stored, and a packet count may be updated. The state of the flow is obtained from the flow-entry is examined by looking at the protocol identifier stored in the flow-entry of database 324. If that value indicates that no more classification is required, then the status for the record is set to indicate the LUE has processed the record. In the preferred embodiment, the protocol identifier is a jump

vector for the state processor to a subroutine to state processing the protocol, and no more classification is indicated in the preferred embodiment by the jump vector being zero. If the protocol identifier indicates more processing, then an indication is made that the UFKB-entry is ready to start state processing and the status for the record is set to indicate the LUE has processed the record.

The state processor 1108 processes information in the cache system according to a UFKB-entry after the LUE has completed. State processor 1108 includes a state processor program counter SPPC that generates the address in the state processor instruction database 1109 loaded by compiler process 310 during initialization. It contains an Instruction Pointer (SPIP) which generates the SPID address. The instruction pointer can be incremented or loaded from a Jump Vector Multiplexor which facilitates conditional branching. The SPIP can be loaded from one of three sources: (1) A protocol identifier from the UFKB, (2) an immediate jump vector form the currently decoded instruction, or (3) a value provided by the arithmetic logic unit (SPALU) included in the state processor.

Thus, after a Flow Key is placed in the UFKB by the LUE with a known protocol identifier, the Program Counter is initialized with the last protocol recognized by the Parser. This first instruction is a jump to the subroutine which analyzes the protocol that was decoded.

The State Processor ALU (SPALU) contains all the Arithmetic, Logical and String Compare functions necessary to implement the State Processor instructions. The main blocks of the SPALU are: The A and B Registers, the Instruction Decode & State Machines, the String Reference Memory the Search Engine, an Output Data Register and an Output Control Register

The Search Engine in turn contains the Target Search Register set, the Reference Search Register set, and a Compare block which compares two operands by exclusive-or-ing them together.

Thus, after the UFKB sets the program counter, a sequence of one or more state operations are be executed in state processor 1108 to further analyze the packet that is in the flow key buffer entry for this particular packet.

FIG. 13 describes the operation of the state processor 1108. The state processor is entered at 1301 with a unified flow key buffer entry to be processed. The UFKB-entry is

new or corresponding to a found flow-entry. This UFKB-entry is retrieved from unified flow key buffer 1103 in 1301. In 1303, the protocol identifier for the UFKB-entry is used to set the state processor's instruction counter. The state processor 1108 starts the process by using the last protocol recognized by the parser subsystem 301 as an offset into a jump table. The jump table takes us to the instructions to use for that protocol. Most instructions test something in the unified flow key buffer or the flow-entry if it exists. The state processor 1108 may have to test bits, do comparisons, add or subtract to perform the test.

The first state processor instruction is fetched in 1304 from the state processor instruction database memory 1109. The state processor performs the one or more fetched operations (1304). In our implementation, each single state processor instruction is very primitive (e.g., a move, a compare, etc.), so that many such instructions need to be performed on each unified flow key buffer entry. One aspect of the state processor is its ability to search for one or more (up to four) reference strings in the payload part of the UFKB entry. This is implemented by a search engine component of the state processor responsive to special searching instructions.

In 1307, a check is made to determine if there are any more instructions to be performed for the packet. If yes, then in 1308 the system sets the state processor instruction pointer (SPIP) to obtain the next instruction. The SPIP may be set by an immediate jump vector in the currently decoded instruction, or by a value provided by the SPALU during processing.

The next instruction to be performed is now fetched (1304) for execution. This state processing loop between 1304 and 1307 continues until there are no more instructions to be performed.

At this stage, a check is made in 1309 if the processing on this particular packet has resulted in a final state. That is, is the analyzer is done processing not only for this particular packet, but for the whole flow to which the packet belongs, and the flow is fully determined. If indeed there are no more states to process for this flow, then in 1311 the processor finalizes the processing. Some final states may need to put a state in place that tells the system to remove a flow—for example, if a connection disappears from a lower level connection identifier. In that case, in 1311, a flow removal state is set and saved in

the flow-entry. The flow removal state may be a NOP (no-op) instruction which means there are no removal instructions.

Once the appropriate flow removal instruction as specified for this flow (a NOP or otherwise) is set and saved, the process is exited at 1313. The state processor 1108 can now obtain another unified flow key buffer entry to process.

If at 1309 it is determined that processing for this flow is not completed, then in 1310 the system saves the state processor instruction pointer in the current flow-entry in the current flow-entry. That will be the next operation that will be performed the next time the LRE 1107 finds packet in the UFKB that matches this flow. The processor now exits processing this particular unified flow key buffer entry at 1313.

Note that state processing updates information in the unified flow key buffer 1103 and the flow-entry in the cache. Once the state processor is done, a flag is set in the UFKB for the entry that the state processor is done. Furthermore, If the flow needs to be inserted or deleted from the database of flows, control is then passed on to the flow insertion/deletion engine 1110 for that flow signature and packet entry. This is done by the state processor setting another flag in the UFKB for this UFKB-entry indicating that the state processor is passing processing of this entry to the flow insertion and deletion engine.

The flow insertion and deletion engine 1110 is responsible for maintaining the flow-entry database. In particular, for creating new flows in the flow database, and deleting flows from the database so that they can be reused.

The process of flow insertion is now described with the aid of FIG. 12. Flows are grouped into bins of buckets by the hash value. The engine processes a UFKB-entry that may be new or that the state processor otherwise has indicated needs to be created. FIG. 12 shows the case of a new entry being created. A conversation record bin (preferably containing 4 buckets for four records) is obtained in 1203. This is a bin that matches the hash of the UFKB, so this bin may already have been sought for the UFKB-entry by the LUE. In 1204 the FIDE 1110 requests that the record bin/bucket be maintained in the cache system 1115. If in 1205 the cache system 1115 indicates that the bin/bucket is empty, step 1207 inserts the flow signature (with the hash) into the bucket and the bucket is marked "used" in the cache engine of cache 1115 using a timestamp that

is maintained throughout the process. In 1209, the FIDE 1110 compares the bin and bucket record flow signature to the packet to verify that all the elements are in place to complete the record. In 1211 the system marks the record bin and bucket as "in process" and as "new" in the cache system (and hence in the external memory). In 1212, the initial statistical measures for the flow-record are set in the cache system. This in the preferred embodiment clears the set of counters used to maintain statistics, and may perform other procedures for statistical operations requires by the analyzer for the first packet seen for a particular flow.

Back in step 1205, if the bucket is not empty, the FIDE 1110 requests the next bucket for this particular bin in the cache system. If this succeeds, the processes of 1207, 1209, 1211 and 1212 are repeated for this next bucket. If at 1208, there is no valid bucket, the unified flow key buffer entry for the packet is set as "drop," indicating that the system cannot process the particular packet because there are no buckets left in the system. The process exits at 1213. The FIDE 1110 indicates to the UFKB that the flow insertion and deletion operations are completed for this UFKB-entry. This also lets the UFKB provide the FIDE with the next UFKB record.

Once a set of operations is performed on a unified flow key buffer entry by all of the engines required to access and manage a particular packet and its flow signature, the unified flow key buffer entry is marked as "completed." That element will then be used by the parser interface for the next packet and flow signature coming in from the parsing and extracting system.

All flow-entries are maintained in the external memory and some are maintained in the cache 1115. The cache system 1115 is intelligent enough to access the flow database and to understand the data structures that exists on the other side of memory interface 1123. The lookup/update engine 1107 is able to request that the cache system pull a particular flow or "buckets" of flows from the unified memory controller 1119 into the cache system for further processing. The state processor 1108 can operate on information found in the cache system once it is looked up by means of the lookup/update engine request, and the flow insertion/deletion engine 1110 can create new entries in the cache system if required based on information in the unified flow key buffer 1103. The cache retrieves information as required from the memory through the memory interface 1123 and the unified memory controller 1119, and updates information as required in the

memory through the memory controller 1119.

There are several interfaces to components of the system external to the module of FIG. 11 for the particular hardware implementation. These include host bus interface 1122,which is designed as a generic interface that can operate with any kind of external processing system such as a microprocessor or a multiplexor (MUX) system. Consequently, one can connect the overall traffic classification system of FIGS. 11 and 12 into some other processing system to manage the classification system and to extract data gathered by the system.

The memory interface 1123 is designed to interface to any of a variety of memory systems that one may want to use to store the flow-entries. One can use different types of memory systems like regular dynamic random access memory (DRAM), synchronous DRAM, synchronous graphic memory (SGRAM), static random access memory (SRAM), and so forth.

FIG. 10 also includes some "generic" interfaces. There is a packet input interface 1012—a general interface that works in tandem with the signals of the input buffer interface control 1022. These are designed so that they can be used with any kind of generic systems that can then feed packet information into the parser. Another generic interface is the interface of pipes 1031 and 1033 respectively out of and into host interface multiplexor and control registers 1005. This enables the parsing system to be managed by an external system, for example a microprocessor or another kind of external logic, and enables the external system to program and otherwise control the parser.

The preferred embodiment of this aspect of the invention is described in a hardware description language (HDL) such as VHDL or Verilog. It is designed and created in an HDL so that it may be used as a single chip system or, for instance, integrated into another general-purpose system that is being designed for purposes related to creating and analyzing traffic within a network. Verilog or other HDL implementation is only one method of describing the hardware.

In accordance with one hardware implementation, the elements shown in FIGS. 10 and 11 are implemented in a set of six field programmable logic arrays (FPGA's). The boundaries of these FPGA's are as follows. The parsing subsystem of FIG. 10 is implemented as two FPGAS; one FPGA, and includes blocks 1006, 1008 and 1012, parts

of 1005, and memory 1001. The second FPGA includes 1002, 1007, 1013, 1011 parts of 1005. Referring to FIG. 11, the unified look-up buffer 1103 is implemented as a single FPGA. State processor 1108 and part of state processor instruction database memory 1109 is another FPGA. Portions of the state processor instruction database memory 1109 are maintained in external SRAM's. The lookup/update engine 1107 and the flow insertion/deletion engine 1110 are in another FPGA. The sixth FPGA includes the cache system 1115, the unified memory control 1119, and the analyzer host interface and control 1118.

Note that one can implement the system as one or more VSLI devices, rather than as a set of application specific integrated circuits (ASIC's) such as FPGA's. It is anticipated that in the future device densities will continue to increase, so that the complete system may eventually form a sub-unit (a "core") of a larger single chip unit.

Operation of the Invention

Fig. 15 shows how an embodiment of the network monitor 300 might be used to analyze traffic in a network 102. Packet acquisition device 1502 acquires all the packets from a connection point 121 on network 102 so that all packets passing point 121 in either direction are supplied to monitor 300. Monitor 300 comprises the parser sub-system 301, which determines flow signatures, and analyzer sub-system 303 that analyzes the flow signature of each packet. A memory 324 is used to store the database of flows that are determined and updated by monitor 300. A host computer 1504, which might be any processor, for example, a general-purpose computer, is used to analyze the flows in memory 324. As is conventional, host computer 1504 includes a memory, say RAM, shown as host memory 1506. In addition, the host might contain a disk. In one application, the system can operate as an RMON probe, in which case the host computer is coupled to a network interface card 1510 that is connected to the network 102.

The preferred embodiment of the invention is supported by an optional Simple Network Management Protocol (SNMP) implementation. Fig. 15 describes how one would, for example, implement an RMON probe, where a network interface card is used to send RMON information to the network. Commercial SNMP implementations also are available, and using such an implementation can simplify the process of porting the preferred embodiment of the invention to any platform.

In addition, MIB Compilers are available. An MIB Compiler is a tool that greatly simplifies the creation and maintenance of proprietary MIB extensions.

Examples of Packet Elucidation

Monitor 300, and in particular, analyzer 303 is capable of carrying out state analysis for packet exchanges that are commonly referred to as "server announcement" type exchanges. Server announcement is a process used to ease communications between a server with multiple applications that can all be simultaneously accessed from multiple clients. Many applications use a server announcement process as a means of multiplexing a single port or socket into many applications and services. With this type of exchange, messages are sent on the network, in either a broadcast or multicast approach, to announce a server and application, and all stations in the network may receive and decode these messages. The messages enable the stations to derive the appropriate connection point for communicating that particular application with the particular server. Using the server announcement method, a particular application communicates using a service channel, in the form of a TCP or UDP socket or port as in the IP protocol suite, or using a SAP as in the Novell IPX protocol suite.

The analyzer 303 is also capable of carrying out "in-stream analysis" of packet exchanges. The "in-stream analysis" method is used either as a primary or secondary recognition process. As a primary process, in-stream analysis assists in extracting detailed information which will be used to further recognize both the specific application and application component. A good example of in-stream analysis is any Web-based application. For example, the commonly used PointCast Web information application can be recognized using this process; during the initial connection between a PointCast server and client, specific key tokens exist in the data exchange that will result in a signature being generated to recognize PointCast.

The in-stream analysis process may also be combined with the server announcement process. In many cases in-stream analysis will augment other recognition processes. An example of combining in-stream analysis with server announcement can be found in business applications such as SAP and BAAN.

"Session tracking" also is known as one of the primary processes for tracking applications in client/server packet exchanges. The process of tracking sessions requires

an initial connection to a predefined socket or port number. This method of communication is used in a variety of transport layer protocols. It is most commonly seen in the TCP and UDP transport protocols of the IP protocol.

During the session tracking, a client makes a request to a server using a specific port or socket number. This initial request will cause the server to create a TCP or UDP port to exchange the remainder of the data between the client and the server. The server then replies to the request of the client using this newly created port. The original port used by the client to connect to the server will never be used again during this data exchange.

One example of session tracking is TFTP (Trivial File Transfer Protocol), a version of the TCP/IP FTP protocol that has no directory or password capability. During the client/server exchange process of TFTP, a specific port (port number 69) is always used to initiate the packet exchange. Thus, when the client begins the process of communicating, a request is made to UDP port 69. Once the server receives this request, a new port number is created on the server. The server then replies to the client using the new port. In this example, it is clear that in order to recognize TFTP; network monitor 300 analyzes the initial request from the client and generates a signature for it. Monitor 300 uses that signature to recognize the reply. Monitor 300 also analyzes the reply from the server with the key port information, and uses this to create a signature for monitoring the remaining packets of this data exchange.

Network monitor 300 can also understand the current state of particular connections in the network. Connection-oriented exchanges often benefit from state tracking to correctly identify the application. An example is the common TCP transport protocol that provides a reliable means of sending information between a client and a server. When a data exchange is initiated, a TCP request for synchronization message is sent. This message contains a specific sequence number that is used to track an acknowledgement from the server. Once the server has acknowledged the synchronization request, data may be exchanged between the client and the server. When communication is no longer required, the client sends a finish or complete message to the server, and the server acknowledges this finish request with a reply containing the sequence numbers from the request. The states of such a connection-oriented exchange relate to the various types of connection and maintenance messages.

Server Announcement Example

The individual methods of server announcement protocols vary. However, the basic underlying process remains similar. A typical server announcement message is sent to one or more clients in a network. This type of announcement message has specific content, which, in another aspect of the invention, is salvaged and maintained in the database of flow-entries in the system. Because the announcement is sent to one or more stations, the client involved in a future packet exchange with the server will make an assumption that the information announced is known, and an aspect of the inventive monitor is that it too can make the same assumption.

Sun-RPC is the implementation by Sun Microsystems, Inc. (Palo Alto, California) of the Remote Procedure Call (RPC), a programming interface that allows one program to use the services of another on a remote machine. A Sun-RPC example is now used to explain how monitor 300 can capture server announcements.

A remote program or client that wishes to use a server or procedure must establish a connection, for which the RPC protocol can be used.

Each server running the Sun-RPC protocol must maintain a process and database called the port Mapper. The port Mapper creates a direct association between a Sun-RPC program or application and a TCP or UDP socket or port (for TCP or UDP implementations). An application or program number is a 32-bit unique identifier assigned by ICANN (the Internet Corporation for Assigned Names and Numbers, www.icann.org), which manages the huge number of parameters associated with Internet protocols (port numbers, router protocols, multicast addresses, *etc.*) Each port Mapper on a Sun-RPC server can present the mappings between a unique program number and a specific transport socket through the use of specific request or a directed announcement. According to ICANN, port number 111 is associated with Sun RPC.

As an example, consider a client (*e.g.*, CLIENT 3 shown as 106 in FIG. 1) making a specific request to the server (*e.g.*, SERVER 2 of FIG. 1, shown as 110) on a predefined UDP or TCP socket. Once the port Mapper process on the sun RPC server receives the request, the specific mapping is returned in a directed reply to the client.

1. A client (CLIENT 3, 106 in FIG. 1) sends a TCP packet to SERVER 2 (110 in FIG. 1) on port 111, with an RPC Bind Lookup Request (rpcBindLookup). TCP or UDP port 111 is always associated Sun RPC. This request specifies the program (as a program identifier), version, and might specify the protocol (UDP or TCP).

2. The server SERVER 2 (110 in FIG. 1) extracts the program identifier and version identifier from the request. The server also uses the fact that this packet came in using the TCP transport and that no protocol was specified, and thus will use the TCP protocol for its reply.

3. The server 110 sends a TCP packet to port number 111, with an RPC Bind Lookup Reply. The reply contains the specific port number (*e.g., port* number 'port') on which future transactions will be accepted for the specific RPC program identifier (*e.g.,* Program 'program') and the protocol (UDP or TCP) for use.

It is desired that from now on every time that port number 'port' is used, the packet is associated with the application program 'program' until the number 'port' no longer is to be associated with the program 'program'. Network monitor 300 by creating a flow-entry and a signature includes a mechanism for remembering the exchange so that future packets that use the port number 'port' will be associated by the network monitor with the application program 'program'.

In addition to the Sun RPC Bind Lookup request and reply, there are other ways that a particular program—say 'program'—might be associated with a particular port number, for example number 'port'. One is by a broadcast announcement of a particular association between an application service and a port number, called a Sun RPC portMapper Announcement. Another, is when some server—say the same SERVER 2—replies to some client—say CLIENT 1—requesting some portMapper assignment with a RPC portMapper Reply. Some other client—say CLIENT 2—might inadvertently see this request, and thus know that for this particular server, SERVER 2, port number 'port' is associated with the application service 'program'. It is desirable for the network monitor 300 to be able to associate any packets to SERVER 2 using port number 'port' with the application program 'program'.

FIG. 9 represents a dataflow 900 of some operations in the monitor 300 of FIG. 3 for Sun Remote Procedure Call. Suppose a client 106 (*e.g.*, CLIENT 3 in FIG. 1) is communicating via its interface to the network 118 to a server 110 (*e.g.*, SERVER 2 in FIG. 1) via the server's interface to the network 116. Further assume that Remote Procedure Call is used to communicate with the server 110. One path in the data flow 900 starts with a step 910 that a Remote Procedure Call bind lookup request is issued by client 106 and ends with the server state creation step 904. Such RPC bind lookup request includes values for the 'program,' 'version,' and 'protocol' to use, *e.g.*, TCP or UDP. The process for Sun RPC analysis in the network monitor 300 includes the following aspects. :

- Process 909: Extract the 'program,' 'version,' and 'protocol' (UDP or TCP). Extract the TCP or UDP port (process 909) which is 111 indicating Sun RPC.
- Process 908: Decode the Sun RPC packet. Check RPC type field for ID. If value is portMapper, save paired socket (*i.e.*, dest for destination address, src for source address). Decode ports and mapping, save ports with socket/addr key. There may be more than one pairing per mapper packet. Form a signature (e.g., a key). A flow-entry is created in database 324. The saving of the request is now complete.

At some later time, the server (process 907) issues a RPC bind lookup reply. The packet monitor 300 will extract a signature from the packet and recognize it from the previously stored flow. The monitor will get the protocol port number (906) and lookup the request (905). A new signature (i.e., a key) will be created and the creation of the server state (904) will be stored as an entry identified by the new signature in the flow-entry database. That signature now may be used to identify packets associated with the server.

The server state creation step 904 can be reached not only from a Bind Lookup Request/Reply pair, but also from a RPC Reply portMapper packet shown as 901 or an RPC Announcement portMapper shown as 902. The Remote Procedure Call protocol can announce that it is able to provide a particular application service. Embodiments of the present invention preferably can analyze when an exchange occurs between a client and a server, and also can track those stations that have received the announcement of a service in the network.

The RPC Announcement portMapper announcement 902 is a broadcast. Such causes various clients to execute a similar set of operations, for example, saving the information obtained from the announcement. The RPC Reply portMapper step 901 could be in reply to a portMapper request, and is also broadcast. It includes all the service parameters.

Thus monitor 300 creates and saves all such states for later classification of flows that relate to the particular service 'program'.

FIG. 2 shows how the monitor 300 in the example of Sun RPC builds a signature and flow states. A plurality of packets 206-209 are exchanged, *e.g.,* in an exemplary Sun Microsystems Remote Procedure Call protocol. A method embodiment of the present invention might generate a pair of flow signatures, "signature-1" 210 and "signature-2" 212, from information found in the packets 206 and 207 which, in the example, correspond to a Sun RPC Bind Lookup request and reply, respectively.

Consider first the Sun RPC Bind Lookup request. Suppose packet 206 corresponds to such a request sent from CLIENT 3 to SERVER 2. This packet contains important information that is used in building a signature according to an aspect of the invention. A source and destination network address occupy the first two fields of each packet, and according to the patterns in pattern database 308, the flow signature (shown as KEY1 230 in FIG. 2) will also contain these two fields, so the parser subsystem 301 will include these two fields in signature KEY 1 (230). Note that in FIG. 2, if an address identifies the client 106 (shown also as 202), the label used in the drawing is "C_1". If such address identifies the server 110 (shown also as server 204), the label used in the drawing is "S_1". The first two fields 214 and 215 in packet 206 are "S_1" and C_1" because packet 206 is provided from the server 110 and is destined for the client 106. Suppose for this example, "S_1" is an address numerically less than address "C_1". A third field "p^1" 216 identifies the particular protocol being used, *e.g.,* TCP, UDP, etc.

In packet 206, a fourth field 217 and a fifth field 218 are used to communicate port numbers that are used. The conversation direction determines where the port number field is. The diagonal pattern in field 217 is used to identify a source-port pattern, and the hash pattern in field 218 is used to identify the destination-port pattern. The order indicates the client-server message direction. A sixth field denoted "i^1" 219 is an element

that is being requested by the client from the server. A seventh field denoted "s_1a" 220 is the service requested by the client from server 110. The following eighth field "QA" 221 (for question mark) indicates that the client 106 wants to know what to use to access application "s_1a". A tenth field "QP" 223 is used to indicate that the client wants the server to indicate what protocol to use for the particular application.

Packet 206 initiates the sequence of packet exchanges, *e.g.*, a RPC Bind Lookup Request to SERVER 2. It follows a well-defined format, as do all the packets, and is transmitted to the server 110 on a well-known service connection identifier (port 111 indicating Sun RPC).

Packet 207 is the first sent in reply to the client 106 from the server. It is the RPC Bind Lookup Reply as a result of the request packet 206.

Packet 207 includes ten fields 224–233. The destination and source addresses are carried in fields 224 and 225, *e.g.*, indicated "C_1" and "S_1", respectively. Notice the order is now reversed, since the client-server message direction is from the server 110 to the client 106. The protocol "p^1" is used as indicated in field 226. The request "i^1" is in field 229. Values have been filled in for the application port number, *e.g.*, in field 233 and protocol ""p^2"" in field 233.

The flow signature and flow states built up as a result of this exchange are now described. When the packet monitor 300 sees the request packet 206 from the client, a first flow signature 210 is built in the parser subsystem 301 according to the pattern and extraction operations database 308. This signature 210 includes a destination and a source address 240 and 241. One aspect of the invention is that the flow keys are built consistently in a particular order no matter what the direction of conversation. Several mechanisms may be used to achieve this. In the particular embodiment, the numerically lower address is always placed before the numerically higher address. Such least to highest order is used to get the best spread of signatures and hashes for the lookup operations. In this case, therefore, since we assume "S_1"<"C_1", the order is address "S_1" followed by client address "C_1". The next field used to build the signature is a protocol field 242 extracted from packet 206's field 216, and thus is the protocol "p^1". The next field used for the signature is field 243, which contains the destination source port number shown as a crosshatched pattern from the field 218 of the packet 206. This pattern will be

recognized in the payload of packets to derive how this packet or sequence of packets exists as a flow. In practice, these may be TCP port numbers, or a combination of TCP port numbers. In the case of the Sun RPC example, the crosshatch represents a set of port numbers of UDS for p^1 that will be used to recognize this flow (*e.g., port* 111). Port 111 indicates this is Sun RPC. Some applications, such as the Sun RPC Bind Lookups, are directly determinable ("known") at the parser level. So in this case, the signature KEY-1 points to a known application denoted "a^1" (Sun RPC Bind Lookup), and a next-state that the state processor should proceed to for more complex recognition jobs, denoted as state "st_D" is placed in the field 245 of the flow-entry.

When the Sun RPC Bind Lookup reply is acquired, a flow signature is again built by the parser. This flow signature is identical to KEY-1. Hence, when the signature enters the analyzer subsystem 303 from the parser subsystem 301, the complete flow-entry is obtained, and in this flow-entry indicates state "st_D". The operations for state "st_D" in the state processor instruction database 326 instructs the state processor to build and store a new flow signature, shown as KEY-2 (212) in FIG. 2. This flow signature built by the state processor also includes the destination and a source addresses 250 and 251, respectively, for server "S_1" followed by (the numerically higher address) client "C_1". A protocol field 252 defines the protocol to be used, *e.g.,* "p^2" which is obtained from the reply packet. A field 253 contains a recognition pattern also obtained from the reply packet. In this case, the application is Sun RPC, and field 254 indicates this application "a^2". A next-state field 255 defines the next state that the state processor should proceed to for more complex recognition jobs, *e.g.,* a state "st^1". In this particular example, this is a final state. Thus, KEY-2 may now be used to recognize packets that are in any way associated with the application "a^2". Two such packets 208 and 209 are shown, one in each direction. They use the particular application service requested in the original Bind Lookup Request, and each will be recognized because the signature KEY-2 will be built in each case.

The two flow signatures 210 and 212 always order the destination and source address fields with server "S_1" followed by client "C_1". Such values are automatically filled in when the addresses are first created in a particular flow signature. Preferably,

large collections of flow signatures are kept in a lookup table in a least-to-highest order for the best spread of flow signatures and hashes.

Thereafter, the client and server exchange a number of packets, *e.g.*, represented by request packet 208 and response packet 209. The client 106 sends packets 208 that have a destination and source address S_1 and C_1, in a pair of fields 260 and 261. A field 262 defines the protocol as "p^2", and a field 263 defines the destination port number.

Some network-server application recognition jobs are so simple that only a single state transition has to occur to be able to pinpoint the application that produced the packet. Others require a sequence of state transitions to occur in order to match a known and predefined climb from state-to-state.

Thus the flow signature for the recognition of application "a^2" is automatically set up by predefining what packet-exchange sequences occur for this example when a relatively simple Sun Microsystems Remote Procedure Call bind lookup request instruction executes. More complicated exchanges than this may generate more than two flow signatures and their corresponding states. Each recognition may involve setting up a complex state transition diagram to be traversed before a "final" resting state such as "st_1" in field 255 is reached. All these are used to build the final set of flow signatures for recognizing a particular application in the future.

Embodiments of the present invention automatically generate flow signatures with the necessary recognition patterns and state transition climb procedure. Such comes from analyzing packets according to parsing rules, and also generating state transitions to search for. Applications and protocols, at any level, are recognized through state analysis of sequences of packets.

Note that one in the art will understand that computer networks are used to connect many different types of devices, including network appliances such as telephones, "Internet" radios, pagers, and so forth. The term computer as used herein encompasses all such devices and a computer network as used herein includes networks of such computers.

Although the present invention has been described in terms of the presently preferred embodiments, it is to be understood that the disclosure is not to be interpreted as

limiting. Various alterations and modifications will no doubt become apparent to those or ordinary skill in the art after having read the above disclosure. Accordingly, it is intended that the claims be interpreted as covering all alterations and modifications as fall within the true spirit and scope of the present invention.

CLAIMS

What is claimed is:

1. A packet monitor for examining packets passing through a connection point on a computer network in real-time, the packets provided to the packet monitor via a packet acquisition device connected to the connection point, the packet monitor comprising:

(a) a packet-buffer memory configured to accept a packet from the packet acquisition device;

(b) a parsing/extraction operations memory configured to store a database of parsing/extraction operations that includes information describing how to determine at least one of the protocols used in a packet from data in the packet;

(c) a parser subsystem coupled to the packet buffer and to the pattern/extraction operations memory, the parser subsystem configured to examine the packet accepted by the buffer, extract selected portions of the accepted packet, and form a function of the selected portions sufficient to identify that the accepted packet is part of a conversational flow-sequence;

(d) a memory storing a flow-entry database including a plurality of flow-entries for conversational flows encountered by the monitor;

(e) a lookup engine connected to the parser subsystem and to the flow-entry database, and configured to determine using at least some of the selected portions of the accepted packet if there is an entry in the flow-entry database for the conversational flow sequence of the accepted packet;

(f) a state patterns/operations memory configured to store a set of predefined state transition patterns and state operations such that traversing a particular transition pattern as a result of a particular conversational flow-sequence of packets indicates that the particular conversational flow-sequence is associated with the operation of a particular application program, visiting each state in a traversal including carrying out none or more predefined state operations;

(g) a protocol/state identification mechanism coupled to the state patterns/operations memory and to the lookup engine, the protocol/state identification engine configured to determine the protocol and state of the conversational flow of the packet; and

(h) a state processor coupled to the flow-entry database, the protocol/state identification engine, and to the state patterns/operations memory, the state processor, configured to carry out any state operations specified in the state patterns/operations memory for the protocol and state of the flow of the packet,

the carrying out of the state operations furthering the process of identifying which application program is associated with the conversational flow-sequence of the packet, the state processor progressing through a series of states and state operations until there are no more state operations to perform for the accepted packet, in which case the state processor updates the flow-entry, or until a final state is reached that indicates that no more analysis of the flow is required, in which case the result of the analysis is announced.

2. A packet monitor according to claim 1, wherein the flow-entry includes the state of the flow, such that the protocol/state identification mechanism determines the state of the packet from the flow-entry in the case that the lookup engine finds a flow-entry for the flow of the accepted packet.

3. A packet monitor according to claim 1, wherein the parser subsystem includes a mechanism for building a hash from the selected portions, and wherein the hash is used by the lookup engine to search the flow-entry database, the hash designed to spread the flow-entries across the flow-entry database.

4. A packet monitor according to claim 1, further comprising:

a compiler processor coupled to the parsing/extraction operations memory, the compiler processor configured to run a compilation process that includes:

receiving commands in a high-level protocol description language that describe the protocols that may be used in packets encountered by the monitor, and

translating the protocol description language commands into a plurality of parsing/extraction operations that are initialized into the parsing/extraction operations memory.

5. A packet monitor according to claim 4, wherein the protocol description language commands also describe a correspondence between a set of one or more application programs and the state transition patterns/operations that occur as a result of particular conversational flow-sequences associated with an application program, wherein the compiler processor is also coupled to the state patterns/operations memory, and wherein the compilation process further includes translating the protocol description language commands into a plurality of state patterns and state operations that are initialized into the state patterns/operations memory.

6. A packet monitor according to claim 1, further comprising:

a cache memory coupled to and between the lookup engine and the flow-entry database providing for fast access of a set of likely-to-be-accessed flow-entries from the flow-entry database.

7. A packet monitor according to claim 6, wherein the cache functions as a fully associative, least-recently-used cache memory.

8. A packet monitor according to claim 7, wherein the cache functions as a fully associative, least-recently-used cache memory and includes content addressable memories configured as a stack.

9. A packet monitor according to claim 1, wherein one or more statistical measures about a flow are stored in each flow-entry, the packet monitor further comprising:

a calculator for updating the statistical measures in a flow-entry of the accepted packet.

10. A packet monitor according to claim 9, wherein, when the application program of a flow is determined, one or more network usage metrics related to said application and determined from the statistical measures are presented to a user for network performance monitoring.

11. A method of examining packets passing through a connection point on a computer network, each packets conforming to one or more protocols, the method comprising:

(a) receiving a packet from a packet acquisition device;

(b) performing one or more parsing/extraction operations on the packet to create a parser record comprising a function of selected portions of the packet;

(c) looking up a flow-entry database comprising none or more flow-entries for previously encountered conversational flows, the looking up using at least some of the selected packet portions and determining if the packet is of an existing flow;

(d) if the packet is of an existing flow, classifying the packet as belonging to the found existing flow; and

(e) if the packet is of a new flow, storing a new flow-entry for the new flow in the flow-entry database, including identifying information for future packets to be identified with the new flow-entry,

wherein the parsing/extraction operations depend on one or more of the protocols to which the packet conforms.

12. A method according to claim 11, wherein each packet passing through the connection point is examined in real time.

13. A method according to claim 11, wherein classifying the packet as belonging to the found existing flow includes updating the flow-entry of the existing flow.

14. A method according to claim 13, wherein updating includes storing one or more statistical measures stored in the flow-entry of the existing flow.

15. A method according to claim 14, wherein the one or more statistical measures include measures selected from the set consisting of the total packet count for the flow, the time, and a differential time from the last entered time to the present time.

16. A method according to claim 11, wherein the function of the selected portions of the packet forms a signature that includes the selected packet portions and that can identify future packers, wherein the lookup operation uses the signature and wherein the identifying information stored in the new or updated flow-entry is a signature for identifying future packets.

17. A method according to claim 11, wherein at least one of the protocols of the packet uses source and destination addresses, and wherein the selected portions of the packet include the source and destination addresses.

18. A method according to claim 17, wherein the function of the selected portions for packets of the same flow is consistent independent of the direction of the packets.

19. A method according to claim 18, wherein the source and destination addresses are placed in an order determined by the order of numerical values of the addresses in the function of selected portions.

20. A method according to claim 19, wherein the numerically lower address is placed before the numerically higher address in the function of selected portions.

21. A method according to claim 11, wherein the looking up of the flow-entry database uses a hash of the selected packet portions.

22. A method according to claim 11, wherein the parsing/extraction operations are according to a database of parsing/extraction operations that includes information describing how to determine a set of one or more protocol dependent extraction operations from data in the packet that indicate a protocol used in the packet.

23. A method according to claim 11, wherein step (d) includes if the packet is of an existing flow, obtaining the last encountered state of the flow and performing any state operations specified for the state of the flow starting from the last encountered state of the flow; and wherein step (e) includes if the packet is of a new flow, performing any state operations required for the initial state of the new flow.

24. A method according to claim 23, wherein the state processing of each received packet of a flow furthers the identifying of the application program of the flow.

25. A method according to claim 23, wherein the state operations include updating the flow-entry, including storing identifying information for future packets to be identified with the flow-entry.

26. A method according to claim 25, wherein the state processing of each received packet of a flow furthers the identifying of the application program of the flow.

27. A method according to claim 23, wherein the state operations include searching the parser record for the existence of one or more reference strings.

28. A method according to claim 23, wherein the state operations are carried out by a programmable state processor according to a database of protocol dependent state operations.

29. A packet monitor for examining packets passing through a connection point on a computer network, each packets conforming to one or more protocols, the monitor comprising:

(a) a packet acquisition device coupled to the connection point and configured to receive packets passing through the connection point;

(b) an input buffer memory coupled to and configured to accept a packet from the packet acquisition device;

(c) a parser subsystem coupled to the input buffer memory and including a slicer, the parsing subsystem configured to extract selected portions of the accepted packet and to output a parser record containing the selected portions;

(d) a memory for storing a database comprising none or more flow-entries for previously encountered conversational flows, each flow-entry identified by identifying information stored in the flow-entry;

(e) a lookup engine coupled to the output of the parser subsystem and to the flow-entry memory and configured to lookup whether the particular packet whose parser record is output by the parser subsystem has a matching flow-entry, the looking up using at least some of the selected packet portions and determining if the packet is of an existing flow; and

(f) a flow insertion engine coupled to the flow-entry memory and to the lookup engine and configured to create a flow-entry in the flow-entry database, the flow-entry including identifying information for future packets to be identified with the new flow-entry,

the lookup engine configured such that if the packet is of an existing flow, the monitor classifies the packet as belonging to the found existing flow; and if the packet is of a new flow, the flow insertion engine stores a new flow-entry for the new flow in the flow-entry database, including identifying information for future packets to be identified with the new flow-entry,

wherein the operation of the parser subsystem depends on one or more of the protocols to which the packet conforms.

30. A monitor according to claim 29, wherein each packet passing through the connection point is accepted by the packet buffer memory and examined by the monitor in real time.

31. A monitor according to claim 29, wherein the lookup engine updates the flow-entry of an existing flow in the case that the lookup is successful.

32. A monitor according to claim 29, further including a mechanism for building a hash from the selected portions, wherein the hash is included in the input for a particular packet to the lookup engine, and wherein the hash is used by the lookup engine to search the flow-entry database.

33. A monitor according to claim 29, further including a memory containing a database of parsing/extraction operations, the parsing/extraction database memory coupled to the parser subsystem, wherein the parsing/extraction operations are according to one or more parsing/extraction operations looked up from the parsing/extraction database.

34. A monitor according to claim 33, wherein the database of parsing/extraction operations includes information describing how to determine a set of one or more protocol dependent extraction operations from data in the packet that indicate a protocol used in the packet.

35. A monitor according to claim 29, further including a flow-key-buffer (UFKB) coupled to the output of the parser subsystem and to the lookup engine and to the flow insertion engine, wherein the output of the parser monitor is coupled to the lookup engine via the UFKB, and wherein the flow insertion engine is coupled to the lookup engine via the UFKB.

36. A method according to claim 29, further including a state processor coupled to the lookup engine and to the flow-entry-database memory, and configured to perform any state operations specified for the state of the flow starting from the last encountered state of the flow in the case that the packet is from an existing flow, and to perform any state operations required for the initial state of the new flow in the case that the packet is from an existing flow.

37. A method according to claim 29, wherein the set of possible state operations that the state processor is configured to perform includes searching for one or more patterns in the packet portions.

38. A monitor according to claim 36, wherein the state processor is programmable, the monitor further including a state patterns/operations memory coupled to the state processor, the state operations memory configured to store a database of protocol dependent state patterns/operations.

39. A monitor according to claim 35, further including a state processor coupled to the UFKB and to the flow-entry-database memory, and configured to perform any state operations specified for the state of the flow starting from the last encountered state of the flow in the case that the packet is from an existing flow, and to perform any state operations required for the initial state of the new flow in the case that the packet is from an existing flow.

40. A monitor according to claim 36, wherein the state operations include updating the flow-entry, including identifying information for future packets to be identified with the flow-entry.

41. A packet monitor according to claim 29, further comprising:

a compiler processor coupled to the parsing/extraction operations memory, the compiler processor configured to run a compilation process that includes:

receiving commands in a high-level protocol description language that describe the protocols that may be used in packets encountered by the monitor and any children protocols thereof, and

translating the protocol description language commands into a plurality of parsing/extraction operations that are initialized into the parsing/extraction operations memory.

42. A packet monitor according to claim 38, further comprising:

a compiler processor coupled to the parsing/extraction operations memory, the compiler processor configured to run a compilation process that includes:

receiving commands in a high-level protocol description language that describe a correspondence between a set of one or more application programs and the state transition patterns/operations that occur as a result of particular conversational flow-sequences associated with an application programs, and

translating the protocol description language commands into a plurality of state patterns and state operations that are initialized into the state patterns/operations memory.

43. A packet monitor according to claim 29, further comprising:

a cache subsystem coupled to and between the lookup engine and the flow-entry database memory providing for fast access of a set of likely-to-be-accessed flow-entries from the flow-entry database.

44. A packet monitor according to claim 43, wherein the cache subsystem is an associative cache subsystem including one or more content addressable memory cells (CAMs).

45. A packet monitor according to claim 44, wherein the cache subsystem is also a least-recently-used cache memory such that a cache miss updates the least recently used cache entry.

46. A packet monitor according to claim 29, wherein each flow-entry stores one or more statistical measures about the flow, the monitor further comprising

a calculator for updating at least one of the statistical measures in the flow-entry of the accepted packet.

47. A packet monitor according to claim 46, wherein the one or more statistical measures include measures selected from the set consisting of the total packet count for the flow, the time, and a differential time from the last entered time to the present time.

48. A packet monitor according to claim 46, further including a statistical processor configured to determine one or more network usage metrics related to the flow from one or more of the statistical measures in a flow-entry.

49. A monitor according to claim 29, wherein:

flow-entry-database is organized into a plurality of bins that each contain N-number of flow-entries, and wherein said bins are accessed via a hash data value created by a parser subsystem based on the selected packet portions, wherein N is one or more.

50. A monitor according to claim 49, wherein the hash data value is used to spread a plurality of flow-entries across the flow-entry-database and allows fast lookup of a flow-entry and shallower buckets.

51. A monitor according to claim 36, wherein the state processor analyzes both new and existing flows in order to classify them by application and proceeds from state-to-state based on a set of predefined rules.

52. A monitor according to claim 29, wherein the lookup engine begins processing as soon as a parser record arrives from the parser subsystem.

53. A monitor according to claim 36, wherein the lookup engine provides for flow state entry checking to see if a flow key should be sent to the state processor, and that outputs a protocol identifier for the flow.

54. A method of examining packets passing through a connection point on a computer network, the method comprising:

(a) receiving a packet from a packet acquisition device;

(b) performing one or more parsing/extraction operations on the packet according to a database of parsing/extraction operations to create a parser record comprising a function of selected portions of the packet, the database of parsing/extraction operations including information on how to determine a set of one or more protocol dependent extraction operations from data in the packet that indicate a protocol is used in the packet;

(c) looking up a flow-entry database comprising none or more flow-entries for previously encountered conversational flows, the looking up using at least some of the selected packet portions, and determining if the packet is of an existing flow;

(d) if the packet is of an existing flow, obtaining the last encountered state of the flow and performing any state operations specified for the state of the flow starting from the last encountered state of the flow; and

(e) if the packet is of a new flow, performing any analysis required for the initial state of the new flow and storing a new flow-entry for the new flow in the flow-entry database, including identifying information for future packets to be identified with the new flow-entry.

55. A method according to claim 54, wherein one of the state operations specified for at least one of the states includes updating the flow-entry, including identifying information for future packets to be identified with the flow-entry.

56. A method according to claim 54, wherein one of the state operations specified for at least one of the states includes searching the contents of the packet for at least one reference string.

57. A method according to claim 55, wherein one of the state operations specified for at least one of the states includes creating a new flow-entry for future packets to be identified with the flow, the new flow-entry including identifying information for future packets to be identified with the flow-entry.

58. A method according to claim 54, further comprising forming a signature from the selected packet portions, wherein the lookup operation uses the signature and wherein the identifying information stored in the new or updated flow-entry is a signature for identifying future packets.

59. A method according to claim 54, wherein the state operations are according to a database of protocol dependent state operations.

ABSTRACT

A monitor for and a method of examining packets passing through a connection point on a computer network. Each packets conforms to one or more protocols. The method includes receiving a packet from a packet acquisition device and performing one or more parsing/extraction operations on the packet to create a parser record comprising a function of selected portions of the packet. The parsing/extraction operations depend on one or more of the protocols to which the packet conforms. The method further includes looking up a flow-entry database containing flow-entries for previously encountered conversational flows. The lookup uses the selected packet portions and determining if the packet is of an existing flow. If the packet is of an existing flow, the method classifies the packet as belonging to the found existing flow, and if the packet is of a new flow, the method stores a new flow-entry for the new flow in the flow-entry database, including identifying information for future packets to be identified with the new flow-entry. For the packet of an existing flow, the method updates the flow-entry of the existing flow. Such updating may include storing one or more statistical measures. Any stage of a flow, state is maintained, and the method performs any state processing for an identified state to further the process of identifying the flow. The method thus examines each and every packet passing through the connection point in real time until the application program associated with the conversational flow is determined. The method

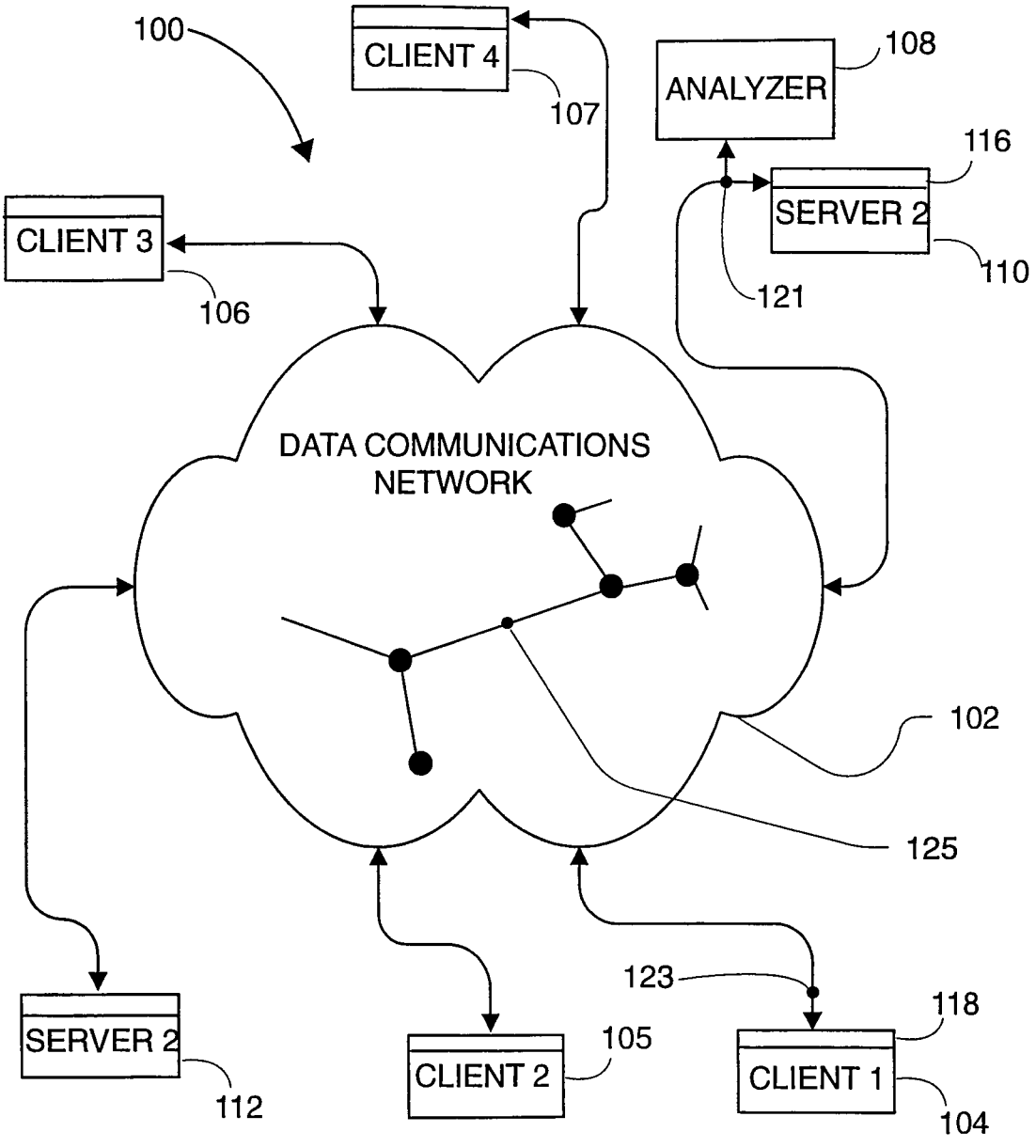

100
CLIENT 4
107
ANALYZER
108
116
SERVER 2
110
121
CLIENT 3
106
DATA COMMUNICATIONS NETWORK
102
125
123
SERVER 2
112
CLIENT 2
105
CLIENT 1
118
104

FIG. 1

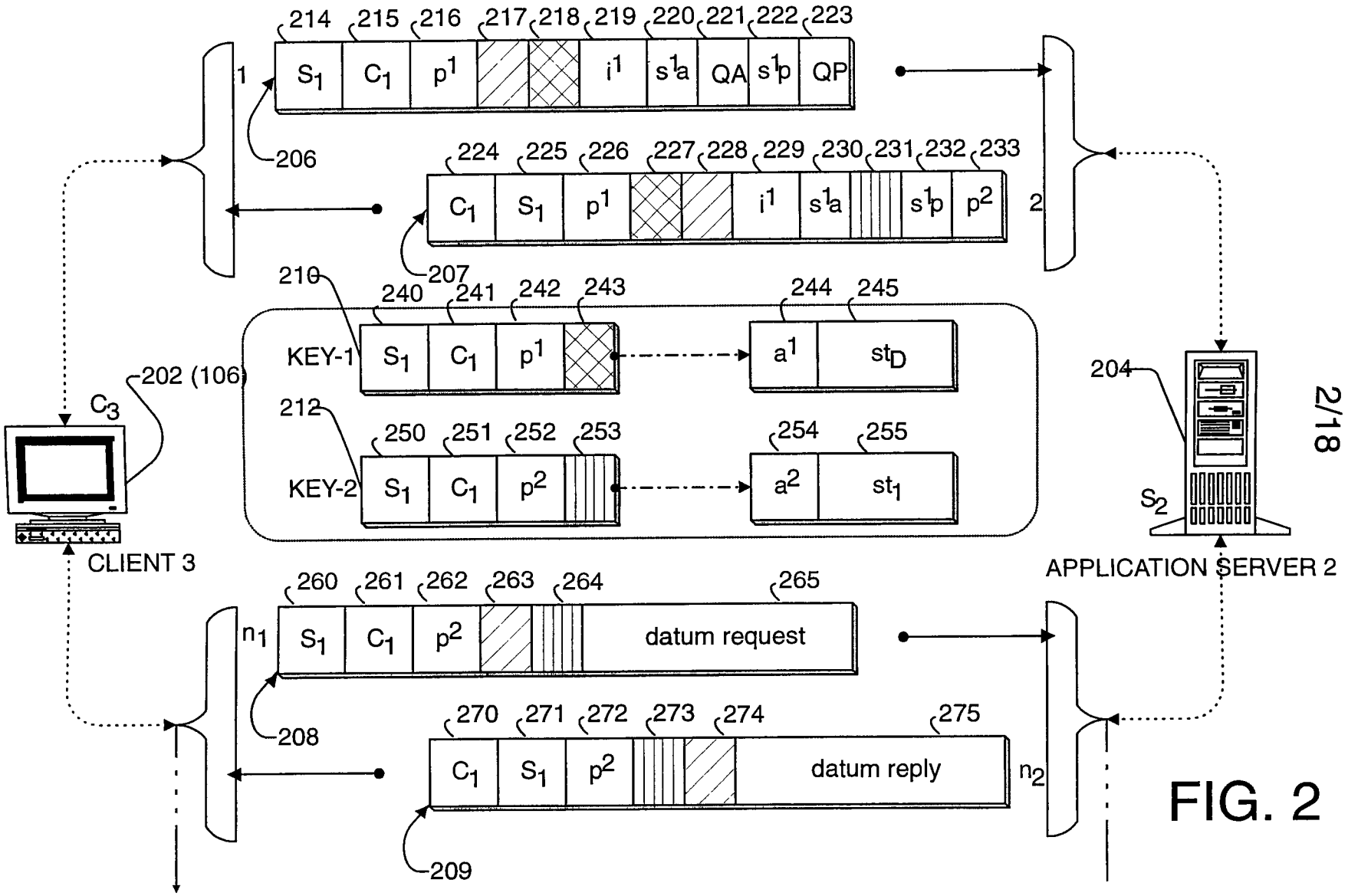

214 215 216 217 218 219 220 221 222 223
S1 C1 p1 i1 s1a QA s1p QP
1
206
224 225 226 227 228 229 230 231 232 233
C1 S1 p1 i1 s1a s1p p2
2
207
210
240 241 242 243
KEY-1 S1 C1 p1
244 245
a1 stD
212
250 251 252 253
KEY-2 S1 C1 p2
254 255
a2 st1
202 (106)
C3
CLIENT 3
204
S2
APPLICATION SERVER 2
260 261 262 263 264 265
n1 S1 C1 p2 datum request
208
270 271 272 273 274 275
C1 S1 p2 datum reply
n2
209

FIG. 2

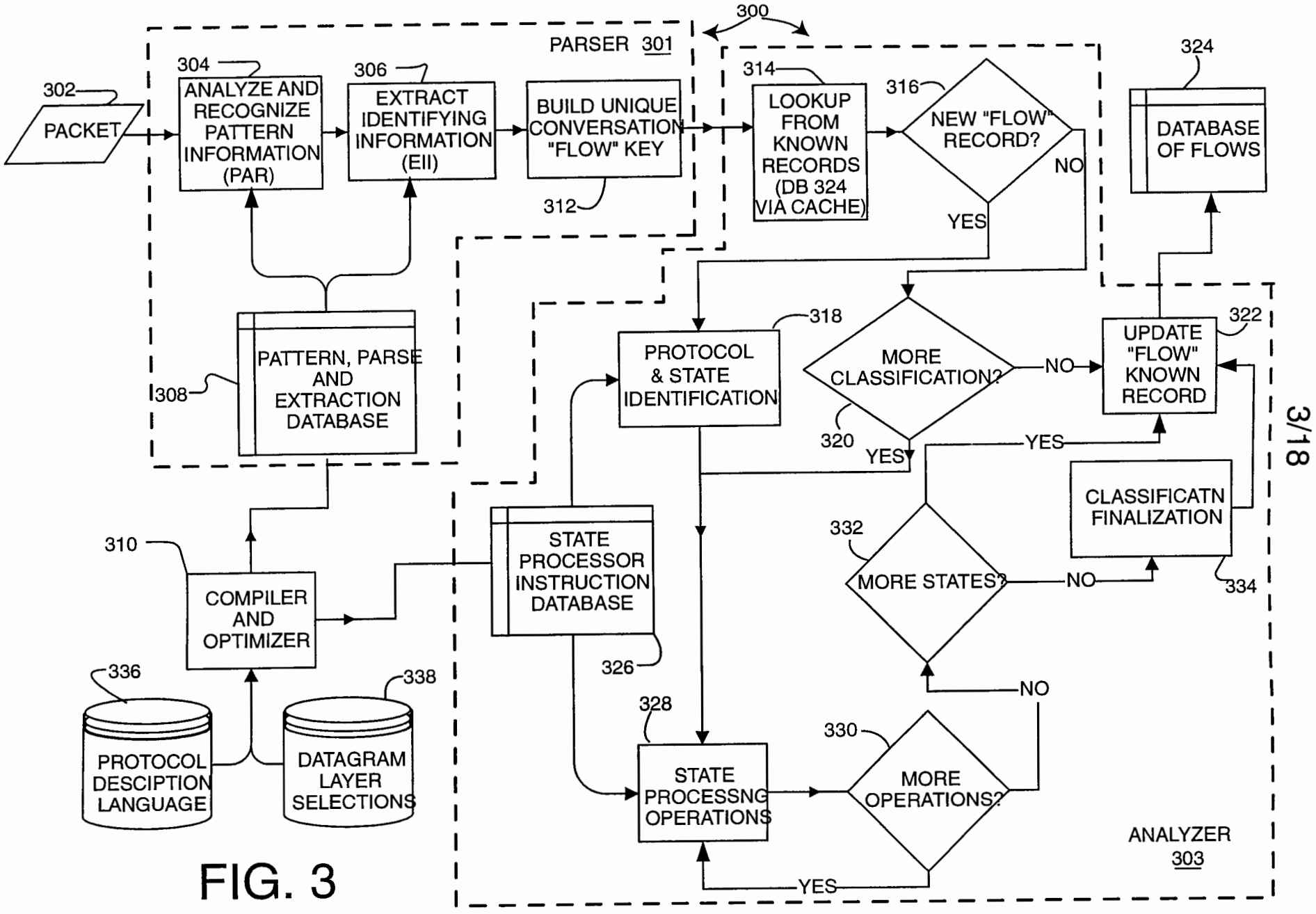

300
PARSER 301
302
PACKET
304
ANALYZE AND RECOGNIZE PATTERN INFORMATION (PAR)
306
EXTRACT IDENTIFYING INFORMATION (EII)
BUILD UNIQUE CONVERSATION "FLOW" KEY
312
314
LOOKUP FROM KNOWN RECORDS (DB 324 VIA CACHE)
316
NEW "FLOW" RECORD?
NO
YES
324
DATABASE OF FLOWS
308
PATTERN, PARSE AND EXTRACTION DATABASE
318
PROTOCOL & STATE IDENTIFICATION
MORE CLASSIFICATION?
320
NO
YES
322
UPDATE "FLOW" KNOWN RECORD
YES
CLASSIFICATN FINALIZATION
334
310
COMPILER AND OPTIMIZER
STATE PROCESSOR INSTRUCTION DATABASE
326
332
MORE STATES?
NO
NO
336
338
328
330
PROTOCOL DESCIPTION LANGUAGE
DATAGRAM LAYER SELECTIONS
STATE PROCESSNG OPERATIONS
MORE OPERATIONS?
YES
ANALYZER 303

FIG. 3

3/18

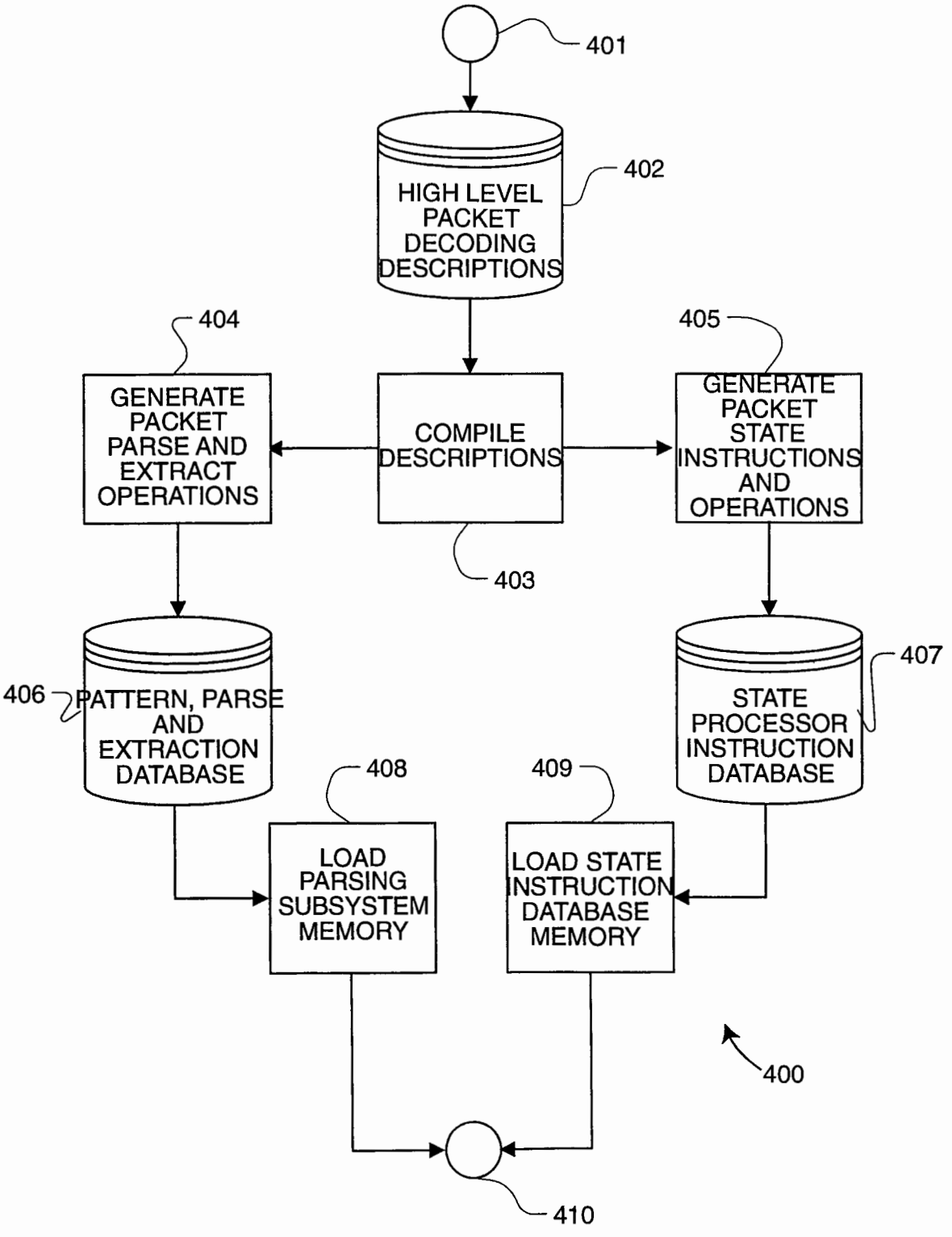

401
HIGH LEVEL PACKET DECODING DESCRIPTIONS
402
404
405
GENERATE PACKET PARSE AND EXTRACT OPERATIONS
COMPILE DESCRIPTIONS
GENERATE PACKET STATE INSTRUCTIONS AND OPERATIONS
403
406
PATTERN, PARSE AND EXTRACTION DATABASE
407
STATE PROCESSOR INSTRUCTION DATABASE
408
409
LOAD PARSING SUBSYSTEM MEMORY
LOAD STATE INSTRUCTION DATABASE MEMORY
400
410

FIG. 4

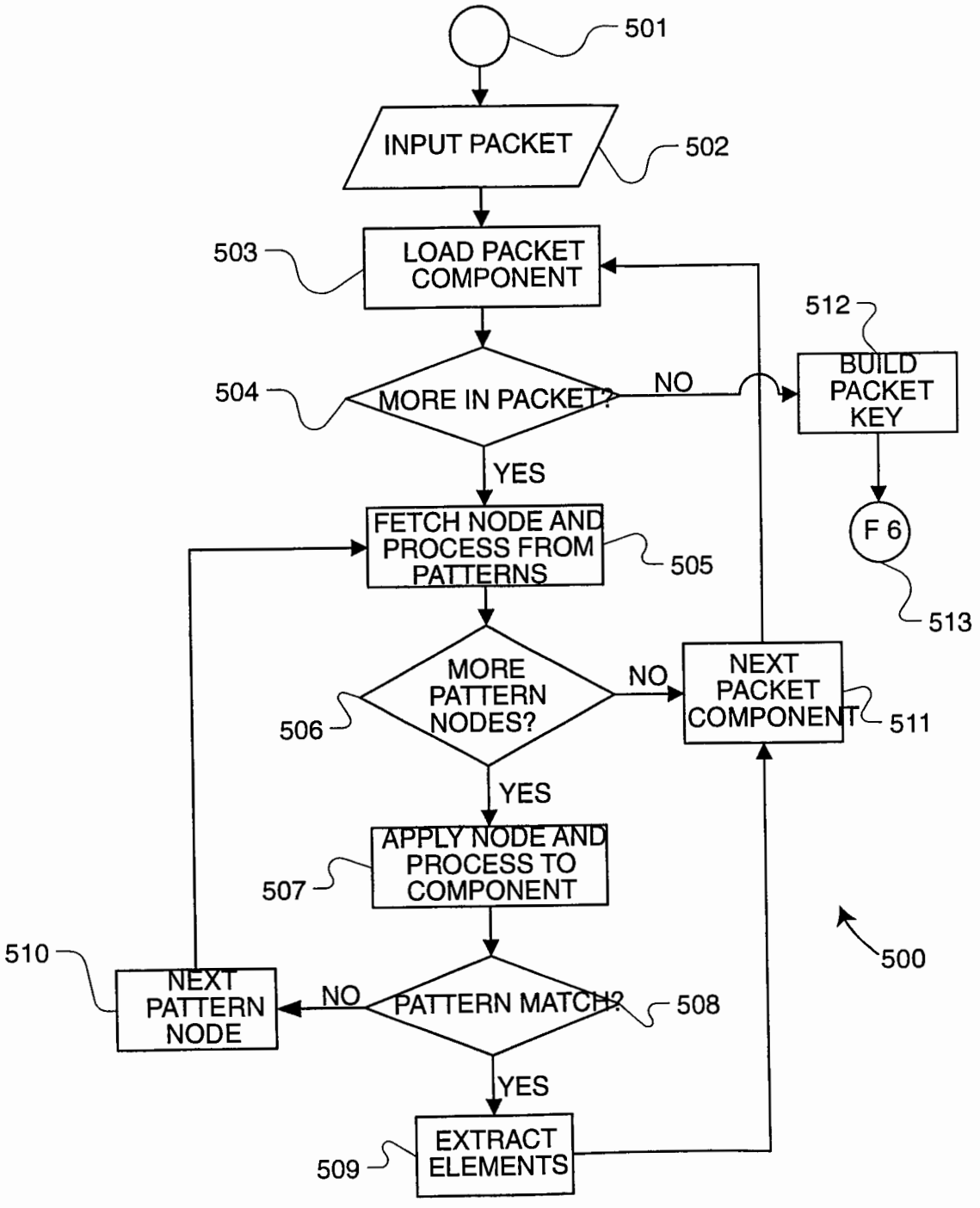

501
INPUT PACKET
502
LOAD PACKET COMPONENT
503
MORE IN PACKET?
504
NO
BUILD PACKET KEY
512
F 6
513
YES
FETCH NODE AND PROCESS FROM PATTERNS
505
MORE PATTERN NODES?
506
NO
NEXT PACKET COMPONENT
511
YES
APPLY NODE AND PROCESS TO COMPONENT
507
500
NEXT PATTERN NODE
510
NO
PATTERN MATCH?
508
YES
EXTRACT ELEMENTS
509

FIG. 5

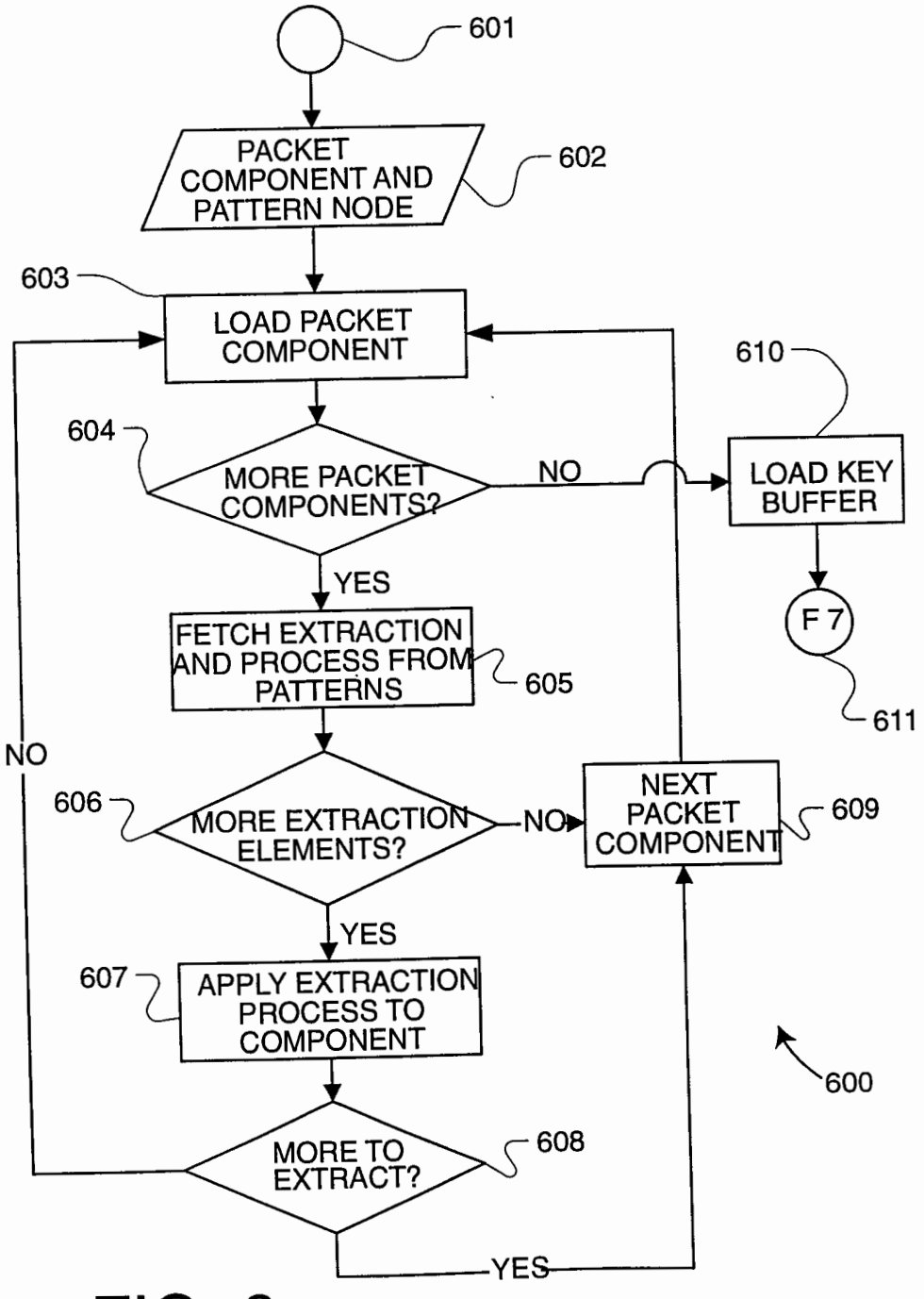

601
PACKET COMPONENT AND PATTERN NODE
602
603
LOAD PACKET COMPONENT
610
604
MORE PACKET COMPONENTS?
NO
LOAD KEY BUFFER
YES
F 7
FETCH EXTRACTION AND PROCESS FROM PATTERNS
605
611
NO
606
MORE EXTRACTION ELEMENTS?
NO
NEXT PACKET COMPONENT
609
YES
607
APPLY EXTRACTION PROCESS TO COMPONENT
600
MORE TO EXTRACT?
608
YES

FIG. 6

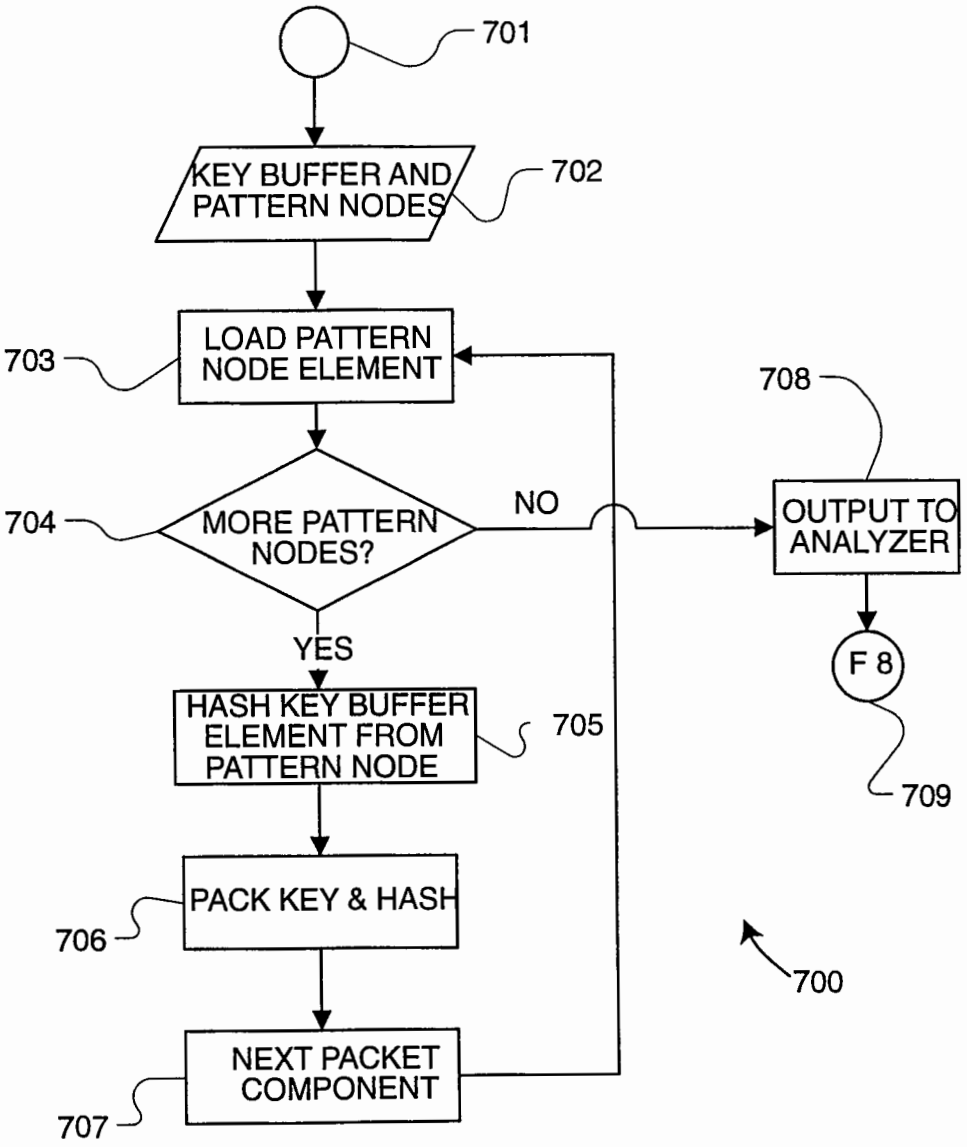

701
KEY BUFFER AND PATTERN NODES
702
LOAD PATTERN NODE ELEMENT
703
MORE PATTERN NODES?
704
NO
YES
OUTPUT TO ANALYZER
708
F 8
709
HASH KEY BUFFER ELEMENT FROM PATTERN NODE
705
PACK KEY & HASH
706
NEXT PACKET COMPONENT
707
700

FIG. 7

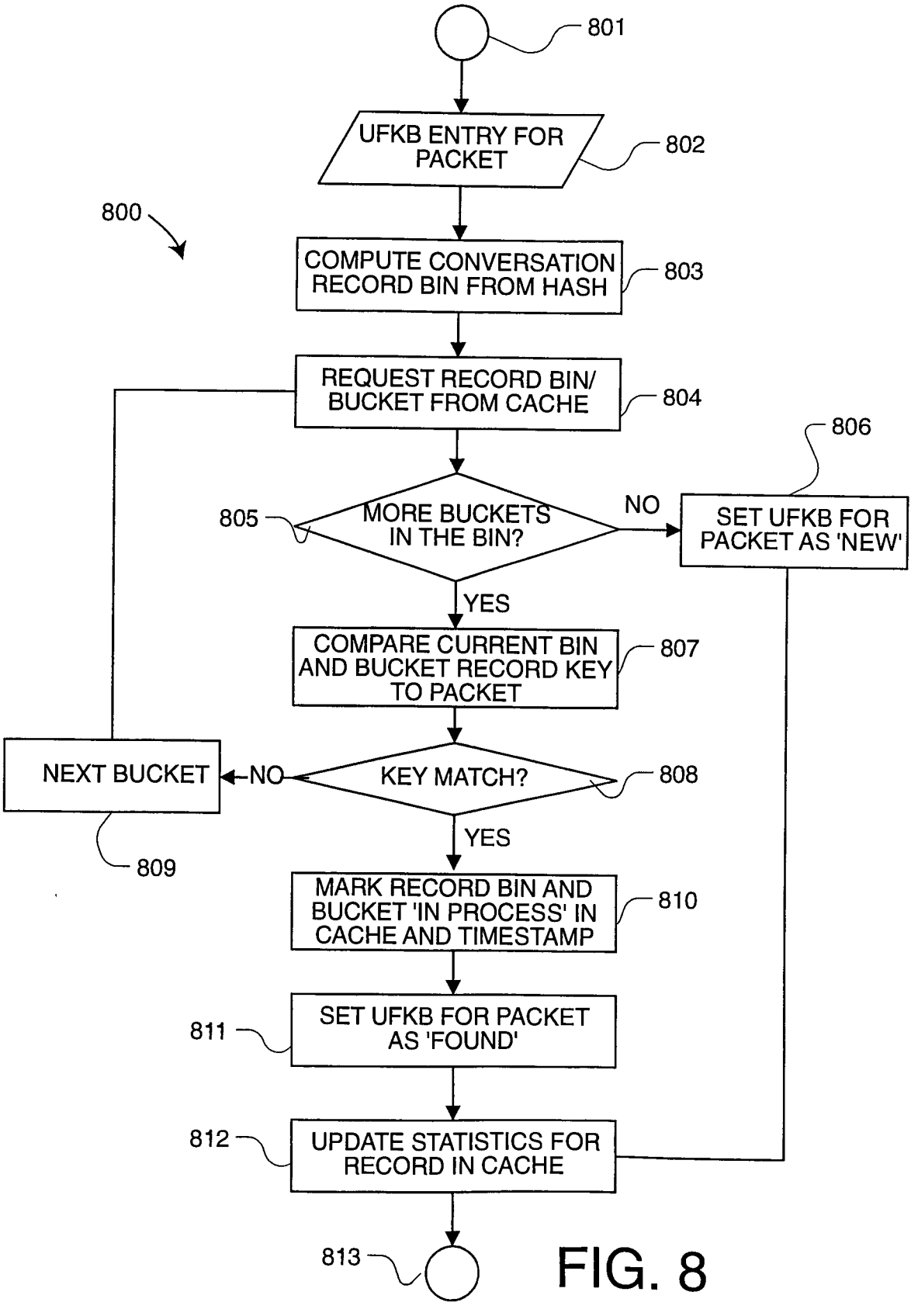

801
UFKB ENTRY FOR PACKET
802
800
COMPUTE CONVERSATION RECORD BIN FROM HASH
803
REQUEST RECORD BIN/ BUCKET FROM CACHE
804
806
805
MORE BUCKETS IN THE BIN?
NO
SET UFKB FOR PACKET AS 'NEW'
YES
COMPARE CURRENT BIN AND BUCKET RECORD KEY TO PACKET
807
NEXT BUCKET
NO
KEY MATCH?
808
YES
809
MARK RECORD BIN AND BUCKET 'IN PROCESS' IN CACHE AND TIMESTAMP
810
811
SET UFKB FOR PACKET AS 'FOUND'
812
UPDATE STATISTICS FOR RECORD IN CACHE
813

FIG. 8

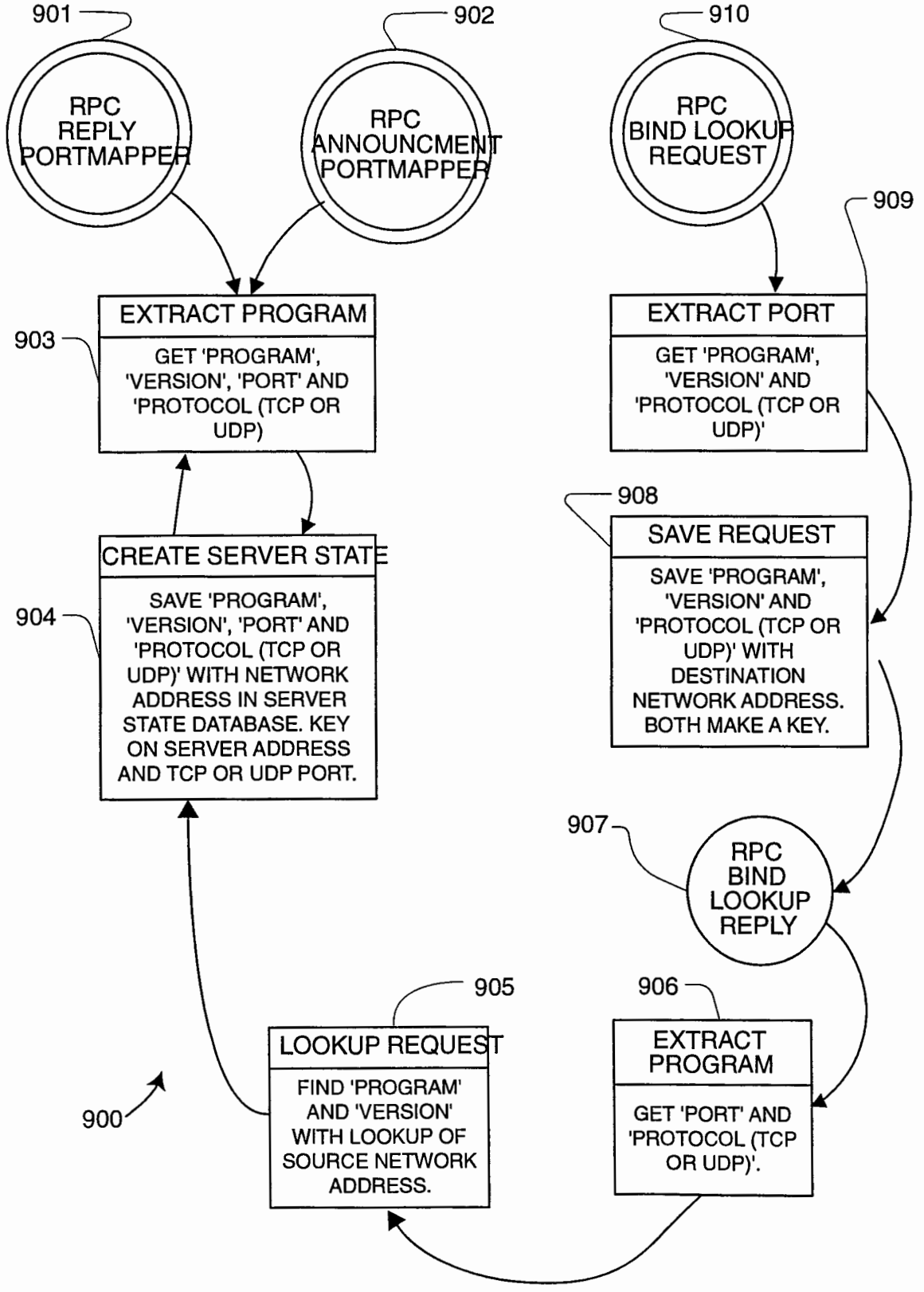

901
RPC REPLY PORTMAPPER
902
RPC ANNOUNCMENT PORTMAPPER
910
RPC BIND LOOKUP REQUEST
909
EXTRACT PROGRAM
903
GET 'PROGRAM', 'VERSION', 'PORT' AND 'PROTOCOL (TCP OR UDP)
EXTRACT PORT
GET 'PROGRAM', 'VERSION' AND 'PROTOCOL (TCP OR UDP)'
908
SAVE REQUEST
SAVE 'PROGRAM', 'VERSION' AND 'PROTOCOL (TCP OR UDP)' WITH DESTINATION NETWORK ADDRESS. BOTH MAKE A KEY.
CREATE SERVER STATE
904
SAVE 'PROGRAM', 'VERSION', 'PORT' AND 'PROTOCOL (TCP OR UDP)' WITH NETWORK ADDRESS IN SERVER STATE DATABASE. KEY ON SERVER ADDRESS AND TCP OR UDP PORT.
907
RPC BIND LOOKUP REPLY
905
LOOKUP REQUEST
FIND 'PROGRAM' AND 'VERSION' WITH LOOKUP OF SOURCE NETWORK ADDRESS.
906
EXTRACT PROGRAM
GET 'PORT' AND 'PROTOCOL (TCP OR UDP)'.
900

FIG. 9

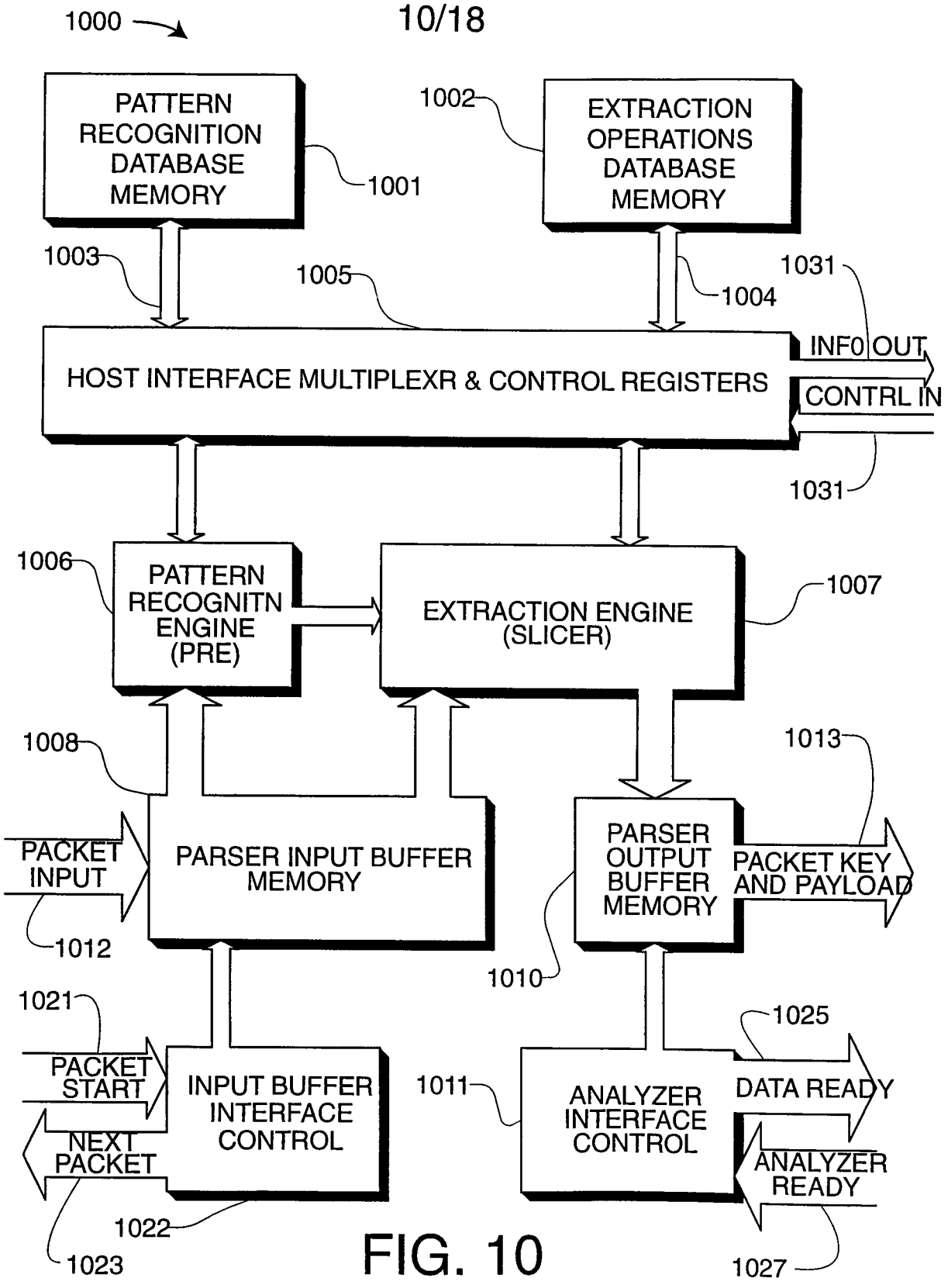

1000
PATTERN RECOGNITION DATABASE MEMORY
1001
1002
EXTRACTION OPERATIONS DATABASE MEMORY
1003
1004
1005
1031
HOST INTERFACE MULTIPLEXR & CONTROL REGISTERS
INF0 OUT
CONTRL IN
1031
1006
PATTERN RECOGNITN ENGINE (PRE)
EXTRACTION ENGINE (SLICER)
1007
1008
1013
PACKET INPUT
PARSER INPUT BUFFER MEMORY
PARSER OUTPUT BUFFER MEMORY
PACKET KEY AND PAYLOAD
1012
1010
1021
1025
PACKET START
INPUT BUFFER INTERFACE CONTROL
1011
ANALYZER INTERFACE CONTROL
DATA READY
NEXT PACKET
ANALYZER READY
1022
1023
1027

FIG. 10

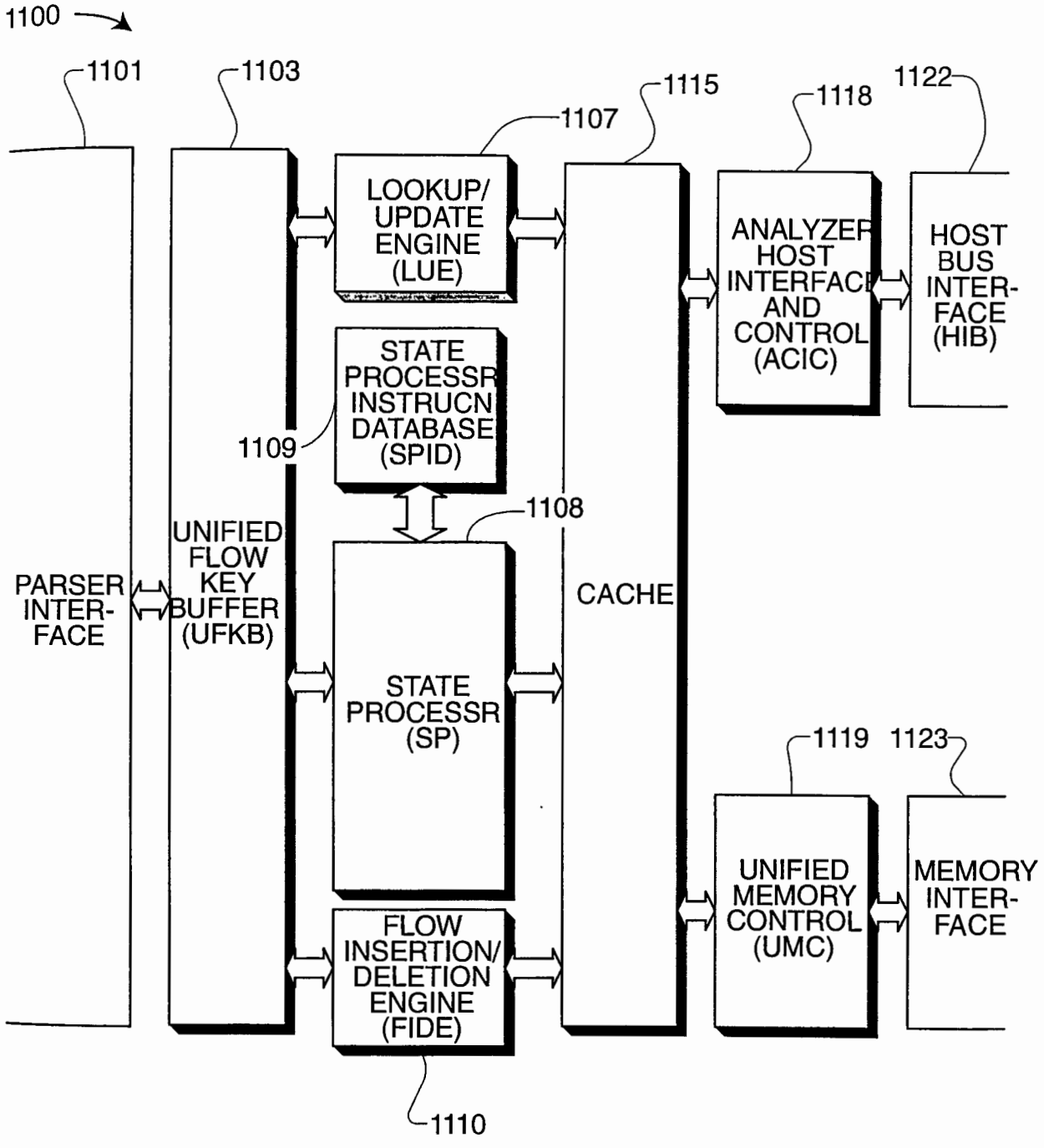

1100
1101
1103
1107
1115
1118
1122
LOOKUP/ UPDATE ENGINE (LUE)
ANALYZER HOST INTERFACE AND CONTROL (ACIC)
HOST BUS INTER-FACE (HIB)
STATE PROCESSR INSTRUCN DATABASE (SPID)
1109
1108
PARSER INTER-FACE
UNIFIED FLOW KEY BUFFER (UFKB)
CACHE
STATE PROCESSR (SP)
1119
1123
UNIFIED MEMORY CONTROL (UMC)
MEMORY INTER-FACE
FLOW INSERTION/ DELETION ENGINE (FIDE)
1110

FIG. 11

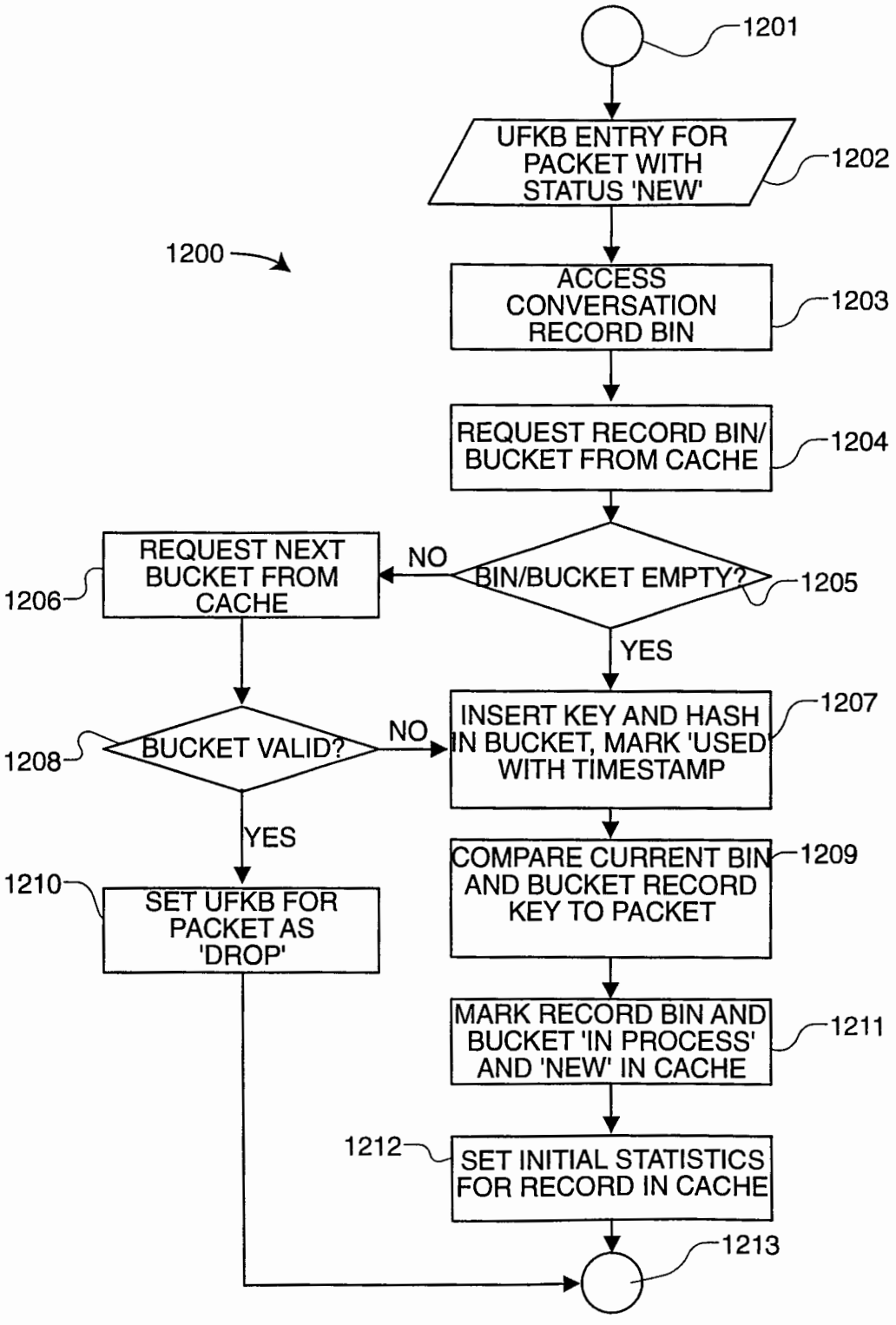

1201
UFKB ENTRY FOR PACKET WITH STATUS 'NEW'
1202
1200
ACCESS CONVERSATION RECORD BIN
1203
REQUEST RECORD BIN/ BUCKET FROM CACHE
1204
REQUEST NEXT BUCKET FROM CACHE
NO
BIN/BUCKET EMPTY?
1205
1206
YES
INSERT KEY AND HASH IN BUCKET, MARK 'USED WITH TIMESTAMP
1207
BUCKET VALID?
NO
1208
YES
COMPARE CURRENT BIN AND BUCKET RECORD KEY TO PACKET
1209
1210
SET UFKB FOR PACKET AS 'DROP'
MARK RECORD BIN AND BUCKET 'IN PROCESS' AND 'NEW' IN CACHE
1211
1212
SET INITIAL STATISTICS FOR RECORD IN CACHE
1213

FIG. 12

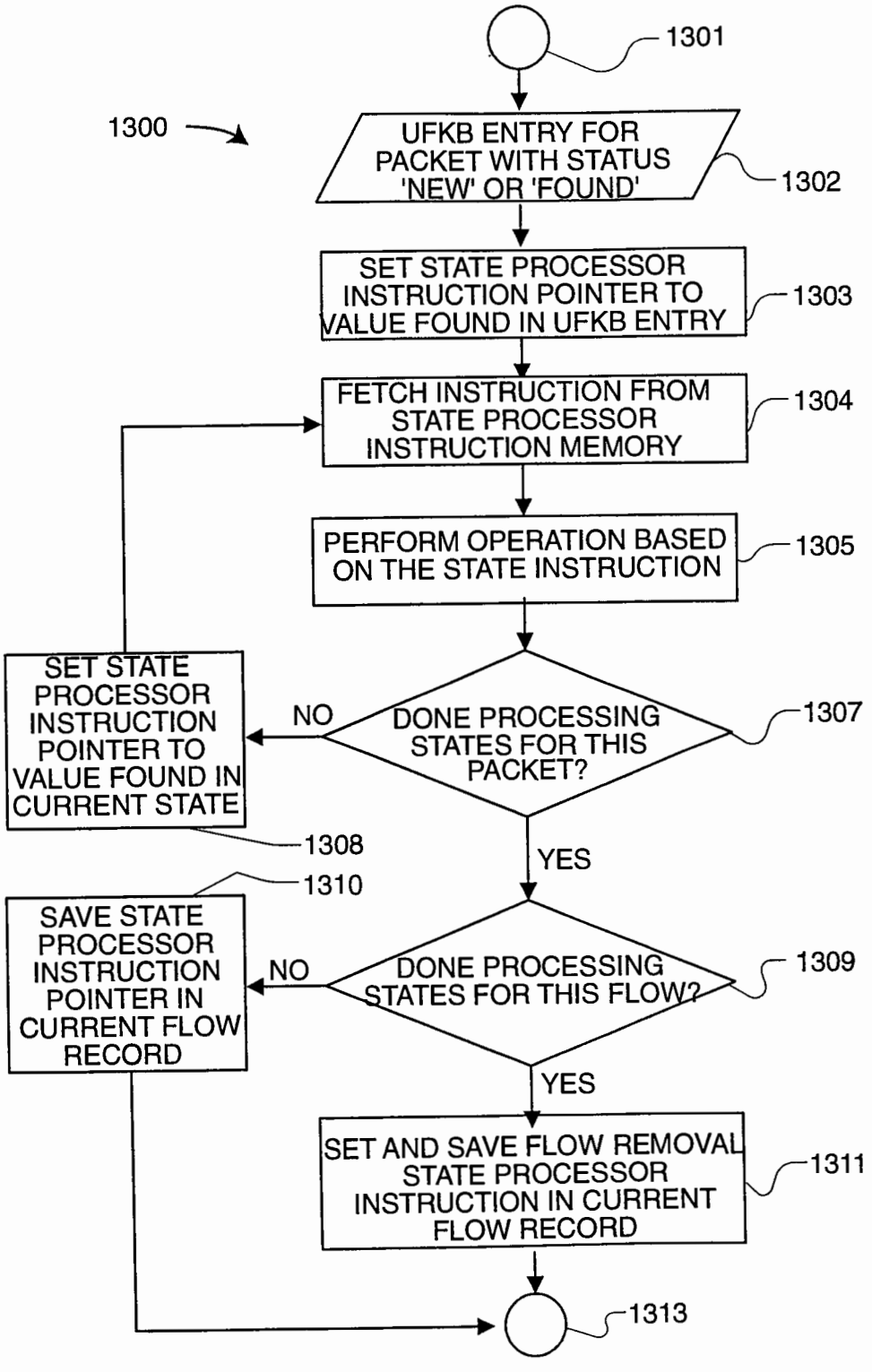

1301
1300
UFKB ENTRY FOR PACKET WITH STATUS 'NEW' OR 'FOUND'
1302
SET STATE PROCESSOR INSTRUCTION POINTER TO VALUE FOUND IN UFKB ENTRY
1303
FETCH INSTRUCTION FROM STATE PROCESSOR INSTRUCTION MEMORY
1304
PERFORM OPERATION BASED ON THE STATE INSTRUCTION
1305
DONE PROCESSING STATES FOR THIS PACKET?
1307
NO
SET STATE PROCESSOR INSTRUCTION POINTER TO VALUE FOUND IN CURRENT STATE
1308
YES
1310
DONE PROCESSING STATES FOR THIS FLOW?
1309
NO
SAVE STATE PROCESSOR INSTRUCTION POINTER IN CURRENT FLOW RECORD
YES
SET AND SAVE FLOW REMOVAL STATE PROCESSOR INSTRUCTION IN CURRENT FLOW RECORD
1311
1313

FIG. 13

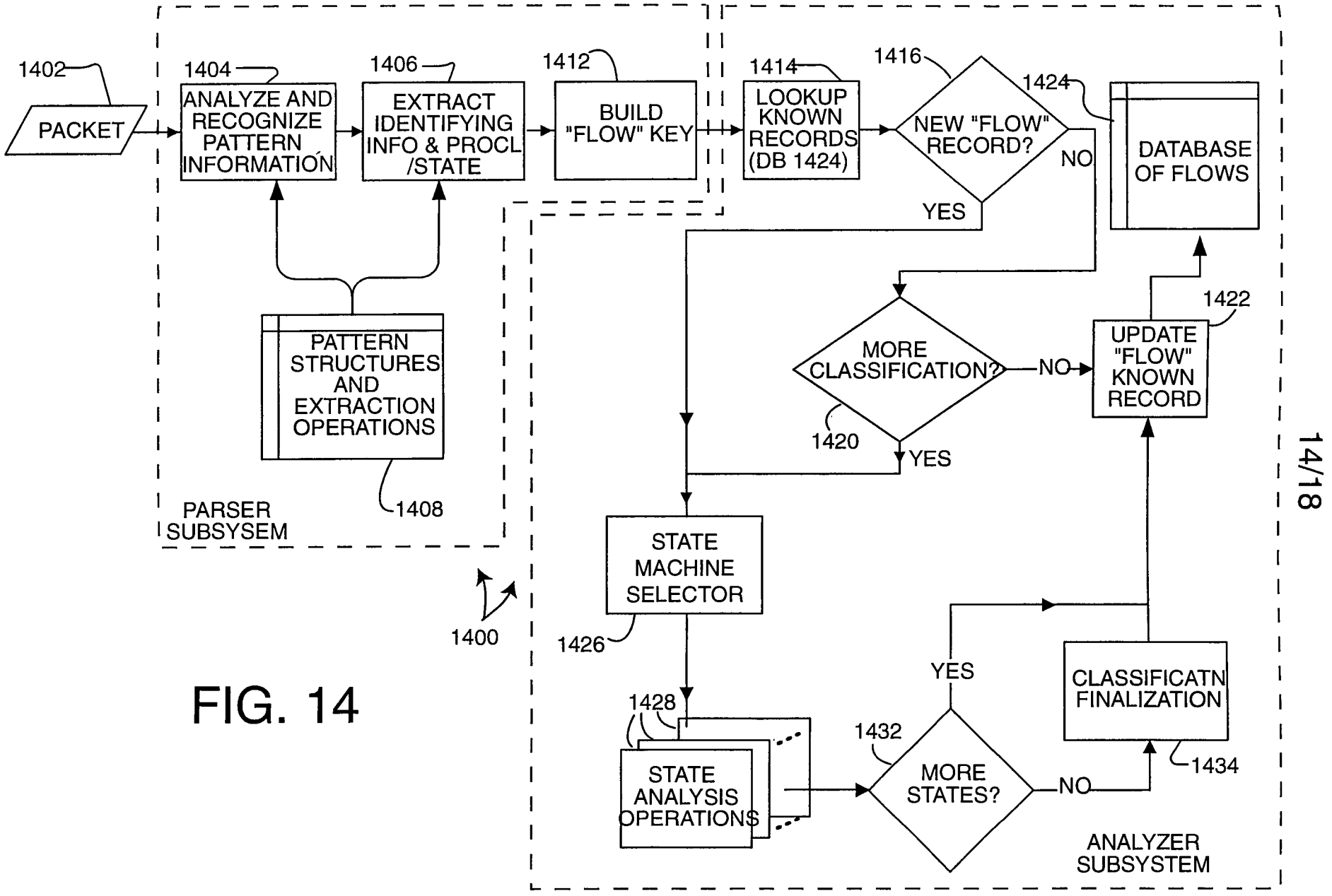

1402
PACKET
1404
ANALYZE AND RECOGNIZE PATTERN INFORMATION
1406
EXTRACT IDENTIFYING INFO & PROCL /STATE
1412
BUILD "FLOW" KEY
1414
LOOKUP KNOWN RECORDS (DB 1424)
1416
NEW "FLOW" RECORD?
NO
YES
1424
DATABASE OF FLOWS
PATTERN STRUCTURES AND EXTRACTION OPERATIONS
1408
PARSER SUBSYSEM
MORE CLASSIFICATION?
NO
YES
1420
1422
UPDATE "FLOW" KNOWN RECORD
1400
STATE MACHINE SELECTOR
1426
1428
STATE ANALYSIS OPERATIONS
1432
MORE STATES?
YES
NO
CLASSIFICATN FINALIZATION
1434
ANALYZER SUBSYSTEM

FIG. 14

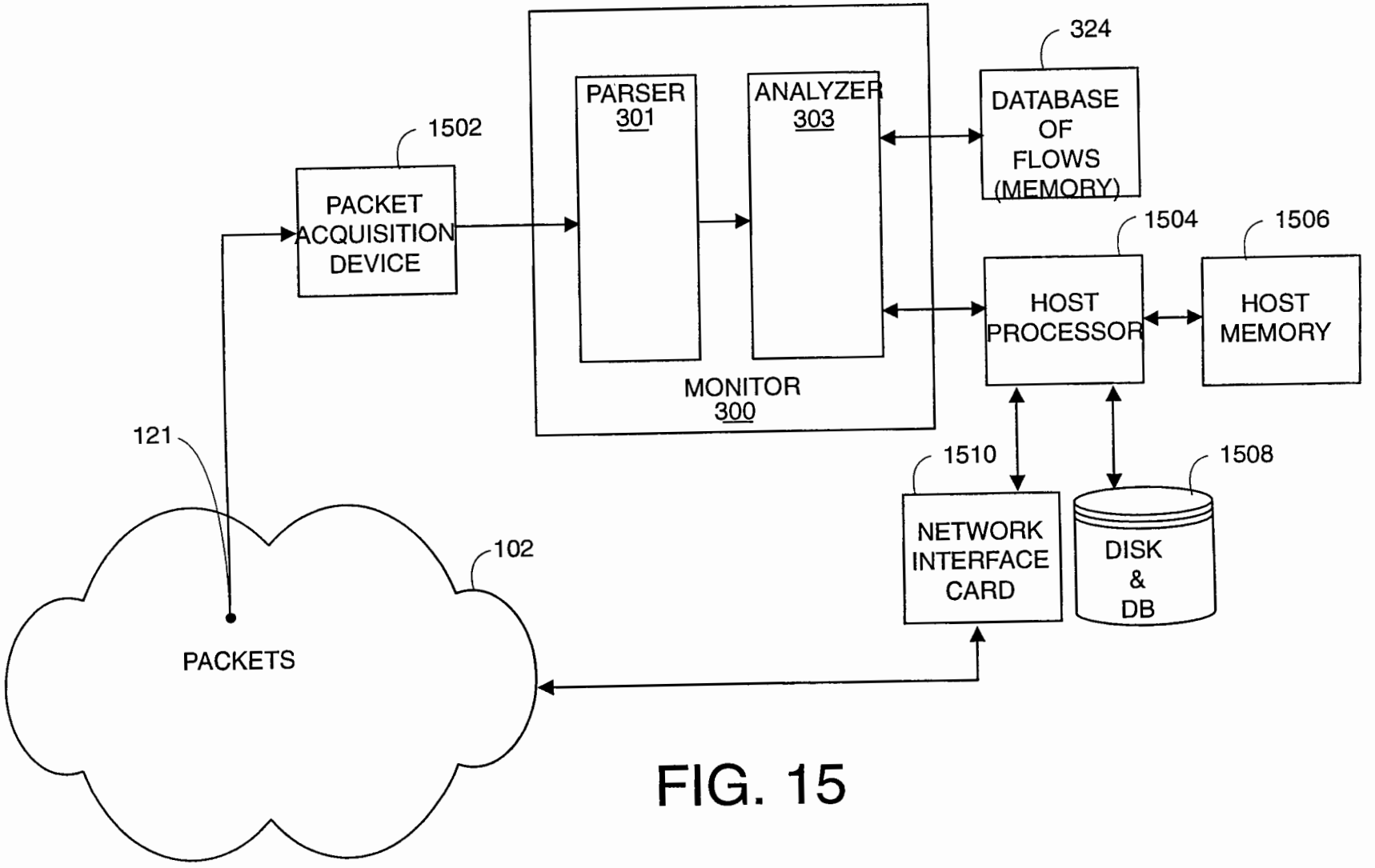

PACKET ACQUISITION DEVICE
1502
PARSER 301
ANALYZER 303
MONITOR 300
324
DATABASE OF FLOWS (MEMORY)
1504
HOST PROCESSOR
1506
HOST MEMORY
121
PACKETS
102
1510
NETWORK INTERFACE CARD
1508
DISK & DB

FIG. 15

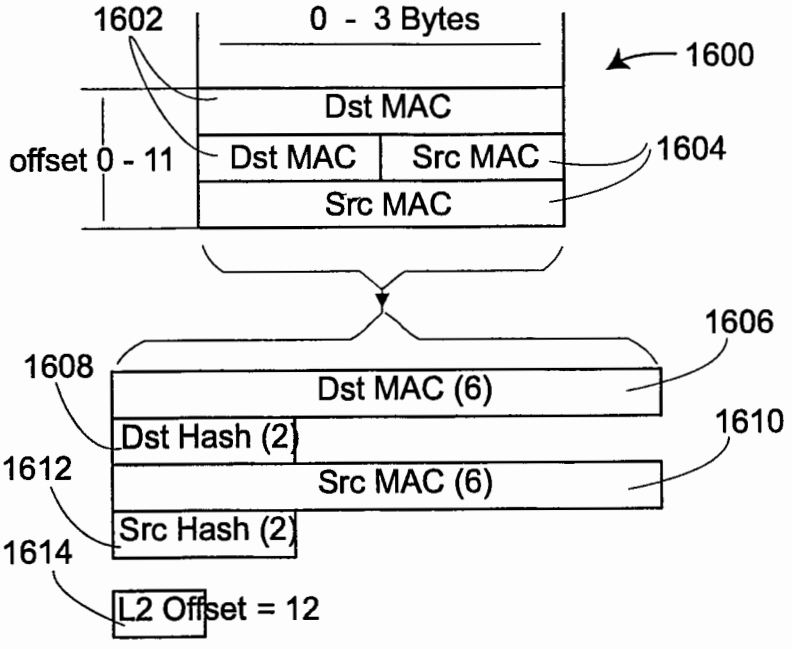

1602
0 - 3 Bytes
1600
Dst MAC
offset 0 - 11
Dst MAC
Src MAC
1604
Src MAC
1606
1608
Dst MAC (6)
Dst Hash (2)
1610
1612
Src MAC (6)
Src Hash (2)
1614
L2 Offset = 12

FIG. 16

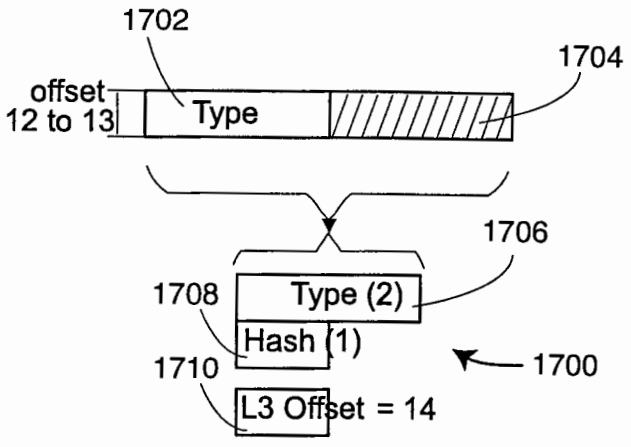


FIG. 17A

1712

IDP = 0x0600*
IP = 0x0800*
CHAOSNET = 0x0804
ARP = 0x0806
VIP = 0x0BAD*
VLOOP = 0x0BAE
VECHO = 0x0BAF
NETBIOS-3COM = 0x3C00 - 0x3C0D#
DEC-MOP = 0x6001
DEC-RC = 0x6002
DEC-DRP = 0x6003*
DEC-LAT = 0x6004
DEC-DIAG = 0x6005
DEC-LAVC = 0x6007
RARP = 0x8035
ATALK = 0x809B*
VLOOP = 0x80C4
VECHO = 0x80C5
SNA-TH = 0x80D5*
ATALKARP = 0x80F3
IPX = 0x8137*
SNMP = 0x814C#
IPv6 = 0x86DD*
LOOPBACK = 0x9000

Apple = 0x080007

* L3 Decoding
# L5 Decoding

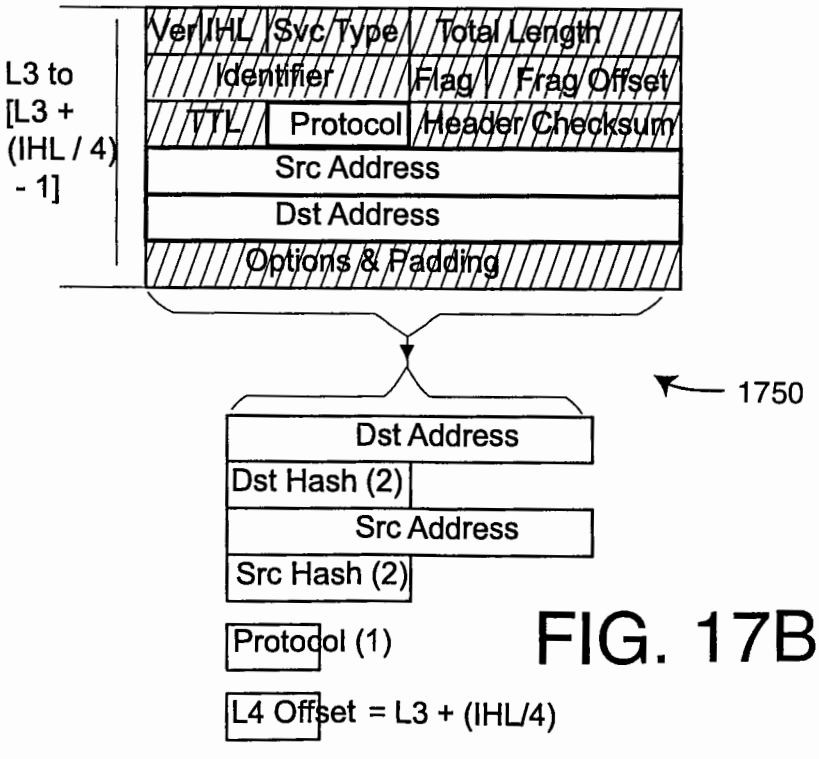


FIG. 17B

1752

ICMP = 1
IGMP = 2
GGP = 3
TCP = 6*
EGP = 8
IGRP = 9
PUP = 12
CHAOS = 16
UDP = 17*
IDP = 22#
ISO-TP4 = 29
DDP = 37#
ISO-IP = 80
VIP = 83#
EIGRP = 88
OSPF = 89

* L4 Decoding
# L3 Re-Decoding

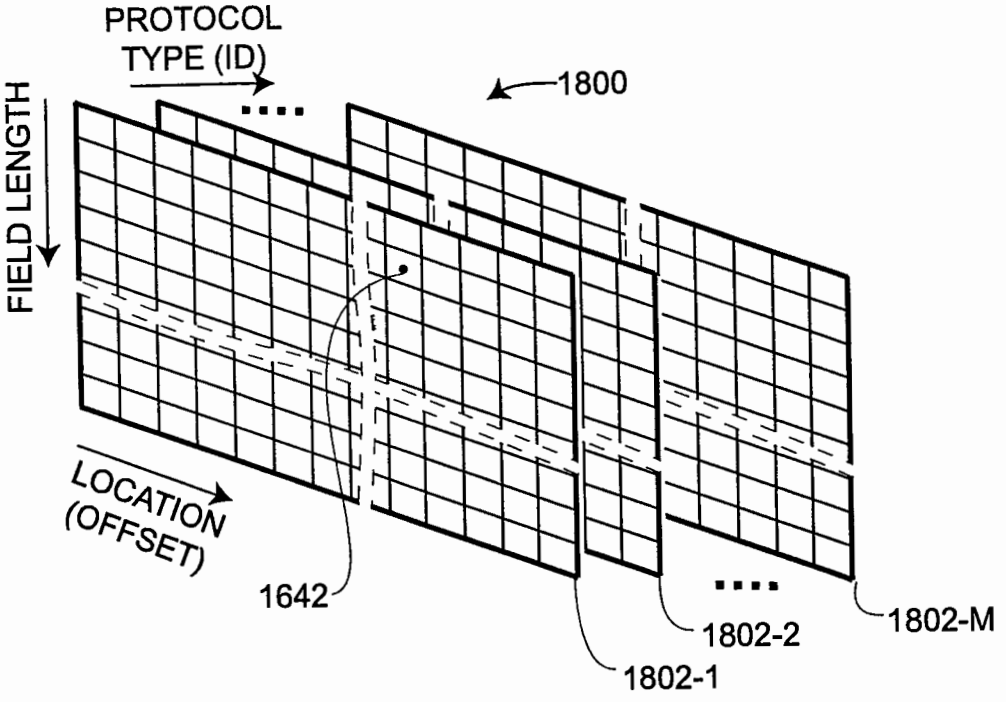


FIG. 18A

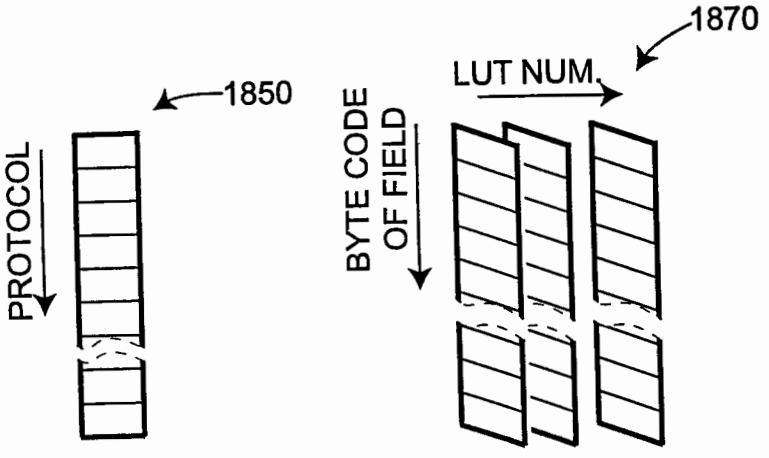


FIG. 18B

PRINT OF DRAWINGS
AS ORIGINALLY FILED 01-1

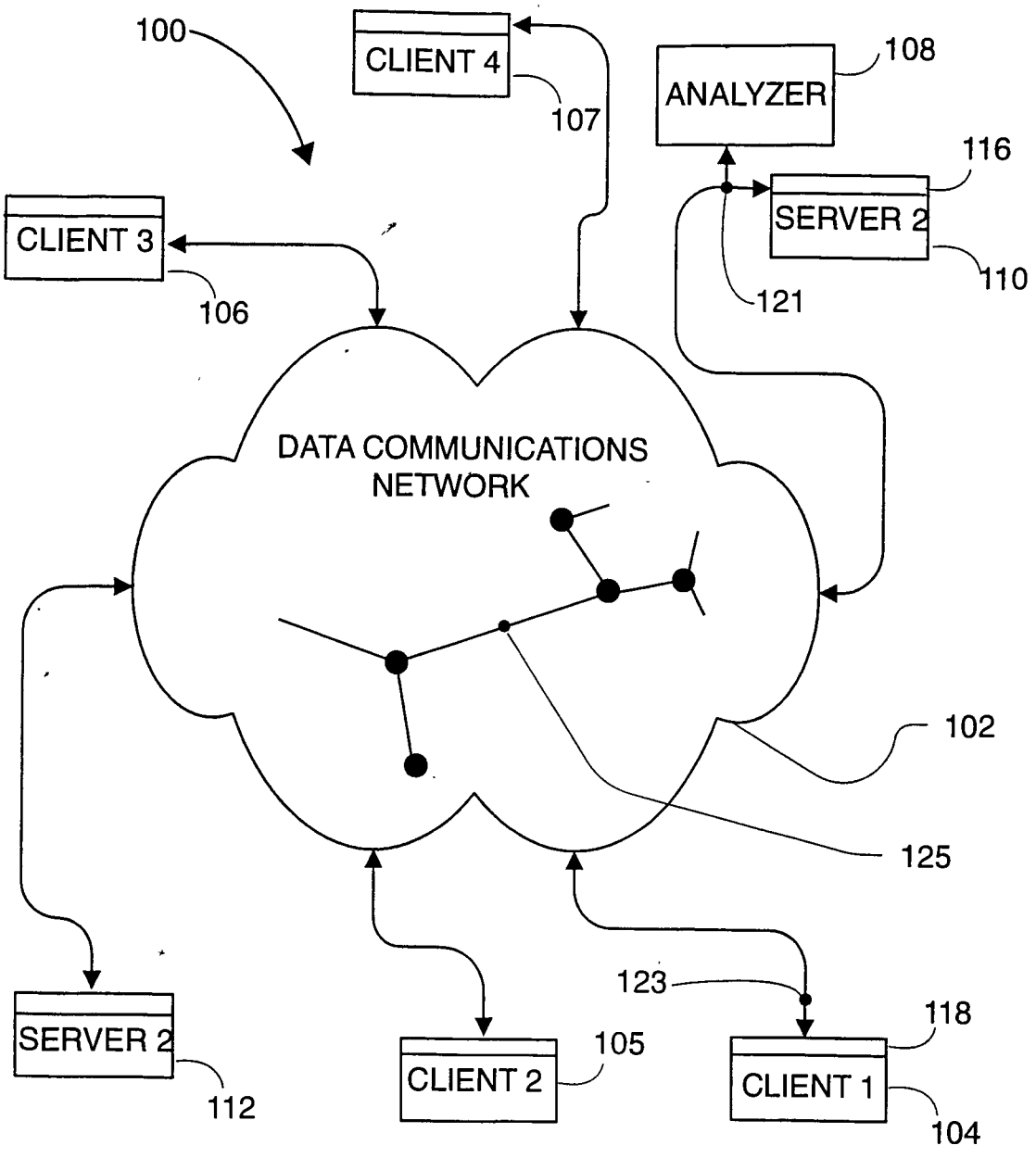


FIG. 1

RENT OF DRAWINGS
AS ORIGINALLY FILED 001-1

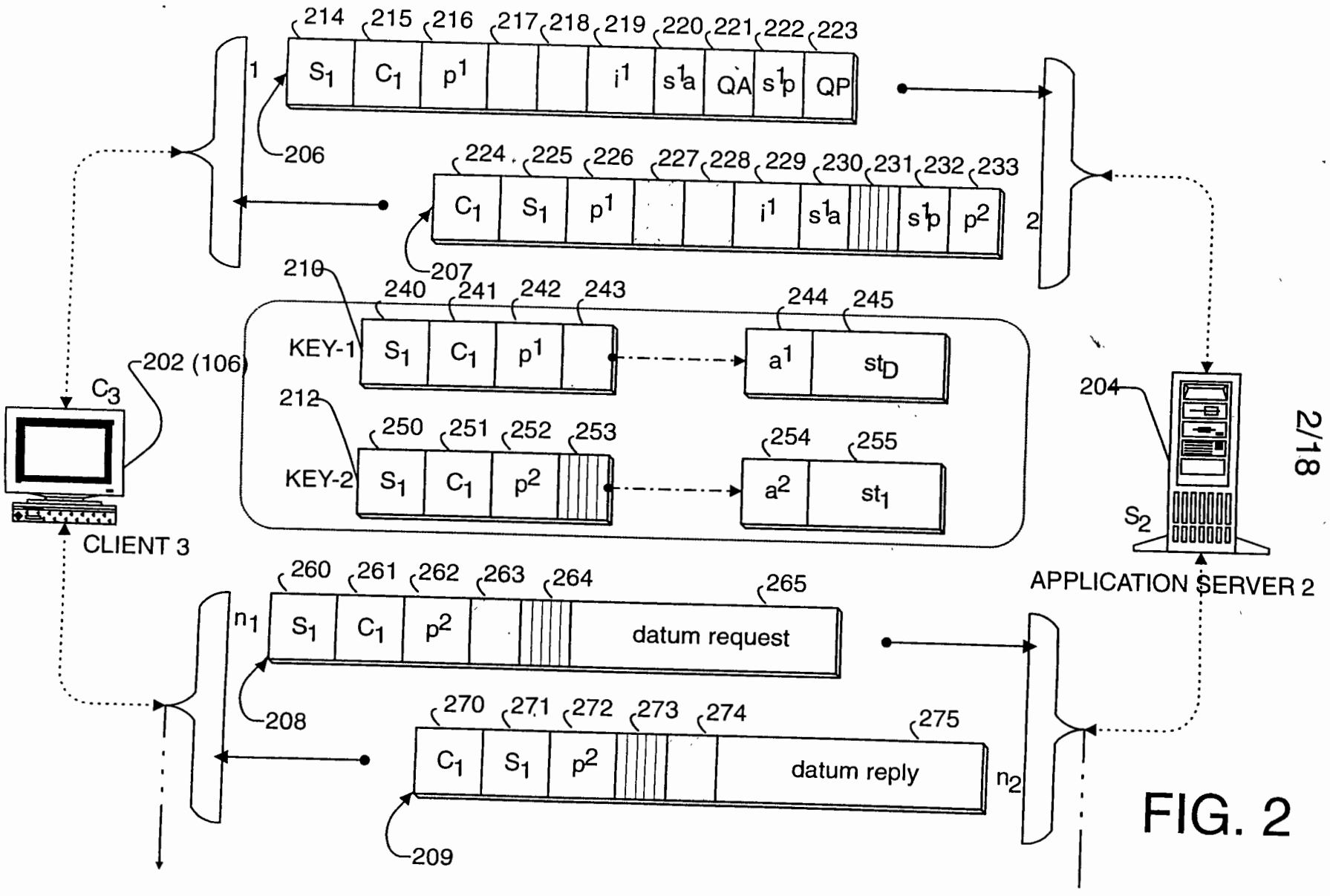


FIG. 2

2/18

NT OF DRAWINGS
S ORIGINALLY FILED 101-1

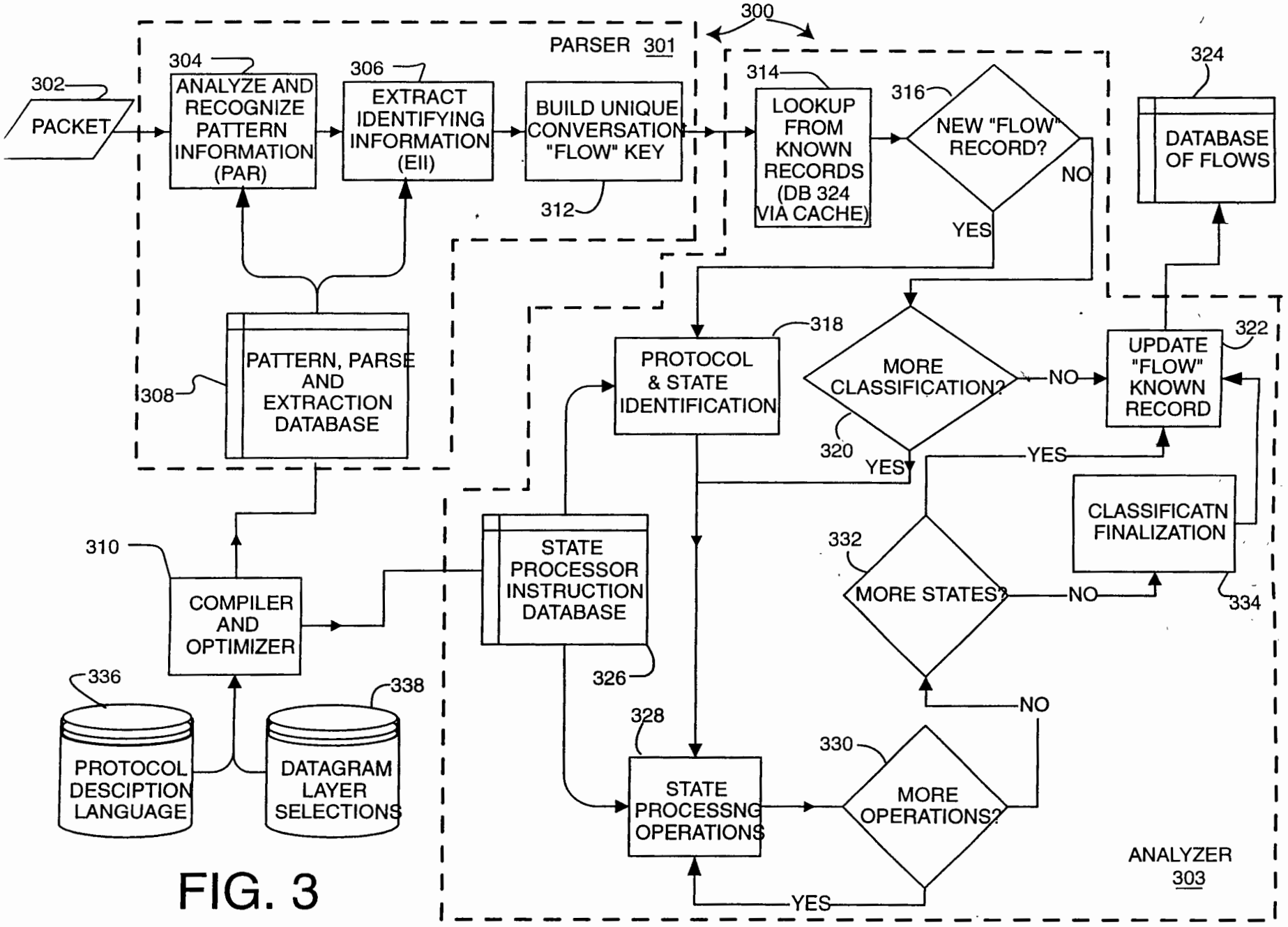


FIG. 3

PRINT OF DRAWINGS
AS ORIGINALLY FILED
101-1

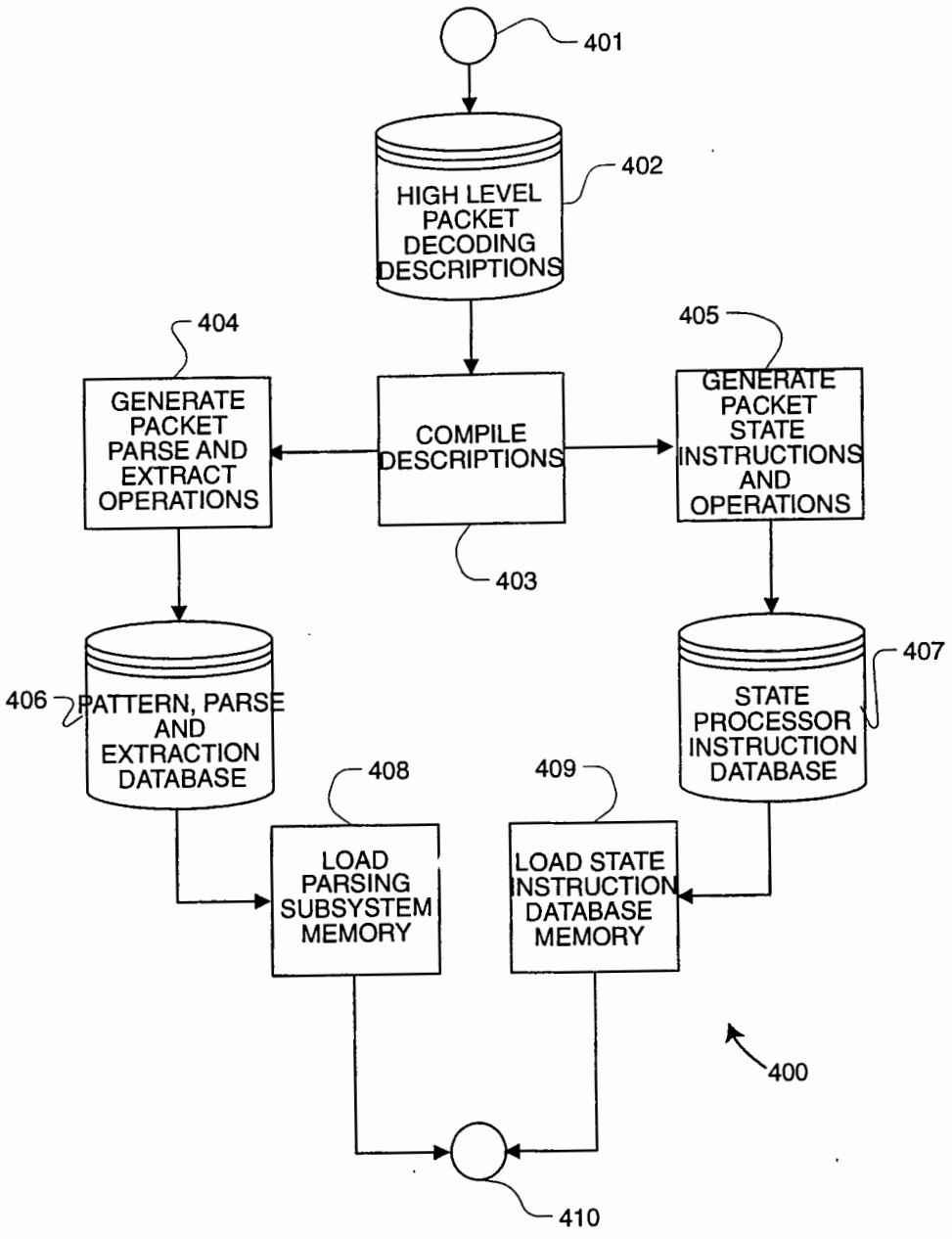


FIG. 4

PRINT OF DRAWINGS -001-1
AS ORIGINALLY FILED

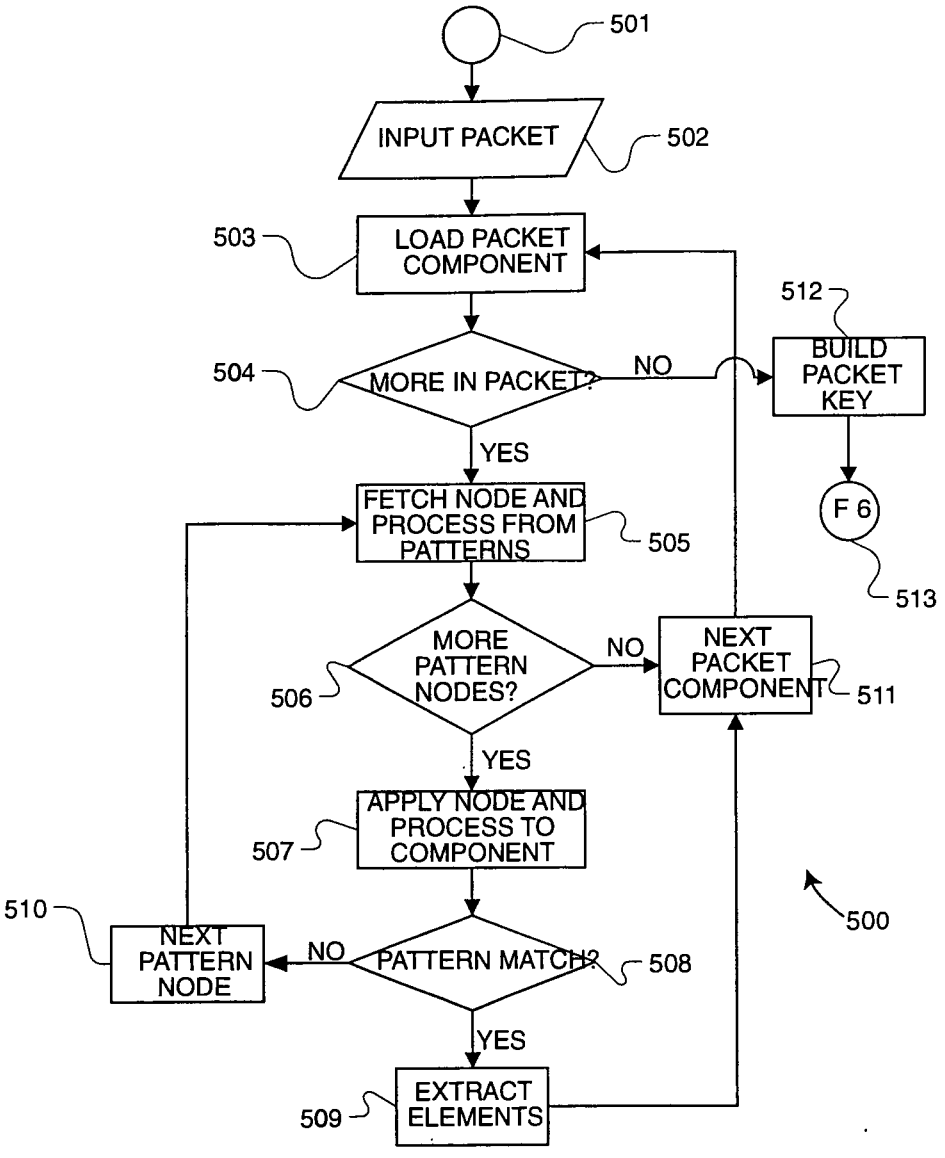


FIG. 5

PRINT OF DRAWINGS
AS ORIGINALLY FILED -001-1

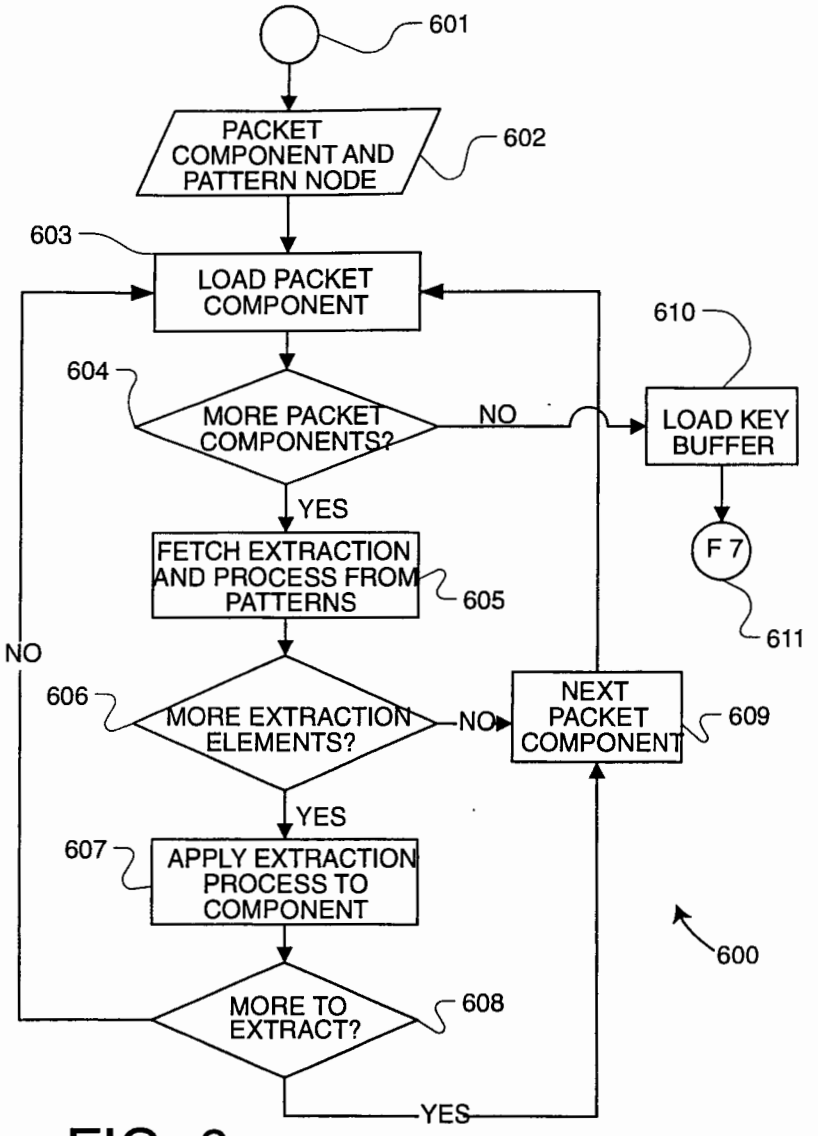


FIG. 6

PRINT OF DRAWINGS
AS ORIGINALLY FILED 001-1

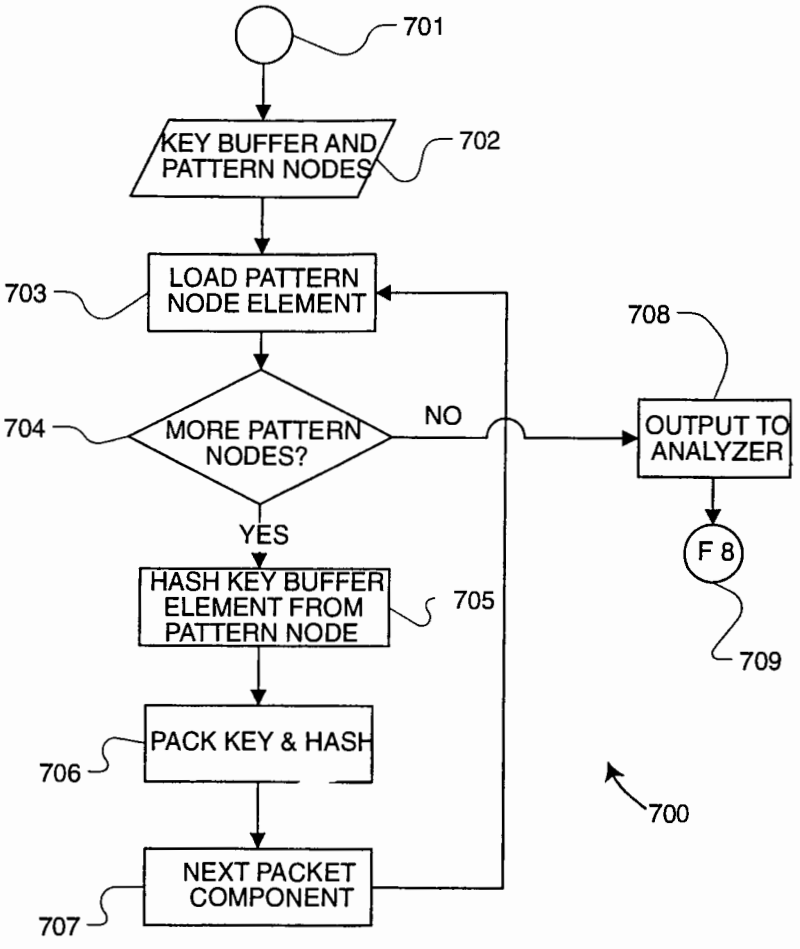


FIG. 7

PRINT OF DRAWINGS
AS ORIGINALLY FILED -001-1

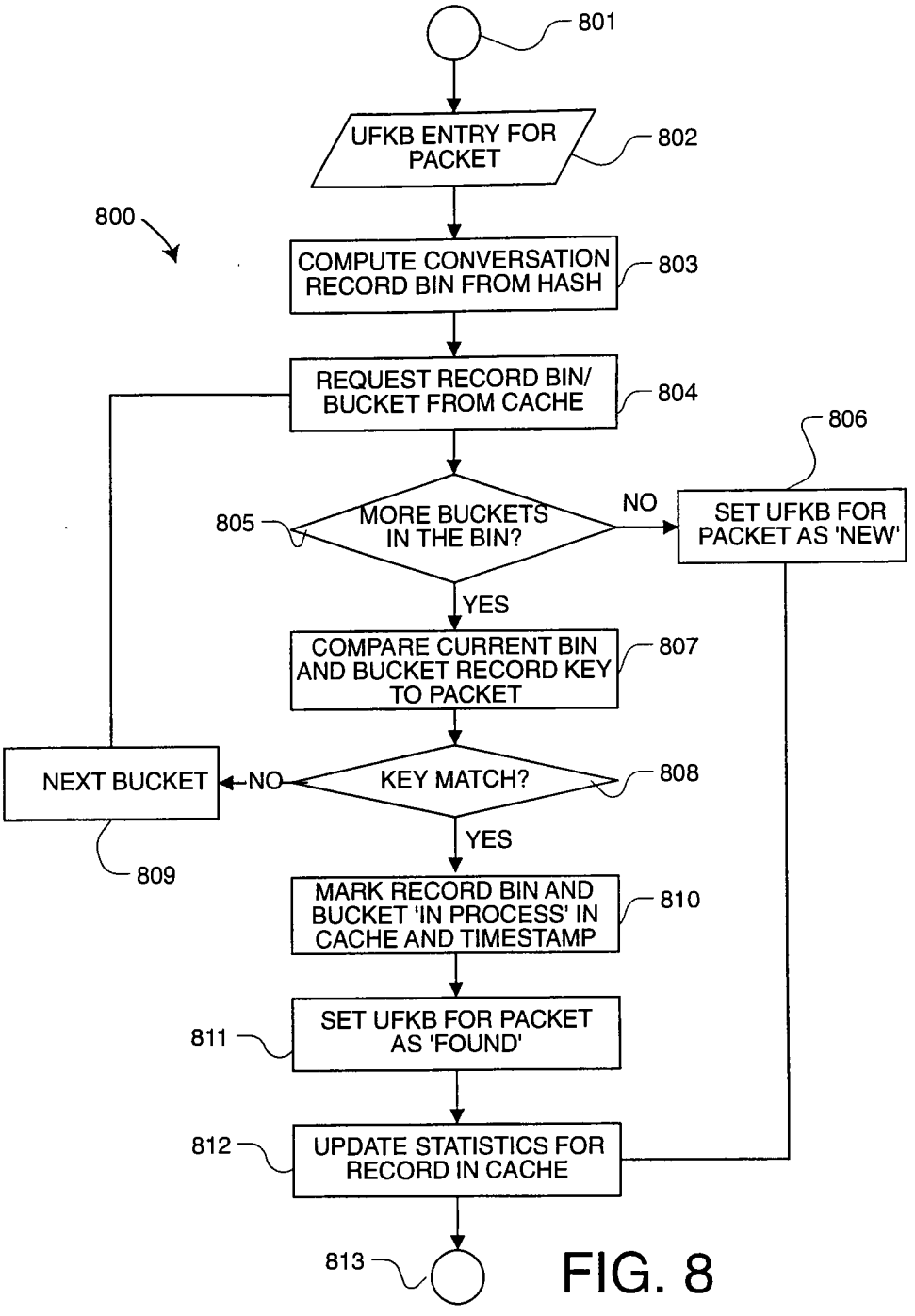


FIG. 8

PRINT OF DRAWINGS
AS ORIGINALLY FILED

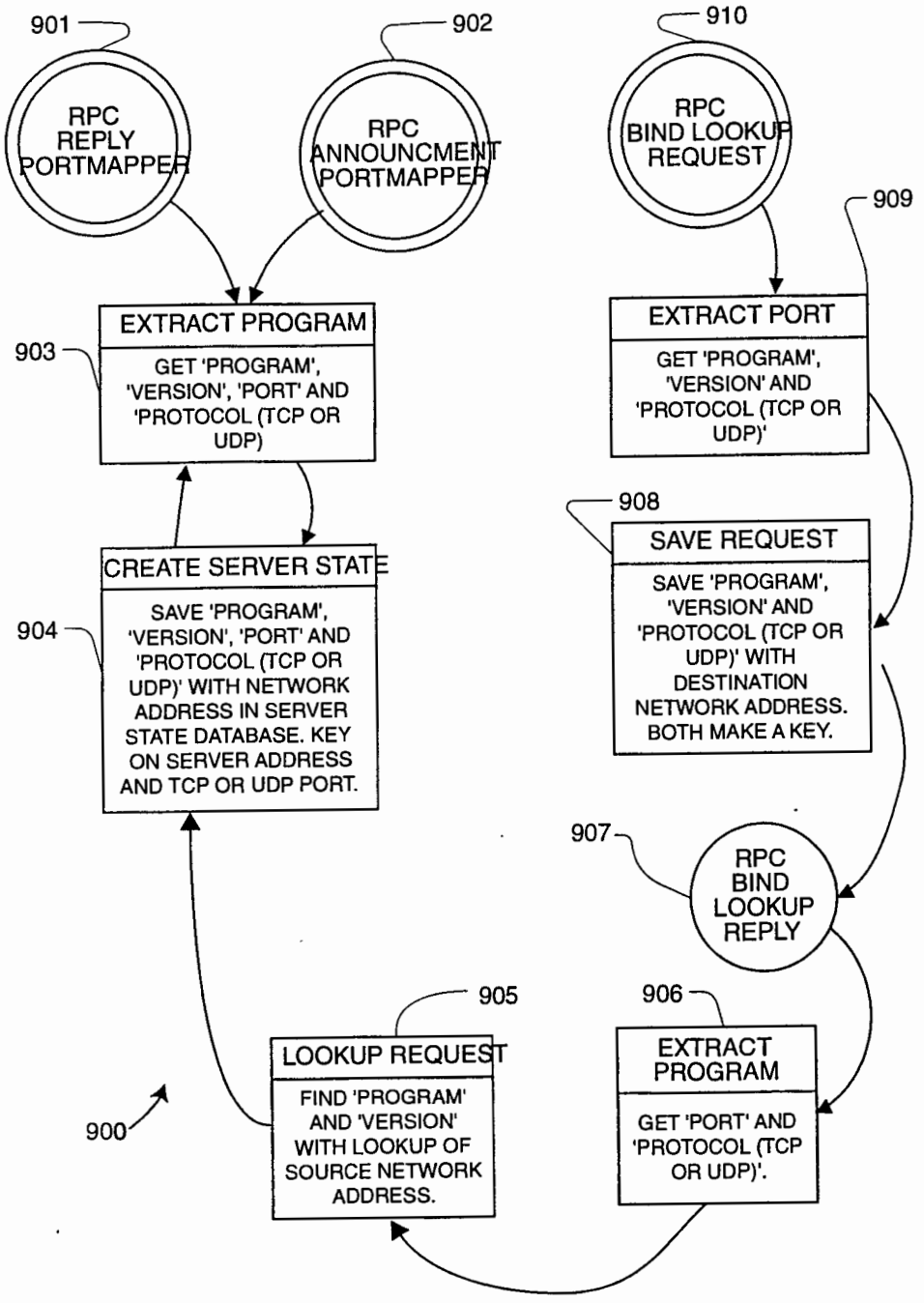


FIG. 9

PRINT OF DRAWINGS
AS ORIGINALLY FILED 01-1

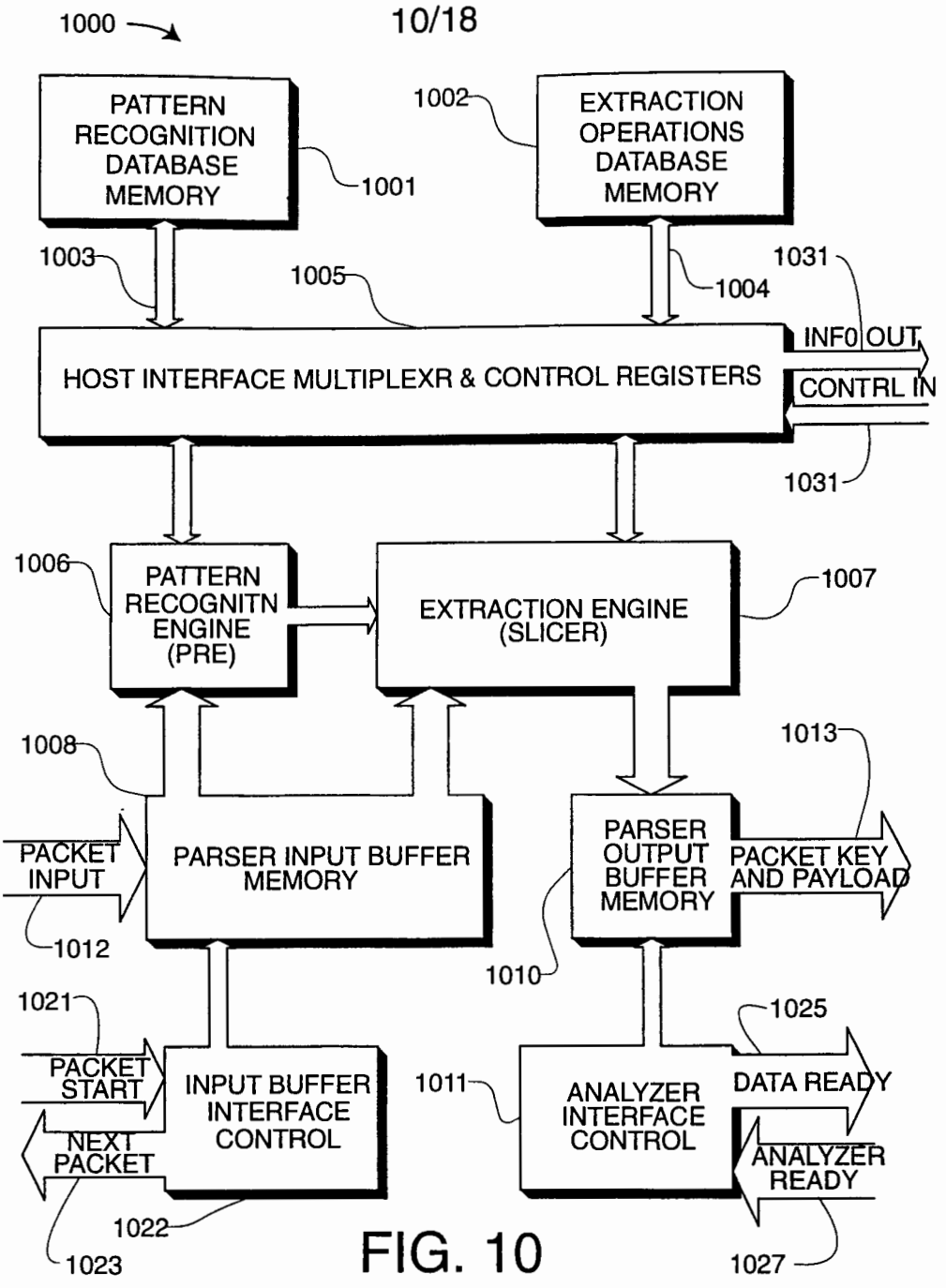


FIG. 10

PRINT OF DRAWINGS
AS ORIGINALLY FILED 001-1

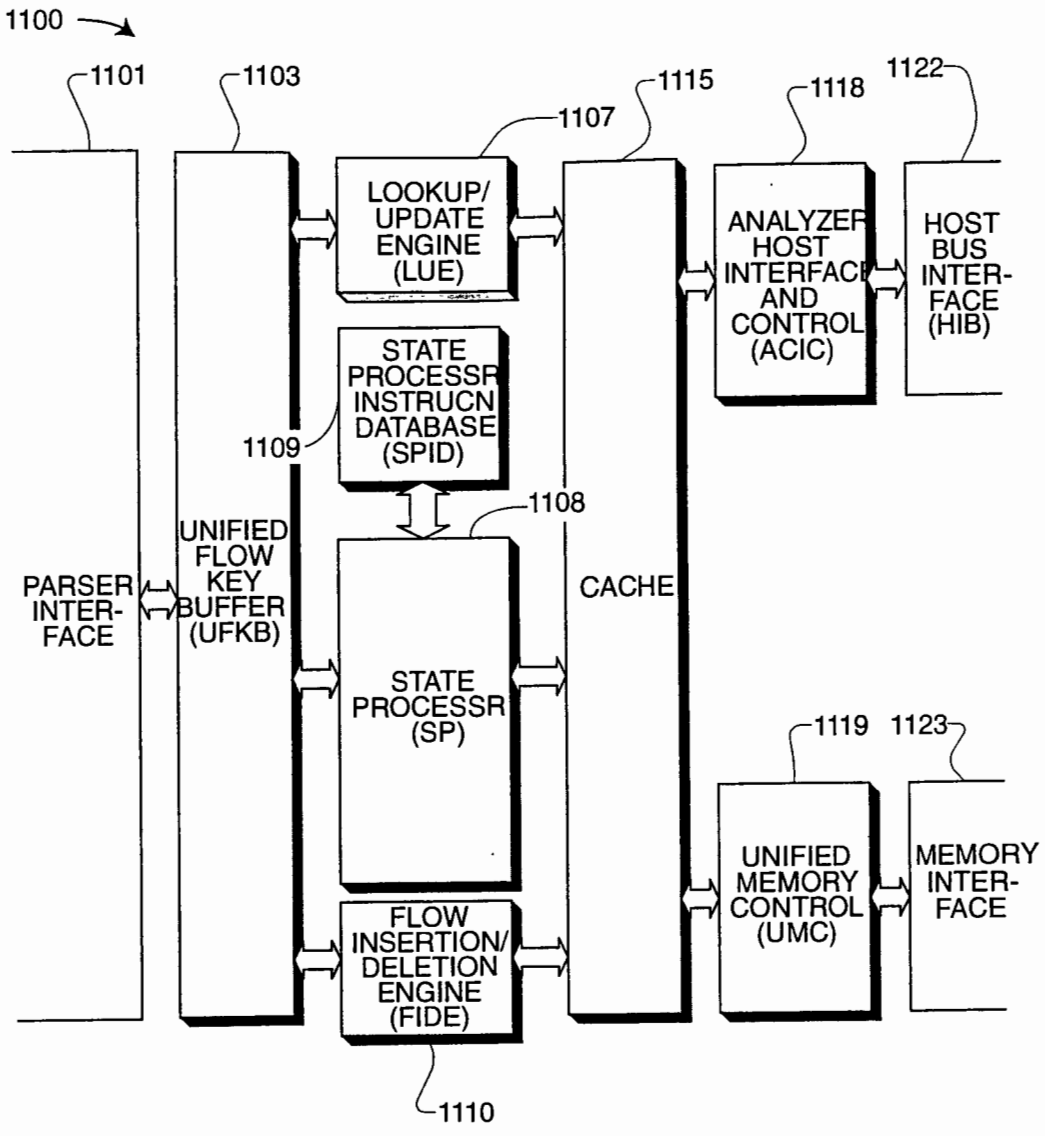


FIG. 11

PRINT OF DRAWINGS 001-1
AS ORIGINALLY FILED

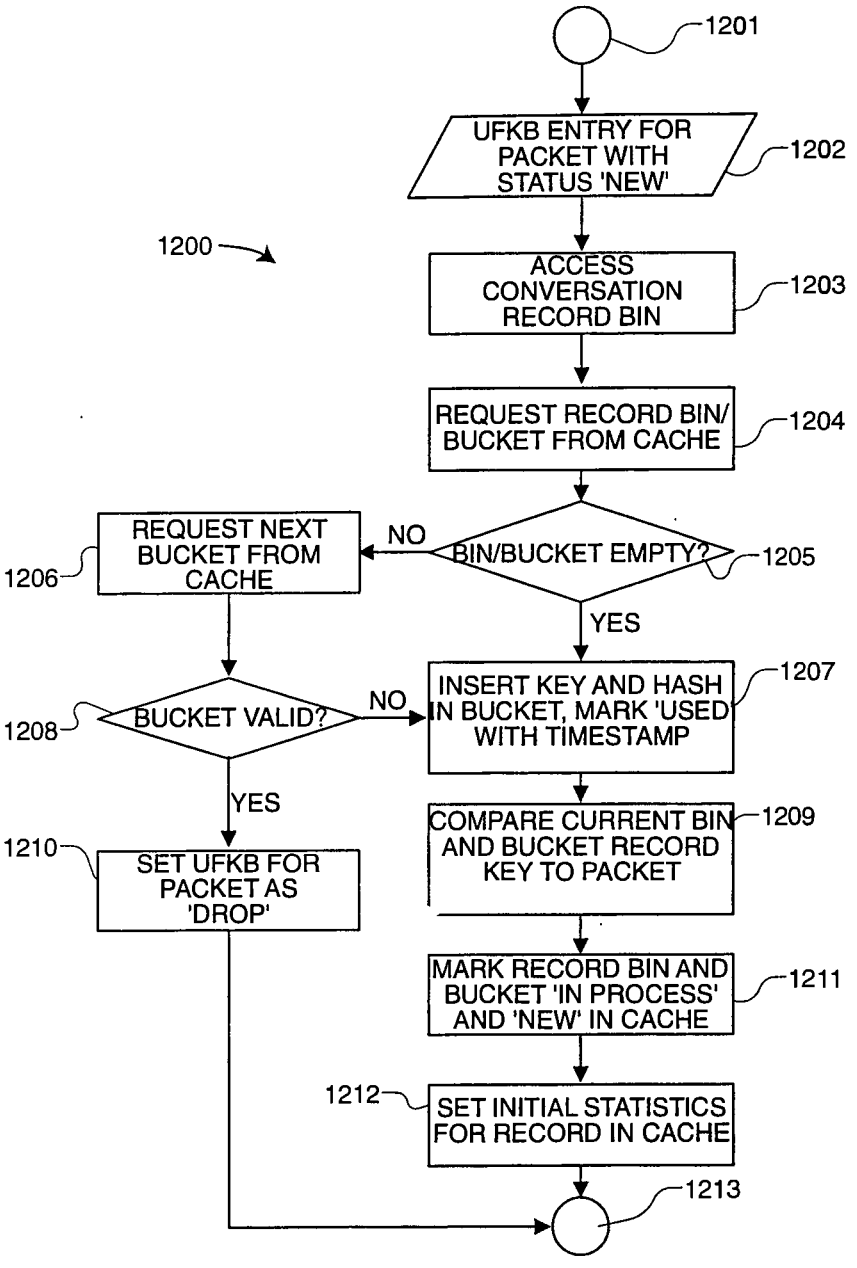


FIG. 12

PRINT OF DRAWINGS
AS ORIGINALLY FILED T-001-1

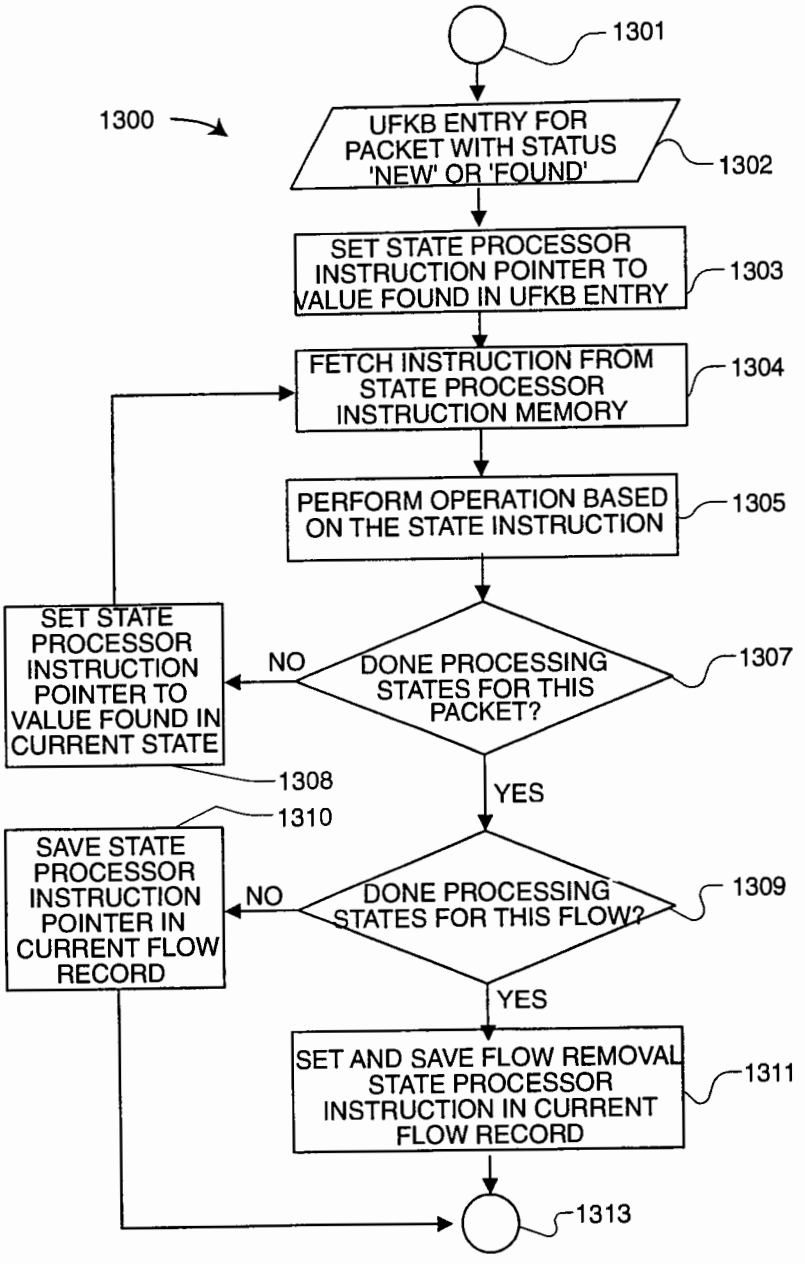


FIG. 13

PRINT OF DRAWINGS
AS ORIGINALLY FILED -001-1

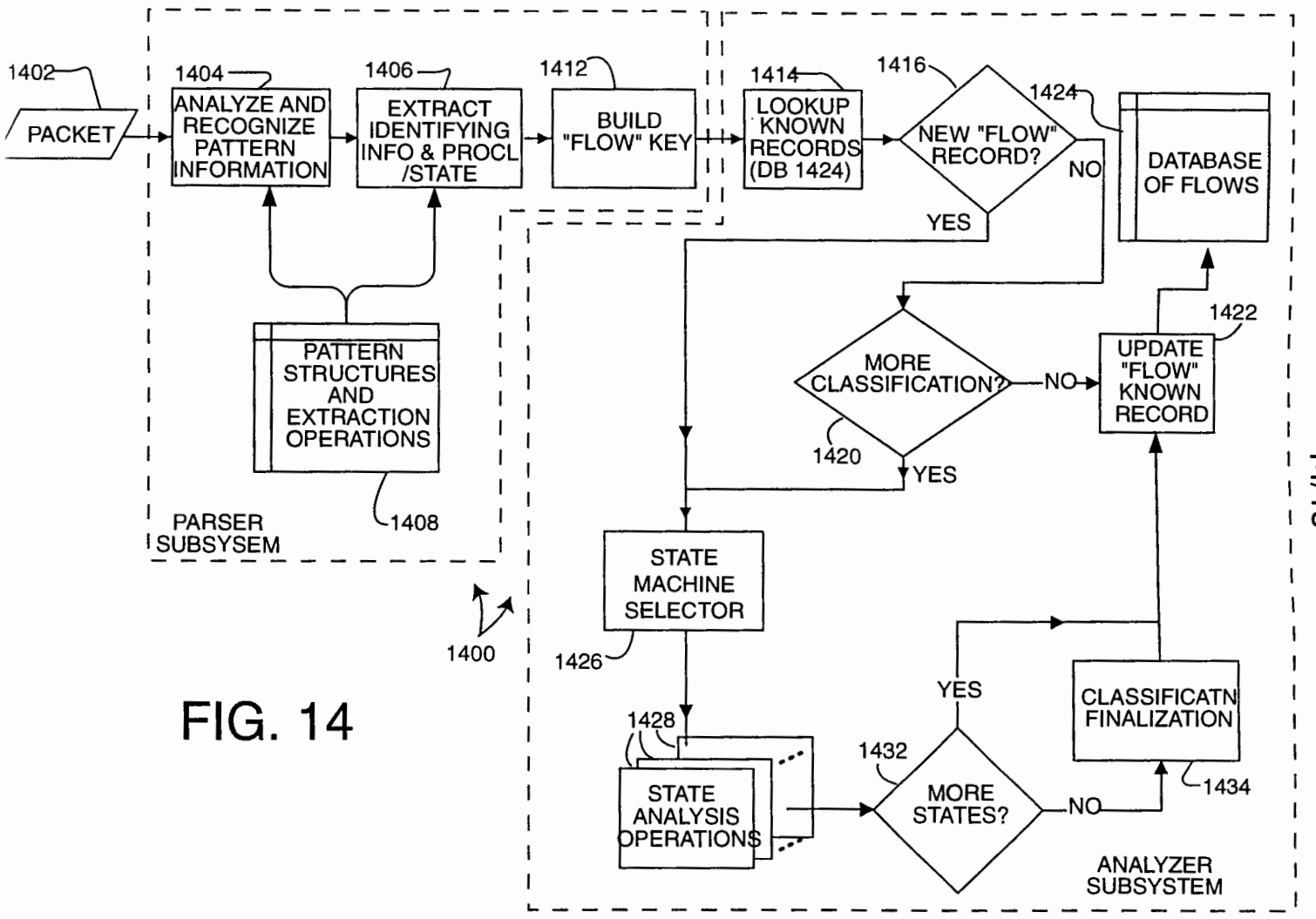


FIG. 14

NOAC Ex. 1015 Page 105

PRINT OF DRAWINGS
AS ORIGINALLY FILED 01-1

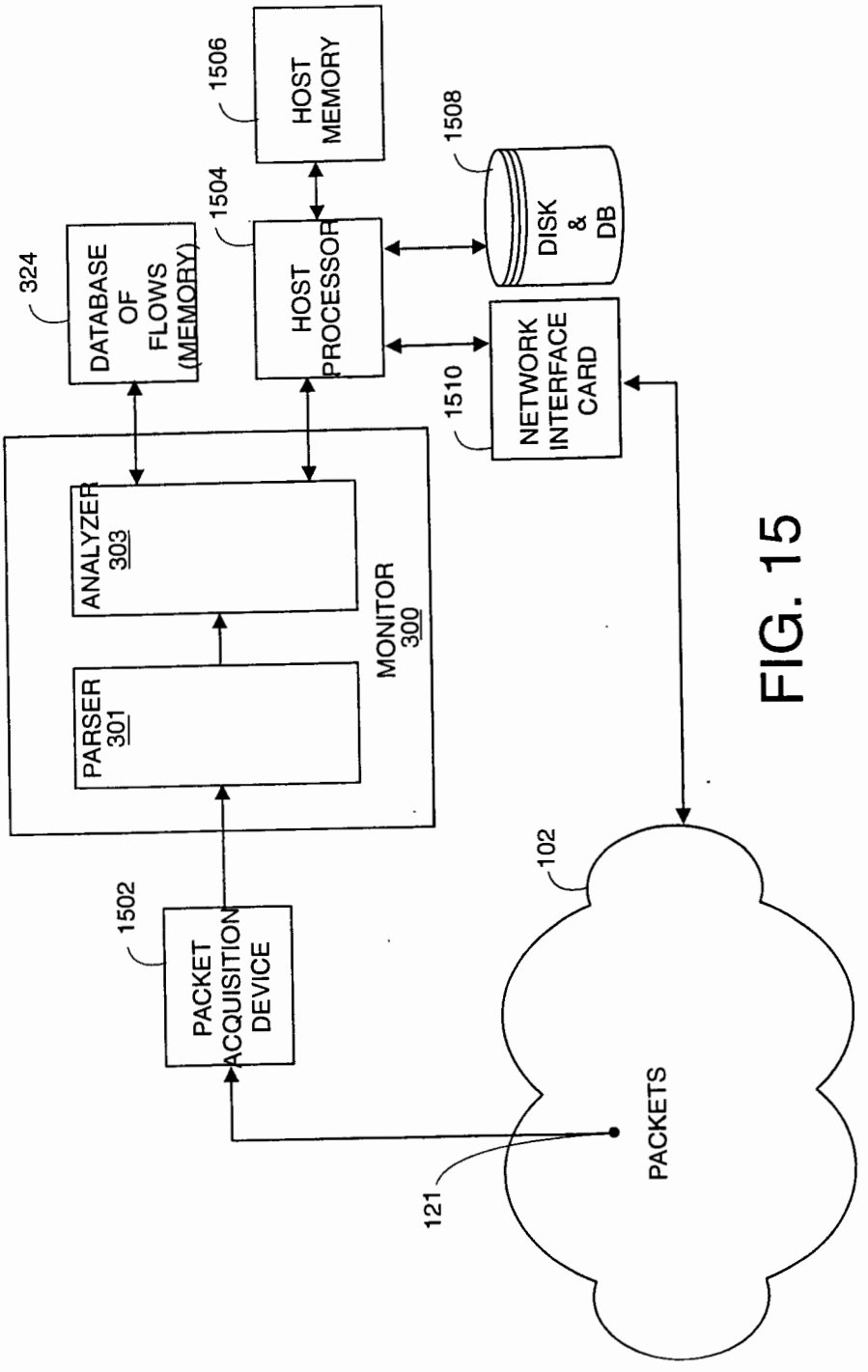


FIG. 15

PRINT OF DRAWINGS -001-1
AS ORIGINALLY FILED

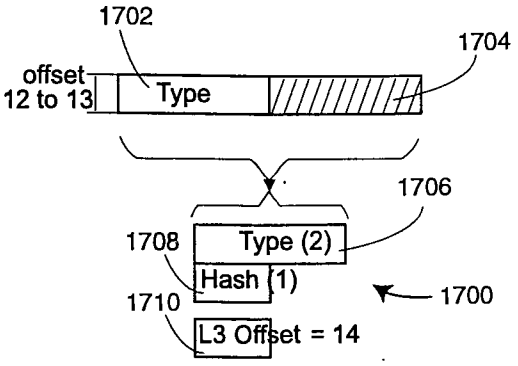


FIG. 17A

1712

IDP = 0x0600*
IP = 0x0800*
CHAOSNET = 0x0804
ARP = 0x0806
VIP = 0x0BAD*
VLOOP = 0x0BAE
VECHO = 0x0BAF
NETBIOS-3COM = 0x3C00 - 0x3C0D#
DEC-MOP = 0x6001
DEC-RC = 0x6002
DEC-DRP = 0x6003*
DEC-LAT = 0x6004
DEC-DIAG = 0x6005
DEC-LAVC = 0x6007
RARP = 0x8035
ATALK = 0x809B*
VLOOP = 0x80C4
VECHO = 0x80C5
SNA-TH = 0x80D5*
ATALKARP = 0x80F3
IPX = 0x8137*
SNMP = 0x814C#
IPv6 = 0x86DD*
LOOPBACK = 0x9000

Apple = 0x080007

* L3 Decoding
# L5 Decoding

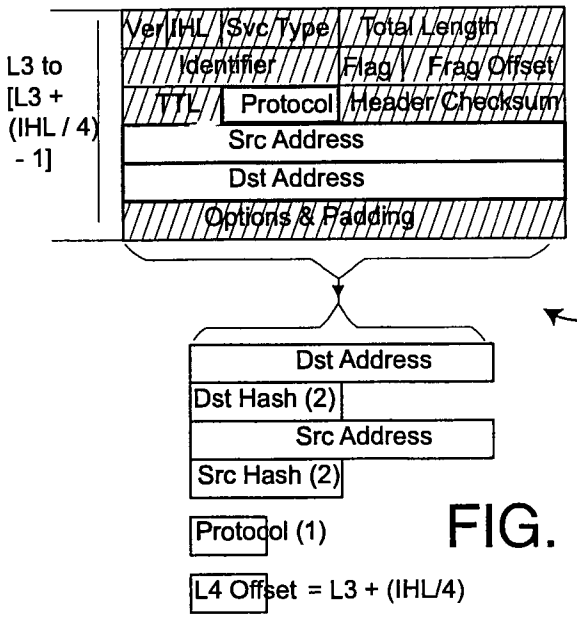


FIG. 17B

1752

ICMP = 1
IGMP = 2
GGP = 3
TCP = 6*
EGP = 8
IGRP = 9
PUP = 12
CHAOS = 16
UDP = 17*
IDP = 22#
ISO-TP4 = 29
DDP = 37#
ISO-IP = 80
VIP = 83#
EIGRP = 88
OSPF = 89

* L4 Decoding
# L3 Re-Decoding

PRINT OF DRAWINGS
AS ORIGINALLY FILED
-001-1

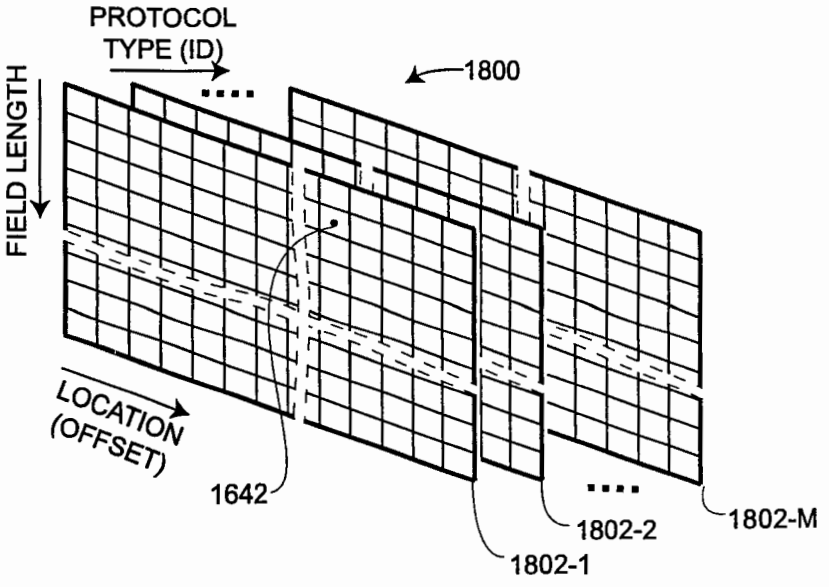


FIG. 18A

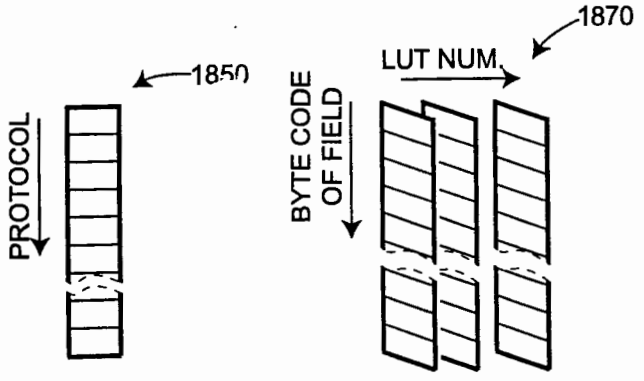


FIG. 18B

This Form is for INTERNAL PTO USE ONLY
It does NOT get mailed to the applicant.

NOTICE OF FILING / CLAIM FEE(S) DUE
(CALCULATION SHEET)

APPLICATION NUMBER: _____

Total Fee Calculation

	Fee Code Sm/Lg	Total # Claims		Number Extra	X	Fee Sm Entity	Fee Lg Entity	= Total
Basic Filing Fee	201/111					____	690	____
Total Claims >20	203/103	59	20 =	39	X	____	7.00	____
Independent Claims >3	202/102	4	3 =	1	X	____	78	____
Multi Dep Claim Present	204/104					____	____	____
Surcharge	205/105					____	130	____
English Translation	139							____
TOTAL FEE CALCULATION								____

Fees due upon filing the application

Total Filing Fees Due = $ 1600

Less Filing Fees Submitted - $ _____

BALANCE DUE = $ 1600

WP

Office of Initial Patent Examination

Figure 7

FORM OIPE-RAM-01 (Rev. 12/97)

FORMALITIES LETTER

OC000000005353894

UNITED STATES DEPARTMENT OF COMMERCE
Patent and Trademark Office
Address COMMISSIONER OF PATENT AND TRADEMARKS
Washington, D C 20231

APPLICATION NUMBER	FILING/RECEIPT DATE	FIRST NAMED APPLICANT	ATTORNEY DOCKET NUMBER
09/608,237	06/30/2000	Russell S. Dietz	APPT-001-1

Dov Rosenfeld
Suite 2
5507 College Avenue
Oakland, CA 94618

Date Mailed: 08/25/2000

NOTICE TO FILE MISSING PARTS OF NONPROVISIONAL APPLICATION

FILED UNDER 37 CFR 1.53(b)

Filing Date Granted

An application number and filing date have been accorded to this application. The item(s) indicated below, however, are missing. Applicant is given TWO MONTHS from the date of this Notice within which to file all required items and pay any fees required below to avoid abandonment. Extensions of time may be obtained by filing a petition accompanied by the extension fee under the provisions of 37 CFR 1.136(a).

- The statutory basic filing fee is missing.
  Applicant must submit **$ 690** *to complete the basic filing fee and/or file a small entity statement claiming such status (37 CFR 1.27).*
- Total additional claim fee(s) for this application is $780.
  - **$702** for **39** total claims over 20.
  - **$78** for **1** independent claims over 3 .
- The oath or declaration is missing.
  A properly signed oath or declaration in compliance with 37 CFR 1.63, identifying the application by the above Application Number and Filing Date, is required.
- To avoid abandonment, a late filing fee or oath or declaration surcharge as set forth in 37 CFR 1.16(e) of $130 for a non-small entity, must be submitted with the missing items identified in this letter.
- **The balance due by applicant is $ 1600.**

A copy of this notice <u>MUST</u> be returned with the reply.

Customer Service Center
Initial Patent Examination Division (703) 308-1202

PART 3 - OFFICE COPY

SECTION #3

Our Ref./Docket No: AI . T-001-1 Patent

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant(s): Dietz, *et al.* Application No.: 09/608237 Filed: June 30, 2000 Title: METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK	Group Art Unit: 2755 Examiner: (Unassigned)

OIPE JC66 NOV 06 2000 PATENT & TRADEMARK OFFICE

RESPONSE TO NOTICE TO FILE MISSING PARTS OF APPLICATION

Assistant Commissioner for Patents
Washington, D.C. 20231
Attn: Box Missing Parts

Dear Assistant Commissioner:

This is in response to a Notice to File Missing Parts of Application under 37 CFR 1.53(f). Enclosed is a copy of said Notice and the following documents and fees to complete the filing requirements of the above-identified application:

_X_ Executed Declaration and Power of Attorney. The above-identified application is the same application which the inventor executed by signing the enclosed declaration;

_X_ Executed Assignment with assignment cover sheet.

_X_ A credit card payment form in the amount of $__1772.00__ is attached, being for:

_X_	Statutory basic filing fee:	$ 710
_X_	Additional claim fee of	$ 782
_X_	Assignment recordation fee of	$ 40
_X_	Extension Fee First Month of	$ 110
_X_	Missing Parts Surcharge	$ 130

____ Applicant(s) believe(s) that no Extension of Time is required. However, this conditional petition is being made to provide for the possibility that applicant has inadvertently overlooked the need for a petition for an extension of time.

_X_ Applicant(s) hereby petition(s) for an Extension of Time under 37 CFR 1.136(a) of:

_X_ one months ($110) ____ two months ($380)

____ two months ($870) ____ four months ($1360)

If an additional extension of time is required, please consider this as a petition therefor.

Certificate of Mailing under 37 CFR 1.8

I hereby certify that this response is being deposited with the United States Postal Service as first class mail in an envelope addressed to the Assistant Commissioner for Patents, Washington, D.C. 20231 on.

Date: Nov 1, 2000 Signed: [signature]

Name: Dov Rosenfeld, Reg. No. 38687

__X__ The Commissioner is hereby authorized to charge payment of any missing fees associated with this communication or credit any overpayment to Deposit Account No. 50-0292

(A DUPLICATE OF THIS TRANSMITTAL IS ATTACHED):

Respectfully Submitted,

Nov 1, 2000
Date

Dov Rosenfeld, Reg. No. 38687

Address for correspondence:
Dov Rosenfeld
5507 College Avenue, Suite 2
Oakland, CA 94618
Tel. (510) 547-3378; Fax: (510) 653-7992

NOV 0 6 2000
PATENT & TRADEMARK OFFICE

#3

PATENT APPLICATION

DECLARATION AND POWER OF ATTORNEY FOR PATENT APPLICATION — **ATTORNEY DOCKET NO. APPT-001-1**

As a below named inventor, I hereby declare that:

My residence/post office address and citizenship are as stated below next to my name;

I believe I am the original, first and sole inventor (if only one name is listed below) or an original, first and joint inventor (if plural names are listed below) of the subject matter which is claimed and for which a patent is sought on the invention entitled:

METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK

the specification of which is attached hereto unless the following box is checked:

(X) was filed on June 30, 2000 as US Application Serial No. 09/608237 or PCT International Application Number ____ and was amended on _____ (if applicable).

I hereby state that I have reviewed and understood the contents of the above-identified specification, including the claims, as amended by any amendment(s) referred to above. I acknowledge the duty to disclose all information which is material to patentability as defined in 37 CFR 1.56.

Foreign Application(s) and/or Claim of Foreign Priority

I hereby claim foreign priority benefits under Title 35, United States Code Section 119 of any foreign application(s) for patent or inventor(s) certificate listed below and have also identified below any foreign application for patent or inventor(s) certificate having a filing date before that of the application on which priority is claimed:

COUNTRY	APPLICATION NUMBER	DATE FILED	PRIORITY CLAIMED UNDER 35
			YES: ____ NO: ____
			YES: ____ NO: ____

Provisional Application

I hereby claim the benefit under Title 35, United States Code Section 119(e) of any United States provisional application(s) listed below:

APPLICATION SERIAL NUMBER	FILING DATE
60/141,903	June 30, 1999

U.S. Priority Claim

I hereby claim the benefit under Title 35, United States Code, Section 120 of any United States application(s) listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States application in the manner provided by the first paragraph of Title 35, United States Code Section 112, I acknowledge the duty to disclose material information as defined in Title 37, Code of Federal Regulations, Section 1.56(a) which occurred between the filing date of the prior application and the national or PCT international filing date of this application:

APPLICATION SERIAL NUMBER	FILING DATE	STATUS(patented/pending/abandoned)

POWER OF ATTORNEY:

As a named inventor, I hereby appoint the following attorney(s) and/or agent(s) listed below to prosecute this application and transact all business in the Patent and Trademark Office connected therewith:

Dov Rosenfeld, Reg. No. 38,687

Send Correspondence to:	**Direct Telephone Calls To:**
Dov Rosenfeld	Dov Rosenfeld, Reg. No. 38,687
5507 College Avenue, Suite 2	Tel: (510) 547-3378
Oakland, CA 94618	

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Name of First Inventor: Russell S. Dietz **Citizenship:** USA

Residence: 6146 Ostenberg Drive, San Jose, CA 95120-2736

Post Office Address: Same

First Inventor's Signature **Date** 10/3/00

ADDITIONAL INVENTOR SIGNATURES:

Name of Second Inventor: Joseph R. Maixner **Citizenship:** USA

Residence: 121 Driftwood Court, Aptos, CA 95003

Post Office Address: Same

_____ _____

Inventor's Signature **Date**

Name of Third Inventor: Andrew A. Koppenhaver **Citizenship:** USA

Residence: 10400 Kenmore Drive, Fairfax, VA 22030

Post Office Address: Same

_____ _____

Inventor's Signature **Date**

Name of Fourth Inventor: William H. Bares **Citizenship:** USA

Residence: 9005 Glenalden Drive, Germantown, TN 38139

Post Office Address: Same

_____ _____

Inventor's Signature **Date**

Name of Fifth Inventor: Haig A. Sarkissian **Citizenship:** USA

Residence: 8701 Mountain Top, San Antonio, Texas 78255

Post Office Address: Same

_____ _____

Inventor's Signature **Date**

Name of Sixth Inventor: James F. Torgerson **Citizenship:** USA

Residence: 227 157th Ave., NW, Andover, MN 55304

Post Office Address: Same

_____ _____

Inventor's Signature **Date**

PATENT APPLICATION

DECLARATION AND POWER OF ATTORNEY FOR PATENT APPLICATION | **ATTORNEY DOCKET NO. APPT-001-1**

NOV 0 6 2000

As a below named inventor, I hereby declare that:

My residence/post office address and citizenship are as stated below next to my name;

I believe I am the original, first and sole inventor (if only one name is listed below) or an original, first and joint inventor (if plural names are listed below) of the subject matter which is claimed and for which a patent is sought on the invention entitled:

METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK

the specification of which is attached hereto unless the following box is checked:

(X) was filed on June 30, 2000 as US Application Serial No. 09/608237 or PCT International Application Number _____ and was amended on _____ (if applicable).

I hereby state that I have reviewed and understood the contents of the above-identified specification, including the claims, as amended by any amendment(s) referred to above. I acknowledge the duty to disclose all information which is material to patentability as defined in 37 CFR 1.56.

Foreign Application(s) and/or Claim of Foreign Priority

I hereby claim foreign priority benefits under Title 35, United States Code Section 119 of any foreign application(s) for patent or inventor(s) certificate listed below and have also identified below any foreign application for patent or inventor(s) certificate having a filing date before that of the application on which priority is claimed:

COUNTRY	APPLICATION NUMBER	DATE FILED	PRIORITY CLAIMED UNDER 35
			YES: ____ NO: ____
			YES: ____ NO: ____

Provisional Application

I hereby claim the benefit under Title 35, United States Code Section 119(e) of any United States provisional application(s) listed below:

APPLICATION SERIAL NUMBER	FILING DATE
60/141,903	June 30, 1999

U.S. Priority Claim

I hereby claim the benefit under Title 35, United States Code, Section 120 of any United States application(s) listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States application in the manner provided by the first paragraph of Title 35, United States Code Section 112, I acknowledge the duty to disclose material information as defined in Title 37, Code of Federal Regulations, Section 1.56(a) which occurred between the filing date of the prior application and the national or PCT international filing date of this application:

APPLICATION SERIAL NUMBER	FILING DATE	STATUS(patented/pending/abandoned)

POWER OF ATTORNEY:

As a named inventor, I hereby appoint the following attorney(s) and/or agent(s) listed below to prosecute this application and transact all business in the Patent and Trademark Office connected therewith:

Dov Rosenfeld, Reg. No. 38,687

Send Correspondence to:	**Direct Telephone Calls To:**
Dov Rosenfeld 5507 College Avenue, Suite 2 Oakland, CA 94618	Dov Rosenfeld, Reg. No. 38,687 Tel: (510) 547-3378

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Name of First Inventor: Russell S. Dietz **Citizenship:** USA

Residence: 6146 Ostenberg Drive, San Jose, CA 95120-2736

Post Office Address: Same

_____ _____

First Inventor's Signature **Date**

OIPE JC66 NOV 06 2000 PATENT & TRADEMARK OFFICE

PATENT APPLICATION

DECLARATION AND POWER OF ATTORNEY FOR PATENT APPLICATION | **ATTORNEY DOCKET NO. APPT-001-1**

As a below named inventor, I hereby declare that:

My residence/post office address and citizenship are as stated below next to my name;

I believe I am the original, first and sole inventor (if only one name is listed below) or an original, first and joint inventor (if plural names are listed below) of the subject matter which is claimed and for which a patent is sought on the invention entitled:

METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK

the specification of which is attached hereto unless the following box is checked:

(X) was filed on June 30, 2000 as US Application Serial No. 09/608237 or PCT International Application Number _____ and was amended on _____ (if applicable).

I hereby state that I have reviewed and understood the contents of the above-identified specification, including the claims, as amended by any amendment(s) referred to above. I acknowledge the duty to disclose all information which is material to patentability as defined in 37 CFR 1.56.

Foreign Application(s) and/or Claim of Foreign Priority

I hereby claim foreign priority benefits under Title 35, United States Code Section 119 of any foreign application(s) for patent or inventor(s) certificate listed below and have also identified below any foreign application for patent or inventor(s) certificate having a filing date before that of the application on which priority is claimed:

COUNTRY	APPLICATION NUMBER	DATE FILED	PRIORITY CLAIMED UNDER 35
			YES: NO:
			YES: NO:

Provisional Application

I hereby claim the benefit under Title 35, United States Code Section 119(e) of any United States provisional application(s) listed below:

APPLICATION SERIAL NUMBER	FILING DATE
60/141,903	June 30, 1999

U.S. Priority Claim

I hereby claim the benefit under Title 35, United States Code, Section 120 of any United States application(s) listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States application in the manner provided by the first paragraph of Title 35, United States Code Section 112, I acknowledge the duty to disclose material information as defined in Title 37, Code of Federal Regulations, Section 1.56(a) which occurred between the filing date of the prior application and the national or PCT international filing date of this application:

APPLICATION SERIAL NUMBER	FILING DATE	STATUS(patented/pending/abandoned)

POWER OF ATTORNEY:

As a named inventor, I hereby appoint the following attorney(s) and/or agent(s) listed below to prosecute this application and transact all business in the Patent and Trademark Office connected therewith:

Dov Rosenfeld, Reg. No. 38,687

Send Correspondence to:	**Direct Telephone Calls To:**
Dov Rosenfeld	Dov Rosenfeld, Reg. No. 38,687
5507 College Avenue, Suite 2	Tel: (510) 547-3378
Oakland, CA 94618	

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Name of First Inventor: Russell S. Dietz **Citizenship:** USA

Residence: 6146 Ostenberg Drive, San Jose, CA 95120-2736

Post Office Address: Same

_____ _____

First Inventor's Signature **Date**

PATENT APPLICATION

DECLARATION AND POWER OF ATTORNEY FOR PATENT APPLICATION

ATTORNEY DOCKET NO. APPT-001-1

As a below named inventor, I hereby declare that:

My residence/post office address and citizenship are as stated below next to my name;

I believe I am the original, first and sole inventor (if only one name is listed below) or an original, first and joint inventor (if plural names are listed below) of the subject matter which is claimed and for which a patent is sought on the invention entitled:

METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK

the specification of which is attached hereto unless the following box is checked:

(X) was filed on June 30, 2000 as US Application Serial No. 09/608237 or PCT International Application Number _____ and was amended on _____ (if applicable).

I hereby state that I have reviewed and understood the contents of the above-identified specification, including the claims, as amended by any amendment(s) referred to above. I acknowledge the duty to disclose all information which is material to patentability as defined in 37 CFR 1.56.

Foreign Application(s) and/or Claim of Foreign Priority

I hereby claim foreign priority benefits under Title 35, United States Code Section 119 of any foreign application(s) for patent or inventor(s) certificate listed below and have also identified below any foreign application for patent or inventor(s) certificate having a filing date before that of the application on which priority is claimed:

COUNTRY	APPLICATION NUMBER	DATE FILED	PRIORITY CLAIMED UNDER 35
			YES: NO:
			YES: NO:

Provisional Application

I hereby claim the benefit under Title 35, United States Code Section 119(e) of any United States provisional application(s) listed below:

APPLICATION SERIAL NUMBER	FILING DATE
60/141,903	June 30, 1999

U.S. Priority Claim

I hereby claim the benefit under Title 35, United States Code, Section 120 of any United States application(s) listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States application in the manner provided by the first paragraph of Title 35, United States Code Section 112, I acknowledge the duty to disclose material information as defined in Title 37, Code of Federal Regulations, Section 1.56(a) which occurred between the filing date of the prior application and the national or PCT international filing date of this application:

APPLICATION SERIAL NUMBER	FILING DATE	STATUS(patented/pending/abandoned)

POWER OF ATTORNEY:

As a named inventor, I hereby appoint the following attorney(s) and/or agent(s) listed below to prosecute this application and transact all business in the Patent and Trademark Office connected therewith:

Dov Rosenfeld, Reg. No. 38,687

Send Correspondence to:	**Direct Telephone Calls To:**
Dov Rosenfeld 5507 College Avenue, Suite 2 Oakland, CA 94618	Dov Rosenfeld, Reg. No. 38,687 Tel: (510) 547-3378

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Name of First Inventor: Russell S. Dietz **Citizenship:** USA

Residence: 6146 Ostenberg Drive, San Jose, CA 95120-2736

Post Office Address: Same

_____ _____

First Inventor's Signature **Date**

ADDITIONAL INVENTOR SIGNATURES:

Name of Second Inventor: Joseph R. Maixner **Citizenship:** USA

Residence: 121 Driftwood Court, Aptos, CA 95003

Post Office Address: Same

Joseph R. Maixner 10/23/2000

Inventor's Signature **Date**

Name of Third Inventor: Andrew A. Koppenhaver **Citizenship:** USA

Residence: 10400 Kenmore Drive, Fairfax, VA 22030

Post Office Address: Same

Inventor's Signature **Date**

Name of Fourth Inventor: William H. Bares **Citizenship:** USA

Residence: 9005 Glenalden Drive, Germantown, TN 38139

Post Office Address: Same

Inventor's Signature **Date**

Name of Fifth Inventor: Haig A. Sarkissian **Citizenship:** USA

Residence: 8701 Mountain Top, San Antonio, Texas 78255

Post Office Address: Same

Inventor's Signature **Date**

Name of Sixth Inventor: James F. Torgerson **Citizenship:** USA

Residence: 227 157th Ave., NW, Andover, MN 55304

Post Office Address: Same

Inventor's Signature **Date**

ADDITIONAL INVENTOR SIGNATURES:

Name of Second Inventor: Joseph R. Maixner **Citizenship:** USA

Residence: 121 Driftwood Court, Aptos, CA 95003

Post Office Address: Same

_____ _____

Inventor's Signature **Date**

Name of Third Inventor: Andrew A. Koppenhaver **Citizenship:** USA

Residence: 9325 W. Hinsdale Place, Littleton, CO 80128

Post Office Address: Same

_____ 10/10/2000

Inventor's Signature **Date**

Name of Fourth Inventor: William H. Bares **Citizenship:** USA

Residence: 9005 Glenalden Drive, Germantown, TN 38139

Post Office Address: Same

_____ _____

Inventor's Signature **Date**

Name of Fifth Inventor: Haig A. Sarkissian **Citizenship:** USA

Residence: 8701 Mountain Top, San Antonio, Texas 78255

Post Office Address: Same

_____ _____

Inventor's Signature **Date**

Name of Sixth Inventor: James F. Torgerson **Citizenship:** USA

Residence: 227 157th Ave., NW, Andover, MN 55304

Post Office Address: Same

_____ _____

Inventor's Signature **Date**

NOV 06 2000
PATENT & TRADEMARK OFFICE

PATENT APPLICATION

DECLARATION AND POWER OF ATTORNEY FOR PATENT APPLICATION | **ATTORNEY DOCKET NO. APPT-001-1**

As a below named inventor, I hereby declare that:

My residence/post office address and citizenship are as stated below next to my name;

I believe I am the original, first and sole inventor (if only one name is listed below) or an original, first and joint inventor (if plural names are listed below) of the subject matter which is claimed and for which a patent is sought on the invention entitled:

METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK

the specification of which is attached hereto unless the following box is checked:

(X) was filed on June 30, 2000 as US Application Serial No. 09/608237 or PCT International Application Number ____ and was amended on _____ (if applicable).

I hereby state that I have reviewed and understood the contents of the above-identified specification, including the claims, as amended by any amendment(s) referred to above. I acknowledge the duty to disclose all information which is material to patentability as defined in 37 CFR 1.56.

Foreign Application(s) and/or Claim of Foreign Priority

I hereby claim foreign priority benefits under Title 35, United States Code Section 119 of any foreign application(s) for patent or inventor(s) certificate listed below and have also identified below any foreign application for patent or inventor(s) certificate having a filing date before that of the application on which priority is claimed:

COUNTRY	APPLICATION NUMBER	DATE FILED	PRIORITY CLAIMED UNDER 35
			YES: ____ NO: ____
			YES: ____ NO: ____

Provisional Application

I hereby claim the benefit under Title 35, United States Code Section 119(e) of any United States provisional application(s) listed below:

APPLICATION SERIAL NUMBER	FILING DATE
60/141,903	June 30, 1999

U.S. Priority Claim

I hereby claim the benefit under Title 35, United States Code, Section 120 of any United States application(s) listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States application in the manner provided by the first paragraph of Title 35, United States Code Section 112, I acknowledge the duty to disclose material information as defined in Title 37, Code of Federal Regulations, Section 1.56(a) which occurred between the filing date of the prior application and the national or PCT international filing date of this application:

APPLICATION SERIAL NUMBER	FILING DATE	STATUS(patented/pending/abandoned)

POWER OF ATTORNEY:

As a named inventor, I hereby appoint the following attorney(s) and/or agent(s) listed below to prosecute this application and transact all business in the Patent and Trademark Office connected therewith:

Dov Rosenfeld, Reg. No. 38,687

Send Correspondence to:	**Direct Telephone Calls To:**
Dov Rosenfeld 5507 College Avenue, Suite 2 Oakland, CA 94618	Dov Rosenfeld, Reg. No. 38,687 Tel: (510) 547-3378

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Name of First Inventor: Russell S. Dietz **Citizenship:** USA

Residence: 6146 Ostenberg Drive, San Jose, CA 95120-2736

Post Office Address: Same

_____ _____

First Inventor's Signature **Date**

ADDITIONAL INVENTOR SIGNATURES:

Name of Second Inventor: Joseph R. Maixner **Citizenship:** USA

Residence: 121 Driftwood Court, Aptos, CA 95003

Post Office Address: Same

_____ _____
Inventor's Signature **Date**

Name of Third Inventor: Andrew A. Koppenhaver **Citizenship:** USA

Residence: 10400 Kenmore Drive, Fairfax, VA 22030

Post Office Address: Same

_____ _____
Inventor's Signature **Date**

Name of Fourth Inventor: William H. Bares **Citizenship:** USA

Residence: 9005 Glenalden Drive, Germantown, TN 38139

Post Office Address: Same

William H. Bares 10/8/00
Inventor's Signature **Date**

Name of Fifth Inventor: Haig A. Sarkissian **Citizenship:** USA

Residence: 8701 Mountain Top, San Antonio, Texas 78255

Post Office Address: Same

_____ _____
Inventor's Signature **Date**

Name of Sixth Inventor: James F. Torgerson **Citizenship:** USA

Residence: 227 157th Ave., NW, Andover, MN 55304

Post Office Address: Same

_____ _____
Inventor's Signature **Date**

OIPE JC68 NOV 06 2000 PATENT & TRADEMARK OFFICE

PATENT APPLICATION

DECLARATION AND POWER OF ATTORNEY FOR PATENT APPLICATION — **ATTORNEY DOCKET NO: APPT-001-1**

As a below named inventor, I hereby declare that:

My residence/post office address and citizenship are as stated below next to my name;

I believe I am the original, first and sole inventor (if only one name is listed below) or an original, first and joint inventor (if plural names are listed below) of the subject matter which is claimed and for which a patent is sought on the invention entitled:

METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK

the specification of which is attached hereto unless the following box is checked:

(X) was filed on June 30, 2000 as US Application Serial No. 09/608237 or PCT International Application Number _____ and was amended on _____ (if applicable).

I hereby state that I have reviewed and understood the contents of the above-identified specification, including the claims, as amended by any amendment(s) referred to above. I acknowledge the duty to disclose all information which is material to patentability as defined in 37 CFR 1.56.

Foreign Application(s) and/or Claim of Foreign Priority

I hereby claim foreign priority benefits under Title 35, United States Code Section 119 of any foreign application(s) for patent or inventor(s) certificate listed below and have also identified below any foreign application for patent or inventor(s) certificate having a filing date before that of the application on which priority is claimed:

COUNTRY	APPLICATION NUMBER	DATE FILED	PRIORITY CLAIMED UNDER 35
			YES: ____ NO: ____
			YES: ____ NO: ____

Provisional Application

I hereby claim the benefit under Title 35, United States Code Section 119(e) of any United States provisional application(s) listed below:

APPLICATION SERIAL NUMBER	FILING DATE
60/141,903	June 30, 1999

U.S. Priority Claim

I hereby claim the benefit under Title 35, United States Code, Section 120 of any United States application(s) listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States application in the manner provided by the first paragraph of Title 35, United States Code Section 112, I acknowledge the duty to disclose material information as defined in Title 37, Code of Federal Regulations, Section 1.56(a) which occurred between the filing date of the prior application and the national or PCT international filing date of this application:

APPLICATION SERIAL NUMBER	FILING DATE	STATUS(patented/pending/abandoned)

POWER OF ATTORNEY:

As a named inventor, I hereby appoint the following attorney(s) and/or agent(s) listed below to prosecute this application and transact all business in the Patent and Trademark Office connected therewith:

Dov Rosenfeld, Reg. No. 38,687

Send Correspondence to:	**Direct Telephone Calls To:**
Dov Rosenfeld 5507 College Avenue, Suite 2 Oakland, CA 94618	Dov Rosenfeld, Reg. No. 38,687 Tel: (510) 547-3378

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Name of First Inventor: Russell S. Dietz **Citizenship:** USA

Residence: 6146 Ostenberg Drive, San Jose, CA 95120-2736

Post Office Address: Same

_____ _____

First Inventor's Signature **Date**

ADDITIONAL INVENTOR SIGNATURES:

Name of Second Inventor: Joseph R. Maixner **Citizenship:** USA

Residence: 121 Driftwood Court, Aptos, CA 95003

Post Office Address: Same

_____ _____

Inventor's Signature **Date**

Name of Third Inventor: Andrew A. Koppenhaver **Citizenship:** USA

Residence: 10400 Kenmore Drive, Fairfax, VA 22030

Post Office Address: Same

_____ _____

Inventor's Signature **Date**

Name of Fourth Inventor: William H. Bares **Citizenship:** USA

Residence: 9005 Glenalden Drive, Germantown, TN 38139

Post Office Address: Same

_____ _____

Inventor's Signature **Date**

Name of Fifth Inventor: Haig A. Sarkissian **Citizenship:** USA

Residence: 8701 Mountain Top, San Antonio, Texas 78255

Post Office Address: Same

Haig A. Sarkissian Sept 21, 2000

Inventor's Signature **Date**

Name of Sixth Inventor: James F. Torgerson **Citizenship:** USA

Residence: 227 157th Ave., NW, Andover, MN 55304

Post Office Address: Same

_____ _____

Inventor's Signature **Date**

NOV 0 6 2000
PATENT & TRADEMARK OFFICE

PATENT APPLICATION

DECLARATION AND POWER OF ATTORNEY FOR PATENT APPLICATION

ATTORNEY DOCKET NO. APPT-001-1

As a below named inventor, I hereby declare that:

My residence/post office address and citizenship are as stated below next to my name;

I believe I am the original, first and sole inventor (if only one name is listed below) or an original, first and joint inventor (if plural names are listed below) of the subject matter which is claimed and for which a patent is sought on the invention entitled:

METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK

the specification of which is attached hereto unless the following box is checked:

(X) was filed on June 30, 2000 as US Application Serial No. 09/608237 or PCT International Application Number ____ and was amended on _____ (if applicable).

I hereby state that I have reviewed and understood the contents of the above-identified specification, including the claims, as amended by any amendment(s) referred to above. I acknowledge the duty to disclose all information which is material to patentability as defined in 37 CFR 1.56.

Foreign Application(s) and/or Claim of Foreign Priority

I hereby claim foreign priority benefits under Title 35, United States Code Section 119 of any foreign application(s) for patent or inventor(s) certificate listed below and have also identified below any foreign application for patent or inventor(s) certificate having a filing date before that of the application on which priority is claimed:

COUNTRY	APPLICATION NUMBER	DATE FILED	PRIORITY CLAIMED UNDER 35
			YES: ___ NO: ___
			YES: ___ NO: ___

Provisional Application

I hereby claim the benefit under Title 35, United States Code Section 119(e) of any United States provisional application(s) listed below:

APPLICATION SERIAL NUMBER	FILING DATE
60/141,903	June 30, 1999

U.S. Priority Claim

I hereby claim the benefit under Title 35, United States Code, Section 120 of any United States application(s) listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States application in the manner provided by the first paragraph of Title 35, United States Code Section 112, I acknowledge the duty to disclose material information as defined in Title 37, Code of Federal Regulations, Section 1.56(a) which occurred between the filing date of the prior application and the national or PCT international filing date of this application:

APPLICATION SERIAL NUMBER	FILING DATE	STATUS(patented/pending/abandoned)

POWER OF ATTORNEY:

As a named inventor, I hereby appoint the following attorney(s) and/or agent(s) listed below to prosecute this application and transact all business in the Patent and Trademark Office connected therewith:

Dov Rosenfeld, Reg. No. 38,687

Send Correspondence to:	**Direct Telephone Calls To:**
Dov Rosenfeld 5507 College Avenue, Suite 2 Oakland, CA 94618	Dov Rosenfeld, Reg. No. 38,687 Tel: (510) 547-3378

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Name of First Inventor: Russell S. Dietz **Citizenship:** USA

Residence: 6146 Ostenberg Drive, San Jose, CA 95120-2736

Post Office Address: Same

_____ _____

First Inventor's Signature **Date**

ADDITIONAL INVENTOR SIGNATURES:

Name of Second Inventor: Joseph R. Maixner **Citizenship:** USA

Residence: 121 Driftwood Court, Aptos, CA 95003

Post Office Address: Same

_____ _____

Inventor's Signature **Date**

Name of Third Inventor: Andrew A. Koppenhaver **Citizenship:** USA

Residence: 10400 Kenmore Drive, Fairfax, VA 22030

Post Office Address: Same

_____ _____

Inventor's Signature **Date**

Name of Fourth Inventor: William H. Bares **Citizenship:** USA

Residence: 9005 Glenalden Drive, Germantown, TN 38139

Post Office Address: Same

_____ _____

Inventor's Signature **Date**

Name of Fifth Inventor: Haig A. Sarkissian **Citizenship:** USA

Residence: 8701 Mountain Top, San Antonio, Texas 78255

Post Office Address: Same

_____ _____

Inventor's Signature **Date**

Name of Sixth Inventor: James F. Torgerson **Citizenship:** USA

Residence: 227 157th Ave., NW, Andover, MN 55304

Post Office Address: Same

James Torgerson 9/21/00

Inventor's Signature **Date**

#3

OIPE JC68
NOV 0 6 2000
PATENT & TRADEMARK OFFICE

FORMALITIES LETTER

OC000000005353894

UNITED STATES DEPARTMENT OF COMMERCE
Patent and Trademark Office
Address COMMISSIONER OF PATENT AND TRADEMARKS
Washington, D C. 20231

APPLICATION NUMBER	FILING/RECEIPT DATE	FIRST NAMED APPLICANT	ATTORNEY DOCKET NUMBER
09/608,237	06/30/2000	Russell S. Dietz	APPT-001-1

Dov Rosenfeld
Suite 2
5507 College Avenue
Oakland, CA 94618

Date Mailed: 08/25/2000

NOTICE TO FILE MISSING PARTS OF NONPROVISIONAL APPLICATION

FILED UNDER 37 CFR 1.53(b)

Filing Date Granted

An application number and filing date have been accorded to this application. The item(s) indicated below, however, are missing. Applicant is given TWO MONTHS from the date of this Notice within which to file all required items and pay any fees required below to avoid abandonment. Extensions of time may be obtained by filing a petition accompanied by the extension fee under the provisions of 37 CFR 1.136(a).

- The statutory basic filing fee is missing.
  Applicant must submit ~~$ 690~~ *to complete the basic filing fee and/or file a small entity statement claiming such status (37 CFR 1.27).* 710
- Total additional claim fee(s) for this application is $780.
  - **$702** for **39** total claims over 20.
  - **$78** for **1** independent claims over 3 . 80
- The oath or declaration is missing.
  A properly signed oath or declaration in compliance with 37 CFR 1.63, identifying the application by the above Application Number and Filing Date, is required.
- To avoid abandonment, a late filing fee or oath or declaration surcharge as set forth in 37 CFR 1.16(e) of $130 for a non-small entity, must be submitted with the missing items identified in this letter.
- **The balance due by applicant is $ 1600.**

20
2
1622
40
$1662 + 110 = 1772

A copy of this notice ***MUST*** *be returned with the reply.*

Linash Retta
Customer Service Center
Initial Patent Examination Division (703) 308-1202

PART 2 - COPY TO BE RETURNED WITH RESPONSE

GP/2768 #4
213' 4-16-0

Our Docket/Ref. No.: APPT-001-1 Patent

OIPE
APR 12 2001
PATENT & TRADEMARK OFFICE

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant(s): Dietz et al. Serial No.: 09/608237 Filed: June 30, 2000 Title: METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK	Group Art Unit: 2755 Examiner:

RECEIVED
APR 16 2001
Technology Center 2100

Commissioner for Patents
Washington, D.C. 20231

TRANSMITTAL: INFORMATION DISCLOSURE STATEMENT

Dear Commissioner:

Transmitted herewith are:

X An Information Disclosure Statement for the above referenced patent application, together with PTO form 1449 and a copy of each reference cited in form 1449.

___ A check for petition fees.

X Return postcard.

X The commissioner is hereby authorized to charge payment of any missing fee associated with this communication or credit any overpayment to Deposit Account 50-0292.
A DUPLICATE OF THIS TRANSMITTAL IS ATTACHED

Date: April 9, 2001

Respectfully submitted,

Dov Rosenfeld
Attorney/Agent for Applicant(s)
Reg. No. 38687

Correspondence Address:
Dov Rosenfeld
5507 College Avenue, Suite 2
Oakland, CA 94618
Telephone No.: +1-510-547-3378

Certificate of Mailing under 37 CFR 1.18

I hereby certify that this correspondence is being deposited with the United States Postal Service as first class mail in an envelope addressed to: Commissioner for Patents, Washington, D.C. 20231.

Date of Deposit: Apr 9, 2001

Signature: _____
Dov Rosenfeld, Reg. No. 38,687

Our Docket/Ref. No.: APPT-001-1 Patent

OIPE JC599 APR 12 2001 PATENT & TRADEMARK OFFICE

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant(s): Dietz et al.	Group Art Unit: 2755
Serial No.: 09/608237	Examiner:
Filed: June 30, 2000	
Title: METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK	

RECEIVED
APR 16 2001
Technology Center 2100

Commissioner for Patents
Washington, D.C. 20231

INFORMATION DISCLOSURE STATEMENT

Dear Commissioner:

This Information Disclosure Statement is submitted:

X under 37 CFR 1.97(b), or
(Within three months of filing national application; or date of entry of international application; or before mailing date of first office action on the merits; whichever occurs last)

___ under 37 CFR 1.97(c) together with either a:
___ Certification under 37 CFR 1.97(e), or
___ a $180.00 fee under 37 CFR 1.17(p)
(After the CFR 1.97(b) time period, but before final action or notice of allowance, whichever occurs first)

___ under 37 CFR 1.97(d) together with a:
___ Certification under 37 CFR 1.97(e), and
___ a petition under 37 CFR 1.97(d)(2)(ii), and
___ a $130.00 petition fee set forth in 37 CFR 1.17(i)(1).
(Filed after final action or notice of allowance, whichever occurs first, but before payment of the issue fee)

X Applicant(s) submit herewith Form PTO 1449-Information Disclosure Citation together with copies, of patents, publications or other information of which applicant(s) are aware, which applicant(s) believe(s) may be material to the examination of this application and for which there may be a duty to disclose in accordance with 37 CFR 1.56.

Certificate of Mailing under 37 CFR 1.18

I hereby certify that this correspondence is being deposited with the United States Postal Service as first class mail in an envelope addressed to: Commissioner for Patents, Washington, D.C. 20231.

Date of Deposit: Apr 9, 2001

Signature: _____
Dov Rosenfeld, Reg. No. 38,687

X Some of the references were cited in a search report from a foreign patent office in a counterpart foreign application. In particular, references AD, AF, AH, CI, EA, EB, EC, and ED were cited in a search report from a foreign patent office in a counterpart foreign application.

It is expressly requested that the cited information be made of record in the application and appear among the "references cited" on any patent to issue therefrom.

As provided for by 37 CFR 1.97(g) and (h), no inference should be made that the information and references cited are prior art merely because they are in this statement and no representation is being made that a search has been conducted or that this statement encompasses all the possible relevant information.

Respectfully submitted,

Date: April 9, 2001

Dov Rosenfeld
Attorney/Agent for Applicant(s)
Reg. No. 38687

Correspondence Address:
Dov Rosenfeld
5507 College Avenue, Suite 2
Oakland, CA 94618
Telephone No.: +1-510-547-3378

Et al. FORM 1449 SHEET 1 OF 5.

OIPE JC59 APR 1 2 2001 PATENT & TRADEMARK OFFICE

INFORMATION DISCLOSURE STATEMENT	ATTY. DOCKET NO. APPT-001-1	SERIAL NO. 09/608237
	APPLICANT Dietz et al.	#4
(Use several sheets if necessary)	FILING DATE 6/30/2000	GROUP ~~2755~~ 2157

RECEIVED APR 1 6 2001 Technology Center 2100

U.S. PATENT DOCUMENTS

*EXAMINER INITIAL		DOCUMENT NUMBER	DATE	NAME	CLASS	SUB-CLASS	FILING DATE IF APPROPRIATE
	AA	4736320	Apr. 5, 1988	Bristol	364	300	Oct. 8, 1985
	AB	4891639	Jan. 2, 1990	Nakamura	340	825.500	Jun. 23, 1988
	AC	5101402	Mar. 31, 1992	Chui et al.	370	17	May 24, 1988
	AD	5247517	Sep. 21, 1993	Ross et al.	370	85.5	Sep. 2, 1992
MMM	AE	5247693	Sep. 21, 1993	Bristol	395	800	Nov. 17, 1992
	AF	5315580	May 24, 1994	Phaal	370	13	Aug. 26, 1991
	AG	5339268	Aug. 16, 1994	Machida	365	49	Nov. 24, 1992
	AH	5351243	Sep. 27, 1994	Kalkunte et. al.	370	92	Dec. 27, 1991
	AI	5365514	Nov. 15, 1994	Hershey et al.	370	17	Mar. 1, 1993
	AJ	5375070	Dec. 20, 1994	Hershey at al.	364	550	Mar. 1, 1993
	AK	5394394	Feb. 28, 1995	Crowther et al.	370	60	Jun. 24, 1993

FOREIGN PATENT DOCUMENTS

		DOCUMENT NUMBER	PUBLI-CATION DATE	COUNTRY	CLASS	SUB-CLASS	TRANS-LATION YES	NO
	AM							
	AN							

OTHER DISCLOSURES (Including Author, Title, Date, Pertinent Pages, Place of Publication, Etc.)

MMM	AR	"Technical Note: the Narus System," Downloaded April 29, 1999 from www.narus.com, Narus Corporation, Redwood City California.
	AS	

EXAMINER M. Mehta	DATE CONSIDERED 6/21/2003

*EXAMINER: initial if citation considered, whether or not citation is in conformance with MPEP 609. Draw line through citation if not in conformance and not considered. Include a copy of this form with next communication to Applicant.

Et al. FORM - 1449 SHEET 2 OF 5.

OIPE JC68
APR 12 2001
PATENT & TRADEMARK OFFICE

INFORMATION DISCLOSURE STATEMENT

(Use several sheets if necessary)

ATTY. DOCKET NO.	SERIAL NO.
APPT-001-1	09/608237
APPLICANT	
Dietz et al. #4	
FILING DATE	GROUP
6/30/2000	~~2755~~ 2157

RECEIVED
APR 16 2001
Technology Center 2100

U.S. PATENT DOCUMENTS

*EXAMINER INITIAL		DOCUMENT NUMBER	DATE	NAME	CLASS	SUB-CLASS	FILING DATE IF APPROPRIATE
	BA	5414650	May 9, 1995	Hekhuis	364	715.02	Mar. 24, 1993
	BB	5430709	Jul. 4, 1995	Galloway	370	13	Jun. 17, 1992
	BC	5432776	Jul. 11, 1995	Harper	370	17	Sep. 30, 1993
	BD	5493689	Feb. 20, 1996	Waclawsky et al.	395	821	Mar. 1, 1993
	BE	5500855	Mar. 19, 1996	Hershey et al.	370	17	Jan. 26, 1994
	BF	5568471	Oct. 22, 1996	Hershey et al.	370	17	Sep. 6, 1995
	BG	5574875	Nov. 12, 1996	Stansfield et al.	395	403	Mar. 12, 1993
	BH	5586266	Dec. 17, 1996	Hershey et al.	395	200.11	Oct. 15, 1993
	BI	5606668	Feb. 25, 1997	Shwed	395	200.11	Dec. 15, 1993
	BJ	5608662	Mar. 4, 1997	Large et al.	364	724.01	Jan. 12, 1995
	BK	5634009	May 27, 1997	Iddon et al.	395	200.11	Oct. 27, 1995

FOREIGN PATENT DOCUMENTS

		DOCUMENT NUMBER	PUBLI-CATION DATE	COUNTRY	CLASS	SUB-CLASS	TRANS-LATION YES	NO
	BM							
	BN							

OTHER DISCLOSURES (Including Author, Title, Date, Pertinent Pages, Place of Publication, Etc.)

	BR	
	BS	

EXAMINER	DATE CONSIDERED
M. Meky	6/24/2003

*EXAMINER: Initial if citation considered, whether or not citation is in conformance with MPEP 609. Draw line through citation if not in conformance and not considered. Include a copy of this form with next communication to Applicant.

OIPE APR 12 2001 PATENT & TRADEMARK OFFICE JC59

INFORMATION DISCLOSURE STATEMENT

(Use several sheets if necessary)

ATTY. DOCKET NO.	SERIAL NO.
APPT-001-1	09/608237
APPLICANT	
Dietz et al.	#4
FILING DATE	**GROUP**
6/30/2000	~~2755~~ 2157

RECEIVED APR 16 2001 Technology Center 2100

U.S. PATENT DOCUMENTS

*EXAMINER INITIAL		DOCUMENT NUMBER	DATE	NAME	CLASS	SUB-CLASS	FILING DATE IF BPPROPRIBTE
	CA	5651002	Jul. 22, 1997	Van Seters et all.	370	392	Jul. 12, 1995
	CB	5684954	Nov. 4, 1997	Kaiserswerth et al.	395	200.2	Mar. 20, 1993
	CC	5732213	Mar. 24, 1998	Gessel et al.	395	200.11	Mar. 22, 1996
	CD	5740355	Apr. 14, 1998	Watanabe et al.	395	183.21	Jun. 4, 1996
	CE	5761424	Jun. 2, 1998	Adams et al.	395	200.47	Dec. 29, 1995
	CF	5764638	Jun. 9, 1998	Ketchum	370	401	Sep. 14, 1995
	CG	5781735	Jul. 14, 1998	Southard	395	200.54	Sep. 4, 1997
	CH	5784298	Jul. 21, 1998	Hershey et al.	364	557	Jul. 11, 1996
	CI	5787253	Jul. 28, 1998	McCreery et al.	395	200.61	May 28, 1996
	CJ	5805808	Sep. 8, 1998	Hansani et al.	395	200.2	Apr. 9, 1997
	CK	5812529	Sep. 22, 1998	Czarnik et al.	370	245	Nov. 12, 1996

FOREIGN PATENT DOCUMENTS

		DOCUMENT NUMBER	PUBLI-CATION DATE	COUNTRY	CLASS	SUB-CLASS	TRANS-LATION YES	NO
	CM							
	CN							

OTHER DISCLOSURES (Including Author, Title, Date, Pertinent Pages, Place of Publication, Etc.)

	CR	
	CS	

EXAMINER	DATE CONSIDERED
M. Meby	6/21/2003

*EXAMINER: Initial if citation considered, whether or not citation is in conformance with MPEP 609. Draw line through citation if not in conformance and not considered. Include a copy of this form with next communication to Applicant.

Et al. FORM - 1449 SHEET 4 OF 5.

OIPE
APR 12 2001
PATENT & TRADEMARK OFFICE U.S.

INFORMATION DISCLOSURE STATEMENT

(Use several sheets if necessary)

ATTY. DOCKET NO.	SERIAL NO.
APPT-001-1	09/60[illegible]
APPLICANT	
Dietz et al.	#4
FILING DATE	GROUP
6/30/2000	~~2755~~ 2157

RECEIVED
APR 16 2001
Technology Center 2100

U.S. PATENT DOCUMENTS

*EXAMINER INITIAL		DOCUMENT NUMBER	DATE	NAME	CLASS	SUB-CLASS	FILING DATE IF BPPROPRIBTE
	DA	5819028	Oct. 6, 1998	Manghirmalani et al.	395	185.1	Apr. 16, 1997
	DB	5825774	Oct. 20, 1998	Ready et al.	370	401	Jul. 12, 1995
	DC	5835726	Nov. 10, 1998	Shwed et al.	395	200.59	Jun. 17, 1996
	DD	5838919	Nov. 17, 1998	Schwaller et al.	395	200.54	Sep. 10, 1996
	DE	5841895	Nov. 24, 1998	Huffman	382	155	Oct. 25, 1996
	DF	5850386	Dec. 15, 1998	Anderson et al.	370	241	Nov. 1, 1996
	DG	5850388	Dec. 15, 1998	Anderson et al.	370	252	Oct. 31, 1996
	DH	5862335	Jan. 19, 1999	Welch, Jr. et al.	395	200.54	Apr. 1, 1993
	DI	5878420	Mar. 2, 1999	de la Salle	707	10	Oct. 29, 1997
	DJ	5893155	Apr. 6, 1999	Cheriton	711	144	Dec. 3, 1996
	DK	5903754	May 11, 1999	Pearson	395	680	Nov. 14, 1997

FOREIGN PATENT DOCUMENTS

		DOCUMENT NUMBER	PUBLI-CATION DATE	COUNTRY	CLASS	SUB-CLASS	TRANS-LATION YES	NO
	DM							
	DN							

OTHER DISCLOSURES (Including Author, Title, Date, Pertinent Pages, Place of Publication, Etc.)

	DR	
	DS	

EXAMINER	DATE CONSIDERED
M. Kelley	6/2/2003

*EXAMINER: initial if citation considered, whether or not citation is in conformance with MPEP 609. Draw line through citation if not in conformance and not considered. Include a copy of this form with next communication to Applicant.

Et al. FORM - 1449 SHEET 5 OF 5.

OIPE
APR 12 2001
PATENT & TRADEMARK OFFICE

INFORMATION DISCLOSURE STATEMENT

(Use several sheets if necessary)

ATTY. DOCKET NO.	SERIAL NO.
APPT-001-1	09/608237
APPLICANT	
Dietz et al.	#4
FILING DATE	GROUP
6/30/2000	~~2155~~ 2157

RECEIVED
APR 16 2001
Technology Center 2100

U.S. PATENT DOCUMENTS

*EXAMINER INITIAL		DOCUMENT NUMBER	DATE	NAME	CLASS	SUB-CLASS	FILING DATE IF BPPROPRIBTE
MMM	EA	5917821	Jun. 29, 1999	Gobuyan et al.	370	392	Aug. 16, 1996
MMM	EB	5414704	May 9, 1995	Spinney	370	60	Apr. 5, 1994
MMM	EC	6014380	Jan 11, 2000	Hendel et al.	370	392	Jun. 30, 1997
MMM	ED	5511215	Apr. 23, 1996	Terasaka et al.	395	800	Oct. 26, 1993
	EE						
	EF						
	EG						
	EH						
	EI						
	EJ						
	EK						

FOREIGN PATENT DOCUMENTS

		DOCUMENT NUMBER	PUBLI-CATION DATE	COUNTRY	CLASS	SUB-CLASS	TRANS-LATION YES	NO
	DM							
	DN							

OTHER DISCLOSURES (Including Author, Title, Date, Pertinent Pages, Place of Publication, Etc.)

	DR	
	DS	

EXAMINER	DATE CONSIDERED
M. Mehr	6/21/2003

*EXAMINER Initial if citation considered, whether or not citation is in conformance with MPEP 609. Draw line through citation if not in conformance and not considered. Include a copy of this form with next communication to Applicant.

#5

Our Docket/Ref. No.: APPT-0001-1 Patent

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant(s): Dietz et al. Serial No.: 09/608237 Filed: June 30, 2000 Title: METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK	Group Art Unit: 2755 Examiner:

RECEIVED
APR 22 2002
Technology Center 2100

Commissioner for Patents
Washington, D.C. 20231

INFORMATION DISCLOSURE STATEMENT

Dear Commissioner:

This Information Disclosure Statement is submitted:

X under 37 CFR 1.97(b), or
(Within three months of filing national application; or date of entry of international application; or before mailing date of first office action on the merits; whichever occurs last)

X Applicant(s) submit herewith Form PTO 1449-Information Disclosure Citation together with copies, of patents, publications or other information of which applicant(s) are aware, which applicant(s) believe(s) may be material to the examination of this application and for which there may be a duty to disclose in accordance with 37 CFR 1.56.

X (Certification under 37 C.F.R. 1.97 (e)) Each item of information contained in this information disclosure statement was first cited in an official communication from a foreign patent office in a counterpart foreign application not more than three months prior to the filing of this information disclosure statement (written opinion from PCT mailed Jan 11,2002).

It is expressly requested that the cited information be made of record in the application and appear among the "references cited" on any patent to issue therefrom.

As provided for by 37 CFR 1.97(g) and (h), no inference should be made that the information and references cited are prior art merely because they are in this statement and no representation is

Certificate of Mailing under 37 CFR 1.18

I hereby certify that this correspondence is being deposited with the United States Postal Service as first class mail in an envelope addressed to: Commissioner for Patents, Washington, D.C. 20231.

Date of Deposit: 30 Mar 2002 Signature: _____
Dov Rosenfeld, Reg. No. 38,687

being made that a search has been conducted or that this statement encompasses all the possible relevant information.

Date: 30 Mar 2002

Respectfully submitted,

Dov Rosenfeld
Attorney/Agent for Applicant(s)
Reg. No. 38687

Correspondence Address:
Dov Rosenfeld
5507 College Avenue, Suite 2
Oakland, CA 94618
Telephone No.: +1-510-547-3378

Et al. FOR: . . . . SHEET 1 OF 1.

INFORMATION DISCLOSURE STATEMENT

(Use several sheets if necessary)

OIPE JC42
APR 18 2002
PATENT & TRADEMARK OFFICE

ATTY. DOCKET NO.	SERIAL NO.
APPT-001-1	09/608237
APPLICANT: Dietz et al. #5	
FILING DATE: 6/30/2000	GROUP: ~~2755~~ 2157

U.S. PATENT DOCUMENTS

*EXAMINER INITIAL		DOCUMENT NUMBER	DATE	NAME	CLASS	SUB-CLASS	FILING DATE IF APPROPRIATE
MMM	AA	5,249,292	Sep. 28, 1993	Chiappa	395	650	Mar.10,1992
MMM	AB	5,511,213	Apr. 23, 1996	Correa	395	800	May 8, 1992
MMM	AC	5,703,877	Dec. 30, 1997	Nuber et al.	370	395	Nov. 22, 1995
MMM	AD	5,802,054	Sep. 1, 1998	Bellenger	370	351	Aug. 16, 1996
	AE						
	AF						
	AG						
	AH						
	AI						
	AJ						
	AK						
	AL						
	AM						
	AN						

RECEIVED
APR 22 2002
Technology Center 2100

FOREIGN PATENT DOCUMENTS

		DOCUMENT NUMBER	PUBLI-CATION DATE	COUNTRY	CLASS	SUB-CLASS	TRANS-LATION YES	NO
	AO							

OTHER DISCLOSURES (Including Author, Title, Date, Pertinent Pages, Place of Publication, Etc.)

	AP	

EXAMINER	DATE CONSIDERED
M. Meby	6/21/2003

*EXAMINER: initial if citation considered, whether or not citation is in conformance with MPEP 609. Draw line through citation if not in conformance and not considered. Include a copy of this form with next communication to Applicant.

US005249292A

United States Patent [19]

Chiappa

[11] Patent Number: 5,249,292

[45] Date of Patent: Sep. 28, 1993

[54] **DATA PACKET SWITCH USING A PRIMARY PROCESSING UNIT TO DESIGNATE ONE OF A PLURALITY OF DATA STREAM CONTROL CIRCUITS TO SELECTIVELY HANDLE THE HEADER PROCESSING OF INCOMING PACKETS IN ONE DATA PACKET STREAM**

[76] Inventor: **J. Noel Chiappa**, 708 E. Woodland Dr., Grafton, Va. 23692

[21] Appl. No.: **847,880**

[22] Filed: **Mar. 10, 1992**

Related U.S. Application Data

[63] Continuation of Ser. No. 332,530, Mar. 31, 1989, abandoned.

[51] **Int. Cl.**[5] **G06F 9/28**; G06F 13/12

[52] **U.S. Cl.** **395/650**; 395/325; 395/800; 370/60; 370/61; 364/DIG. 1; 364/228; 364/229.2; 364/230.3; 364/230.4; 364/266

[58] **Field of Search** ..... 364/DIG. 1, DIG. 2; 340/825.52, 825.1; 370/60, 61, 80; 395/200, 325, 650, 800

[56] **References Cited**

U.S. PATENT DOCUMENTS

3,846,763	11/1974	Riikonen	395/275
4,281,315	7/1981	Bauer et al.	364/200
4,312,065	1/1982	Ulug	370/94
4,456,957	6/1984	Schieltz	364/200
4,493,030	1/1985	Barrart et al.	364/200
4,494,230	1/1985	Turner	370/60
4,499,576	2/1985	Fraser	370/60
4,601,586	7/1986	Bahr et al.	364/200
4,630,258	12/1986	McMillen et al.	370/60
4,630,260	12/1986	Toy et al.	370/60
4,777,595	10/1988	Strecker et al.	364/200
4,807,282	2/1989	Kazan et al.	379/284
4,851,997	7/1989	Tatara	364/200
4,858,112	8/1989	Puerzer et al.	364/200
4,899,333	2/1990	Roediger	370/60
4,975,828	12/1990	Wishneusky et al.	395/325
4,979,100	12/1990	Makris et al.	395/325
4,991,133	2/1991	Davis et al.	395/375

OTHER PUBLICATIONS

"Hyperchannel Net Is Plugged Into the Open-Systems World," *Electronics*, Oct. 1, 1987, pp. 96-97.

"Cisco Introduces High-Performance Desktop Gateway That Allows Remote Users to Access World-Wide Networks", ciscoSystems, Inc., Mar. 4, 1988.

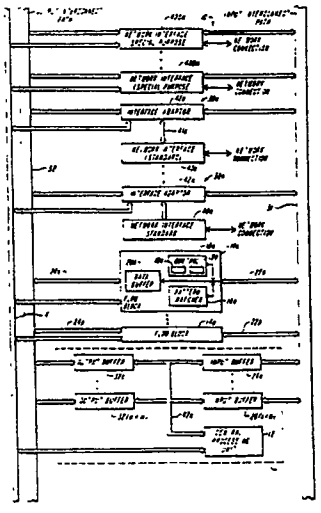
"Company Backgrounder Mar. 1988", ciscoSystems, Inc. Network Systems brochures.

Primary Examiner—Thomas C. Lee
Assistant Examiner—John C. Loomis
Attorney, Agent, or Firm—Fish & Richardson

[57] **ABSTRACT**

A high speed data packet switching circuit has a software controlled primary processing unit, a plurality of network interface units connected to a plurality of networks for receiving incoming data packet streams and for transmitting outgoing data packet streams, a plurality of high speed data stream hardware control circuits for processing data packets in response to instructions from the primary processing unit and circuitry for interconnecting the primary processing unit, the interface units, and the data stream control circuits. The primary processing unit receives from the network interface unit at least a first one of the data packets of each new data packet stream and assigns that stream to be processed by one of the data stream control circuits without further processing by the primary processing unit. The apparatus and method thus perform routine, repetitive processing steps on the further packets of the data stream using the high speed hardware circuitry, while the initial processing and other non-repetitive or special processing of the data packets are performed in software. Particular hardware is described for effecting the high speed hardware processing of the data packets.

17 Claims, 5 Drawing Sheets



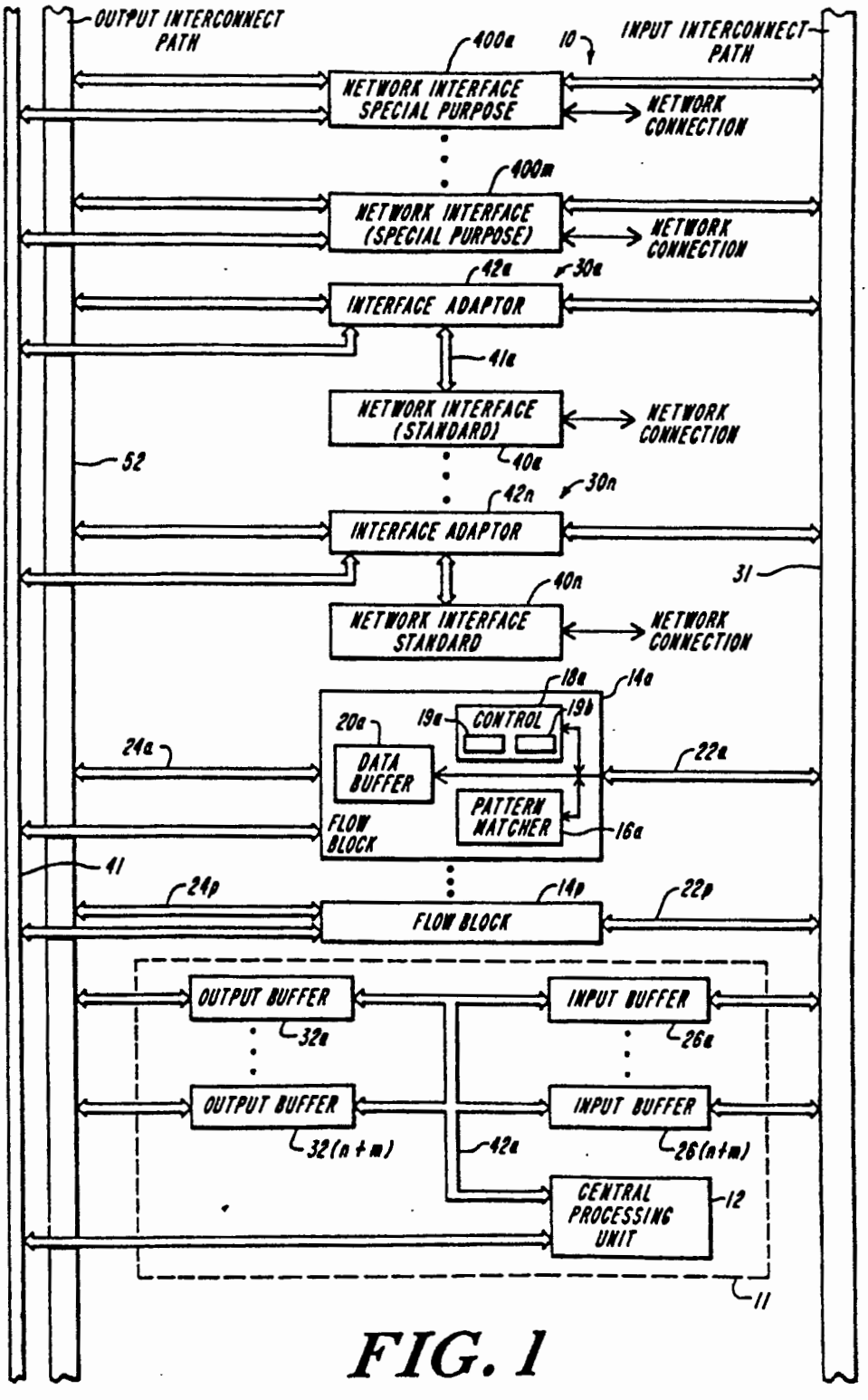

OUTPUT INTERCONNECT PATH
INPUT INTERCONNECT PATH
400a
10
NETWORK INTERFACE SPECIAL PURPOSE
NETWORK CONNECTION
400m
NETWORK INTERFACE (SPECIAL PURPOSE)
NETWORK CONNECTION
42a
30a
INTERFACE ADAPTOR
41a
NETWORK INTERFACE (STANDARD)
NETWORK CONNECTION
40a
52
42n
30n
INTERFACE ADAPTOR
31
40n
NETWORK INTERFACE STANDARD
NETWORK CONNECTION
18a
14a
20a
19a
CONTROL
19b
24a
DATA BUFFER
22a
PATTERN MATCHER
16a
FLOW BLOCK
41
24p
14p
22p
FLOW BLOCK
OUTPUT BUFFER
32a
INPUT BUFFER
26a
OUTPUT BUFFER
32(n+m)
INPUT BUFFER
26(n+m)
42a
CENTRAL PROCESSING UNIT
12
11

FIG. 1

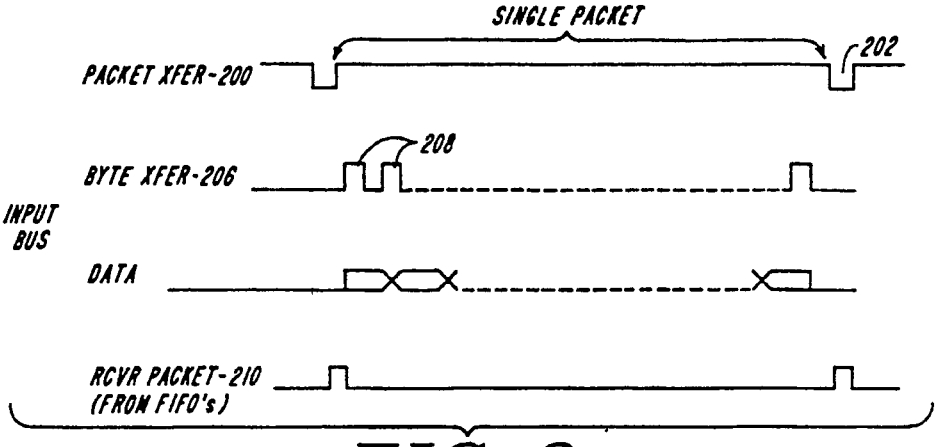

SINGLE PACKET
202
PACKET XFER-200
208
BYTE XFER-206
INPUT BUS
DATA
RCVR PACKET-210 (FROM FIFO's)

FIG. 2

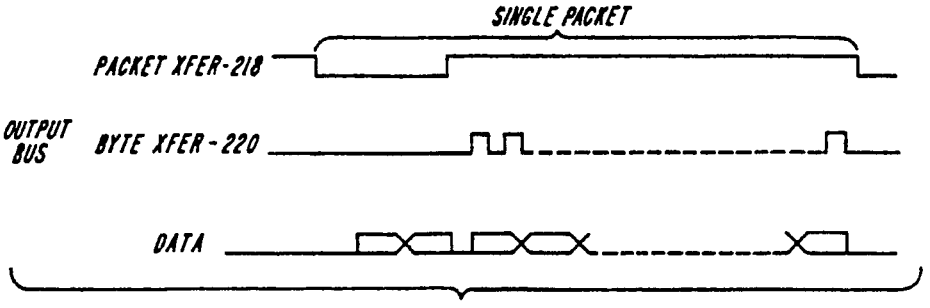

SINGLE PACKET
PACKET XFER-218
OUTPUT BUS
BYTE XFER-220
DATA

FIG. 3

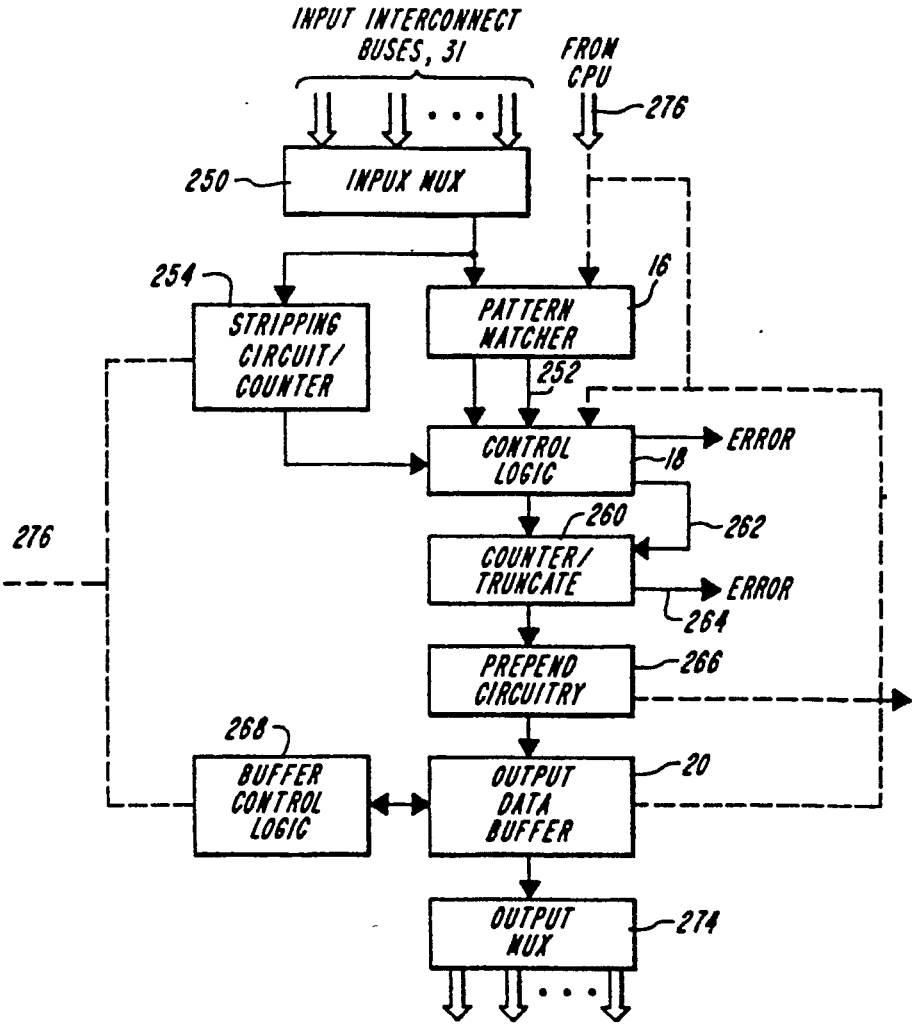

INPUT INTERCONNECT BUSES, 31
FROM CPU
276
250
INPUX MUX
254
STRIPPING CIRCUIT/ COUNTER
16
PATTERN MATCHER
252
CONTROL LOGIC
18
ERROR
260
262
276
COUNTER/ TRUNCATE
ERROR
264
PREPEND CIRCUITRY
266
268
BUFFER CONTROL LOGIC
OUTPUT DATA BUFFER
20
OUTPUT MUX
274

FIG. 4

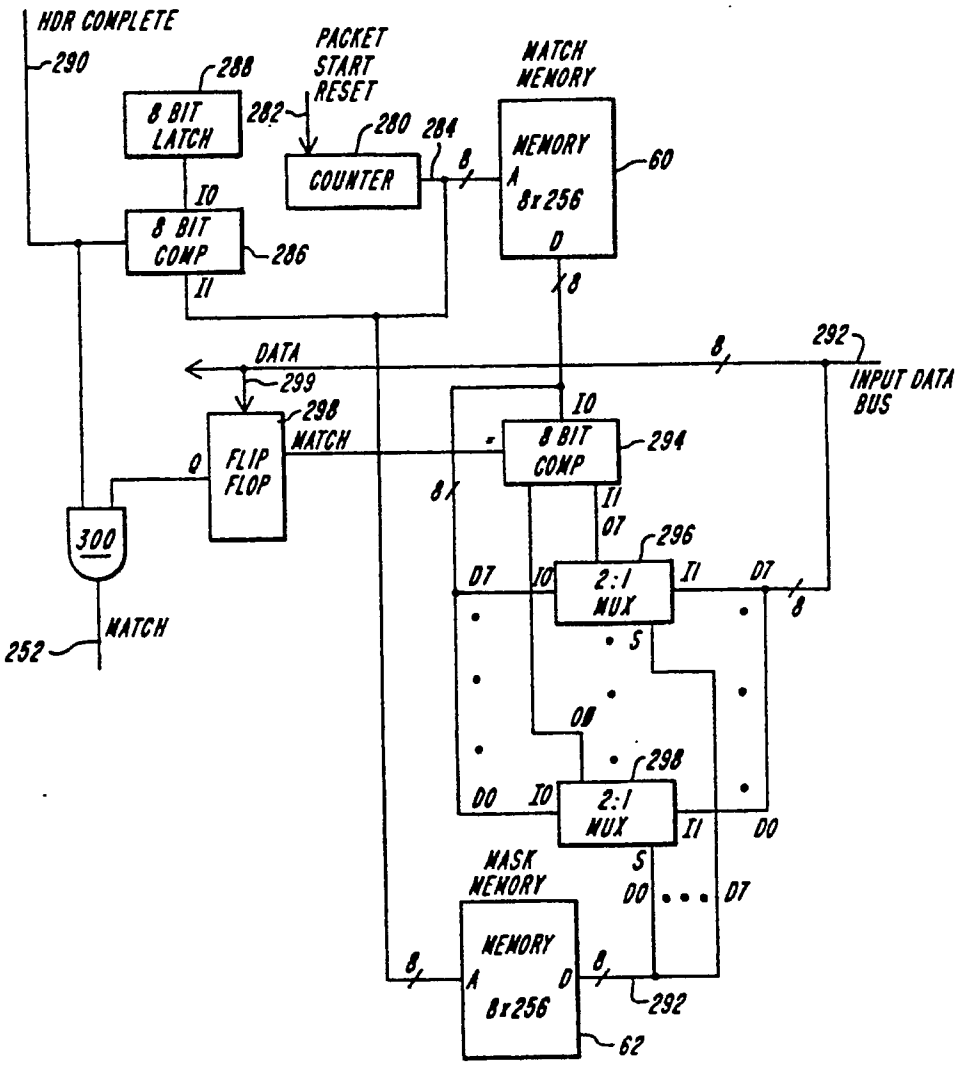

HDR COMPLETE
290
PACKET START RESET
288
8 BIT LATCH
282
280
COUNTER
284
8
MATCH MEMORY
MEMORY
A
8x256
D
60
IO
8 BIT COMP
286
II
8
DATA
299
8
292
INPUT DATA BUS
298
MATCH
FLIP FLOP
Q
IO
8 BIT COMP
294
8
II
07
296
300
07
IO
2:1 MUX
II
07
8
S
MATCH
252
08
298
00
IO
2:1 MUX
II
00
S
MASK MEMORY
00 ... 07
MEMORY
A
8x256
D
8
8
292
62

FIG. 5

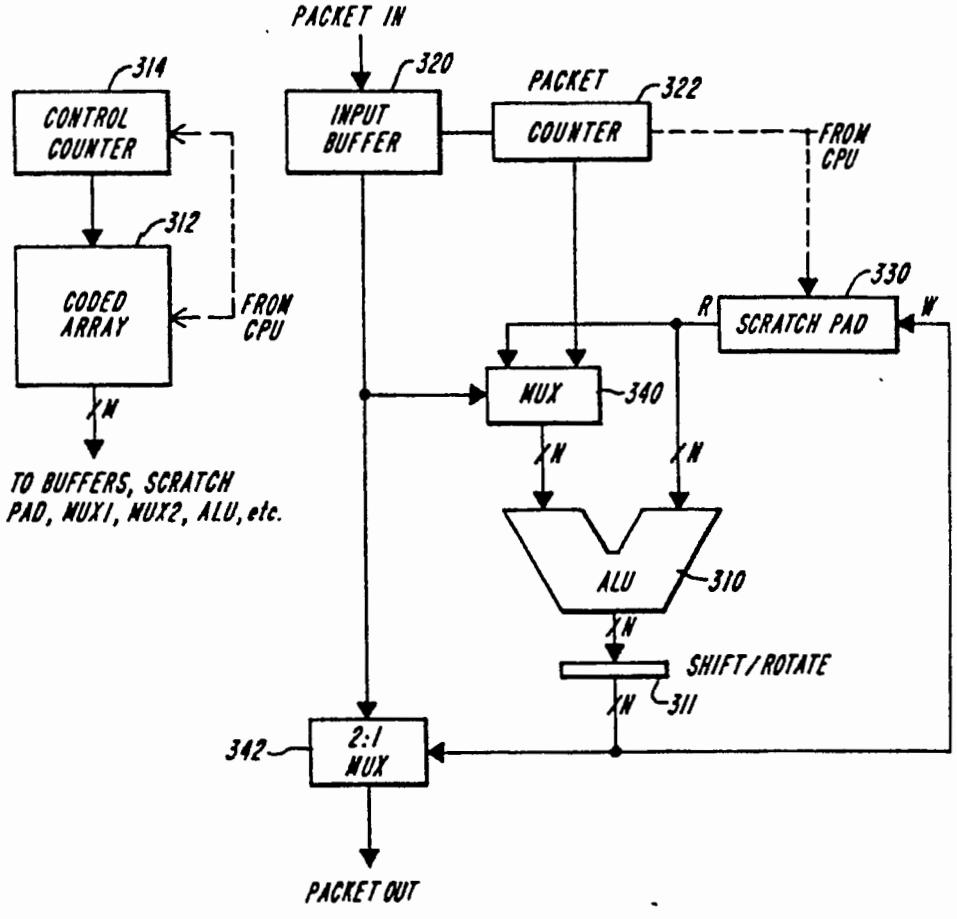

PACKET IN
314
CONTROL COUNTER
320
INPUT BUFFER
PACKET
322
COUNTER
FROM CPU
312
CODED ARRAY
FROM CPU
330
R
SCRATCH PAD
W
MUX
340
XN
XN
TO BUFFERS, SCRATCH PAD, MUX1, MUX2, ALU, etc.
ALU
310
XN
SHIFT/ROTATE
XN
311
342
2:1 MUX
PACKET OUT

FIG. 6

DATA PACKET SWITCH USING A PRIMARY PROCESSING UNIT TO DESIGNATE ONE OF A PLURALITY OF DATA STREAM CONTROL CIRCUITS TO SELECTIVELY HANDLE THE HEADER PROCESSING OF INCOMING PACKETS IN ONE DATA PACKET STREAM

This is a continuation of co-pending application Ser. No. 332,530 field on Mar. 31, 1989 now abandoned.

BACKGROUND OF THE INVENTION

The invention relates generally to data communications networks and in particular to the packet switch used to direct the flow of data packets along data paths in a data communications network.

In a data communications network, a data packet switch directs data packets from one network node to another. The throughput for a packet switch can be measured in the number of either data packets per second or bits per second which pass through the switch. The former measure is important because in a typical network traffic, the bulk of the packets are small. However, when the traffic is weighted by packet size, the bulk of the data is carried in large data packets. In large bulk data transfers, the second measure is thus more important. This is a continuing dichotomy in throughput measurement. For example. the amount of work needed to switch packets is fairly constant, independent of the packet size.

The average desired values for both of these measures of packet throughput are going up quickly, just as other basic measures of computer power have been increasing. As the volume of the data transfers increases, increasingly higher throughput rates are being demanded. The increase in the volume of data transfers results as experience is gained in new systems, and more and more applications, with more and more expansive needs, are being developed. Also, quickly changing technology has made the basic underlying data transmission resource very inexpensive. Fiber optics, for example, offers data rates in the gigabyte per second range. Finally, many difficult problems in the organization of large systems can be bypassed by the free consumption of resources. The typical drop in cost of such resources has always made this an attractive path for meeting difficult system requirements.

Accordingly, the need for throughput rates substantially higher than currently available in a packet switch is presently sought. Switches more than an order of magnitude faster than current switches would seem to be required.

The present invention is directed to a class of packet switch which differs substantially from the other two classes of devices often commonly (and confusingly) referred to as packet switches.

One class of packet switch is that commonly used in digital telephone exchanges. This switch is intended only to transfer packets among the devices in a single station, such as a telephone exchange. The format of the packet in these systems is chosen to make the hardware in the switch as simple as possible; and this usually means that the packets include fields designed for direct use by the hardware. The capabilities of this class of switches (for example, in such areas as congestion control) are very limited to keep the hardware simple.

The second class of packet switch is used in networks such as X.25 networks. In some sense, these switches are little different from the switch described above, but there is a substantial difference. The format of the packets (that is, the protocols) handled by these switches is much more complex. The greater complexity is necessary since the protocols are designed to work in less restricted environments and in a much larger system, and provide a greater range of services. While the formats interpreted by the first class of switches above are chosen for easy implementation in hardware, the data packets handled by this second class of switches are generally intended to be interpreted by software (which can easily and economically handle the greater complexity).

In the third class of packet switch, the packet protocols are intended to be used in very large data networks having many very dissimilar links (such as a mix of very high speed LAN's and low speed long distance point to point lines). Examples of such protocols are the United States designed TCP/IP, and the International Standards Organization's IP/CLNS protocols.

In addition, this third class of switches (called routers) often handle multiple protocols simultaneously. Just as there are many human languages, there are many computer protocols. While a single set of telephone links and exchanges suffice to handle all human languages, in computer communication systems the switches are more involved in the carrying of data, and must understand some of the details of each protocol to be able to correctly handle data in that protocol. The routers often have to make fairly complex changes to the packets as they pass through the switch.

It is this latter class of packet switch to which this invention primarily relates. In current conventional packet switch design, a programmed general purpose processor examines each packet as it arrives over the network interface and processes the packet. Packet processing requires assignment to an outbound network interface for transmission over the next communications link in the data path. While attempts are being made to build higher speed packet switches, based on this general architecture, the attempts have not been very successful. One approach is to use faster processors; another is to make the software run faster; and a third is to apply multiple processors to the processing task. All of these approaches fail to meet the need for the reasons noted below.

The approach which uses faster processors simply keeps pace with processor dependent (future) demands since the traffic which the packet switch will handle will depend upon the speed of the user processors being used to generate the traffic. Those user processors, like the processors in the packet switches, will increase in speed at more or less the same rate and accordingly no overall increase in the ability of the future packet switch over the present packet switch, relative to traffic load, will be available. Furthermore, this approach may be impractical as not being cost-effective for wide spread use. For example, two high speed machines, distant from each other, must have intermediate switches which are all equally as powerful; deployment on a large scale of such expensive switches is not likely to be practicable.

The approach which increases the execution rate of the software itself by, for example, removing excess instructions or writing the code in assembly language, leads to a limit beyond which an increase in execution rate cannot be made. The gains which result are typically small (a few percent) and the engineering costs of

such distortions in the software are significant in the long term.

The use of multiple processors to avoid the "processor bottleneck" provides some gains but again has limits. Given a code path to forward a packet, it is not plausible to split that path into more than a few stages. Three is typical: network input; protocol functions; and network output. The basis for this limitation is the overhead incurred to interface the different processors beyond a limited number of task divisions; that is, after a certain point, the increase in interface overhead outweighs the savings obtained from the additional stage. This is particularly true because of the need to tightly integrate the various components, for example, congestion control at the protocol level requires close coordination with the output device. Also, the interface overhead costs are made more severe by the complication of the interface which is required.

In general then, the multiprocessor approach is not, as expected, the answer to substantially increasing the throughput of the packet switching network. This has been borne out by several attempts by technically well-regarded groups to build packet switches using this approach. While aggregate throughput over a large number of interfaces can be obtained, this is, in reality, little different than having a large number of small switches. It has thus far proven implausible to substantially speed up a single stream using this approach.

Accordingly, it is a primary object of the present invention to increase the throughput of a data packet switch while maintaining reasonable cost, and avoiding a high complexity of circuitry.

Other objects of the invention are a high speed data packet switching circuitry and method which can handle large numbers of input streams, large numbers of output destinations and lines, and large and small data packets at high bit and packet throughput rates.

SUMMARY OF THE INVENTION

The invention relates to a method and apparatus for effecting high speed data packet switching. The switching circuit features a software controlled primary processing unit; a plurality of network interface units for receiving incoming data packet streams and for transmitting outgoing data packet streams from and to network paths respectively; a plurality of data stream control circuits or flow blocks for processing data packets in response to the primary processing unit; and circuitry for interconnecting the primary processing unit and the plurality of interface units and data stream control circuits. The primary processing unit is adapted to receive from the network interface units, and to process, at least a first one of the data packets of each new data packet stream and to assign this stream to be processed by a data stream control circuit without further intervention or processing by the primary processing unit. It is important to note that this first packet is not necessarily a "connection set up" packet or any other similar explicit direction to the switch to set up a stream. Rather, as is usual in the connectionless datagram model, this first packet is just another user data packet.

In particular aspects of the invention, the data stream control circuit features a pattern matching circuit, responsive to pattern setting signals from the primary processing unit and to the incoming data packets from the network interface units, for identifying those packets of a packet stream which will be processed by the control circuit. The data stream control circuit further features a processing unit responsive control circuit for controlling, in response to control signals sent by the primary processing unit, the congestion control and header modification, stripping and prepending functions of the data stream control circuit. The data stream control circuit further features a data buffer responsive to the pattern matching circuitry and the processing unit responsive control circuit for storing data and protocol elements of an incoming data packet stream and for outputting a data packet stream to be forwarded along a communications path.

The network interface unit features, in one aspect of the invention, a network interface circuit for communicating with a network channel and an interface adapter for receiving channel data from the network interface circuit and for transmitting that channel data over the interconnecting circuit structure to the data stream control circuits and the primary processing unit, and for receiving network data from the data stream control circuits and the primary processing unit over the interconnecting circuit structure and for providing received data to the associated network interface circuit for transmission over a network channel.

In another particular aspect of the invention, the software controlled primary processing unit features a central processing unit, bus circuitry, a plurality of input storage units for receiving respectively each of the plurality of data streams from the network interface units and each storage unit having its output connected to the bus circuitry, elements for connecting the central processing unit to the bus circuitry, and a plurality of output storage units for receiving data from the central processing unit over the bus circuitry and for providing the data to the network interface units.

The method of the invention features the step of separating from a software controlled primary processing unit used in a high speed data packet switching circuit a portion of the functionality which is repetitively used in connection with the processing of the second and further packets of an input data stream and implementing that portion of the functionality in hardware elements.

BRIEF DESCRIPTION OF THE DRAWINGS

Other objects, features, and advantages of the invention will be apparent from the following description taken together with the drawings in which:

FIG. 1 is an electrical block diagram of an overall packet switching circuitry in accordance with a particular embodiment of the invention;

FIG. 2 is a timing diagram of an input interconnect circuitry according to a particular embodiment of the invention;

FIG. 3 is a timing diagram of an output interconnect circuitry in accordance with a particular embodiment of the invention;

FIG. 4 is a detailed block diagram of the control circuitry according to a particular embodiment of the invention;

FIG. 5 is a detailed block diagram of the pattern matching circuitry according to a particular embodiment of the invention; and

FIG. 6 is a detailed block diagram of the control circuitry of the flow blocks according to a particular embodiment of the invention.

DESCRIPTION OF A PARTICULAR PREFERRED EMBODIMENT BACKGROUND

According to the invention, a selected portion of the packet forwarding operation, previously performed by the processor software, is moved from the software to the packet switch hardware. In this manner, all of the load on the software is removed for "normal user data packets;" and since hardware can operate at a substantially greater speed than software, substantial performance gain can be achieved. However, any attempt to translate into hardware all of the functions currently performed in software would not be possible. Typical packet switches contain tens of thousands of lines of code, and are thus of extreme complexity. To implement all of this software in hardware would require either programmability of the hardware, thus reintroducing the problem of a software system, or require an unmanageable and uneconomic configuration of hardware circuitry. Accordingly, it is necessary is to select that amount of software which can efficiently and effectively be performed in hardware and thus reduce to hardware only a small, but effective, portion of the software function.

If the software code of a typical packet switching system were monitored, most of it is exercised infrequently. It is there to handle errors, or to handle the control traffic which forms a small, albeit vital, share of the packets in the system. Very little of the code, a few percent, is used in connection with processing a "normal" packet through the switch. And it is precisely those "normal" packets which form a preponderance of the packets with which the switch deals. Thus, in one aspect, the invention herein is to select that portion of the software which will be reproduced in hardware and leave the remaining functionality in software where it is more appropriate for reasons of efficiency and support. In particular, the illustrated embodiment attempts to do so with the minimum number of circuit elements.

One way to reduce the functionality which must be reproduced in hardware is to not implement in hardware the code which handles packets other than normal data packets. It is feasible to produce a device which would handle all normal user data packets entirely in hardware. This would allow a far faster router than is available with current means.

However, even that level of reduction can be surpassed, producing an even more efficient implementation (the illustrated embodiment of the invention) if a further observation is made. In the handling of a single data packet, several operations are necessary to forward each packet. In accordance with the invention, it is recognized that many of these forwarding operations are completely repetitive when performed on individual packets which are part of a common connection path, that is, part of a data stream having a common source and often the same destination.

Thus, most packets in the system are part of ongoing transfers in which as many as thousands of similar packets flow through the switch. While the meaning of the various packets at higher levels of the communications system can be quite different, the portion of the packet protocol which concerns the packet switch is usually identical from packet to packet. Thus, judicious retention of information about a traffic stream passing through the switch is often both necessary and useful. It is necessary to implement some required functions such as flow and congestion control. It is further useful to prevent the repetitive computation of identical information for packets belonging to the same traffic stream.

It is further important to recognize that although the complexity of the functionality provided at the packet protocol layer is increasing, it does so (a) because network systems are getting larger and more mechanisms are required to make the larger systems work correctly, (b) because the user community is becoming more sophisticated, and (c) because systems are being deployed with extra functionality. This complexity has a direct bearing on the cost of forwarding packets, since many added functions are performed on each packet.

System Description

Accordingly, the illustrated embodiment of the invention operates using two important assumptions. First, that traffic streams exist and are of sufficient duration to be useful. Second, that the majority of the traffic in the network is in the streams. Both of these assumptions are reasonably descriptive of most data communications networks.

Referring to FIG. 1, in accordance with a particular embodiment of the invention, a specialized hardware 10 does all the work necessary for forwarding a "normal" packet in a previously identified packet stream from one network interface to another. All packets which the specialized hardware 10 cannot process are passed to a software controlled primary processing unit 11, including a central processing unit, CPU, 12, running software code which is more or less similar to the current software code run by the processors of most packet switches. If the packet looks like it is part of a new traffic stream, the central processing unit 12 provides the specialized hardware 10 with the necessary data parameters to deal with further packets from that packet traffic stream. Accordingly, any further packets seen from that data stream are dealt with automatically by the specialized hardware 10.

In operation, a packet switch normally examines the low level network header of an incoming packet at the input network, and removes that header from the packet. The packet is then passed to the software of the appropriate "protocol." The software generally checks the packet for errors, does certain bookkeeping on the packet, ensures that the packet is not violating flow or access controls, generates a route for the packet, and passes it to the output network. The output network constructs the outgoing network header, attaches it to the packet, and sends the packet on to the next packet switch or other destination. At all stages in the process, the packet switch must guard against data congestion.

Most of these functions are identical on packets of the same stream and can therefore be separated from those functions which vary from packet to packet in the same packet stream. The repetitive functions can be performed once in software at CPU 12, at the time the hardware is first set up for a packet stream, that is, at the time the first packet of the stream is being processed. At this time, the hardware itself has very little that it is able to do. Thereafter, the hardware will handle all succeeding packets of the stream without any further intervention from the central processing unit.

The illustrated specialized hardware 10 has a plurality of data stream control circuits (flow blocks) 14*a*, 14*b*, . . . ,14*p*, each flow block having a pattern matching hardware 16, a control circuitry 18, and a data buffer 20. An input bus 22 connects, as described below, to any of the inbound network interfaces, and an output bus 24

NOAC Ex. 1015 Page 146

can connect to any outbound network interface. There is further associated with each input network interface a CPU input storage buffer **26**, the output of which is directed to the CPU **12** for handling special packets, that is, packets which are not "normal," and a CPU output storage buffer **32**, for receiving special packets from the CPU **12** for transmission to the network interfaces.

The network interface devices **30** or **400**, as viewed from the packet processing elements, (either flow blocks **14**, or CPU **12** and storage buffers **26**, **32**), are pure sources or sinks of data. They are always functioning autonomously, and accordingly no intervention is required on the part of the flow blocks **14** or storage buffers **26**, **32** and CPU **12** to keep these network interface devices operating. The flow blocks **14** should not interact with the network interfaces since that interaction would require extra complexity in the flow block, a cost to be paid for in each flow block, and not by the network interface. Further, the central processor **12** should not control the network operation since that control inevitably slows the central processor operation, as well as the network. Accordingly, each network interface device is an autonomous unit.

In the illustrated embodiment, two classes of network interface devices are illustrated. The network interfaces **30***a*, . . . ,**30***n*, each include a network interface adapter **42**, and a standard network interface circuit **40**. The network interfaces **30** connect to an input interconnect path **31**, an output interconnect path **52**, and a CPU standard bus **41** for complete communications with all other circuit elements of the packet switch, and receive data from and transmit data to the associated standard network interface circuit **40**. The other class of network interface device is the special purpose network interface **400** which connects to the input interconnect path **31**, the output interconnect path **52**, the CPU standard bus **41**, and also to the associated network.

In the illustrated embodiment, the packet switch is configured so that it can be expanded as necessary to support more traffic streams. The expansion can be easily implemented by adding additional flow block circuitries, and if necessary, additional network interface devices. This allows an existing unit to be scaled up as the traffic load grows.

In operation, a traffic stream is received and first identified by the CPU **12**, as it receives the first packet of a new traffic stream from a CPU input buffer **26** connected to the input interconnect path **31**. A free flow block **14** is selected to handle future packets of that traffic stream and all of the necessary information to handle the traffic stream, including the identification of the stream, is loaded into the pattern matching circuitry **16** and the control circuitry **18** of the selected flow block over the CPU bus **41**.

As each subsequent packet of the stream arrives at the packet switch interface circuit, it is handled by the network interface **30** (for ease of explanation it is generally assumed that the receiving network device will be an interface **30**) and flow block **14** without intervention by the CPU **12**. In particular, as it is received at interface circuit **30**, it passes through the network interface circuitry **30** and is placed on the input interconnect path **31** so that each flow block **14**, assigned to that interface, can check the packet, in parallel, to determine if any one of those flow blocks recognizes the packet as being assigned to it. If a match is found, the packet is accepted by that flow block and the data, usually modified by the control circuitry **18** of the flow block, is read and stored by the flow block. Further circuitry of control circuitry **18** will remove the packet from the data buffer **20** of the flow block **14**, with a new header prepended thereto, when the system is ready to send the packet over the next link of the data communications path.

Any packet which is not recognized by any of the flow blocks is available to the CPU from the one of the CPU input buffers **26** assigned for receiving data from that network interface. The CPU input buffer for each network automatically starts to copy each packet from the input interconnect path **31** each time a packet arrives, and continues to do so until one of the flow blocks **14** for that network interface accepts, or all flow blocks assigned to that network interface reject, the packet. If the packet was accepted by one of the assigned flow block circuitries, the portion of the data stored in the associated CPU input buffer **26** is discarded, and the CPU input buffer resets to await the next packet from that network interface. If the packet is rejected by those flow blocks assigned to that network interface, the associated buffer **26** passes the packet to the processor **12** which will analyze the packet and process it accordingly. It is important to note that no conflict arises from trying to put two packets into a CPU input buffer at the same time since each network interface has its own associated buffer **26** and a network interface **30** can receive only one packet at a time.

The CPU **12** further has access to the set of output buffers **32** (one buffer for each output network) over a bus **420**, through which it can send packets to the network interfaces over output interconnect path **52** for transmission along a link of the transmission chain.

Description of Detailed Elements

Network Interface

Data enters the packet switch from a network through the network interface. As noted above, these units are autonomous. They can be constructed either by building the special purpose hardware **400**, one for each network, which enables a network to connect directly with the respective interconnect paths, or by providing the standard adapter **42**, into which an existing off-the-shelf hardware network interface **40** can be inserted. The two classes of hardware can both be advantageously used in the same embodiment.

Referring to FIG. **1**, the second approach employs a standard network interface element **40** (typically an off-the-shelf commercially available circuitry) which connects over lines **41***a* (which is usually a standard bus) to the associated interface adapter **42**. Each adapter **42** has a standard interface connection which connects to the input interconnect path **31** for eventual connection to an as yet unknown one of the flow blocks **14** and to the network associated storage buffer **26**. The interface adapter also has a standard bus interface which connects to CPU bus **41**. The interface adapter **42** also provides a third interface for receiving packets from the flow blocks over the output interconnect path **52**. Adapter **42** provides those received packets, to the associated network interface **40** for transmission over a network path to the next network connection. The choice of this second interface approach is convenient and allows for modular expansion and network interface card interchangeability; however, use of the adapter **42** with a separate network interface **40** is likely to be more expensive than a special purpose network interface card **400**.

The choice of which network interface approach is adapted thus depends upon both cost and speed. The interface adapter **42** with its various bus connections is, most likely, the slower path unless the bus **41***a* is very fast; and even then, most current network interfaces for high speed networks cannot keep up with a network which is running at maximum capacity. Additionally, the use of several cards is likely to be more expensive. Accordingly, it may be desirable to provide the special purpose network interfaces, such as a special network interface **400**, which connect to interconnect paths **31** and **52**, for high volume networks where speed is more important; whereas the slower network interface, employing off the shelf components, can be employed where speed is not as important or where the construction of special purpose hardware is not cost justified.

The autonomous interface network unit is, as noted above, responsible, on the input side, only for ensuring that all packets destined for the switch are received from the network and are fed to the flow blocks **14** and storage buffers **26**. Congestion and control are the responsibility of the flow blocks **14** and the control devices **18** therein. Similarly, the output side of the network interfaces **30** needs only to read data packets sent by the flow blocks **14** and buffers **32**, and transmits them over the selected network.

It is also possible that inexpensive and slow network interfaces can be connected directly to the standard bus **41** and be run by the general purpose CPU **12** rather than by the interface adapter **42**. These packets would then be sent on whichever path the processor normally uses to send packets which it originates. This is an acceptable alternative, subject to the speed and time requirements imposed upon the central processor. The standard bus also provides the central processor unit with full access to the standard network interfaces **40** and special network interfaces **400** through the network adapter **42** so that any network interface can be controlled by the CPU **12** when unusual functions, such as problems with the transmission layer, fault isolation, or other complex testing tasks must be performed.

The Interconnect Path

As noted above, each interface adapter **42** or special network interface **400** connects to each of the flow blocks **14** in a most general form of illustrated structure. Depending upon the economics and speed desired, the interconnect circuitry can take a variety of forms using a number of techniques which are already known in the art. One particular approach, using "brute force," is to use a full crossbar switch to make all possible connections between each of the network interface adapters and each of the flow blocks, both on the input and the output interconnect paths. As the flow blocks are assigned, and reassigned, between interface adapter units and special network interfaces, the various points of the crossbar can be opened and closed.

An alternate approach, used in digital telephone systems, is to interface all of the functional units to a high speed, time division, multiplexed bus. This approach requires less switch hardware but necessitates a bus speed comparable to the maximum speed of an interface times the number of interfaces. Such speed requirements may make it less economical to build such a bus than might otherwise appear.

The input interconnect path is conceptually simple in that flow blocks **14** are assigned to but a single network interface at a time. The relationship is not symmetrical, however. The input network interface thus feeds at most one input packet at a time to the flow blocks; however, the input packet can be read by many different flow blocks, all of which are assigned to that network interface. The output side of the flow blocks is slightly more complex since several flow blocks, each connected to a different network interface at its input, may present a packet to the same output network interface simultaneously. The output interconnect must thus have some method for choosing which, and in what order, to send the packets. For example, it can service the flow blocks in the same order specified by the CPU when the processor sets up the traffic stream; or preferably, a grant passing ring configuration can be employed. It is important, however, to be sure to allocate appropriate bandwidths to each stream so that acceptable operation is attained. There are various concepts for performing this function, well known to those practiced in the art, and they will not be discussed here in any further detail.

The Flow Blocks 14

Each flow block **14** consists, as noted above, of a pattern matching circuit, the flow block data buffer **20**, and the control device **18**. The pattern matching hardware, in the illustrated embodiment of the invention, contains two variable length bit strings: a "match" bit string and a "mask" bit string. Those fields in the packet which can vary among the packets of a single stream, are set "on" in the "mask" string. Values in these bits are then ignored. The values in the fields which identify a stream, and which are always present in a packet of the stream, are stored in the "match" bit string. Several functions can thus be performed by the pattern matching circuitry **16**, in addition to merely checking the assignment of a packet to a traffic stream. For example, certain error checks (for valid values) can be performed. Also, since a flow block **14** is assigned by the CPU **12** to forward a traffic stream only if a route for the traffic stream exists, and if the traffic stream is authorized by the access control software in the CPU **12**, a match by the circuitry **16** immediately implies that the packet is allowed by the access control to pass through the switch, and that a route for the packet exists.

The data buffer **20** of a flow block can be implemented in a variety of ways. The simplest approach, is to have associated with each flow block a separate memory array having head and tail registers to allow reading and writing. Two disadvantages to this approach are that it requires additional hardware and the buffer memory is split up into many small memory banks rather than employing a single large memory bank for the entire packet switch.

Nevertheless, the use of a large memory bank, from which each flow block buffer memory is allocated, results in a complex storage management problem. It is necessary in such a memory structure to maintain a list of unused blocks, a mapping of the used blocks, etc. In addition, the flow control mechanism must be more complex, particularly if there is less total buffering than the sum of the maximum storages of all of the data streams. It must therefore deal with a global resource shortage of buffer memory in the switch. This problem can thus remove a primary advantage of having a large memory bank. In addition, with separate memory banks, each bank need only be able to support two simultaneous accesses: a read and a write. With a single

large bank, all of the network interface accesses must be handled simultaneously.

A number of practical operating problems exist with the circuitry illustrated in FIG. 1. Thus, if there are more identified traffic streams than there are flow blocks, or if a single packet stream is to be routed over multiple paths by the network protocol, appropriate hardware must be available to deal with the various circumstances. In particular, if there are more identified traffic streams than there are flow blocks **14**, it is important to avoid "thrashing" as the streams compete for the flow blocks. If the protocol has adequate flow and congestion control mechanisms, these can be used to inhibit the excess streams. Also, the flow blocks should be packaged and interfaced to the rest of the system so that additional flow blocks can be installed as load patterns change or as switches experience higher usage rates than they are able to handle.

Further, the software can maintain a record of the streams including the time when each flow block was last used, so that periodic scans can be made by the software to find flow blocks which are associated with streams that are no longer active and list those flow blocks as ready for reuse. Further, the software can maintain a record of the stream parameters so that if a previously terminated stream should restart, it would not be necessary to recompute everything. Preferably, the CPU stores this information in its local memory.

It may also be desirable to avoid assigning a stream to a flow block until a minimum number of packets relating to a stream have been counted. In this instance the CPU **12** can maintain the necessary information regarding the stream (and pass the packets of the stream on to the next network node) and dedicate a flow block to that stream only after the length of the stream has been assessed.

There are also instances when a single packet is forwarded over multiple paths. The situation can thus exist when packets of the same data stream are received over two different network interfaces and/or where a single packet stream must be divided and forwarded to two or more output networks. The first problem can be handled simply by allocating one flow block to each interface. The second problem is somewhat harder to handle; however, in most protocols, there is a sequence field in each packet wherein it is possible to assign two different flow blocks to the stream in which the sequence field was masked out except for, for example, the lowest bit. In one flow block the bit would have to match to "zero" and in the other flow block to "one." Thereafter, each flow block can be assigned to a different output stream, the split being roughly into two divisions. More complex and controlled splitting requires more sophisticated mechanisms to effect proper queuing and sequencing on the output.

The Flow Block Circuitry

In the description of this particular embodiment, the width of the various buses, the number of identical interface units or flow blocks, the length of a counter, etc., are subject to the particular switching system environment and capacity. Accordingly, while specific numbers are suggested from time to time, the values "N", "n", "P", etc. are variable and may be equal to each other or not as the particular embodiment requires.

Referring to FIG. 1, the flow block control device circuitry **18** effects bookkeeping functions at the protocol level and flow and congestion control. One functional unit **19***a* of each control circuitry **18** strips the input header from a packet before it enters the flow block data buffer **20** and another functional unit **19***b* of the control circuitry prepends the output header to the data packet before it exits the flow block data buffer.

In addition, each protocol tends to have certain bookkeeping functions which must be performed on each packet of the stream; however, these functions tend to be slightly different for each protocol. The design of the illustrated control device provides flexibility for handling the different protocols, including in particular the capability of computing the checksum (and signaling an error should one occur), and updating the "hop" count. The control circuitry **18** also needs to be flexible enough to handle the different protocols in a very short time. Accordingly, the design preferably allows for additional circuitry to be added to the protocol function circuitry **19***a* and **19***b*. The additional circuitry can also be implemented in the state machine controller for the flow block.

The flow block control circuitry also acts as a flow control agent. Thus, if packets are entering the flow block at too fast a rate, an error is caused. The specific hardware configuration depends on the protocol architecture and the policy implemented therein. One effective mechanism uses the error alarm signal to show that the flow block buffer is filled past a selected level. The control circuitry also needs to set a so-called "discard" level. This is necessary to protect the congestion control mechanism of the switch from being overloaded by a single, out of control, data stream. Such a mechanism would cause a selectable percentage of the incoming packets of a stream to be ignored completely rather than passed, over bus **41**, to the congestion control mechanism of the CPU **12**, which it could overload.

The Interconnect Path Operation

In the illustrated embodiment of the invention, the presently preferred embodiment of the interconnect paths **31** and **52** uses the simple, brute force, approach; that is, a full cross bar is provided for each interconnect path by assigning a separate bus to each network interface adapter **30**, to which each of flow blocks **14** and buffers **26** is connected. Each bus has a full set of control lines for, and in addition to, the data lines of the bus. The illustrated interconnect circuitry thus consists of a set of, for example, "N" identical buses. The interconnect further can include some general signal lines such as, for example, a system reset line. The full cross bar is also large enough to support the maximum complement of interface circuitries simultaneously, each interface being able to proceed at full speed with no buffering.

Considering in particular the input interconnect **31**, there are R buses, "R" being equal to the sum of the number of special network interface units **400** and interface adapter **42**. Each interface data bus is "M" bits wide, and is driven only by the associated network adapter **30** or interface **400**. In addition to the data from each network interface **30**, each bus also has a plurality of control signals for controlling the transfer of the incoming packets from the network to the flow blocks **14** and buffers **26**. The control signals allow a flow block **14** to indicate to the associated CPU input buffer **26** (and CPU **12**) whether the packet has been accepted.

Referring to FIG. 2, the control signal functions can be performed with two lines, both driven by the network interface or adapter and "listened" to by all of the flow blocks assigned to that network (including the

corresponding CPU input buffer 26 assigned to that network). One line 200 indicates when a packet is being transferred and is active for the duration of the packet transfer. A non-active period 202 has a minimum time duration so that the next active period (and the next packet) is spaced in time from the previous active period. The beginning of the inactive period indicates the end of the packet. A second line 206 is a "word transfer" line and each active transition 208 on that line signals that a new word (a byte in the illustrated embodiment) of data has been placed on the bus by the network interface.

There is further a common control line 210 which can be driven by any of the flow blocks 14 and listened to by the CPU input buffer 26 for that network. When going active, it signals to the CPU that the current packet has been accepted by a flow block and the packet may thus be ignored by the CPU 12. The timing must be carefully controlled, especially if faults are to be detected. For example, if the packet length in the protocol header is to be verified, it is necessary to wait until the entire packet has been received before accepting the packet. However, by that time, the next packet is starting. This problem also arises when verifying header check sums for packets with no data. The timing can be resolved by having the accept line driven at a time during the mandatory non-active period of the packet line, that is, after the packet has completely arrived and before the next packet begins.

Referring to FIG. 3, the output interconnect 52 has a slightly more complex data bus. The bus is "P" bits wide and is driven by a sequentially varying one of the flow blocks 14 and buffers 32 (the "driving circuits") assigned to the connected network interface. The output of the driving circuit is read by the associated network interface 30 or 400. Preferably, the driving circuits are arranged as, and include the circuitry to form, a grant-passing ring. In addition, there are other control lines which are used to control the transfer of the packet from the drive circuit having the grant. These other lines 218, 220 are substantially the same as those control lines 200, 206 of the input interconnect bus. After a packet has been transferred to a network interface, the "grant" advances to the next driving circuit. If the identified driving circuit has a packet waiting at the time the grant line becomes active (typically the rising edge), it begins a transfer. Otherwise, the grant is passed to the next driving circuit which repeats the process.

Flow Block Details

As noted above, the flow blocks 14 has several major functional units. The stages, in the illustrated embodiment, are connected asynchronously since the throughput of the stages is not constant and some stages have buffering between them. Referring to FIG. 4, the circuit structure of flow block 14, considered in more detail, has an input multiplexor 250 which selects the current input bus and passes the data to both the pattern matcher 16 and the rest of the flow block. The pattern matcher, as noted above, examines the header of the incoming packet. If it matches the pattern to be handled by this flow block, the match is indicated by a signal over a line 252 to the control device logic 18.

Simultaneously, data from the input bus flows through a stripping circuit 254 which includes a counter and which discards the first "n" bytes of data (the header) allowing the remainder of the packet to pass through unmodified. The packet then passes to the control logic 18 where the higher level protocol functions such as check sum computation and hop count modification occur. The control logic 18, pattern matcher 16, and stripping circuit 254 have all been previously loaded with other necessary data from CPU 12 over bus 41. The input to the control device has a small amount of buffering to allow the control device to take more than one cycle when processing certain bytes in the data stream. The packet passing through this stage of processing may be modified; for example, this stage may abort further processing of the packet if an error is found, as described in more detail below. The packet then passes to a counter/truncate circuitry 260 which contains a counter loaded by the control logic over circuitry 262. The counter serves two functions: any unused trailer in the packet is discarded, and, if the packet is truncated, an error flag is raised over a line 264. The next stage of processing, a circuitry 266, prepends "n" bytes of data, the new output header, loaded from the CPU 12 in a similar manner to stripping circuit 254, to the packet as it passes therethrough. It also contains some buffering on the input to allow the new packet header to be inserted. In those instances where the new packet is substantially larger than the old one, the buffering is a necessity. The packet next passes to the output data buffer 20 which consists of a dual port (one read-only and one write-only) memory, along with a control logic 268 to keep track of the packets in the buffer. The buffer 20 is organized in a ring structure and a hardware queue of "t" buffer pointer/size pairs keeps track of the utilization of the buffer. Additional control circuitry within the buffer keeps track of the current start and end of the "free space". The packet then passes to an output multiplexor 274 which has output bus control logic and a set of drivers, one for each output bus in the output interconnect 52. When the flow block receives the "grant," for the appropriate output network interface 30, as described above, packets which are in the output buffer are read out and passed along the bus. Throughout the flow block, there are, in addition, data paths 276 which allow the CPU 12, over bus 41, to load memories, etc. in order to maintain proper operation of the flow block.

Referring to FIG. 5, the pattern matcher 16 has two small memories 60, 62 each "a" bits wide and "b" bytes long. In the illustrated embodiment, 8×256 bit RAM's are employed. One memory 62 contains the "masked" bits and the other memory 60 contains the "match" bits. More precisely, for those header positions for which a bit is "on" in the mask memory, the packet can have any value in the header whereas, if a bit is "off" in the mask memory, those corresponding bits in the packet header must match the CPU predetermined values stored in the match memory.

The pattern matcher can operate with varying quantities of data in the memories 60, 62, and if all the mask "off" bits in the header match the "match" memory bits, the header is a "match", as indicated over line 252, and the flow block continues to read the packet. In the illustrated embodiment, an "n" bit counter 280 is reset over a line 282 when the packet begins arriving and counts up "one" for each byte received from the bus. The output of the counter over lines 284 is used as an index into the two memories and is directed, also, to an "n" bit comparator 286. Comparator 286 compares the output of counter 280 with the output of an "n" bit latch 288 which holds the current header size count. When

the count reaches the header count, a header complete signal is generated over a line 290.

The comparison of the input header to the match word is effected byte-by-byte, using an eight bit comparator 294 and a series of eight identical two-to-one multiplexors 296. The output of the match memory is one input of the identical two-to-one multiplexors 296 with the "n" bits (typically eight bits) from the data bus 292 as the other input. In each multiplexor, the select input is driven by the corresponding output bit over lines 292 of the mask memory; so that if a mask bit is "off", the data bus bit is selected. Otherwise, the match bit is selected. The "n" selected bits are then fed into the "n" bit (illustrated as eight bits) comparator 294 which, as noted above, receives the original match data word as the other input.

The output of the comparator is fed to a flip flop 298 which is set by a signal over a line 299 when the packet begins to be read. If any byte of the header fails to have a complete match (of the selected bits), the output of the comparator goes low and clears (resets) the flip flop. If the flip flop is still set when the counter 280 has also reached a match (the end of the header), the packet header is accepted and the logical "AND" circuit 300 provides the match indication signal over line 252.

In addition, the pattern matcher further contains data pads, not shown, which allow the CPU 12 to load (through bus 41) the match and mask memories 60, 62, the length latch 288, and other data as well.

Referring now to FIG. 6, the data stream control unit 18 (and stripping circuitry 254) has an arithmetic logic unit (ALU) 310, special purpose logic which will be described hereinafter, and a control table stored in a memory 312. The ALU and the control store act like a general purpose microcode engine, but one which has been specialized to create a very minimal, high speed processor for packet headers. The functions to be performed, as described above, are very limited.

The illustrated circuitry allows the processing of the headers in the transmission time of a complete packet having no data, thus allowing the flow block to operate at full network bandwidth with minimum sized packets. In addition, the control device keeps its required cycle time as high as possible (that is, operates as slow as possible) to keep its costs down. .

In the illustrated typical circuitry, the control table 312 is the heart of the control device. It consists of an array of words, which are horizontal microcode, used to directly control the ALU and gates of the control circuit as is well known in the art. While some fields of the control word will perform standard operations, such as selecting which ALU operation is to be performed on each cycle, other fields will control the special logic associated with packet forwarding.

The illustrated control circuitry further includes a control counter 314 which is set at the start of each packet. The counter selects one of the control words in the control array (the output of the control word controlling the logic elements of the control device). While processing a packet, this counter is incremented at the cycle speed of the control device, essentially asynchronous to the rest of the system, thereby stepping through the control table at a fixed rate. The input data packet flows through an input FIFO buffer 320, the output of which is controlled by a bit in the control table 312. If the bit is "on," a byte is read out of the input buffer. This function, which is thus not performed automatically when data is read from the buffer, allows data to be passed through under control of the local processor, and allows certain bytes of the packet to be operated on by more than one control word without the necessity of storing the byte in an intermediate location. A second counter 322, cleared at the start of each packet, counts the current data byte and provides that count for use by the rest of the control device 18.

Another bit of the control word from array 312, effectively disables the control device, thereby allowing the rest of the packet to pass through to the next stage of processing. This bit is set in the last control word of the process sequence, that is, once processing of the header has been completed. Another field of the control word controls the logic which cancels the packet if certain conditions are true. This field is thus used to cancel processing of the packet when fatal errors are detected.

The control circuitry also includes several scratch pad registers 330. These registers allow accumulation of results etc., and provide constants for use by the ALU 310. The scratch pad registers 330 can be loaded by the CPU 12 during that process by which the CPU selects a flow block to receive a data packet stream.

The apparatus further has a multiplexor 340 to allow selection from the variety of data sources available as inputs to the ALU. The results of the ALU processing can be sent to a number of circuitries. In particular, inputs to the multiplexor 340 come from either the input data buffer 320, count register 322, or the scratch pad registers 330. Data may be written from the ALU 310, through a shift/rotate register 311, to either the scratch pad registers, or output from the control unit through an output multiplexor 342. Further, a pass around path 343 allows the result of an ALU calculation to be sent to a register while a data byte is sent to the output. Other data paths not shown are available which allow the CPU 12 to load the control table, the scratch pad registers 330, the counters 314, 322, etc. when a flow block is selected to receive a data packet stream.

As noted above, the illustrated embodiment provides for a flexible flow block configuration which, when loaded from CPU 12 with protocol setting data signals, enables the flow block to handle a particular one of a plurality of packet stream protocols. In an alternative embodiment of the invention, each flow block can have implemented therein, in hardware, the necessary circuitry to enable it to handle one (or more) particular protocols. Accordingly therefore, different hardware modules would be needed for different protocols; however, some speed advantage can be obtained by reducing the flexibility of the hardware controlled flow block.

In addition, further circuit efficiency can be obtained, without loss of flexibility, if those flow blocks which can be assigned to a particular interface share the same ALU circuitry (FIG. 6). Recalling that ALU 310 operates to process an incoming data packet, and, since only one data packet can be received from a network at a time, all of the flow blocks assigned to a particular network interface can then share the same ALU since only one of the assigned flow blocks will be active for receiving a data packet at any particular instant. This savings in circuitry can, for example, be advantageously implemented when a plurality of flow blocks are provided on the same card module. In that configuration, all flow blocks of a card module which share an ALU should be used in connection with the same selected network interface, and in particular, as noted above, the

card module may be implemented fully in hardware with different flow blocks of the card module being used for different protocols.

Additions, subtractions, deletions and other modifications to the illustrated embodiment of the invention will be apparent to those practiced in the art and are within the scope of the following claims.

What is claimed is:

1. A high speed data packet switching circuit comprising:

a software controlled primary processing units,

a plurality of network interface units for receiving incoming data packet streams and for transmitting outgoing data packet streams, each of said data packet streams having a selected protocol and all of the data packets in a said stream having the identical protocol,

a plurality of data stream control circuits for concurrently receiving at least a portion of a header of the data packets and selectively processing the received packets only wherein each said data stream control circuit processes the data packets of one data stream having one of said selected protocol in response to previously generated electrical signals from the primary processing unit based upon header identification information in the at least first data packet of the new data packet stream for designating and initializing one of said data stream control circuits to process a remainder of the data packets of the new data packet stream,

means for interconnecting said primary processing unit, said plurality of interface units and said plurality of data stream control circuits,

said primary processing unit receiving from said network interface units, and for processing, at least a first one of the data packets of a new data packet stream and having means for generating said electrical signals means in each said designated and initialized data stream control circuit for receiving and processing only those data packets which include said header identification information upon which said designated and initializing is based.

2. The packet switching circuit of claim 1 further wherein each data stream control circuit comprises

a pattern machining circuit responsive to pattern setting signals from the primary processing unit and to incoming data packets from said network interface units for identifying and receiving a packet stream which will be processed by said control circuit,

a processing unit responsive control circuit for controlling, in response to control signals sent by the primary processing unit, a congestion control means, and a header stripping and prepending functions means for the data stream control circuit, and

a data buffer responsive to said pattern matching circuit and the processing unit responsive control circuit for receiving and storing data and protocol elements for an incoming data packet stream and for outputting a data packet stream to a said network interface unit to be forwarded to a next network node.

3. The packet switching circuit of claim 2 wherein said pattern matching circuit comprises

a mask bit memory,

a match bit memory, and

means for comparing data bits of incoming data packets, not masked by a data word from the mask bit memory with an output of the match bit memory for determining the validity of an incoming data stream packet.

4. The packet switching circuit of claim 2 wherein said pattern matching circuit comprises

a match memory

a mask memory,

a comparator circuitry, and

means for inputting, to the comparator circuitry, data bits from the match memory and corresponding data bits from an input packet, said corresponding data bits being selected in accordance with the bit values in the mask memory, for determining the acceptability of an input packet.

5. The packet switching circuit of claim 4, wherein said pattern matching circuit further comprises

means for determining the end of an input header for an input packet,

to the comparator circuit for determining whether all of the matched bits in the input header are valid, and

means for providing an acceptance signal in response to a valid output of the comparator responsive means and the header determining means.

6. The high speed data packet switching circuit of claim 2 wherein the processing unit responsive control circuit comprises

a table array storage for storing horizontal microcode,

a control counter for selecting words of the table array storage,

an arithmetic logic unit, and

means for controlling operation of the processing unit responsive control circuit using horizontal microcode output of the table array memory.

7. The packet switching circuit of claim 1 wherein said data stream control circuit comprises

an input multiplexor for selecting a data packet stream source from among the interconnecting means accessible to the control circuit;

a pattern matching circuit responsive to pattern setting signals from the primary processing unit and to incoming data packets from the input multiplexor for identifying those data packets which will be processed by the control circuit.

a header stripping circuitry for removing the header from each data packet from the input multiplexor.

control logic, responsive to the pattern matching circuit and to the stripping circuitry, for passing the data packet, without the header, for further processing by the control circuit,

a counter/truncator circuit for determining whether the data packet from the control logic is truncated and for providing an error signal in the event the packet is truncated,

a prepend circuitry for adding a new header to the data packet from the counter/truncator circuit,

an output data buffer for buffering the data packet from the prepend circuitry and responsive to a buffer control logic, for maintaining accurate status data regarding the contents of the buffer, and for outputting a next data packet for transmission over a network, and

an output demultiplexor connected to the output data buffer for transmitting data from the output data buffer over the output interconnecting path.

8. The packet switching circuit of claim 1 further wherein said network interface unit comprises

a network interface circuit for communicating with a network channel in accordance with a said selected protocol and delivering data from the channel in a predetermined format, and

an interface adapter for receiving data from the channel through the network interface circuit in said predetermined format and for transmitting that data from the channel over the interconnecting means to said data stream control circuits and said primary processing unit, for receiving data, to be sent over a network channel, over said interconnecting means from the data stream control circuit and the primary processing unit and for delivering data received from said interconnecting means to said network interface circuit for transmission over a said network channel.

9. The packet switching circuit of claim 8 wherein said network interface unit further comprises

a single network special purpose hardware interface circuit having

means for communicating with a network channel,

means for transmitting received network data over the interconnecting means to said data stream control circuits and said primary processing unit,

means for receiving network data packets from the data stream control circuits and the primary processing unit, and

means for processing the received data packets for transmission over a network channel.

10. The packet switching circuit of claim 1 wherein said software controlled primary processing unit further includes

a central processing unit,

a bus means;

a plurality of input storage units for selectively receiving ones of said plurality of data streams from the network interface units and each storage unit having its output connected to said bus means,

means for connecting the central processing unit to said bus means,

a plurality of output storage units for selectively receiving data from said central processing unit over said bus means, and for providing said data to said network interface units, and

means for controlling the input of data to said input and output storage units.

11. The packet switching circuit of claim 1 wherein said interconnecting means comprises

an input bus for interconnecting the outputs of said network interface units, the inputs of said data stream control circuits, and the primary processing unit, and

an output bus for interconnecting the outputs of said data stream control circuits, the inputs to said network interface units, and the primary processing unit.

12. The packet switching circuit of claim 11 wherein said interconnecting means further comprises a central processing unit bus interconnecting said data stream control circuits, said network interface units, and a central processing unit of said primary processing unit.

13. The packet switching circuit of claim 12 wherein said input and output bus means each comprises data lines and control lines.

14. A high speed data packet switching method for switching data packet stream among communication paths comprising the steps of

receiving each packet stream from one of a plurality of networks,

processing at least a first packet of each received data packet stream using a software controlled, primary processing unit,

designating that performance of routine, repetitive header processing of the further packets of one of said received packet steams, said processing including packet forwarding processing to effect routing of said packet,

receiving and examining by each said high speed hardware circuitry at least a portion of each packet of each said received data packet stream, determining based on said examination of said at least a portion of each packet by each of said high speed hardware circuitry, which said high speed hardware circuitry has been designated to process each further packet of each received data packet stream, receiving in said designated high speed hardware circuitry said each further packet.

15. The high speed data packet switching method of claim 4 further comprising the step of

controlling at leat the initialization of a said high speed hardware circuitry assigned to process a packet stream from the software controlled, primary processing unit.

16. A high speed data packet switching method comprising the steps of

receiving incoming packet streams from network interface units;

processing ones of the received data packets in response to a software controlled primary processing unit using a plurality of hardware data stream control circuits,

interconnecting the primary processsing unit, the interface units, and the data stream control circuits for communications therebetween,

processing at least a first one of the data packets from the receiving step for each new data packet stream in the primary processing unit,

identifying, using the primary processing unit, one of the data stream control circuits for processing the incoming data packet stream,

determining by each said data stream control circuit the one data stream control circuit which will process each packet of that portion of said incoming data packet stream which is not processed by said primary processing unit,

processing that portion of a said data packet stream which is not processed by said primary processing unit by said identified data stream control circuit, and

outputting the results of the data stream control circuit processing and the primary processing unit processing to form an output data stream for transmission along a communications path.

17. A high speed data packet switching circuit for receiving data packet streams from a plurality of input communication paths and for transmitting data packet streams to a plurality of output communication paths, said circuit comprising

a plurality of network interface units for receiving the incoming data packet streams and for transmitting outgoing data packet streams,

a software controlled primary processing unit, having a bus means,

a central processing unit,

a plurality of input storage units for receiving respectively each of said plurality of data streams from the network interface units and each input storage unit having its output connected to said bus means,

means for connecting the central processing unit to said bus means, and

a plurality of output storage units for receiving data from said central processing unit over said bus means, and for providing said data to said network interface units,

a plurality of data stream control circuits for manipulating data packet stream in response to the primary processing unit,

said data stream control circuits comprising

a pattern matching circuit responsive to pattern setting signals from the central processing unit and to incoming streams of data packets from said network interface units for identifying a data packet to be processed by said control circuit,

means for transferring identified data packets to said control circuit,

a processor responsive control circuit for controlling, in response to control signals sent by the primary processing unit, means for congestion control, and means for header stripping and prepending functions for the data stream control circuit, and

a data buffer responsive to said pattern matching circuit and the processor responsive control circuit for storing an incoming data packet stream from said control circuit and for outputting a stored data packet stream to be forwarded to a network interface unit,

means for interconnecting said primary processing unit, said plurality of network interface units and said plurality of data stream control circuits, and

said primary processing unit receiving from said network interface units at least a first one of the data packets of each new data packet stream and having means for designating those data packets of the stream which are not processing by the primary processing unit to be processed by a said data stream control circuit without further processing by said primary processing unit.

* * * * *

US005511213A

United States Patent [19]

Correa

[11] Patent Number: 5,511,213

[45] Date of Patent: Apr. 23, 1996

[54] **ASSOCIATIVE MEMORY PROCESSOR ARCHITECTURE FOR THE EFFICIENT EXECUTION OF PARSING ALGORITHMS FOR NATURAL LANGUAGE PROCESSING AND PATTERN RECOGNITION**

[76] Inventor: **Nelson Correa**, Carrera 6a No 57-11 Apt. 402, Santa Fe de Bogota, D.C., Colombia

[21] Appl. No.: **880,711**

[22] Filed: **May 8, 1992**

[51] **Int. Cl.**[6] **G06F 15/38**

[52] **U.S. Cl.** **395/800**; 395/700; 364/253; 364/274.8; 364/DIG. 1

[58] **Field of Search** ..... 395/800, 700; 364/253, 274.8, DIG. 1

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,686,623	8/1987	Wallace	395/700
4,914,590	4/1990	Loatman et al.	364/419.08
4,994,966	2/1991	Hutchins	364/419.08
5,105,353	4/1992	Charles et al.	395/700
5,239,298	8/1993	Wei	341/51
5,239,663	8/1993	Faudemay et al.	395/800

Primary Examiner—Alyssa H. Bowler

Assistant Examiner—John Harrity

Attorney, Agent, or Firm—Beveridge, DeGrandi, Weilacher & Young

[57] **ABSTRACT**

An associative memory processor architecture is disclosed for the fast and efficient execution of parsing algorithms for natural language processing and pattern recognition applications. The architecture consists of an associative memory unit for the storage of parsing state representations, a random access memory unit for the storage of the grammatical rules and other tables according to which the parsing is done, a finite state parsing control unit which embodies the chosen parsing algorithm, and a communications unit for communication with a host processor or external interface. The use of associative memory for the storage of parsing state representations allows the architecture to reduce the algorithmic time complexity of parsing algorithms both with respect to grammar size and input string length, when compared to standard software implementations on general purpose computers. The disclosed architecture provides for a fast and compact computer peripheral or system, particularly when physically realized in one or a small number of integrated circuit chips, and thus contributes to the technical feasibility of real time applications in speech recognition, machine translation, and syntactic pattern recognition.

7 Claims, 5 Drawing Sheets

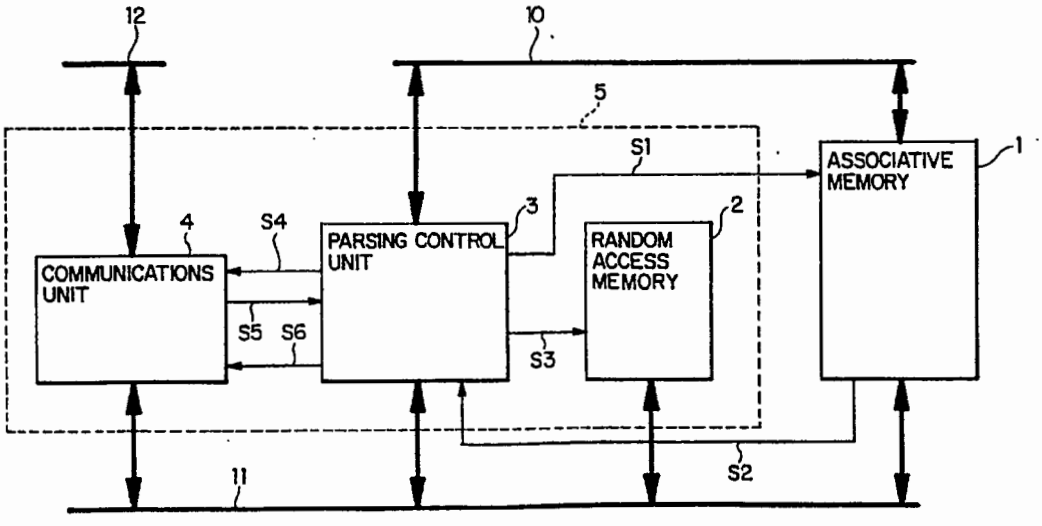
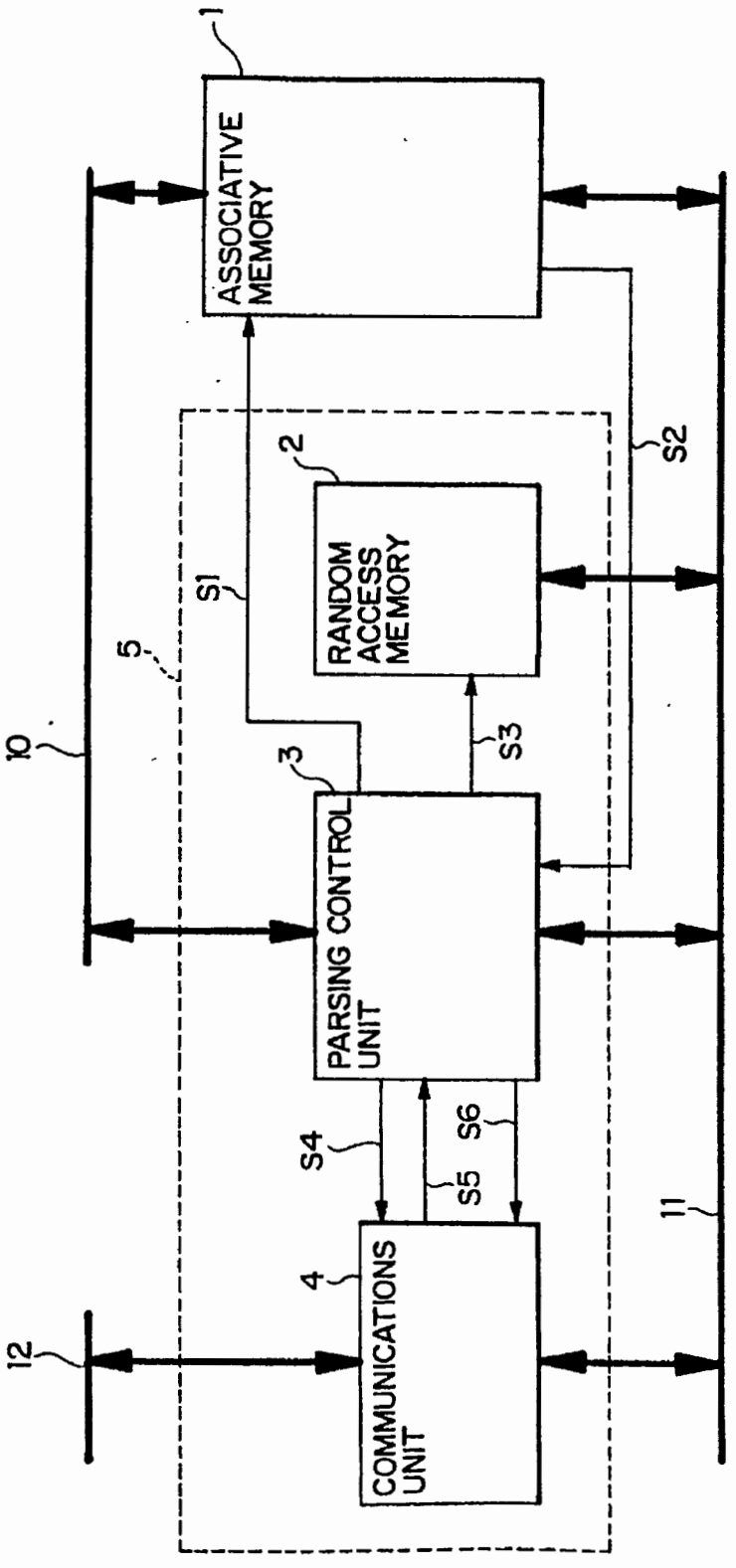


FIG. 1


1
ASSOCIATIVE MEMORY
2
RANDOM ACCESS MEMORY
3
PARSING CONTROL UNIT
4
COMMUNICATIONS UNIT
5
10
11
12
S1
S2
S3
S4
S5
S6

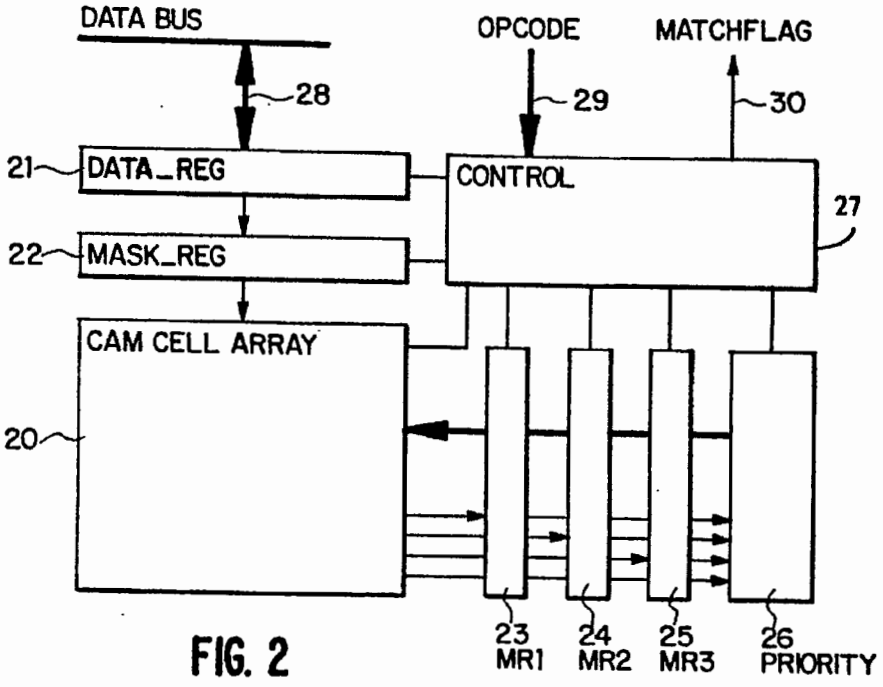

DATA BUS
28
OPCODE
29
MATCHFLAG
30
21
DATA_REG
22
MASK_REG
CONTROL
27
CAM CELL ARRAY
20
23
MR1
24
MR2
25
MR3
26
PRIORITY

FIG. 2

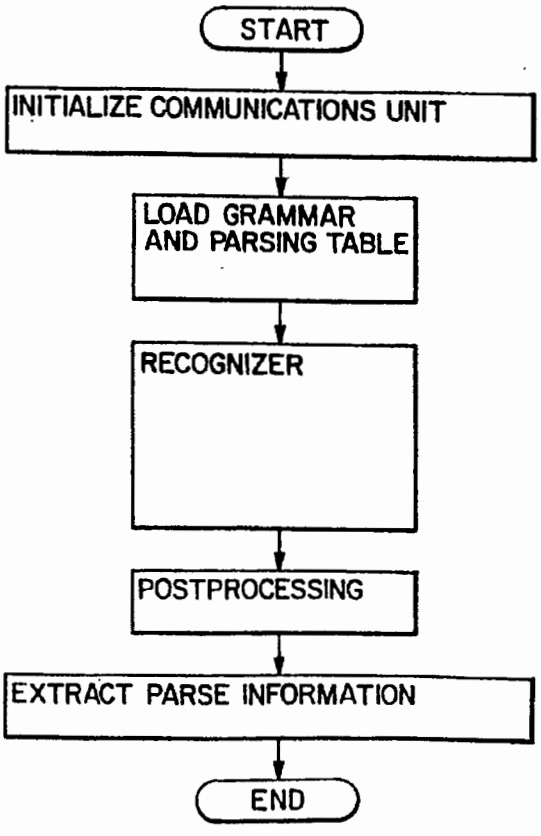

START
INITIALIZE COMMUNICATIONS UNIT
LOAD GRAMMAR AND PARSING TABLE
RECOGNIZER
POSTPROCESSING
EXTRACT PARSE INFORMATION
END

FIG. 6

FIG. 3

NUMBER	RULE
0	Z → S $
1	S → NP VP
2	NP → "JOHN"
3	VP → "THINKS"

"JOHN THINKS $"
0 1 2 3

FIG. 5

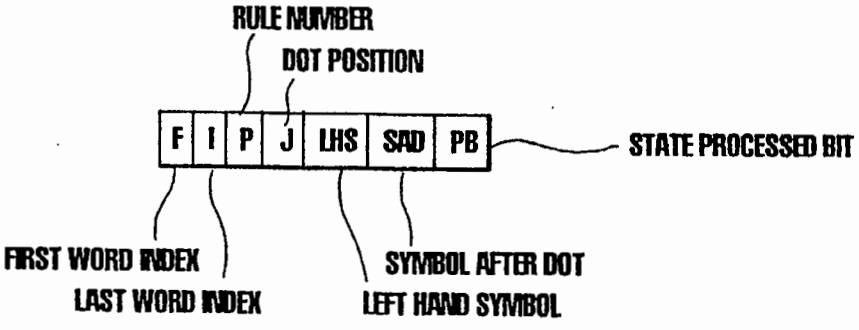


FIG. 7A

CAM

<0, 0, 0, 0, Z, S, 0>
. . .

FIG. 7B

CAM

<0, 0, 0, 0, Z, S, 1>
<0, 0, 0, 0, S, NP, 0>
. . .

FIG. 4

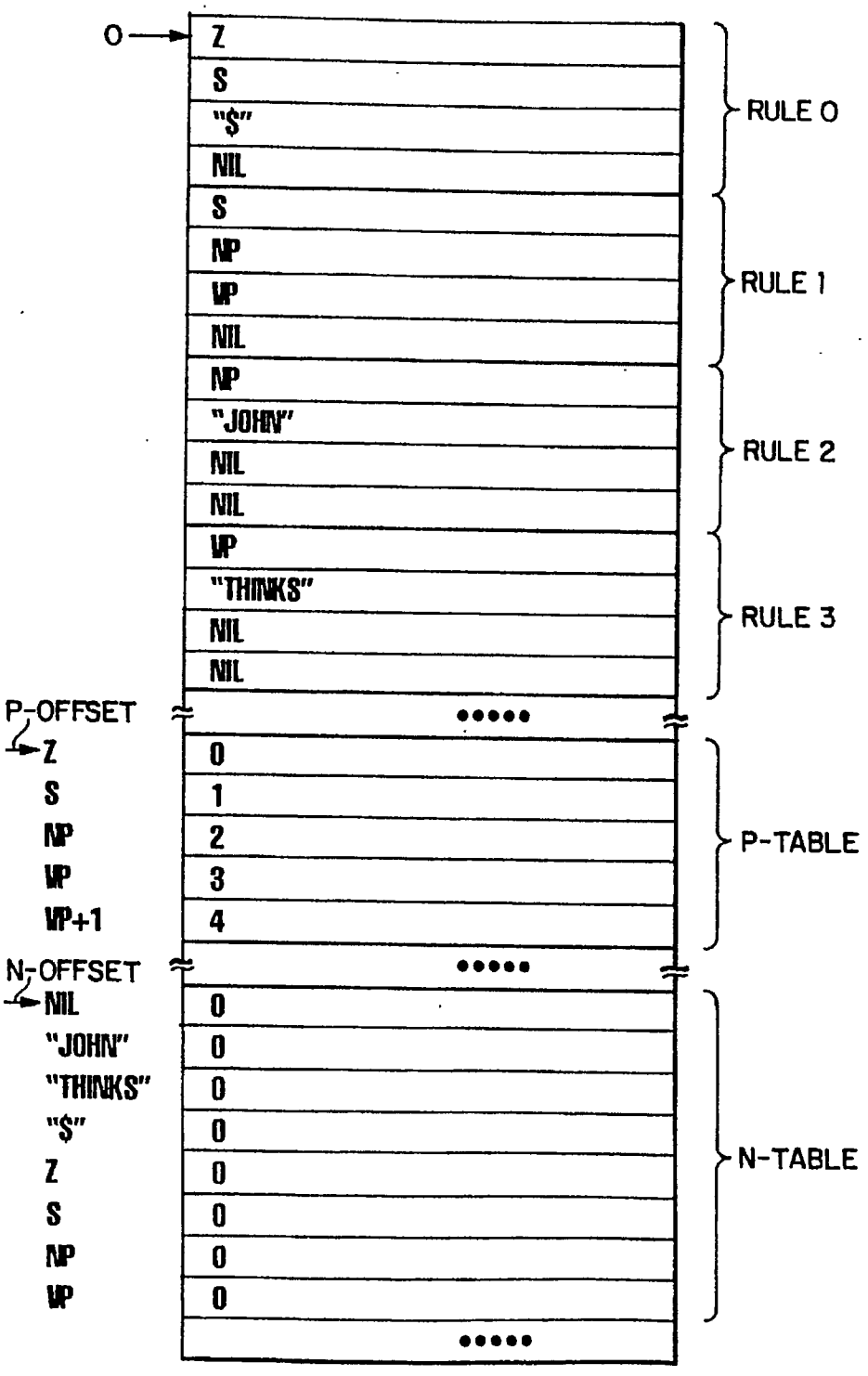

0
Z
S
"$"
NIL
S
NP
VP
NIL
NP
"JOHN"
NIL
NIL
VP
"THINKS"
NIL
NIL
RULE 0
RULE 1
RULE 2
RULE 3
P-OFFSET
Z
S
NP
VP
VP+1
0
1
2
3
4
P-TABLE
N-OFFSET
NIL
"JOHN"
"THINKS"
"$"
Z
S
NP
VP
0
0
0
0
0
0
0
0
N-TABLE

FIG. 7C

"JOHN THINKS $"

CAM	ITEMS	ACTION
<0, 0, 0, 0, Z, S, 1>	Z ----- •S"$"	PREDICT
<0, 0, 1, 0, S, NP, 1>	S ----- •NP VP	PREDICT
<0, 0, 2, 0, NP, "JOHN", 1>	NP ----- •"JOHN"	EXAMINE
<0, 1, 2, 1, NP, NIL, 1>	NP ----- "JOHN"•	COMPLETE
<0, 1, 1, 1, S, VP, 1>	S ----- NP • VP	PREDICT
<1, 1, 3, 0, VP,"THINKS",1>	VP ----- • "THINKS"	EXAMINE
<1, 2, 3, 1, VP, NIL, 1>	VP -----"THINKS"•	COMPLETE
<0, 2, 1, 2, S, NIL, 1>	S ----- NP VP•	COMPLETE
<0, 2, 0, 1, Z, "$", 1>	Z ----- S • "$"	EXAMINE
<0, 3, 0, 2, Z, NIL, 1>	Z ----- S "$" •	ACCEPT

ASSOCIATIVE MEMORY PROCESSOR ARCHITECTURE FOR THE EFFICIENT EXECUTION OF PARSING ALGORITHMS FOR NATURAL LANGUAGE PROCESSING AND PATTERN RECOGNITION

BACKGROUND OF THE INVENTION

The present invention relates broadly to computer hardware architectures using parallel processing techniques and very large scale integration (VLSI) microelectronic implementations of them. More particularly, the invention relates to an integrated associative memory processor architecture for the fast and efficient execution of parsing algorithms used in parsing intensive and real time natural language processing and pattern recognition applications, including speech recognition, machine translation, and natural language interfaces to information systems. Parsing is a technique for the analysis of speech, text, and other patterns, widely used as a key process in contemporary natural language processing systems and in syntactic pattern recognition for the identification of sentence structure and ultimately the semantic content of sentences.

Parsing is done with respect to a fixed set of rules that describe the grammatical structure of a language. Such a set of rules is called a grammar for the language. In a standard parsing model, the parser accepts a string of words from its input and verifies that the string can be generated by the grammar for the language, according to its rules. In such case the string is said to be recognized and is called a sentence of the language. There exist many forms of grammar that have been used for the description of natural languages and patterns, each with its own generative capacity and level of descriptive adequacy for the grammatical description given languages. A hierarchy of grammars has been proposed by N. Chomsky, "On Certain Formal Properties of Grammar," Information and Control, Vol. 2, 1959, p. 137–167, and some of the formalisms that have been or are currently in use for the description of natural language are transformational grammar, two-level grammar, unification grammar, generalized attribute grammar, and augmented transition network grammar. Nonetheless, the formalism most widely used is that of context-free grammars; the formalisms just cited, and others, are in some sense augmentations of or based on context-free grammars.

Likewise, many parsing methods have been reported in the literature for the parsing of natural languages and syntactic pattern recognition. For context-free grammars there are three basic parsing methods, as may be inspected in "The Theory of Parsing, Translation and Compiling," Vol. 1, A. V. Aho and J. D. Ullman, 1972. The universal parsing methods, represented by the Cocke-Kasami-Younger algorithm and Earley's algorithm, do not impose any restriction on the properties of the analysis grammar and attempt to produce all derivations of the input string. The two other methods, known as top-down or bottom-up, attempt as their names indicate to construct derivations for the input string from the start symbol of the grammar towards the input words, or from the input words towards the start symbol of the grammar. The parsing state representations used by the parsing methods include, in general, a triple consisting of the first and last word positions in the input string covered by the parsing state, and a parsing item which may be a grammatical category symbol or a "dotted" grammatical rule, that shows how much of the item has been recognized in the segment of the input string marked by the first and last word positions.

In contrast to the parsing of some artificial languages, such as programming languages for computers, the chief problems encountered in parsing natural languages are due to the size of the grammatical descriptions required, the size of the vocabularies of said languages and several sorts of ambiguity such as part of speech, phrase structure, or meaning found in most sentences. The handling of ambiguity in the description of natural language is by far one of the most severe problems encountered and requires the adoption of underlying grammatical formalisms such as general context-free grammars and the adoption of universal parsing methods for processing.

Even the most efficient universal parsing methods known for context-free grammars (Cocke-Kasami-Younger and Earley's algorithms) are too inefficient for use on general purpose computers due to the amount of time and computer resources they take in analyzing an input string, imposing serious limitations on the size of the grammatical descriptions allowed and the types of sentences that may be handled. The universal parsing methods produce a number of parsing state representations which is in the worst case proportional to the size of the grammatical description of the language and proportional to the square of the number of input words in the string being analyzed. The set of parsing states actually generated in typical applications is, however, a sparse subset of the potential set. Other universal parsing methods used in some systems, including chart parsers, augmented transition network parsers, and top-down or bottom-up backtracking or parallel parsers encounter problems similar to or worse than the standard parsing methods already cited. Since parsing algorithms in current art are typically executed on general purpose computers with a yon Neumann architecture, the number of steps required for the execution of these algorithms while analyzing an input sentence can be as high as proportional to the cube of the size of the grammatical description of the language and proportional to the cube of the number of words in the input string.

The existing von Neumann computer architecture is constituted by a random access memory device (RAM) which may be accessed by location for the storage of program and data, a central processing unit (CPU) for fetching, decoding and execution of instructions from the RAM, and a communications bus between the CPU and RAM, comprising address, control, and dam lines. Due to its architecture, the yon Neumann type computer is restricted to serial operation, executing one instruction on one data item at a time, the communications bus often acting as a "bottleneck" on the speed of the serial operation.

With a clever choice of data structure for the representation of sets of parsing states on a von Neumann computer, such as the use of an array of boolean quantities used to mark the presence or absence of a given item from the set of parsing states, it is possible to reduce the number of steps required to perform basic operations on a set of parsing states to a time that is proportional only to the logarithm of the number of states in the set, and therefore to reduce the total time required for the execution parsing algorithms on the von Neumann computer. However, the number of parsing states that may be generated by universal parsing algorithms is dependent on grammar size and input string length and can be quite high. For the type of grammars and inputs envisioned in language and pattern recognition applications, this number can be of the order of two to the power of thirty, or several thousands of millions of parsing items. This amount of memory space is beyond the capabilities of current computers and, where available, it would be ineffi-

ciently used. The speedup technique suggested is well known and illustrates the tradeoff of processing memory space for reduction of execution time. Universal parsing algorithms, furthermore, require multiple patterns of access to their parsing state representations. This defeats the: purpose of special data structures as above, unless additional memory space is traded off for a fast execution time.

In the technical article "Parallel Parsing Algorithms and VLSI Implementations for Syntactic Pattern Recognition," Y. T. Chiang and K. S. Fu, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 6, No. 3, 1984, p. 302–314, a parallel processing architecture consisting of a triangular-shaped VLSI systolic array is devised for the execution of a variant of the universal parsing algorithm due to Earley. In the Chiang-Fu architecture, the systolic array has a number of rows and a number of columns equal to the number of symbols in the string to be analyzed. Each processing cell of the systolic array is assigned to compute one matrix element of the representation matrix computed by the algorithm. Each cell is a complex VLSI circuit that includes a control and data paths to implement the operators used in the parsing algorithm, and storage cells for the storage of cell data corresponding to matrix elements. The architecture has a regular communication geometry, with each cell communicating information only to its upper and right-hand side neighbors. In order to achieve its processing efficiency requirements, allowing as many processing cells of the array as possible to operate in parallel, the Chiang-Fu architecture must use a weakened form of Earley's algorithm. Furthermore, in order to meet the VLSI design requirement that each processor perform a constant time operation, the architecture restricts the grammar to be free of null productions, i.e., those whose fight-hand sides have exactly zero symbols.

In addition to the two disadvantages of the Chiang-Fu architecture noted above, its main disadvantage, however, is the complexity of each cell in the processing array and the required size of the array. The cell design uses complex special purpose hardware devices such as programmable logic arrays, shift registers, arithmetic units, and memories. This approach yields the fastest execution speed for each cell, but due to its complexity and the highly irregular pattern of interconnections between the cell's components the design is not the best suited for VLSI implementation. Since the systolic array has a number of rows and a number of columns equal to the number of symbols in the string to be analyzed, the number of cells in the array is proportional to the square of the number of symbols in the string.

Associative processing is a technique of parallel computation that seeks to remove some problems of the von Neumann computer by decentralizing the computing resources and allowing the execution of one operation on multiple data items at a time. An associative memory processor has distributed computation resources in its memory, such that the same operation may be executed simultaneously on multiple data items, in situ, The operations that may be executed in the memory are fairly simple, usually restricted to comparison of a stored data word against a given search pattern. The distributed computation approach eliminates two major obstacles to computation speed in the von Neumann computer, namely the ability to operate only on one data item at a time, and the need to move the data to be processed to and from memory. Since associative memory is essentially a memory device, it is the best suited type of circuit for large scale VLSI implementation. Associative processing is currently used in some special purpose computations such as address translation in current computer systems, and is especially well suited for symbolic applications such as string searching, data and knowledge base applications, and artificial intelligence computers. In contrast to addressing by location in a random access memory, associative processing is particularly effective when the sets of data elements to be processed are sparse relative to the set of potential values of their properties, and when the data elements are associated with several types of access patterns or keys.

An associative memory processor architecture for parsing algorithms, as has been proposed by N. Correa, "An Associative Memory Architecture for General Context-free Language Recognition," Manuscript, 1990, stores sets of parsing state representations in an associative memory, permitting inspection of the membership of or the search for a given parsing state in a time which is small and constant, independent of the number of state representations generated by the algorithm. Additionally, the parsing method chosen is implemented in a finite state parsing control unit, instead of being programmed an executed by instruction sequences in the central processing unit of a general purpose computer or microprocessor. This allows for a maximally parallel scheduling of the microoperations required by the algorithm, and eliminates the need for instruction fetching and decoding in the general purpose computer. Furthermore, since the associative memory need be dimensioned only for the number of parsing states that may actually be generated by the parsing algorithms, and since the finite state control unit contains only the states and hardware required for the execution of the algorithm, said machine may be fabricated and programmed more compactly and economically with integrated circuit technology.

It is apparent from the above that prior art approaches to the execution of universal parsing algorithms are neither fast enough nor compact enough for the technical and economic feasibility of complex symbolic applications requiring a parsing step, such as real-time voice recognition and understanding, real-time text and voice-to-voice machine translation, massive document processing, and other pattern recognition applications. The general purpose von Neumann computer and other previous proposals for the parallel execution of those algorithms are not fast enough and not compact enough. The associative processing architecture for the execution of universal parsing algorithms herein disclosed has the potential to offer significant speed improvements in the execution of universal parsing algorithms and is furthermore more compact and better suited for large scale VLSI implementation.

SUMMARY OF THE INVENTION

It is therefore an object of the present invention to provide a new and improved parallel processor architecture that executes parsing algorithms faster than the prior art approaches.

It is a further object of the present invention to provide a new and improved parallel processor architecture which is dedicated exclusively to the execution of parsing algorithms and is physically more compact, smaller, and better suited for large scale VLSI implementation than the prior art approaches.

It is still a further object of the present invention to show a particular embodiment of a universal parsing algorithm in said architecture and the method by which this is achieved.

In accordance with the above objects, the present invention is addressed to an associative memory processor archi-

tecture consisting of an associative memory unit for the storage of parsing state representations, a random access memory unit for the storage of the grammatical rules and other parsing data and tables according to which the parsing is done, a finite state parsing control unit which embodies the chosen parsing algorithm, and a communications unit for communication with a host processor or external interface.

The associative memory unit (CAM) is used for the storage of parsing state representations, dynamically computed by the parsing algorithm according to the input string and grammar. Each parsing state representation consists of a tuple of a first word index to a position in the input string, a last word index to a position in the input string, a parsing item, a left-hand side symbol field., a next symbol field, a state-not-processed field, and optional fields to store other information related to the parsing process, such as context and lookahead symbols, attributes of the parsing state, and information for parse tree extraction. Each parsing state representation is storm in one logical CAM word, which permits fast and easy inspection of the parsing states already generated by the algorithm. The parsing item in the third field of a parsing state representation may be a grammar symbol or a dotted rule, consisting of a rule number and an index to a position on the right hand side of the rule.

The random access memory unit (RAM) is used for the storage of the grammatical rules according to which the parsing is done. This memory unit is also used to store other parsing data and tables used by the parsing algorithm, as detailed below; alternatively, a second random access memory unit may be used for the storage of such information. Each grammatical rule consists of one left-hand side symbol and a right-hand side of zero or more symbols. Each grammatical rule is stored in one logical RAM record, with one RAM word allocated to store each of the rule's symbols. In this manner, it is possible to retrieve the j-th symbol of the p-th grammatical rule from the j-th word of the p-th record in the RAM. The RAM may be accessed by the communications unit for the purpose of allowing the host processor writing into the RAM the grammatical rules according to which the parsing is done. Alternatively, the RAM may be a read-only memory, which permanently stores a predefined set of grammatical rules and tables.

The finite state parsing control unit (PCU) is connected to the CAM and the RAM and is a finite state machine that embodies the chosen parsing algorithm. The PCU accesses the CAM for the purposes of initializing it, inserting initial or seed parsing states for the parsing process, and requesting parsing states marked unprocessed for processing. When an unprocessed parsing state is retrieved from the CAM, the PCU may access the RAM and may request input symbols from the communications unit for the purpose of generating new parsing states to be added to the CAM, as unprocessed. Each access to the RAM allows the inspection of the grammatical rules, if any, that may be applicable for processing of the current parsing state. The input symbols requested form the communications unit allow verification that the next input symbol is compatible with the current parsing state. When the PCU has generated the number of parsing state sets required by the input string and all parsing states in the CAM axe marked processed—i.e., there are no unprocessed states—the PCU performs a test on the contents of the CAM to decide acceptance of the input string, may optionally execute some post-processing operations, as detailed below, signals the communications unit that the parsing of the current input string is complete, and terminates execution. The exact order and the precise nature of the operations performed by the parsing control unit, generically described above, depend on the particular parsing algorithm embodied in the finite state parsing control unit.

The communications unit (CU) is connected to the CAM, RAM, and PCU and is used for communication with a host processor or external interface. The communications unit may be as simple as an interface to a given computer interconnection bus, or as complex as a system that implements a computer communications protocol. The communications unit accesses the RAM for the purpose of allowing the host processor writing into the RAM the grammatical rules according to which the parsing is done. Alternatively, the RAM may be a read-only memory, which permanently stores a predefined set of grammatical rules, in which case the CU need not have access to the RAM. The CU also accesses the finite state control unit for the purposes of initializing it and supplying to it input symbols from the input string to be analyzed. The CU also accesses the CAM at the end of a parsing process for the purpose of reading out and sending to the host processor the parsing state representations and any other information that may be relevant to further processing of the input string. An optional additional function of the communications unit is its ability to issue commands and data to the RAM, CAM and PCU for the purpose of testing their functionality and correctness of operation.

Preferably, the associative memory unit is formed on a single integrated circuit chip, and the random access memory unit, finite state parsing control unit, and a communications unit are formed together or programmed on a separate integrated circuit controller chip. Alternatively, all system components may be integrated on a single chip, with optional provision for external expansion of the RAM or CAM memories. In either case, the operation of the finite state parsing control unit may allow for the execution of parse extraction algorithms and useless parsing state marking and elimination algorithms, to simplify further processing of the parsing result by the host processor.

BRIEF DESCRIPTION OF THE DRAWINGS

In the detailed description of the preferred embodiment of the invention presented below, reference is made to drawings as presently detailed. The drawings are not necessarily to scale, emphasis being placed instead upon illustrating the principles of construction and operation of the invention.

FIG. **1** is a complete schematic illustration of the associative memory processing system for parsing,algorithms, object of the present invention.

FIG. **2** shows the general organization of the associative memory unit assumed by the preferred embodiment.

FIG. **3** is a small example context-free grammar and shows a sample input string with annotated string positions.

FIG. **4** is a schematic illustration of the RAM memory map corresponding to the example grammar in FIG. **3**

FIG. **5** is a schematic illustration of the parsing state encodings to be stored in the associative memory, for the preferred embodiment where the processor embodies Earley's algorithm.

FIG. **6** is a flow chart of the steps followed by the system during loading of a grammar, parsing, and extraction of the parse information.

FIGS. **7***.a–c* are a schematic illustration of a series of CAM memory maps of the associative processing system at different times during parsing an input string, according to the example grammar in FIG. **3**.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

FIG. **1** illustrates an embodiment of the present invention suitable for the execution of a wide family of parsing algorithms. Referring to the same figure, the system includes an associative memory unit **1** and a communications and parsing control unit **5**. The communications and parsing control unit includes a random access memory unit **2**, a finite state parsing control unit **3**, a communications unit **4**, a first data bus **10**, a second data bus **11**, and other signals further detailed below.

Associative memory unit **1** is connected by the internal data bus **10** and by control fines **S1** and **S2** to the parsing control unit The associative memory unit (CAM) is used for the storage of parsing state representations and its word width is commensurate with the number of bits required for the representation of parsing states. The parsing state representations produced by the parsing control unit may be transferred, i.e., written, to the associative memory through the internal data bus **10**. Likewise, parsing states stored in the associative memory may be transferred in the opposite direction, i.e., read, to the parsing control unit by means of the same internal data bus **10**. To provide for fast data transfers between the associative memory and the parsing control unit, in one bus cycle, the width of the first data bus **10** is equal to the width of one CAM word. Control line **S1** from the parsing control unit to the associative memory is the operation select code for the operation requested of the associative memory. Control line **S2** from the associative memory to the parsing control unit is a match flag produced by the associative memory after a match operation. Because an associative memory is used for the storage of parsing state representations, operations such as the insertion of a new parsing state into the CAM may be performed in constant time, independent of the number of parsing states already generated, and the performance degradation resulting from the use of random access memory in avon Neumann computer for the storage of the same representations is mitigated. Also, because an associative memory is used, multiple access patterns are permitted into the parsing state representations, without the overhead of additional data structures. These multiple access patterns play a role in the implementation of some optimizations of the parsing algorithm to be embedded in the finite state parsing control unit.

The general organization of the associative memory unit assumed by the preferred embodiment is shown in FIG. **2**. This device has one array **20** of content addressable memory cells, one data register **21**, one mask register **22**, three general purpose match registers **23**, **24**, and **25**, a priority encoder **26** for multiple response resolution, and an internal control section **27** for control of CAM operations. The device has an associative method of word selection for read and write operations, in which word selection is achieved by the use of one of the match registers **23**, **24**, or **25**, and the priority encoder **26**. The memory receives data and control signals from the outside through data and control buses **28** and **29**, and produces one match signal MATCHFLAG **30** after the execution of match operations. The set of operations provided by the associative memory unit is further detailed below in the description of the parsing control unit.

Random access memory unit **2** in FIG. **1** is connected to the parsing control unit and other system components by a second internal data bus **11** and by address and control lines **S3** from the PCU. Physically, the RAM is organized as a linear array of words, divided into logical records of several words. The number of bits per RAM word must be selected according to the number of terminal and non-terminal symbols in the grammar; for example, with a word width of ten bits a total of **1024** different symbols may be encoded. We let PLEN be the number of words in one logical RAM record and require that it be at least one more than the number of symbols in the longest grammatical rule to be represented. The grammatical rules are ordered by their left-hand side symbol and numbered from zero to some positive integer PMAX, so that the number of RAM words required to store the grammatical rules is PLEN times PMAX.

The rules of FIG. **3** constitute a simple grammar with four non-terminal symbols Z, S, NP, and VP, and three terminal symbols "$", "John", and "thinks"; Z is the start symbol of the grammar, and "$" is the "end-of-input-string" marker. Each grammatical rule is stored in one logical RAM record, as shown in FIG. **4**, with one RAM word used to store each of the rule's symbols. In this preferred embodiment, the logical records have a fixed number of words, such that the j-th symbol of the p-th grammatical rule may be retrieved from the RAM word at address p times PLEN plus j. The symbol NIL, not in the vocabulary of the grammar, is used to mark the end of each rule's right-hand side. The RAM may be accessed by the communications unit through the second internal data bus **11** for the purpose of allowing the host processor to write into the RAM the grammatical rules according to which the parsing is done. Alternatively, the RAM may be a mad-only memory, which permanently stores a predefined set of grammatical rules.

In this embodiment, the random access memory unit contains additional space for the storage of two parsing tables, P-TABLE and N-TABLE. P-TABLE relates the non-terminal symbols of the grammar to the number of the record of the fast production in their list of alternatives in the RAM. This information is used by the parsing algorithm and is stored at an offset P_OFFSET from the first word of the RAM, beyond the end of the space used to store the grammatical rules. N-TABLE is a table of all the symbols in the grammar and the special symbol NIL that indicates for each one whether it may derive the empty string after one or more derivation steps (i.e., whether it may be nulled). This table is storm at an offset N_OFFSET from the fast word of the RAM, beyond the end of the space used to store the P-TABLE. These tables are also shown in FIG. **4**.

The parsing control unit **3** in FIG. **1** is connected to the associative memory unit and to the random access memory unit as already described. The parsing control unit is also connected by the second internal data bus **11** and by control lines **S4**, **S5**, and **S6** to the communications unit. The second internal data bus **11** is used to transfer commands and input symbols to the parsing control unit, and to read status information from the same. Control line **S4** is the SYMBOL_REQUEST line from the PCU to the communications unit, while **S5** is the SYMBOL_READY line in the converse direction. Control line **S6** is the END_OF_PARSE line from the PCU to the communications unit. Because the parsing control unit is a finite state machine that embodies the chosen parsing algorithm, it is optimized both with regard to speed and size. In this preferred embodiment, the parsing control unit is designed to execute a version of Earley's algorithm, "An Efficient Context-free Parsing Algorithm," Communications of the Association for Computing Machinery, Vol. 13, No. 2, p. 94–102, known in the art, and includes some optimizations of the original algorithm, suggested by S: Graham et al., "An Improved Context-free Recognizer," A CM Transactions on Programming Languages and Systems, Vol. 2, No. 3, 1980, p. 415–462. According to Earley's algorithm, in the preferred embodiment the parsing control

unit has a main procedure that initializes the machine, writes an initial parsing state into the associative memory unit, and then reads unprocessed states from the CAM and processes them according to one of three actions: PREDICT, COMPLETE, and EXAMINE, to be detailed below. The embodiment is most general, allowing arbitrary context-free grammar rules, including grammar rules with zero right-hand side symbols. In this version, the algorithm uses a number k of "lookahead" symbols equal to zero. Modification of this feature of the algorithm is within the state of current an and may be made by those skilled in the art.

The parsing state representations stored in the associative memory unit are bit patterns arranged into seven fields named "first-word-index", "last-word-index", "rule-number", "dot-position", "left-hand-side" symbol, "symbol-after-dot", and "processed-bit", as shown in FIG. 5. The data in the fifth and sixth fields, "left-hand-side" symbol and "symbol-after-dot", respectively, are redundant, since they may be obtained from the grammar rules stored in the random access memory knowing the "rule-number" and "dot-position" values. However, the operation of retrieving the symbol on the right side of the dot is essential to the three actions of the algorithm, particularly the COMPLETER, and hence the "symbol-after-dot" field is included in the parsing state representations to facilitate and speed up the execution of this operation. Similarly, the inclusion of the fifth field, "left-hand-side" symbol, allows the implementation of an important optimization to the COMPLETER step. A complete behavioral description of the parsing control unit, corresponding to Earley's algorithm with the noted optimizations, appears below in TABLE 1, pans A through G. The first data bus 10 of FIG. 1 is referred to as CAM_BUS in the descriptive code, and the second data bus 11 of the same figure is referred to as D_BUS in the same code. The behavioral description assumes the purely associative memory of FIG. 2, with one data and one mask register, and with three match registers MR1, MR2, and MR3, which may be used for word selection in the CAM operations. The behavioral description of the CAM operation codes assumed by the parsing control unit is given in TABLE 2, below.

The communications unit is connected to the associative memory unit, the random access memory unit, and the parsing control unit through the second internal dam bus 11. The CU accesses, through said second internal data bus 11, the finite state parsing control unit for the purposes of initializing it and supplying to it input symbols of the input string to be analyzed. The unit also accesses the CAM at the end of a parsing process for the purpose of reading out and sending to the host processor the parsing state representations and any other information that may be relevant to further processing of the input string. In this embodiment, the communications unit implements a communications protocol for computer peripherals that may be supported by small computers and workstations. This allows the use of the associative processor object of the present invention as an internal or external peripheral device for a wide variety of computers.

The operation of the associative parsing machine, according to the behavioral description of its components given in TABLE 1 and TABLE 2 below, with the grammar of FIG. 3 and for the input string "John thinks $" will now be described with reference to FIG. 6 and FIGS. 7A to 7C.

When the associative parsing machine starts its operation in response to a command from the host processor or external interface, it requires that the parsing grammar, the productions table (P-TABLE), and the nullable symbols table (N-TABLE) have already been loaded into the random access memory. Thus, for the grammar of FIG. 3, the RAM configuration is that shown in FIG. 4. FIG. 6 is a flow chart that shows the general operation of the system, including loading of the analysis grammar, invocation of the main recognizer procedure, execution of optional post-processing actions, and extraction of the parse information.

The parsing control unit of the machine uses an associative memory with one data register DATA_REG, one mask register MASK_REG, and three match registers MR1, MR2, and MR3. MR1 is used as a general "match" register, MR2 as a temporary "match" register, and MR3 as a "free words" register. The parsing control unit contains three registers CURRENT_SET, INPUT_SYMBOLS, and NEXT_SYMBOL which are used to store the number of the current parsing state set being processed (last-word-index), the number of symbols from the input string already seen, and the next input symbol from the input string. A one bit flag EXIST_SYMBOL is use to indicate that the NEXT_SYMBOL register currently contains the next input symbol from the input string. The parsing control unit also has a data register DR used for storing parsing state representations and a STATUS register with "accept" and "error" fields, into which the result of recognition of the input string is deposited, in the "accept" field of the register. An END_OF_PARSE one bit flag is used to signal the communications unit the end of the parsing process for the input string.

The descriptive code corresponding to the top level of the parsing control unit (RECOGNIZER) is shown in TABLE 1, part A. The code contains steps to initialize the recognizer, write an initial parsing state representation into the CAM, dynamically compute the set of all parsing state representations, and test for acceptance of the input string, depending on the set of parsing states computed. The intialization steps of the recognizer in the code of INITIALIZE_RECOGNIZER, shown in TABLE 1, part B, reset the CURRENT_SET and other registers of the machine, reset the STATUS accept and END_OF_PARSE flags, clear the associative memory, and according to the operation CLEARCAM, in TABLE 1, part G, set the "free words" register MR3 of the CAM, indicating that initially all CAM words are free. Immediately thereafter the parsing control unit assembles and writes into the CAM an initial parsing state representation that corresponds to the application of the production for the initial symbol of the grammar in a top-down derivation. This is shown in the code of WRITE_INITIAL_STATE, also in TABLE 1, part B. This initial parsing state corresponds to the zero-th production of the grammar in FIG. 3 and has first and last word indices equal to zero, rule number equal to zero, dot position equal to zero, left-hand-side symbol equal to the numeric code of Z, symbol-after-dot equal to numeric code of S, and processed-bit mark in zero. The contents of the CAM after insertion of this parsing state are shown in FIG. 7A.

The principal part of the RECOGNIZE-R code consists of an iteration cycle in which the CAM is searched for unprocessed parsing states in the current state set and, if any are found, these are processed, one at a time, according to one of three actions: PREDICT, COMPLETE, and EXAMINE, depending to the type of the symbol found in the "symbol-after-dot" field of the unprocessed parsing state. PREDICT is applied when the symbol after the dot is a non-terminal symbol, COMPLETE when there is no symbol (i.e., NIL) after the dot, and EXAMINE when the symbol is a terminal symbol. The processing of each state includes toggling its processed-bit mark to one (i.e., marking it as processed). The descriptive code for the three actions PREDICT, COMPLETE, and EXAMINE is shown in TABLE 1, part C. The

descriptive code for the search of unprocessed parsing states from the current state set appears in the code of MATCH_UNPROCESSED_STATES in TABLE 1, part F.

The first parsing state to be processed by the machine is the initial state inserted into the CAM, as part of the initialization steps of the RECOGNIZER code. This parsing state is first read from the CAM into register DR of the parsing control unit, and then processed according to the PREDICT operation, since the symbol S found in the "symbol-after-dot" field is a non-terminal symbol. The PREDICT operation first searches the CAM to verify if the "symbol-after-dot" in the state (S in this case) has not already been predicted during processing of the current parsing state set, and then marks the state processed by toggling its "processed-bit" field to one and rewriting it into the CAM. If the symbol has been predicted during processing of the current parsing state set no further action is done by the PREDICT operation. Otherwise, the operation seeks grammar rules with the "symbol-after-dot" on the left-hand side and for each one generates a new parsing state representation, to be added to the CAM as unprocessed. The new states are added into the CAM by the operation ADD_STATE, shown in TABLE 1, part D. According to this operation, a new parsing state representation is not added into the CAM if it is already found there, ignoring its "processed-bit". The ADD_STATE operation may also add some additional states into the CAM, if some symbols after the dot in the original state to be added are nullable. Since in the grammar of FIG. 3 there is only one rule for the symbol S of the initial parsing state representation, and there are no nullable symbols, there is only one new parsing state added into the CAM by the PREDICT operation, and the CAM contents after execution of this operation are the two parsing states shown in FIG. 7B.

After one more iteration in the RECOGNIZER code, in which the production for the NP non-terminal symbol is predicted, the associative processor is ready to apply the EXAMINE operation to the first symbol "John" of the input string. Symbols from the input string are obtained from the communications unit by the GET_INPUT_SYMBOL operation of TABLE 1, part E. If the symbol is not already in the NEXT_SYMBOL register, the operation raises the SYMBOL_REQUEST signal to the communications unit and waits until the unit responds with the SYMBOL_READY signal in the converse direction, at which time the symbol must be present on the data bus 11 (D_BUS) of FIG. 1 and is loaded into the NEXT_SYMBOL register.

The parsing control unit continues operating as made explicit in its behavioral description of TABLE 1, parts A through G, until no parsing states axe found unprocessed in the current parsing state set and the value of the CURRENT_SET register is greater than the value in the INPUT_SYMBOLS register. This condition signals the end of the dynamic computation of parsing state representations for the input string read. For the input string "John thinks $", assumed as input to the associative parsing machine, the parsing state representations computed, and hence the contents of the CAM at the end of the iterations of the RECOGNIZER, are shown in FIG. 7C. The last two steps of the of the parsing control unit, as shown in the RECOGNIZER code of TABLE 1, part A, are a test for acceptance of the input string, by searching the CAM for presence of a particular parsing state representation, and to signal the end of the parsing process, by setting the END_OF_PARSE flag to one. The details of the test for acceptance appear in TABLE 1, part F.

Throughout TABLE 1, the interaction between the operation of the parsing control unit and the associative memory unit is done through the operations of TABLE 1, part G. These operations assume the basic operation codes of TABLE 2 for the associative memory unit, and are macro codes that utilize those primitive operations of the associative memory.

Two optimizations of Earley's original algorithm appear in the steps CHECK_IF_ALREADY_PREDICTED and CHECK_IF_ALREADY_COMPLETED of the PREDICT and COMPLETE operations in TABLE 1, part C. The two steps, shown in TABLE 1, part F, help to avoid lengthy computations in which a non-terminal symbol already predicted during computation of the current parsing state set is tried to be predicted again, or a non-terminal symbol already completed from a given parsing state set is tried to be completed again. A third optimization of the algorithm appears in the operation ADD_STATE of TABLE 1, part D. This operation handles in an efficient way what would otherwise be a series of predict and complete operations on nullable symbols, using the precomputed information on nullable symbols from the N-TABLE.

In addition to the execution of the selected parsing algorithm, the finite state parsing control unit may optionally execute some post-processing operations, such as parse extraction algorithms and useless parsing state marking and elimination algorithms, to simplify further processing of the parsing result by the host processor.

The chief advantage of the associative memory parsing processor over a traditional von Neumann computer is that it reduces the theoretical and practical time complexity of universal parsing algorithms both with respect to grammar size and input string length, in a compact manner. The hardware implementation of the parsing algorithm to be used also contributes significantly to speed of operation. Additionally, when attached to the central processing unit of a standard computer, the associative processor acts as a dedicated parallel processor that frees general computing resources of the host computer for other user tasks. An advantage of the associative memory processor over other parallel architectures for the execution of parallel parsing algorithms, such as the systolic array architecture of Chiang and Fu, is that the parallel processing element in the associative processor is its associative memory, which is better suited for large scale VLSI implementation, due to its regularity of layout and interconnection patterns and its wide range of applications. For the purposes of illustration, but not of limitation, in the following TABLE 1, parts A through G, an example behavioral description of the associative processor in accordance with the invention is given. It should be noted by those skilled in the art that the description admits man), different structural realizations and that, therefore, in the interest of generality, none such is given.

TABLE 1

part A

Behavioral Description of Parsing Control Unit (PCU): RECOGNIZER

```
RECOGNIZER:
/*  Data register fields DR: <f, i, p, j, lhs, sad, pb>
    CAM MR1: General match register
    CAM MR2: Temporary match register
    CAM MR3: Free words register
    ------------------------------------------------------------ */
    INITIALIZE_RECOGNIZER;
    WRITE_INITIAL_STATE;
    repeat
        MATCH_UNPROCESSED_STATES;
```

TABLE 1-continued

part A

Behavioral Description of Parsing Control Unit (PCU): RECOGNIZER

```
        while MATCHED_STATES do begin
                READCAM MR1;
                switch CLASSIFY(DR.sad) begin
                        NON_TERMINAL:   PREDICT;
                        NIL:            COMPLETE;
                        TERMINAL:       EXAMINE;
                        default         ERROR( 0);
                endswitch;
                MATCH_UNPROCESSED_STATES;
        endwhile;
        CURRENT_SET := CURRENT_SET + 1;
        EXIST_SYMBOL := 0;
    until CURRENT_SET > INPUT_SYMBOLS;
    TEST_ACCEPTANCE;
    END_OF_PARSE := 1;
END.
```

TABLE 1

part B

Behavioral Description of PCU: Initialization routines

```
INITIALIZE_RECOGNIZER:
        CURRENT_SET := 0;
        INPUT_SYMBOLS := 0;
        EXIST_SYMBOL := 0,
        SYMBOL_REQUEST := 0;
        END_OF_PARSE := 0;
        STATUS.accept := 0;
        STATUS.error[0] := 0;
        CLEARCAM;
END.
WRITE_INITIAL_STATE:
        DR.f := 0;
        DRj := 0;
        DR.p := 0;
        DR.j := 0;
        DR.lhs := RULE[ 0, 0];
        DR.sad := RULE[ 0, 1];
        DR.pb := 0;
        ADD_STATE;
END.
```

TABLE 1

part C

Behavioral Description of PCU: PREDICT, COMPLETE, EXAMINE

```
PREDICT:
    CHECK_IF_ALREADY_PREDICTED;
    MARK_STATE_PROCESSED;
    if not( MATCHED_STATES) begin
        FIRST_P := P_TABLE[ DR.sad];
        LAST_P := P_TABLE[ DR.sad + 1];
        DR.f := CURRENT_SET;
        DR.i := CURRENT_SET;
        DR.j := 0;
        DR.lhs := DR.sad;
        repeat
            DR.p := FIRST_P;
            DR.sad := RULE[ FIRST_P, 1];
            DR.pb := (DR.sad == NIL);
            ADD_STATE;
            FIRST_P := FIRST_P + 1,
        until FIRST_P = LAST_P;
    endif;
END.
COMPLETE:
    CHECK_IF_ALREADY_COMPLETED;
    MARK_STATE_PROCESSED;
    ifnot( MATCHED_STATES) begin
        DR.i := DR.f;
        DR.sad := DR.lhs;
        MATCHCAM MR1, DR, < 1, 0, 1, 1, 1, 0, 1>;
        while MATCHED_STATES do begin
            READCAM MR1;
            DR.i := CURRENT_SET;
            DR.j := DR.j + 1;
            DR.sad := RULE[ DR.p, DR.j + 1];
            DR.pb := 0;
            ADD_STATE;
            SELECTNEXTCAM MR1;
        endwhile;
    endif;
END.
EXAMINE:
    MARK_STATE_PROCESSED;
    GET_INPUT_SYMBOL;
    if DR.sad = NEXT_SYMBOL begin
        DR.i := CURRENT_SET + 1;
        DR.j := DR.j + 1;
        DR.sad := RULE[ DR.p, DR.j + 1];
        DR.pb := 0;
        ADD_STATE;
    endif;
END.
```

TABLE 1

part D

Behavioral Description of PCU: ADD_STATE

```
ADD_STATE:
    WRITESETCAM MR3, DR, < 0, 0, 0, 0, 0, 0, 1>;
    if not( MATCHED_STATES) begin
        repeat
            NULLABLE := N_TABLE[ DR.sad];
            if NULLABLE begin
                DR.j := DR.j + 1
                DR.sad := RULE[ DR.p, DR.j + 1];
                WRITESETCAM MR3, DR, < 0, 0, 0, 0, 0, 0, 1>;
            endif;
        until not( NULLABLE) OR MATCHED_STATES;
    endif;
END.
```

TABLE 1

part E

Behavioral Description of PCU: GET_INPUT_SYMBOL, CLASSIFY

```
GET_INPUT_SYMBOL:
    if not( EXIST_SYMBOL) begin
            SYMBOL_REQUEST := 1;
            wait on SYMBOL_READY;
            NEXT_SYMBOL := D_BUS,
            SYMBOL_REQUEST := 0;
            EXIST_SYMBOL := 1;
            INPUT_SYMBOLS := INPUT_SYMBOLS + 1;
    endif;
END.
CLASSIFY( SYMBOL):
    /*  Assumes an n-bit encoding of 'SYMBOL' as follows
        Start symbol (ZETA):    2^ (n-1)
        Other non-terminals:    2^ (n-1), . . ., 2^ n - 1
        Terminals:              1, . . ., 2^ (n-1) - 1
        End-of-string (NIL):    0
        ------------------------------------------------- */
    NT = SYMBOL[ n-1];
    ZERO = not( OR( SYMBOL[ n-2], . . ., SYMBOL[ 0]));
    if (NT AND ZERO) begin return(ZETA) endif;
    if (NT AND not ZERO)) begin return( NON_TERMINAL) endif;
    if (not NT) AND not( ZERO)) begin return( TERMINAL) endif;
    if (not( NT) AND ZERO) begin return( NIL) endif;
END.
```

TABLE 1

part F

Behavioral Description of PCU: Other Macros

```
MATCH_UNPROCESSED_STATES:
    DR.i := CURRENT_SET;
    DR.pb := 0;
    MATCHCAM MR1, DR, < 1, 0, 1, 1, 1, 1, 0>;
END.
MARK_STATE_PROCESSED:
    DR.pb := 1;
    WRITECAM MR1, DR;
END.
CHECK_IF_ALREADY_PREDICTED:
    DR.pb := 1;
    MATCHCAM MR2, DR, < 1, 0, 1, 1, 1, 0, 0>;
END.
CHECK_IF_ALREADY COMPLETED:
    DR.pb := 1;
    MATCHCAM MR2, DR, < 0, 0, 1, 1, 0, 0, 0>;
END.
ERROR( i):
    STATUS.error[ i] := 1:
```

TABLE 1-continued

part F

Behavioral Description of PCU: Other Macros

```
END.
TEST_ACCEPTANCE:
    DR.f := 0;
    DR.i := INPUT_SYMBOLS;
    DR.p := 0;
    DR.j := 2;
    MATCHCAM MR2, DR, < 0, 0, 0, 0, 1, 1, 1>;
    STATUS.accept := MATCHED_STATES;
END.
```

TABLE 1

part G

Behavioral Description of PCU: CAM Macros

These macros are expanded into primitive CAM operation codes, with the following usage of the three match registers: MR1 =

TABLE 1-continued

part G

Behavioral Description of PCU: CAM Macros

```
match register, MR2 = temporary match register, MR3 =
free words register.
CLEARCAM:
        CLEAR;
        SETREG MR3;
END.
READCAM REG:
        READ REG;
        DR := CAM_BUS;
END.
WRITECAM REG, DATA:
        CAM_BUS = DATA;
        WRITE REG;
END.
SELECTNEXTCAM REG:
        SELECTNEXT REG;
        MATCHED_STATES := MATCHFLAG;
END.
MATCHCAM REG, DATA, MASK:
        CAM_BUS = MASK;
        LOADMASK;
        CAM_BUS = DATA;
        MATCH REG;
```

TABLE 1-continued

part G

Behavioral Description of PCU: CAM Macros

```
        MOVEREG REG, (REG AND not( MR3));
        MATCHED_STATES := MATCHFLAG;
END.
WRITESETCAM REG, DATA, MASK:
        CAM_BUS = MASK;
        LOADMASK;
        CAM_BUS = DATA;
        MATCH MR2;
        MOVEREG MR2, (MR2 AND not( MR3));
        MATCHED_STATES := MATCHFLAG;
        if not( MATCHFLAG) begin
                WRITE REG;
                SELECTNEXT REG,
        endif;
END.
```

Also, for the purposes of illustration, but not of limitation, in the following TABLE 2, a behavioral description of the CAM operation codes assumed by the parsing control unit is given.

TABLE 2

Behavioral Description of CAM Operation Codes

```
/*      CAM registers: DATA_REG, MASK_REG, MR1, MR2, MR3
        CAM width: WCAM (bits per word)
        CAM height: HCAM (number of words)
        CAM[ i] is the i-th CAM word, for i = 1, . . ., HCAM
        ---------- */
CLEAR:
        DATA_REG := 0;
        MASK_REG := 0;   /* MASK register: "0" don't mask; "1" mask      */
        MR1[ i] := 0;    /* MATCH register 1, for i = 1, . . ., HCAM     */
        MR2[ i] := 0;    /* MATCH register 2, for i = 1, . . ., HCAM     */
        MR3[ i] := 0;    /* MATCH register 3, for i = 1, . . ., HCAM     */
END.
READ REG:                                   /* REG = MR1, MR2, or MR3   */
        DATA_REG := CAM[ PRIORITY[ REG]];
        CAM_BUS = DATA_REG;
END.
WRITE REG:                                  /* REG = MR1, MR2, or MR3   */
        DATA_REG := CAM_BUS;
        CAM[ PRIORITY[ REG]] := DATA_REG;
END.
SELECTNEXT REG:                             /* REG = MR1, MR2, or MR3   */
        REG := SELECT_NEXT( REG);           /* resets LSB of REG set to "1" */
        MATCHFLAG := OR(REG[ 1], . . ., REG[ HCAM]);
END.
LOADMASK:
        MASK_REG := CAM_BUS;
        END.
SETREG REG:                                 /* REG = MR1, MR2, or MR3   */
        REG[ i] := 1;                       /* for i = 1, . . ., HCAM   */
END.
RESETREG REG:                               /* REG = MR1, MR2, or MR3   */
        REG[ i] := 0;                       /* for i = 1, . . ., HCAM   */
END.
MOVEREG REG, expression:                    /* REG = MR1, MR2, or MR3   */
                                            /* expression: register, Boolean */
        REG[ i] := expression[ i];          /* for i = 1, . . ., HCAM   */
        MATCHFLAG := OR(REG[ 1], . . ., REG[ HCAM]);
END.
MATCH REG:                                  /* REG = MR1, MR2, or MR3   */
        DATA_REG := CAM_BUS;
        SEARCH_PATTERN = DATA_REG * MASK_REG;
        MLINE[ i] = MATCH( CAM[ i], SEARCH_PATTERN);
        MATCHFLAG := OR( MLINE[ 1], . . ., MLINE[ HCAM]);
        REG[ i] := MLINE[ i];               /* for i = 1, . . ., HCAM   */
```

TABLE 2-continued

Behavioral Description of CAM Operation Codes
END.

While this invention has been shown particularly and described with reference to a preferred embodiment, it shall be understood by those skilled in the an that numerous modifications may be made in form and details of the architecture, in the choice of the parsing algorithm to be used, and in the particular embodiment of said algorithm, that are within the scope and spirit of the inventive contribution, as defined by the appended claims. For example, the associative memory unit has been shown with a particular organization and set of operation codes it can execute, but this does not preclude the use of other associative memory means that can implement the required operations. Likewise, different arrangements in the number and nature of the control signals used to interconnect the system components are possible. Variations and optimizations in the choice of the parsing algorithm are possible, which may affect the time and space complexity of the device. Some of the optimizations referred to may require minor changes to the architecture of the preferred embodiment, such as the inclusion of additional tables for the parsing process. One such optimization worth noting is the inclusion of a table or other means in the random access memory to store the relation FIRSTk between non-terminal and terminal symbols, to avoid useless predictions.

Finally, the behavioral description of the parsing control unit shown in Table 1, corresponding to the particular parsing algorithm chosen, or any other alternative one, admits of many distinct physical realizations, such as may be obtained by manual transformation of the specification into structural, logical, electrical, and geometrical levels of description, or as the same descriptions may be obtained by means of automated synthesis tools for silicon compilation.

What is claimed is:

1. An associative memory processing system for executing parsing algorithms and real time context-free language processing and pattern recognition of an input symbol string, said system comprising:

an associative memory unit logically arranged as an array of words for storing parsing state representations, each associative memory word being compared, in parallel with all other words, to an input search pattern corresponding to a parsing state representation;

a random access memory unit for storing parsing data including context-free language grammatical rules according to which parsing is done for the context-free language of the input symbol string;

a parsing control unit, connected to said associative memory unit and said random access memory unit, for accessing said associative memory unit to store and retrieve parsing state representations according to an input symbol string said parsing control unit being a finite state machine that executes a parsing algorithm, corresponding to the context-free language of the input symbol string, for syntactically recognizing the input symbol string; and

a communications unit for providing communication between said associative memory processing system and an external device.

2. An associative memory processing system as claimed in claim 1 wherein said parsing control unit executes parsing algorithms for natural language processing and pattern recognition applications.

3. An associative memory processing system as claimed in claim 1 wherein said associative memory unit is formed of one or more banks of integrated circuit semiconductor chips.

4. An associative memory processing system as claimed in claim 1 wherein said associative memory unit is formed of one or more banks of associative memory chips, and said random access memory unit and said parsing control unit are formed on a separate integrated circuit semiconductor chip.

5. An associative memory processing system as claimed in claim 1 wherein all system components are formed on a single integrated circuit semiconductor chip.

6. An associative memory processing system as claimed in claim 1, wherein said parsing control unit accesses said associative memory unit in an amount of time that is constant and independent of an amount of parsing data stored in said associative memory unit.

7. An associative memory processing system as claimed in claim 1, wherein said parsing control unit performs post-processing actions.

* * * * *

US005703877A

United States Patent [19]

Nuber et al.

[11] Patent Number: 5,703,877

[45] Date of Patent: *Dec. 30, 1997

[54] **ACQUISITION AND ERROR RECOVERY OF AUDIO DATA CARRIED IN A PACKETIZED DATA STREAM**

[75] Inventors: **Ray Nuber**, La Jolla; **Paul Moroney**, Olivenhain; **G. Kent Walker**, Escondido, all of Calif.

[73] Assignee: **General Instrument Corporation of Delaware**, Chicago, Ill.

[*] Notice: The term of this patent shall not extend beyond the expiration date of Pat. No. 5,517,250.

[21] Appl. No.: **562,611**

[22] Filed: **Nov. 22, 1995**

[51] **Int. Cl.**[6] **H04J 3/06**; H04N 7/12

[52] **U.S. Cl.** **370/395**; 370/510; 370/514; 375/366; 348/423; 348/462; 348/466; 348/467

[58] **Field of Search** 370/389, 395, 370/503, 509, 510, 514, 516; 375/362, 365, 366, 368, 371; 348/423, 461, 462, 464, 466, 467

[56] **References Cited**

U.S. PATENT DOCUMENTS

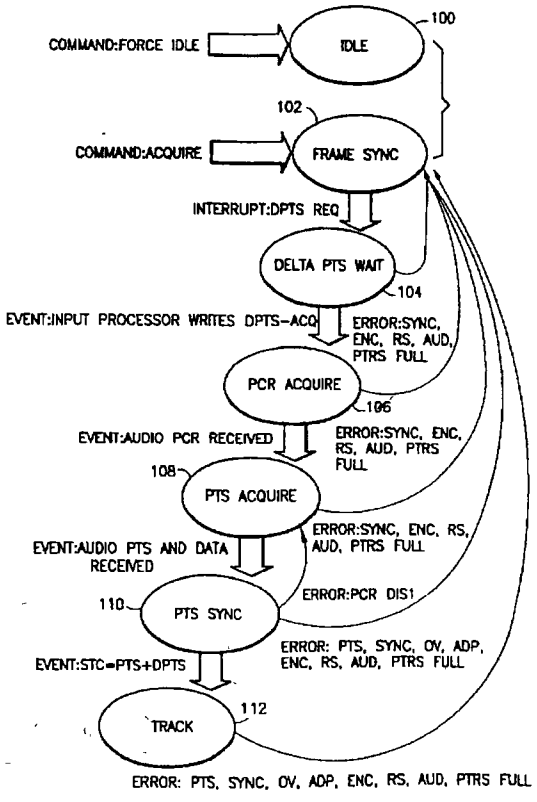
5,365,272	11/1994	Siracusa	348/461
5,376,969	12/1994	Zdepski	348/466
5,467,342	11/1995	Logston et al.	370/253
5,517,250	5/1996	Hoogenboom et al.	348/467
5,537,409	7/1996	Moriyama et al.	370/471

Primary Examiner—Alpus H. Hsu
Attorney, Agent, or Firm—Barry R. Lipsitz

[57] **ABSTRACT**

Audio data is processed from a packetized data stream carrying digital television information in a succession of fixed length transport packets. Some of the packets contain a presentation time stamp (PTS) indicative of a time for commencing the output of associated audio data. After the audio data stream has been acquired, the detected audio packets are monitored to locate subsequent PTS's for adjusting the timing at which audio data is output, thereby providing proper lip synchronization with associated video. Errors in the audio data are processed in a manner which attempts to maintain synchronization of the audio data stream while masking the errors. In the event that the synchronization condition cannot be maintained, for example in the presence of errors over more than one audio frame, the audio data stream is reacquired while the audio output is concealed. An error condition is signaled to the audio decoder by altering the audio synchronization word associated with the audio frame in which the error has occurred.

25 Claims, 4 Drawing Sheets



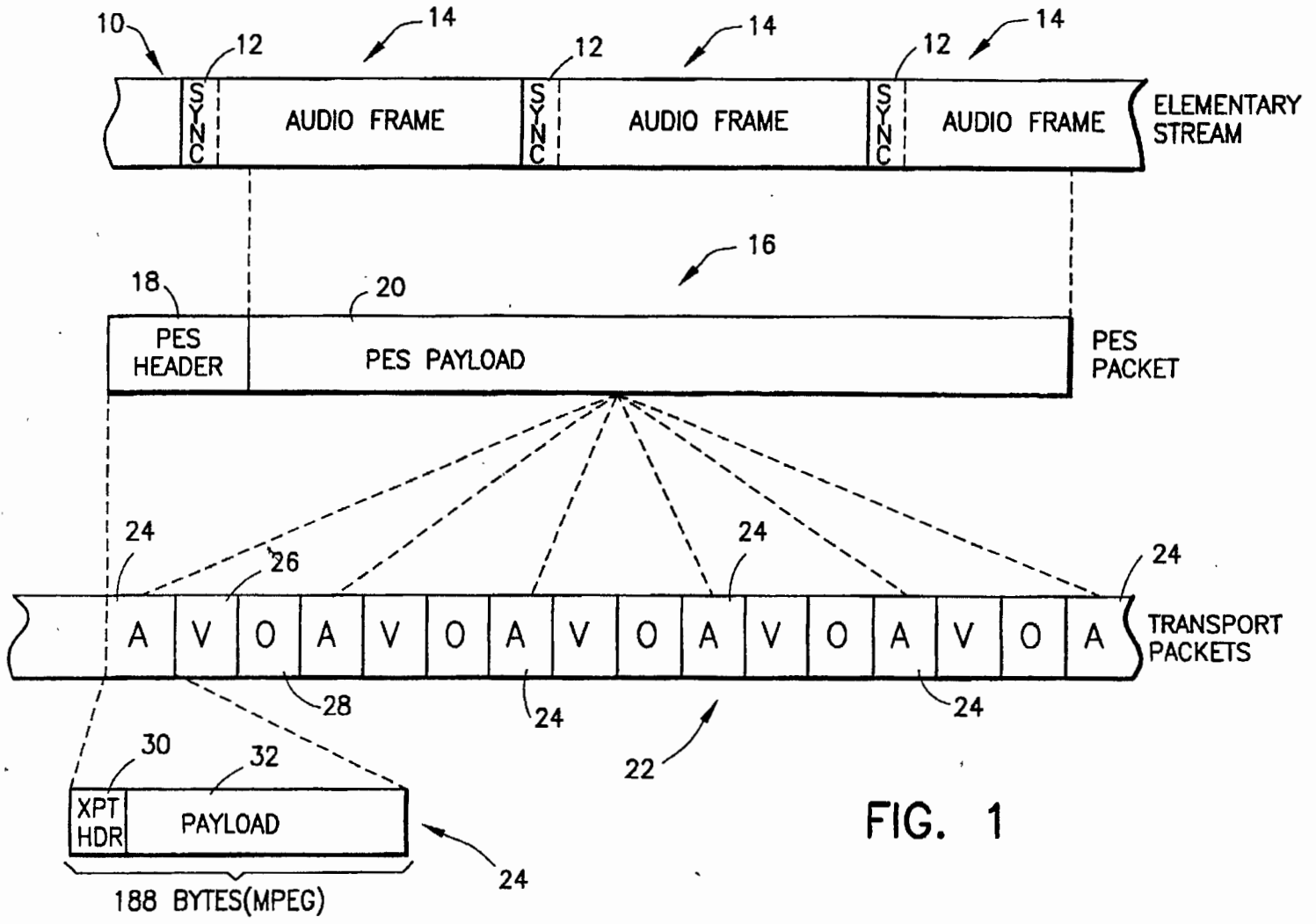

10
12
14
SYNC
AUDIO FRAME
ELEMENTARY STREAM
16
18
20
PES HEADER
PES PAYLOAD
PES PACKET
24
26
A V O A V O A V O A V O A V O A
TRANSPORT PACKETS
28
22
30
32
XPT HDR
PAYLOAD
188 BYTES(MPEG)

FIG. 1

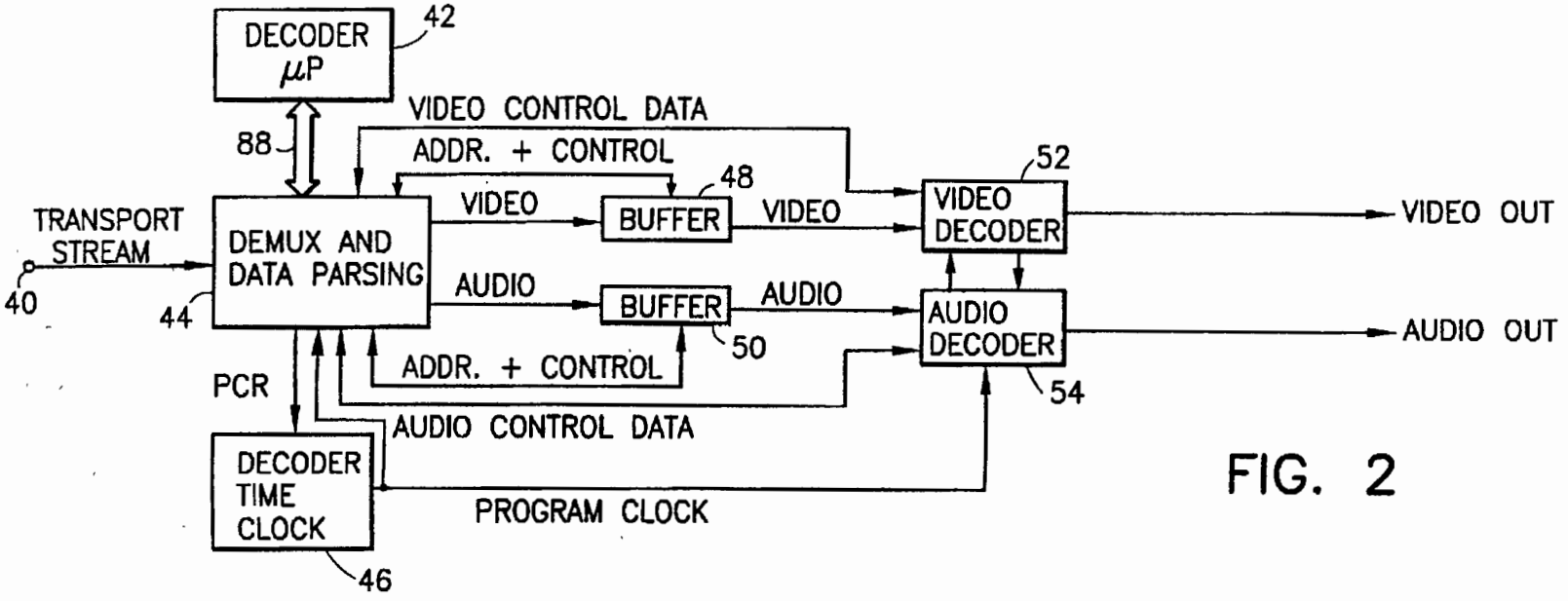

DECODER
μP
42
88
VIDEO CONTROL DATA
ADDR. + CONTROL
48
52
TRANSPORT
STREAM
40
DEMUX AND
DATA PARSING
44
VIDEO
BUFFER
VIDEO
VIDEO
DECODER
VIDEO OUT
AUDIO
BUFFER
AUDIO
AUDIO
DECODER
AUDIO OUT
50
ADDR. + CONTROL
PCR
54
AUDIO CONTROL DATA
DECODER
TIME
CLOCK
PROGRAM CLOCK
46

FIG. 2

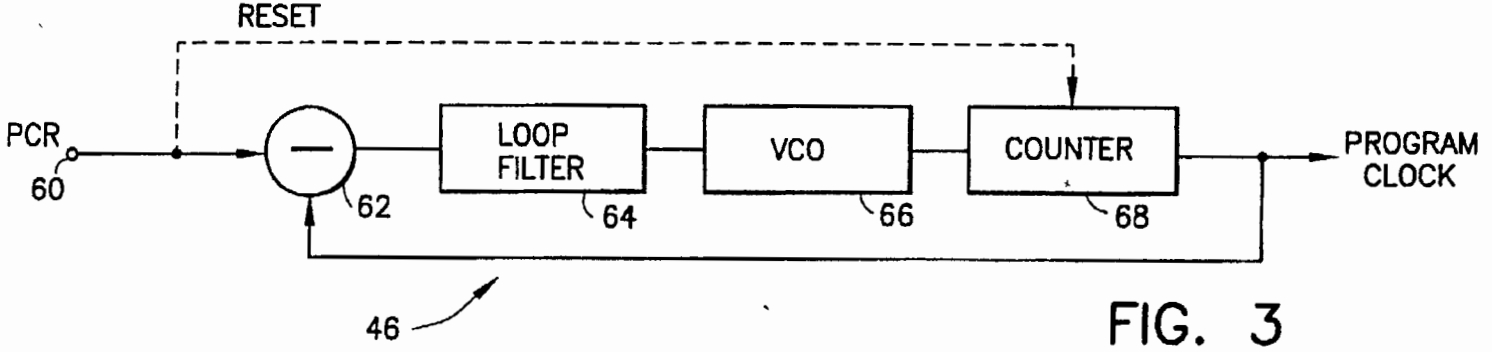
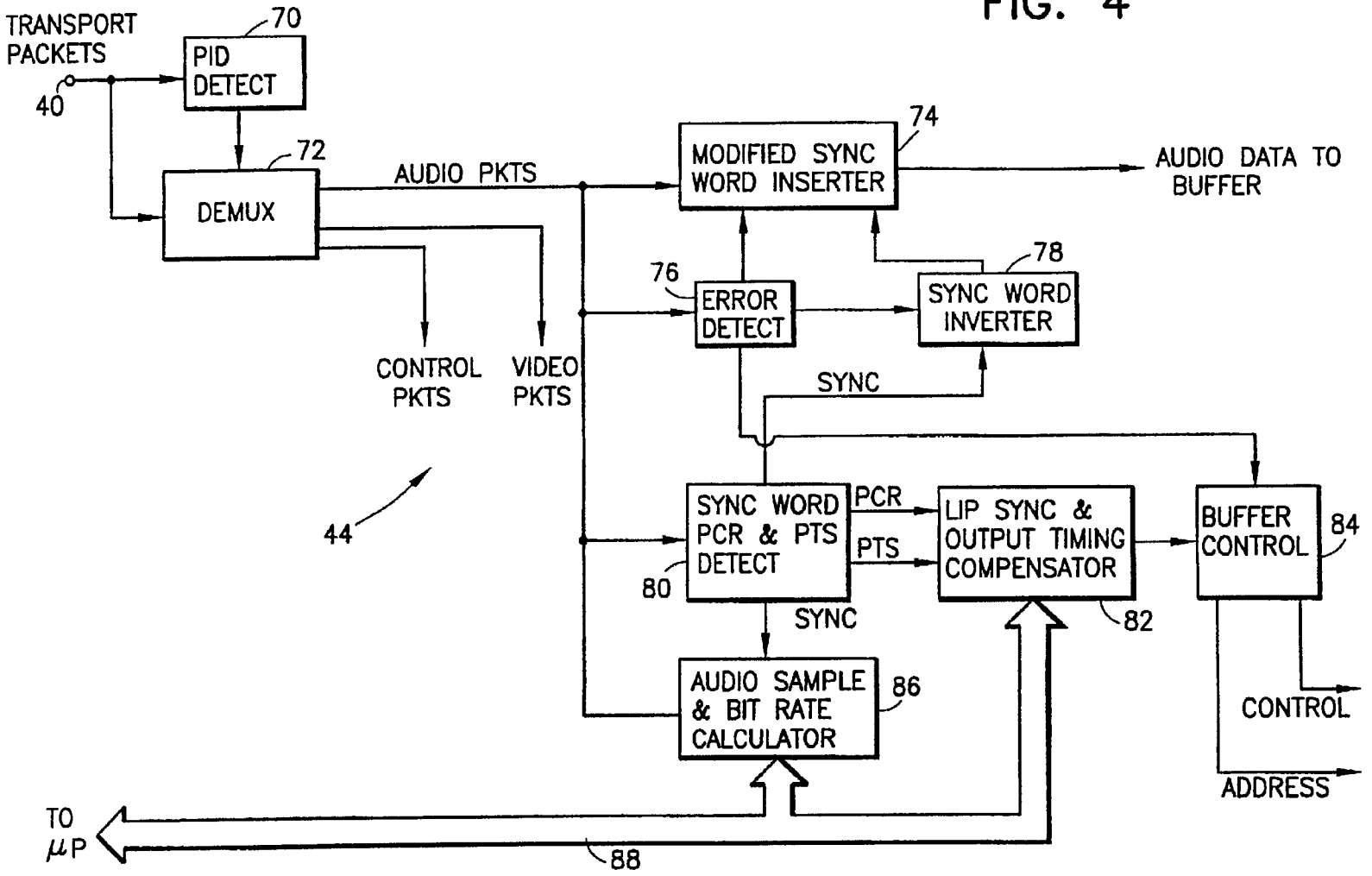

RESET
PCR
60
62
LOOP
FILTER
64
VCO
66
COUNTER
68
PROGRAM
CLOCK
46

FIG. 3

FIG. 4


TRANSPORT PACKETS
40
PID DETECT
70
DEMUX
72
AUDIO PKTS
MODIFIED SYNC WORD INSERTER
74
AUDIO DATA TO BUFFER
ERROR DETECT
76
SYNC WORD INVERTER
78
CONTROL PKTS
VIDEO PKTS
SYNC
44
SYNC WORD PCR & PTS DETECT
80
PCR
PTS
LIP SYNC & OUTPUT TIMING COMPENSATOR
82
BUFFER CONTROL
84
SYNC
AUDIO SAMPLE & BIT RATE CALCULATOR
86
CONTROL
ADDRESS
TO μP
88

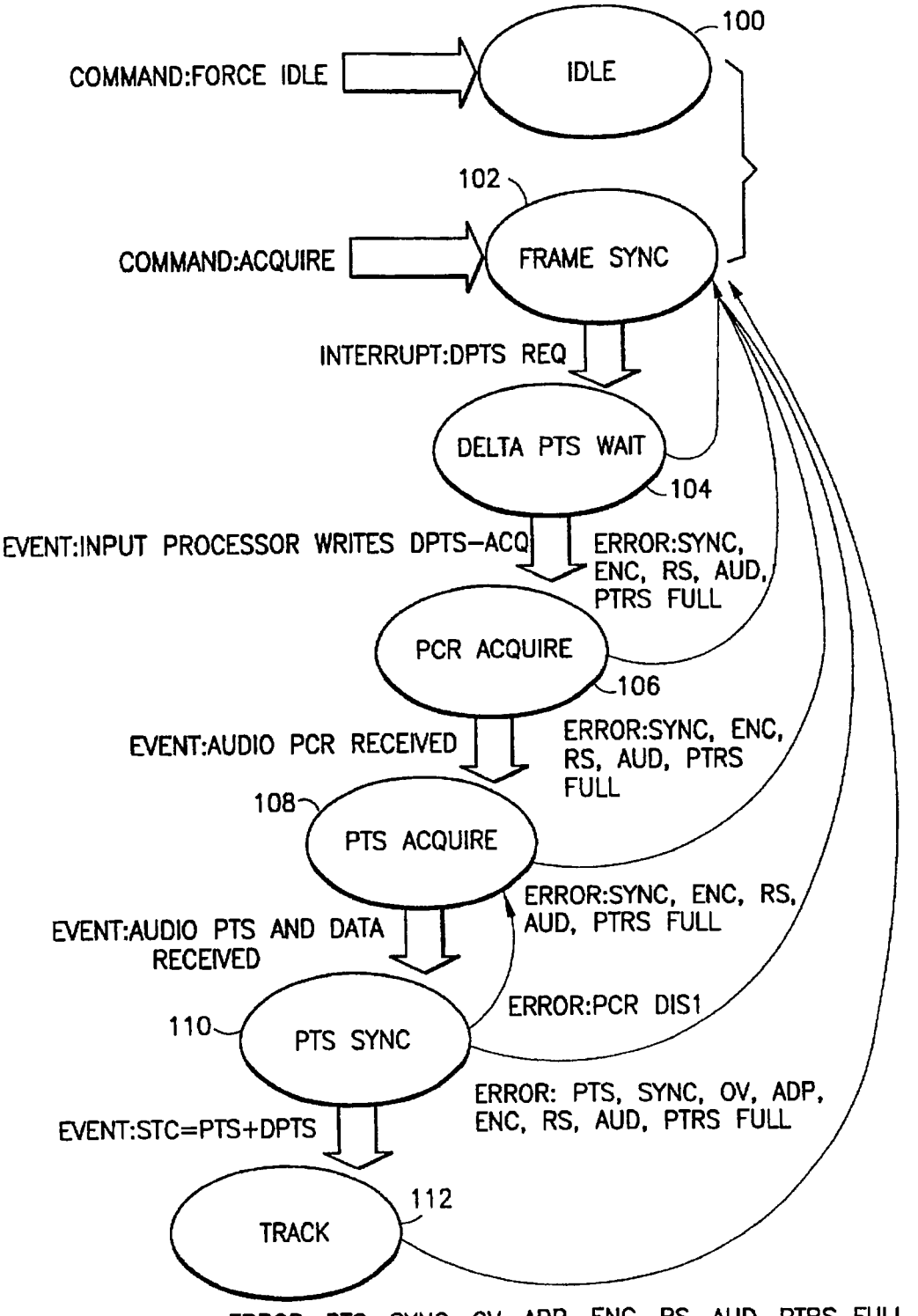

100
IDLE
COMMAND:FORCE IDLE
102
FRAME SYNC
COMMAND:ACQUIRE
INTERRUPT:DPTS REQ
DELTA PTS WAIT
104
EVENT:INPUT PROCESSOR WRITES DPTS-ACQ
ERROR:SYNC, ENC, RS, AUD, PTRS FULL
PCR ACQUIRE
106
EVENT:AUDIO PCR RECEIVED
ERROR:SYNC, ENC, RS, AUD, PTRS FULL
108
PTS ACQUIRE
ERROR:SYNC, ENC, RS, AUD, PTRS FULL
EVENT:AUDIO PTS AND DATA RECEIVED
ERROR:PCR DIS1
110
PTS SYNC
ERROR: PTS, SYNC, OV, ADP, ENC, RS, AUD, PTRS FULL
EVENT:STC=PTS+DPTS
112
TRACK
ERROR: PTS, SYNC, OV, ADP, ENC, RS, AUD, PTRS FULL

FIG. 5

ACQUISITION AND ERROR RECOVERY OF AUDIO DATA CARRIED IN A PACKETIZED DATA STREAM

BACKGROUND OF THE INVENTION

The present invention relates to a method and apparatus for acquiring audio data from a packetized data stream and recovery from errors contained in such data.

Various standards have emerged for the transport of digital data, such as digital television data. Examples of such standards include the Moving Pictures Experts Group (MPEG) standards and the DigiCipher® II standard proprietary to General Instrument Corporation of Chicago, Ill., U.S.A., the assignee of the present invention. The DigiCipher® II standard extends the MPEG-2 systems and video standards, which are widely known and recognized as transport and video compression specifications specified by the International Standards Organization (ISO) in Document series ISO 13818. The MPEG-2 specification's systems "layer" provides a transmission medium independent coding technique to build bitstreams containing one or more MPEG programs. The MPEG coding technique uses a formal grammar ("syntax") and a set of semantic rules for the construction of bitstreams. The syntax and semantic rules include provisions for demultiplexing, clock recovery, elementary stream synchronization and error handling.

The MPEG transport stream is specifically designed for use with media that can generate data errors. Many programs, each comprised of one or more elementary streams, may be combined into a transport stream. Examples of services that can be provided using the MPEG format are television services broadcast over terrestrial, cable television and satellite networks as well as interactive telephony-based services. The primary mode of information carriage in MPEG broadcast applications will be the MPEG-2 transport stream. The syntax and semantics of the MPEG-2 transport stream are defined in International Organisation for Standardisation, ISO/IEC 13818-1, International Standard, 1994 entitled "Generic Coding of Moving Pictures and Associated Audio: Systems," recommendation H.222, incorporated herein by reference.

Multiplexing according to the MPEG-2 standard is accomplished by segmenting and packaging elementary streams such as compressed digital video and audio into packetized elementary stream (PES) packets which are then segmented and packaged into transport packets. As noted above, each MPEG transport packet is fixed at 188 bytes in length. The first byte is a synchronization byte having a specific eight-bit pattern, e.g., 01000111. The sync byte indicates the beginning of each transport packet.

Following the sync byte is a three-byte field which includes a one-bit transport packet error indicator, a one-bit payload unit start indicator, a one-bit transport priority indicator, a 13-bit packet identifier (PID), a two-bit transport scrambling control, a two-bit adaptation field control, and a four-bit continuity counter. The remaining 184 bytes of the packet may carry the data to be communicated. An optional adaptation field may follow the prefix for carrying both MPEG related and private information of relevance to a given transport stream or the elementary stream carried within a given transport packet. Provisions for clock recovery, such as a program clock reference (PCR), and bitstream splicing information are typical of the information carried in the adaptation field. By placing such information in an adaptation field, it becomes encapsulated with its associated data to facilitate remultiplexing and network routing operations. When an adaptation field is used, the payload is correspondingly shorter in length.

The PCR is a sample of the system time clock (STC) for the associated program at the time the PCR bytes are received at the decoder. The decoder uses the PCR values to synchronize a decoder system time clock (STC) with the encoder's system time clock. The lower nine bits of a 42-bit STC provide a modulo-300 counter that is incremented at a 27 MHz clock rate. At each modulo-300 rollover, the count in the upper 33 bits is incremented, such that the upper bits of the STC represent time in units of a 90 kHz clock period. This enables presentation time stamps (PTS) and decode time stamps (DTS) to be used to dictate the proper time for the decoder to decode access units and to present presentation units with the accuracy of one 90 kHz clock period. Since each program or service carried by the data stream may have its own PCR, the programs can be multiplexed asynchronously.

Synchronization of audio, video and data presentation within a program is accomplished using a time stamp approach. Presentation time stamps (PTSs) and/or decode time stamps (DTSs)are inserted into the transport stream for the separate video and audio packets. The PTS and DTS information is used by the decoder to determine when to decode and display a picture and when to play an audio segment. The PTS and DTS values are relative to the same system time clock sampled to generate the PCRs.

All MPEG video and audio data must be formatted into a packetized elementary stream (PES) formed from a succession of PES packets. Each PES packet includes a PES header followed by a payload. The PES packets are then divided into the payloads of successive fixed length transport packets.

PES packets are of variable and relatively long length. Various optional fields, such as the presentation time stamps and decode time stamps may be included in the PES header. When the transport packets are formed from the PES, the PES headers immediately follow the transport packet headers. A single PES packet may span many transport packets and the subsections of the PES packet must appear in consecutive transport packets of the same PID value. It should be appreciated, however, that these transport packets may be freely multiplexed with other transport packets having different PIDs and carrying data from different elementary streams within the constraints of the MPEG-2 Systems specification.

Video programs are carried by placing coded MPEG video streams into PES packets which are then divided into transport packets for insertion into a transport stream. Each video PES packet contains one or more coded video pictures, referred to as video "access units." A PTS and/or a DTS value may be placed into the PES packet header that encapsulates the associated access units. The DTS indicates when the decoder should decode the access unit into a presentation unit. The PTS is used to actuate the decoder to present the associated presentation unit.

Audio programs are provided in accordance with the MPEG Systems specification using the same specification of the PES packet layer. PTS values may be included in those PES packets that contain the first byte of an audio access unit (sync frame). The first byte of an audio access unit is part of an audio sync word. An audio frame is defined as the data between two consecutive audio sync words, including the preceding sync word and not including the succeeding sync word.

In DigiCipher® II, audio transport packets include one or both of an adaptation field and payload field. The adaptation field may be used to transport the PCR values. The payload field transports the audio PES, consisting of PES headers and PES data. PES headers are used to transport the audio PTS values. Audio PES data consists of audio frames as specified, e.g., by the Dolby® AC-3 or Musicam audio syntax specifications. The AC-3 specifications are set forth in a document entitled Digital Audio Compression (AC-3), ATSC Standard, Doc. A/52, United States Advanced Television Systems Committee. The Musicam specification can be found in the document entitled "Coding of Moving Pictures and Associated Audio for Digital Storage Media at Up to About 1.5 MBIT/s," *Part* 3 Audio, 11172-3 (MPEG-1) published by ISO. Each syntax specifies an audio sync frame as audio sync word, followed by audio information including audio sample rate, bit rate and/or frame size, followed by audio data.

In order to reconstruct a television signal from the video and audio information carried in an MPEG/DigiCipher® II transport stream, a decoder is required to process the video packets for output to a video decompression processor (VDP) and the audio packets for output to an audio decompression processor (ADP). In order to properly process the audio data, the decoder is required to synchronize to the audio data packet stream. In particular, this is required to enable audio data to be buffered for continuous output to the ADP and to enable the audio syntax to be read for audio rate information necessary to delay the audio output to achieve proper lip synchronization with respect to the video of the same program.

Several events can result in error conditions with respect to the audio processing. These include loss of audio transport packets due to transmission channel errors. Errors will also result from the receipt of audio packets which are not properly decrypted or are still encrypted. A decoder must be able to handle such errors without significantly degrading the quality of the audio output.

The decoder must also be able to handle changes in the audio sample rate and audio bit rate. The audio sample rate for a given audio elementary stream will rarely change. The audio bit rate, however, can often change at program boundaries, and at the start and end of commercials. It is difficult to maintain synchronization to the audio stream through such rate changes, since the size of the audio sync frames is dependent on the audio sample rate and bit rate. Handling undetected errors in the audio stream, particularly in systems where error detection is weak, complicates the tracking of the audio stream through rate changes. When a received bitstream indicates that an audio rate has changed, the rate may or may not have actually changed. If the decoder responds to an indication from the bitstream that the audio rate has changed when the indication is in error and the rate has not changed, a loss of audio synchronization will likely occur. This can result in an audio signal degradation that is noticeable to an end user.

To support an audio sample rate change, the audio clock rates utilized by the decoder must be changed. This process can take significant time, again degrading the quality of the audio output signal. Still further, such a sample rate change will require the audio buffers to be cleared to establish a different sample-rate-dependent lip sync delay. Thus, it may not be advantageous to trust a signal in the received bitstream indicating that the audio sample rate has changed.

With respect to bit rate changes, the relative frequency of such changes compared to undetected errors in the bit rate information will be dominated by whether the receiver has adequate error detection. Thus, it would be advantageous to provide a decoder having two modes of operation. In a robust error detection environment such as for satellite communications or cable media, where error detection is robust, a seamless mode of operation can be provided by trusting a bit rate change indication provided by the data. In a less robust error detection environment, indications of bit rate changes can be ignored, at the expense of requiring resynchronization of the audio in the event that the bit rate has actually changed.

It would be further advantageous to provide an audio decoder in which synchronization to the audio bitstream is maintained when the audio data contains errors. Such a decoder should conceal the audio for those sync frames in which an error has occurred, to minimize the aural impact of audio data errors.

It would be still further advantageous to provide a decoder in which the timing at which audio data is output from the decoder's audio buffer is adjusted on an ongoing basis. The intent of such adjustments would be to insure correct presentation time for audio elementary streams.

The present invention provides methods and apparatus for decoding digital audio data from a packetized transport stream having the aforementioned and other advantages.

SUMMARY OF THE INVENTION

In accordance with the present invention, a method is provided for processing digital audio data from a packetized data stream carrying television information in a succession of fixed length transport packets. Each of the packets includes a packet identifier (PID). Some of the packets contain a program clock reference (PCR) value for synchronizing a decoder system time clock (STC). Some of the packets contain a presentation time stamp (PTS) indicative of a time for commencing the output of associated data for use in reconstructing a television signal. In accordance with the method, the PID's for the packets carried in the data stream are monitored to identify audio packets associated with the desired program. The audio packets are examined to locate the occurrence of at least one audio synchronization word therein for use in achieving a synchronization condition. The audio packets are monitored after the synchronization condition has been achieved to locate an audio PTS. After the PTS is located, the detected audio packets are searched to locate the next audio synchronization word. Audio data following the next audio synchronization word is stored in a buffer. The stored audio data is output from the buffer when the decoder system time clock reaches a specified time derived from the PTS. The detected audio packets are continually monitored to locate subsequent audio PTS's for adjusting the timing at which the stored audio data is output from the buffer on an ongoing basis.

A PTS pointer can be provided to maintain a current PTS value and an address of the buffer identifying where the sync word of an audio frame referred to by the current PTS is stored. In order to provide the timing adjustment, the PTS value in the PTS pointer is replaced with a new PTS value after data stored at the address specified by the PTS pointer has been output from the buffer. The address specified by the PTS pointer is then replaced with a new address corresponding to the sync word of an audio frame referred to by the new PTS value. The output of data from the buffer is suspended when the new buffer address is reached during the presentation process. The output of data from the buffer is recommenced when the decoder's system time clock reaches a specified time derived from the new PTS value.

In an illustrated embodiment, the output of data from the buffer is recommenced when the decoder's system time clock reaches the time indicated by the sum of the new PTS value and an offset value. The offset value provides proper lip synchronization by accounting for any decoder video signal processing delay. In this manner, after the audio and video data has been decoded, the audio data can be presented synchronously with the video data so that, for example, the movement of a person's lips in the video picture will be sufficiently synchronous to the sound reproduced.

The method of the present invention can comprise the further step of commencing a reacquisition of the audio synchronization condition if the decoder's system time clock is beyond the specified time derived from the new PTS value before the output of data from the buffer is recommenced. Thus, if a PTS designates that an audio frame should be presented at a time which has already passed, reacquisition of the audio data will automatically commence to correct the timing error, thus minimizing the duration of the resultant audio artifact.

In the illustrated embodiment, two consecutive audio synchronization words define an audio frame therebetween, including the preceding sync word, but not including the succeeding sync word. The occurrence of errors may be detected in the audio packets. Upon detecting a first audio packet of a current audio frame containing an error, the write pointer for the buffer is advanced by the maximum number of bytes (N) contained in one of the fixed length transport packets. At the same time, the current audio frame is designated as being in error. The subsequent audio packets of the current audio frame are monitored for the next audio synchronization word after the error has been detected. If the synchronization word is not received at the expected point in the audio elementary stream, subsequent data is not stored in the buffer until the sync word is located. Storage of audio data into the buffer is resumed with the next sync word if the next audio synchronization word is located within N bytes after the commencement of the search therefor. If the next audio synchronization word is not located within N bytes after the commencement of the search therefor, a reacquisition of the synchronization condition is commenced. These steps will insure the buffer is maintained at the correct fullness when as many as one transport packet is lost per audio sync frame, even with the sync frame size changes such as with a sample rate of 44.1 ksps, and will resynchronize the audio when too many audio transport packets are lost.

Whenever the audio data from which the television audio is being reconstructed is in error, it is preferable to conceal the error in the television audio. In the illustrated embodiment, a current audio frame is designated as being in error by altering the audio synchronization word for that frame. For example, every other bit of the audio synchronization word can be inverted. The error in the television audio for the corresponding audio frame may then be concealed in response to an altered synchronization word during the decoding and presentation process. This method allows the buffering and error detection process to signal the decoding and presentation process when errors occur via the data itself, without the need for additional interprocess signals.

The audio data can include information indicative of an audio sample rate and audio bit rate, at least one of which is variable. In such a situation, it is advantageous to maintain synchronization within the audio elementary stream during a rate change indicated by the audio data. This can be accomplished by ignoring an audio sample rate change indicated by the audio data on the assumption that the sample rate has not actually changed, and concealing the audio frame containing the data indicative of an audio sample rate change while attempting to maintain the synchronization condition. This strategy will properly respond to an event in which the audio sample rate change or bit rate change indication is the result of an error in the indication itself, as opposed to an actual rate change.

Similarly, audio data can be processed in accordance with a new rate indicated by the audio data in the absence of an error indication pertaining to the audio frame containing the new rate, while attempting to maintain the synchronization condition. The audio data is processed without changing the rate if an error indication pertains to the audio frame containing the new rate. At the same time, the audio frame to which the error condition pertains is concealed while the decoder attempts to maintain the synchronization condition. If the synchronization condition cannot be maintained, a reacquisition of the synchronization condition is commenced, as desired when the sample rate actually changes.

Apparatus in accordance with the present invention acquires audio information carried by a packetized data stream. The apparatus also handles errors contained in the audio information. Means are provided for identifying audio packets in the data stream. An audio elementary stream is recovered from the detected audio packets for storage in a buffer. An audio presentation time stamp (PTS) is located in the detected audio packets. Means responsive to the PTS are provided for commencing the output of audio data from the buffer at a specified time. Means are provided for monitoring the detected audio packets after the output of audio data from the buffer has commenced, in order to locate subsequent audio PTS's for use in governing the output of audio data from the buffer to insure audio is presented synchronous to any other elementary streams of the same program and to maintain correct buffer fullness.

The apparatus can further comprise means for maintaining a PTS pointer with a current PTS value and an address of the buffer identifying where a portion of audio data referred to by the current PTS is stored. Means are provided for replacing the PTS value in the PTS pointer with a new current PTS value after data stored at the address set forth in the PTS pointer has been output from the buffer. The address in the PTS pointer is then replaced with a new address corresponding to a portion of audio data referred to by the new current PTS value. Means responsive to the PTS pointer are provided for suspending the output of data from the buffer when the new address is reached. Means are provided for recommencing the output of data from the buffer at a time derived from the new current PTS value. In the event that the new current PTS value is outside a predetermined range, means provided in the apparatus conceal the audio signal and reestablish synchronization.

In an illustrated embodiment, the audio transport packets have a fixed length of M bytes. The transport packets carry a succession of audio frames each contained wholly or partially in said packets. The audio frames each begin with an audio synchronization word. Means are provided for detecting the occurrence of errors in the audio packets. A write pointer for the buffer is advanced by the maximum number of audio frame bytes per audio transport packet (N) and a current audio frame is designated as being in error upon detecting an error in an audio packet of the current audio frame. Means are provided for monitoring the detected audio packets of the current audio frame for the next audio synchronization word after the error has been detected. If the

synchronization word is not received where expected within the audio elementary stream, subsequent audio data is not buffered until the next audio synchronization word is received. This process compensates for too many audio bytes having been buffered when the errored audio packet was detected. Such an event will occur each time the lost packet does not carry the maximum number of possible audio data bytes. Means are provided for resuming the storage of audio data in the buffer if the next audio synchronization word is located within N bytes after the commencement of the search therefor. If the next audio synchronization word is not located within said N bytes after the commencement of the search therefor, the audio timing will be reacquired. In this manner, the size of the sync frames buffered will be maintained including for those frames that are marked as being in error, unless the next sync word is not located where expected in the audio elementary stream to recover from the error before buffering any of the next successive frame. This algorithm allows the decode and presentation processes to rely on buffered audio frames being the correct size in bytes, even when data errors result in the loss of an unknown amount of audio data.

Means can also be provided for concealing error in an audio signal reproduced from data output from the buffer when the data output from the buffer is in error. Means are further provided for altering the audio synchronization word associated with a current audio frame, to signal the decode and presentation process that a particular frame is in error. The concealing means are responsive to altered synchronization words for concealing audio associated with the corresponding audio frame.

Decoder apparatus in accordance with the invention acquires audio information carried by a packetized data stream and handles errors therein. Means are provided for identifying audio packets in the data stream. The successive audio frames are extracted from the audio transport packets. Each audio frame is carried by one or more of the packets, and the start of each audio frame is identified by an audio synchronization word. Means responsive to the synchronization words obtain a synchronization condition enabling the recovery of audio data from the detected audio packets for storage in a buffer. Means are provided for detecting the presence of errors in the audio data. Means responsive to the error detecting means control the flow of data through the buffer when an error is present, to attempt to maintain the synchronization condition while masking the error. Means are provided for reestablishing the audio timing if the controlling means cannot maintain the synchronization condition.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a diagrammatic illustration showing how audio transport packets are formed from an elementary stream of audio data;

FIG. 2 is a block diagram of decoder apparatus that can be used in accordance with the present invention;

FIG. 3 is a more detailed block diagram of the decoder system time clock (STC) illustrated in FIG. 2;

FIG. 4 is a more detailed block diagram of the demultiplexing and data parsing circuit of FIG. 2; and

FIG. 5 is a state diagram illustrating the processing of audio data in accordance with the present invention.

DETAILED DESCRIPTION OF THE INVENTION

FIG. 1 is a diagrammatic illustration showing how one or more digital programs can be multiplexed into a stream of transport packets. Multiplexing is accomplished by segmenting elementary streams such as coded video and audio into PES packets and then segmenting these into transport packets. The figure is illustrative only, since a PES packet, such as PES packet **16** illustrated, will commonly translate into other than the six transport packets **24** illustrated.

In the example of FIG. **1**, an elementary stream generally designated **10** contains audio data provided in audio frames **14** delineated by synchronization words **12**. Similar elementary streams will be provided for video data and other data to be transported.

The first step in forming a transport packet stream is to reconfigure the elementary stream for each type of data into a corresponding packetized elementary stream (PES) formed from successive PES packets, such as packet **16** illustrated. Each PES packet contains a PES header **18** followed by a PES payload **20**. The payload comprises the data to be communicated. The PES header **18** will contain information useful in processing the payload data, such as the presentation time stamp (PTS).

The header and payload data from each PES packet are encapsulated into transport packets **24**, each containing a transport header **30** and payload data **32**. The payload data of the transport packet **24** will contain a portion of the payload data **20** and/or PES header **18** from PES packet **16**. In an MPEG implementation, the transport header **30** will contain the packet identifier (PID) which identifies the transport packet, such as an audio transport packet **24**, a video transport packet **26**, or other data packet **28**. In FIG. **1**, only the derivation of the audio transport packets **24** is shown. In order to derive video packets **26** and other packets **28**, corresponding elementary streams (not shown) are provided which are processed into PES packets and transport packets in essentially the same manner illustrated in FIG. **1** with respect to the formation of the audio transport packets

Each MPEG transport packet contains 188 bytes of data, formed from the four-byte transport header **30** and payload data **32**, which can be up to 184 bytes. In the MPEG implementation, an adaptation field of, e.g., eight bytes may be provided between the transport header **30** and payload **32**. The variable length adaptation field can contain, for example, the program clock reference (PCR) used for synchronization of the decoder system time clock (STC).

The plurality of audio transport packets **24**, video transport packets **26** and other packets **28** is multiplexed as illustrated in FIG. **1** to form a transport stream **22** that is communicated over the communication channel from the encoder to the decoder. The purpose of the decoder is to demultiplex the different types of transport packets from the transport stream, based on the PID's of the individual packets, and to then process each of the audio, video and other components for use in reconstructing a television signal.

FIG. **2** is a block diagram of a decoder for recovering the video and audio data. The transport stream **22** is input to a demultiplexer and data parsing subsystem **44** via terminal **40**. The demultiplexing and data parsing subsystem communicates with a decoder microprocessor **42** via a data bus **88**. Subsystem **44** recovers the video and audio transport packets from the transport packet stream and parses the PCR, PTS and other necessary data therefrom for use by other decoder components. For example, PCR's are recovered from adaptation fields of transport packets for use in synchronizing a decoder system time clock (STC) **46** to the system time clock of the encoder. Presentation time stamps for the video and audio data streams are recovered from the

respective PES packet headers and communicated as video or audio control data to the video decoder **52** and audio decoder **54**, respectively.

The decoder time clock **46** is illustrated in greater detail in FIG. **3**. An important function of the decoder is the reconstruction of the clock associated with a particular program. This clock is used to reconstruct, for example, the proper horizontal scan rate for the video. The proper presentation rate of audio and video presentation units must also be assured. These are the audio sample rate and the video frame rate. Synchronization of the audio to the video, referred to as "lip sync", is also required.

In order to generate a synchronized program clock, the decoder system time clock (STC) **46** receives the PCR's via terminal **60**. Before the commencement of the transport stream decoding, a PCR value is used to preset a counter **68** for the decoder system time clock. As the clock runs, the value of this counter is fed back to a subtracter **62**. The local feedback value is then compared with subsequent PCR's in the transport stream as they arrive at terminal **60**. When a PCR arrives, it represents the correct STC value for the program. The difference between the PCR value and the STC value, as output from subtracter **62**, is filtered by a loop filter **64** and used to drive the instantaneous frequency of a voltage controlled oscillator **66** to either decrease or increase the STC frequency as necessary. The STC has both a 90 kHz and 27 MHz component, and the loop filter **64** converts this to units in the 27 Mhz domain. The output of the VCO **66** is a 27 MHz oscillator signal which is used as the program clock frequency output from the decoder system time clock. Those skilled in the art will recognize that the decoder time clock **46** illustrated in FIG. **3** is implemented using well known phase locked loop (PLL) techniques.

Before beginning audio synchronization, the decoder of FIG. **2**, and particularly subsystem **44**, will remain idle until it is configured by decoder microprocessor **42**. The configuration consists of identifying the type of audio data stream to be processed (e.g., Dolby AC-3 or Musicam audio), identifying the PID of packets from which the audio PCR values are to be extracted, and identifying the PID for audio packets.

During the idle state, subsystem **44** will instruct audio decoder **54** to conceal the audio output. Concealment can be accomplished by zeroing all of the audio samples. Subsequent digital signal processing will result in a smooth aural transition from no sound to sound, and back to no sound. The concealment of the audio output will be terminated when the synchronization process reaches a tracking state. Decoder microprocessor **42** configures the audio format as AC-3 or Musicam, depending on whether audio decoder **54** is an AC-3 or Musicam decoder. Microprocessor **42** determines the audio PID and audio PCR PID from program map information provided in the transport stream. The program map information is essentially a directory of PID's, and is identified via its own PID.

Once the demultiplexer and data parsing subsystem **44** is commanded to enter a Frame Sync state via an acquire command, it will begin searching for two consecutive audio sync words and will supply the decoder microprocessor **42** with the audio sampling rate and audio bit rate indicated within the audio elementary stream. To locate the sync words, subsystem **44** will receive transport packets on the audio PID and extract the PES data, searching for the occurrence of the audio sync word, which is a predetermined, fixed word. For example, the AC-3 audio sync word is 0000 1011 0111 0111 (16 bits) while the Musicam sync word is 1111 1111 1111 (12 bits).

The number of bits between the first bit of two consecutive audio sync words is referred to as the frame size. The frame size depends on whether the audio stream is AC-3 or Musicam and has a different value for each combination of audio sample and bit rate. In a preferred embodiment, subsystem **44** is required to synchronize to AC-3 and Musicam sample rates of 44.1 ksps and 48 ksps. The AC-3 audio syntax conveys the audio sample rate and audio frame size while the Musicam audio syntax conveys the audio sample rate and audio bit rate. Both AC-3 and Musicam specify one sync frame size for each bit rate when the sample rate is 48 ksps. However, AC-3 and Musicam specify two sync frame sizes for each bit rate when the sample rate is 44.1 ksps, a fact which complicates synchronization, especially through packet loss. When the sample rate is 44.1 ksps, the correct sync frame size between the two possibilities is indicated by the least significant bit of the AC-3 frame size code or by a Musicam padding bit.

Once two consecutive audio sync words have been received with the correct number of bytes in between, as specified by the sync frame size, subsystem **44** will store the audio sample rate and audio bit rate implied by the audio syntax for access by the decoder microprocessor **42**, interrupting the microprocessor to indicate that subsystem **44** is waiting for the microprocessor to supply it with an audio PTS correction factor. The correction factor is necessary in order to know when to output audio data to the audio decoder **54** during initial acquisition and during tracking for proper lip synchronization. The value is denoted as dPTS. The lip sync value used for tracking is slightly less than that used for initial acquisition to allow for time errors which will exist between any two PTS values, namely that which is used for acquisition and those which are used for tracking.

Decoder microprocessor **42** sets the correction factors such that audio and video will exit the decoder with the same time relationship as it entered the encoder, thus achieving lip synchronization. These correction factors are determined based on audio sample rate and video frame rate (e.g., 60 Hz or 50 Hz). These dependencies exist because the audio decompression processing time required by audio decoder **54** potentially depends on audio sample and bit rate while the video decompression implemented by video decoder **52** potentially depends on video frame rate and delay mode. In a preferred implementation, the PTS correction factors consist of 11 bits, representing the number of 90 kHz clock periods by which audio data is to be delayed before output to the audio decoder **54**. With 11 bit values, the delay can be as high as 22.7 milliseconds.

Once the demultiplexing and data parsing subsystem **44** requests the decoder microprocessor **42** to supply the correction factors, it will monitor reception of consecutive sync words at the expected positions within the audlo elementary stream. If an error condition occurs during this time, subsystem **44** will transition to searching for two consecutive audio sync words with the correct number of data bytes in between. Otherwise, subsystem **44** remains in State dPTS-wait until the decoder microprocessor services the interrupt from subsystem **44** by writing $dPTS_{acq}$ to subsystem **44**.

Once subsystem **44** is provided with the PTS correction factors, it checks whether a transport packet has been received on the audio PCR PID containing a PCR value, carried in the adaptation field of the packet. Until this has occurred, reception of consecutive sync words will continue [State=PCR Acquire]. If an error condition occurs during this time, subsystem **44** will transition to searching for two consecutive audio sync words [State=Frame Sync]. Otherwise, it will remain in State=PCR Acquire until it receives a PCR value on the audio PCR PID.

After a PCR has been acquired, subsystem **44** will begin searching for a PTS [State=PTS Acquire], which is carried in the PES header of the audio transport packets. Until this has occurred, subsystem **44** will monitor the reception of consecutive sync words. If an error condition occurs during this time, it will transition to an error handling algorithm [State=Error Handling]. Otherwise, it will remain in the PTS acquire state until it receives a PTS value on the audio PID.

When subsystem **44** receives an audio PTS value, it will begin searching for reception of the next audio sync word. This is important since the PTS defines the time at which to output the data which begins with the next audio frame. Since audio frames are not aligned with the audio PES, the number of bytes which will be received between the PTS and the next audio sync word varies with time. If an error condition occurs before reception of the next audio sync word, subsystem **44** returns to searching for audio frame synchronization [State=Frame Sync]. It should be appreciated that since audio sync frames and PES headers are not aligned, it is possible for a PES header, and the PTS which it may contain, to be received between the 12 or 16 bits which form an audio sync word. In this case, the sync word to which the PTS refers is not the sync word which is split by the PES header, but rather the following sync word.

When subsystem **44** receives the next sync word, it has acquired PTS. At this point, it will store the received PTS and the PES data (starting with the sync word which first followed the PTS) into an audio buffer **50**, together with the buffer address at which it writes the sync word. This stored PTS/buffer address pair will allow subsystem **44** to begin outputting audio PES data to the audio decoder **54** at the correct time, starting with the audio sync word. In a preferred embodiment, the buffer **50** is implemented in a portion of dynamic random access memory (DRAM) already provided in the decoder.

Once subsystem **44** begins buffering audio data, a number of parameters must be tracked which will allow it to handle particular error conditions, such as loss of an audio transport packet to transmission errors. These parameters can be tracked using audio pointers including a PTS pointer, a DRAM offset address pointer, and a valid flag pointer discussed in greater detail below.

After PTS is acquired, subsystem **44** begins waiting to synchronize to PTS [State=PTS Sync]. In this state, the demultiplexer and data parsing subsystem **44** continues to receive audio packets via terminal **40**, writes their PES data into buffer **50**, and maintains the error pointers. When this state is entered, subsystem **44** compares its audio STC to the correct output start time, which is the PTS value in the PTS pointer plus the acquisition PTS correction factor ($dPTS_{acq}$). If subsystem **44** discovers that the correct time has passed, i.e., $PCR > PTS + dPTS_{acq}$, one or more of the three values is incorrect and subsystem **44** will flag decoder microprocessor **42**. At this point, the state will revert to State=Frame Sync, and subsystem **44** will return to searching for two consecutive audio sync words. Otherwise, until $PCR = PTS + dPTS_{acq}$, subsystem **44** will continue to receive audio packets, write their PES data into the buffer **50**, maintain the error pointers, and monitor the reception of consecutive sync words.

When $PCR = PTS + dPTS_{acq}$, subsystem **44** has synchronized to PTS and will begin tracking the audio stream [State=Track]. At this time, subsystem **44** will begin transferring the contents of the audio buffer to the audio decoder **54** upon the audio decoder requesting audio data, starting with the sync word located at the buffer address pointed to by the PTS pointer. In the tracking state, subsystem **44** will continue to receive audio packets, write their PES data into the buffer **50**, maintain the error pointers, and monitor reception of consecutive sync words. If an error condition occurs during this time, subsystem **44** will transition to error processing. Otherwise, it will remain in State=Track until an error occurs or microprocessor **42** commands it to return to the idle state.

As subsystem **44** outputs the sync word of each sync frame to the audio decoder **54** as part of the "audio" referred to in FIG. **2**, it will signal the error status of each audio sync frame to the audio decoder using the sync word. The sync word of audio sync frames in which subsystem **44** knows of no errors will be output as specified by the Dolby AC-3 or Musicam specification, as appropriate. The sync word of audio sync frames in which subsystem **44** knows of errors will be altered relative to the correct sync words. As an example, and in the preferred embodiment, every other bit of the sync word of sync frames to which an error pointer points will be inverted, starting with the most significant bit of the sync word. Thus, the altered AC-3 sync word will be 1010 0001 1101 1101 while the altered Musicam sync word will be 0101 0101 0101. Only the bits of the sync word will be altered. The audio decoder **54** will conceal the audio errors in the sync frame which it receives in which the sync word has been altered in this manner. However, the audio decoder will continue to maintain synchronization with the audio bitstream. Synchronization will be maintained assuming the audio bit rate did not change, and knowing that two sync frame sizes are possible when the audio sample rate is 44.1 ksps.

In accordance with the preferred embodiment, audio decoder **54** will maintain synchronization through sample and bit rate changes if this feature is enabled by the decoder microprocessor **42**. If the microprocessor disables sample rate changes, audio decoder **54** will conceal the audio errors in each sync frame received with a sample rate that does not match the sample rate of the sync frame on which the audio decoder last acquired, and will assume that the sample rate did not change in order to maintain synchronization. The audio decoder is required to process through bit rate changes. If an error in the bit rate information is indicated, e.g., through the use of a cyclic redundancy code (CRC) as well known in the art, audio decoder **54** will assume that the bit rate of the corresponding sync frame is the same bit rate as the previous sync frame in order to maintain synchronization. If the decoder microprocessor **42** has enabled rate changes, the audio decoder **54** will assume that the rates indicated in the sync frame are correct, will process the sync frame, and use the appropriate sync frame size in maintaining synchronization with the audio bitstream.

Demultiplexer and data parsing subsystem **44** will also aid microprocessor **42** in checking that audio data continues to be output at the correct time by resynchronizing with the PTS for some PTS values received. To accomplish this, when a PTS value is received it will be stored in the PTS pointer, along with the audio offset address at which the next sync word is written in audio buffer **50**, if the PTS pointer is not already occupied. In doing this, subsystem **44** will ensure that the next sync word is received at the correct location in the audio PES bitstream. Otherwise, the PTS value will not be stored and subsystem **44** will defer resynchronization until the next successful PTS/DRAM offset address pair is obtained. Subsystem **44** will store the PTS/DRAM offset address pair in the PTS pointer until it begins to output the associated audio sync frame. Once it begins outputting audio data to the audio decoder **54**, subsystem **44** will continue to service the audio decoder's requests for

audio data, outputting each audio sync frame in sequence. This will continue until the sync frame pointed to by the PTS pointer is reached. When this occurs, subsystem **44** will stop outputting data to the audio decoder **54** until PCR=PTS+$dPTS_{track}$. This will detect audio timing errors which may have occurred since the last resynchronization by this method.

If PCR>PTS+$dPTS_{acq}$ when subsystem **44** completes output of the previous sync frame, the audio decoder **54** is processing too slow or an undetected error has occurred in a PCR or PTS value. After this error condition, subsystem **44** will flag microprocessor **42**, stop the output to the audio decoder **54**, clear audio buffer **50** and the pointers, and return to searching for two consecutive sync words separated by the correct number of audio data bytes. If the audio decoder **54** is not requesting data when the buffer read pointer equals the address pointed to by the PTS pointer, an audio processing error has occurred and subsystem **44** will maintain synchronization with the audio stream, clear its audio buffer and pointers, and return to searching for two consecutive audio sync words [State=Frame Sync].

In order to handle errors, subsystem **44** sets a unique error flag for each error condition, which is reset when microprocessor **42** reads the flag. Each error condition which interrupts microprocessor **42** will be maskable under control of the microprocessor. Table 1 lists the various error conditions related to audio synchronization and the response by subsystem **44**. In this table, "Name" is a name assigned to each error condition as referenced in the state diagram of FIG. **5**. "Definition" defines the conditions indicating that the corresponding error has occurred. "INT" is an interrupt designation which, if "yes", indicates that subsystem **44** will interrupt microprocessor **42** when this error occurs. "Check State" and "Next State" designate the states in which the error will be detected (checked) and the audio processor will enter, respectively, with the symbol ">" that the designated error will be detected when the audio processing state of subsystem **44** is higher than the designated state. The audio processing state hierarchy, from lowest to highest, is:

1. Idle
2. Frame Sync
3. $dPTS_{wait}$
4. PCR_{acq}
5. PTS_{acq}
6. PTS Sync
7. Track

The symbol "≧" preceding a state indicates that the error will be detected when the audio processing state of subsystem **44** is equal to or higher than the designated state. The designated state(s) indicate(s) that the error will be detected in this state or that the audio processing of subsystem **44** will proceed to this state after the associated actions are carried out. The designation "same" indicates that the audio processing of subsystem **44** will stay in the same state after the associated actions are carried out.

The heading "Buffer Action" indicates whether the audio buffer is to be flushed by setting its read and write pointers to be equal to the base address of the audio buffer. The designation "none" indicates no change from normal audio buffer management.

The heading "Pointer Action" indicates by the term "reset" that the PTS pointer, error pointers or both will be returned to the state specified as if subsystem **44** had been reset. The designation "none" indicates no change from normal pointer management. The designation "see other actions" indicates that other actions under the "Other Actions" heading may indicate a pointer to be set or reset. The "Other Actions" heading states any additional actions required of the subsystem **44** as a result of the error.

TABLE 1

SUMMARY OF ERRORS, EXCEPTIONS, AND ACTIONS.

Name	Definition	Int	Check State	Next State	Buffer Action	Pointer Action	Other Actions
pta_err	PCR > PTS + $dPTS_{acq}$	yes	pts_sync	frame_sync	flush	reset	none
pts_err	PCR > PTS + $dPTS_{acq}$	yes	track	frame_sync	flush	reset	Stop output to Audio Decoder (ADP).
sync_err	Input processor loses sync with input audio frames	yes	>idle	frame_sync	flush	reset	Stop output to ADP.
ov_err	Audio Buffer overflows	yes	≥pts_sync	frame_sync	flush	reset	Input processor maintains synchronization with the audio bitstream. Stop output to ADP.
under_err	Audio Buffer underflows	no	track	same	none	none	Input processor maintains synchronization with the audio bitstream. Stop output to ADP.
fa_err	Input processor reaches Audio PBS data which indicates the audio sample rate has changed since the current PID was acquired	yes	>frame_sync	same	none	none	Continue processing as if the audio sample rate had not changed.
fb_err	Input processor receives Audio PES data which indicates the audio bit rate has changed relative to the last audio sync frame reached	yes	>frame_sync	same	none	none	If bit rate changes are enabled, input processor will continue processing, trusting that the bit rate in fact changed and using the appropriate sync frame size to maintain synchronization. If bit rate changes are not enabled, input processor will continue processing using the bit rate indicated by the last audio sync frame received.
pts_miss	Sync word not found due to loss of audio data after a PTS is received	no	≥pts_acquire	same	none	none	None but other error conditions may also apply in this case
pcr_dis1	Input processor reaches a transport packet on the Audio PCR PID with the discontinuity_indicator bit of its adaptation_field set	no	pts_sync	pts acquire	flush	pts:reset error:none	Input processor stops storing PTS values in the PTS pointer until after reception of the next Audio PCR value.
pcr_dis2	Input processor receives a transport packet on the Audio PCR PID with the discontinuity_indicator bit of its adaptation_field set	no	track	same	none	pts:reset error:none	Input processor stops storing PTS values in the PTS pointer until after reception or the next Audio PCR value.
aud_err1a	Audio data of one transport packet of the current input sync frame is lost due to errors	See other actions	>idle	same or frame_sync; see other actions	none	pts:none error:see other actions	Maintain Audio Buffer fullness by advancing the FIFO write pointer by 184 bytes (MPEG), use an error pointer to mark the current sync frame as in error, and continue processing without generating an interrupt. If it is possible that more than one audio sync word was lost with the missing audio transport packet, such as when supporting Musicam Layer II at less than 64 kbps or AC-3 at less than 48 kbps, return to the Frame Sync state and generate an interrupt. If the next audio sync word is not received when expected, begin a byte-by-byte search for the audio sync word during the reception of subsequent audio data. Once the sync byte search is started, stop storing audio data in the buffer until the sync word in found. Do not store the first byte examined during the search. Resume storing audio data when the sync byte is found, starting with the sync word itself. If the sync word is not found during the first 184 bytes searched, return to the Frame Sync state[1] and generate an interrupt

TABLE 1-continued

SUMMARY OF ERRORS, EXCEPTIONS, AND ACTIONS.

Name	Definition	Int	Check State	Next State	Buffer Action	Pointer Action	Other Actions
aud_err1b	Audio data of one transport packet of the current input sync frame is lost due to errors after aud_err1a has occurred during the same input sync frame	yes	>idle	frame_sync	flush	pts:reset error:none	none
aud_err2	Audio data of more than one transport packet of the current input sync frame is lost due to errors	yes	>idle	frame_sync	flush	pts:reset error:see other actions	Use an error pointer to mark the current sync frame as in error.
ptrs_full	Audio data of one transport packet is lost while Error Mode is Unprotected	yes	≥pts_sync	frame_sync	flush	reset	Input processor maintains synchronization with the audio bitstream. Stop output to ADP.

[1]To implement the above error processing for MPEG or DigiCipher II implementations, the Input Processor can maintain an audio frame byte count by:
setting a counter's value so the sync frame size in bytes as each sync word is received,
decrementing the counter as each received audio byte is stored in the Audio Buffer (FIFO),
decrementing the counter by 184 bytes when a single audio transport packet is lost to compensate for the advancement of the FIFO write pointer by 184,
incrementing the counter by the smaller of the two sync frame sizes in bytes corresponding to the current bit rate if the above decrement resulted in a negative counter value (indicating the lost transport packet possibly contained the next audio sync word and accounting for the possibility that the audio sample rate is 44.1 Ksps and the sync frame size has changed from the larger value to the smaller value),
returning to the Frame Sync state if the above increment resulted in a counter value which was still negative (indicating the lost transport packet possibly contained more than one audio sync word), and
beginning the byte-by-byte sync word search when the counter is zero.

NOAC Ex. 1015 Page 184

As indicated above, the demultiplexing and data parsing subsystem **44** of FIG. **2** maintains several pointers to support audio processing. The PTS pointer is a set of parameters related to a PTS value, specifically a PTS value, a DRAM offset address, and a validity flag. In the illustrated embodiment, the PTS value comprises the 17 least significant bits of the PTS value received from the audio PES header. This value is associated with the audio sync frame pointed to by the pointer's DRAM offset address field. The use of 17 bits allows this field to specify a 1.456 second time window ($(2^{17}-1)/90$ kHz), which exceeds the maximum audio time span which the audio buffer **50** is sized to store.

The DRAM offset address maintained by the PTS pointer is a 13-bit offset address, relative to the audio buffer base address, into the DRAM at which the first byte of the audio sync frame associated with the pointer's PTS value is stored. The 13 bits allows the pointer to address an audio buffer as large as 8192 bytes.

The PTS pointer validity flag is a one-bit flag indicating whether or not this PTS pointer contains a valid PTS value and DRAM offset address. Since MPEG does not require PTS values to be transported more often than every 700 milliseconds, subsystem **44** may find itself not having a valid PTS value for some intervals of time.

After the decoder is reset, the valid flag of the PTS pointer is set to invalid. When a new PTS value is received, if the valid flag is set, the newly received PTS value is ignored. If the valid flag is not set, the newly received PTS value is stored into the PTS pointer but its valid flag is not yet set to valid. After a new PTS value is stored into the PTS pointer, the processing of audio data is continued and each audio data byte is counted. If the next audio sync frame is received and placed into the buffer correctly, the DRAM offset address (which corresponds to the buffer address into which the first byte of the sync word of this sync frame is stored) is stored into the pointer's DRAM offset address field. Then, the pointer's valid flag is set to valid. The next audio sync frame is received and placed into the buffer correctly when no data is lost for any reason between reception of the PTS value and reception of a subsequent sync word before too many audio bytes (i.e., the number of audio bytes per sync frame) are buffered. If the next audio, sync frame is not received or placed into the buffer correctly, the valid flag is not set to valid.

After the PTS pointer is used to detect any audio timing errors which may have occurred since the last resynchronization, the valid flag is set to invalid to allow subsequent PTS pointers to be captured and used. This occurs whether the PTS pointer is in the PTS sync or tracking state.

The error pointers are parameters related to an audio sync frame currently in the buffer and known to contain errors. The error pointers comprise a DRAM offset address and a validity flag. The DRAM offset address is a 13-bit offset address, relative to the audio buffer base address, into the DRAM at which the first byte of the audio sync frame known to contain errors is stored. Thirteen bits allows the pointer to address an audio buffer as large as 8192 bytes. The validity flag is a one-bit flag indicating whether or not this error pointer contains a valid DRAM offset address. When receiving data from a relatively error free medium, subsystem **44** will find itself not having any valid error pointers for some intervals of time.

Subsystem **44** is required to maintain a total of two error pointers and one error mode flag. After reset, the validity flag is set to invalid and the error mode is set to "protected." When a sync word is placed into the audio buffer, if the valid flag of one or more error pointers is not set, the buffer address of the sync word is recorded into the DRAM offset address of one of the invalid error pointers. At the same time, the error mode is set to protected. If the validity flag of both error pointers is set when a sync word is placed into the buffer, the error mode is set to unprotected but the DRAM offset address of the sync word is not recorded.

When audio data is placed into the buffer and any error is discovered in the audio data, such as due to the loss of an audio transport packet or the reception of audio data which has not been properly decrypted, subsystem **44** will revert to the PTS acquire state if the error mode is unprotected. Otherwise, the validity bit of the error pointer which contains the DRAM offset address of the sync word which starts the sync frame currently being received is set. In the rare event that an error is discovered in the data for an audio sync frame during the same clock cycle that the sync word for the sync frame is removed from the buffer, the sync word will be corrupted as indicated above to specify that the sync frame is known to contain an audio error. At the same time, the validity bit is cleared such that it does not remain set after the sync frame has been output. This avoids the need to reset subsystem **44** in order to render the pointer useful again.

When audio data is being removed from the audio buffer, the sync word is corrupted if the DRAM offset address of any error pointer matches that of the data currently being removed from the buffer. At the same time, the validity bit is set to invalid.

The decoder of FIG. **2** also illustrates a video buffer **58** and video decoder **52**. These process the video data at the same time the audio data is being processed as described above. The ultimate goal is to have the video and audio data output together at the proper time so that the television signal can be reconstructed with proper lip synchronization.

FIG. **4** is a block diagram illustrating the demultiplexing and data parsing subsystem **44** of FIG. **2** in greater detail. After the transport packets are input via terminal **40**, the PID of each packet is detected by circuit **70**. The detection of the PIDs enables demultiplexer **72** to output audio packets, video packets and any other types of packets carried in the data stream, such as packets carrying control data, on separate lines.

The audio packets output from demultiplexer **72** are input to the various circuits necessary to implement the audio processing as described above. Circuit **74** modifies the sync word of each audio frame known to contain errors. The modified sync words are obtained using a sync word inverter **78**, which inverts every other bit in the sync words output from a sync word, PCR and PTS detection circuit **80**, in the event that the audio frame to which the sync word corresponds contains an error. Error detection is provided by error detection circuit **76**.

The sync word, PCR and PTS detection circuit **80** also outputs the sync word for each audio frame to an audio sample and bit rate calculator **86**. This circuit determines the audio sample and bit rate of the audio data and passes this information to decoder microprocessor **42** via data bus **88**.

The PCR and PTS are output from circuit **80** to a lip sync and output timing compensator **82**. Circuit **82** also receives the dPTS values from microprocessor **42**, and adds the appropriate values to the PTS in order to provide the necessary delay for proper lip synchronization. Compensator **82** also determines if the delayed presentation time is outside of the acceptable range with respect to the PCR, in which case an error has occurred and resynchronization will be required.

Buffer control **84** provides the control and address information to the audio output buffer **50**. The buffer control **84** is signaled by error detection circuit **76** whenever an error occurs that requires the temporary suspension of the writing of data to the buffer. The buffer control **84** also receives the delay values from lip sync and output timing compensator **82** in order to control the proper timing of data output from the buffer.

FIG. **5** is a state diagram illustrating the processing of audio data and response to errors as set forth in Table 1. The idle state is represented by box **100**. Acquisition of the audio data occurs during the frame sync state **102**. The dPTS-wait state is indicated by box **104**. Boxes **106**, **108** and **110** represent the PCR_{acq}, PTS_{acq}, and PTS sync states, respectively. Once audio synchronization has occurred, the signal is tracked as indicated by the tracking state of box **112**. The outputs of each of boxes **104**, **106**, **108**, **110** and **112** indicate the error conditions that cause a return to the frame synchronization state **102**. The error PCR DIS1 during the PTS sync state **110** will cause a return to the PTS acquire state, as indicated in the state diagram of FIG. **5**.

It should now be appreciated that the present invention provides methods and apparatus for acquiring and processing errors in audio data communicated via a transport packet scheme. Transport packet errors are handled while maintaining audio synchronization. During such error conditions, the associated audio errors are concealed. Corrupted data in an audio frame is signaled by altering the sync pattern associated with the audio frame. PTS's are used to check the timing of processing and to correct audio timing errors.

Although the invention has been described in connection with various specific embodiments, it should be appreciated and understood that numerous adaptations and modifications may be made thereto, without departing from the spirit and scope of the invention as set forth in the claims.

We claim:

1. A method for processing digital audio data from a packetized data stream carrying digital television information in a succession of fixed length transport packets, each of said packets including a packet identifier (PID), some of said packets containing a program clock reference (PCR) value for synchronizing a decoder system time clock (STC), and some of said packets containing a presentation time stamp (PTS) indicative of a time for commencing the output of associated data for use in reconstructing a television signal, said method comprising the steps of:

monitoring the PID's for the packets carried in said data stream to detect audio packets, some of said audio packets carrying an audio PTS;

storing audio data from the detected audio packets in a buffer for subsequent output;

monitoring the detected audio packets to locate audio PTS's;

comparing a time derived from said STC with a time derived from the located audio PTS's to determine whether said audio packets are too early to decode, too late to decode, or ready to be decoded; and

adjusting the time at which said stored audio data is output from said buffer on an ongoing basis in response to said comparing step.

2. A method in accordance with claim **1** wherein a PTS pointer is provided to maintain a current PTS value and an address of said buffer identifying where a portion of audio data referred to by said current PTS is stored, said timing adjustment being provided by the further steps of:

replacing said PTS value in said PTS pointer with a new current PTS value after data stored at said address has been output from said buffer;

replacing said address in said PTS pointer with a new address corresponding to a portion of audio data referred to by said new current PTS value;

suspending the output of data from said buffer when said new address is reached; and

recommencing the output of data from said buffer when said decoder system time clock reaches a presentation time derived from said new current PTS value.

3. A method in accordance with claim **2** wherein said presentation time is determined from the sum of said new current PTS value and an offset value that provides proper lip synchronization by accounting for a video signal processing delay.

4. A method in accordance with claim **1** wherein the time at which the audio data is output from said buffer is dependent on an offset value added to said PTS for providing proper lip synchronization by accounting for a video signal processing delay.

5. A method in accordance with claim **1** comprising the further steps of:

examining the detected audio packets to locate the occurrence of at least one audio synchronization word therein for use in achieving a synchronization condition prior to locating said audio PTS's;

commencing a reacquisition of said synchronization condition if said comparing step determines that said audio packets are too late to decode.

6. A method in accordance with claim **5** wherein two consecutive audio synchronization words with a correct number of audio data bytes in between define an audio frame, said audio frame including only one of said two consecutive audio synchronization words, said method comprising the further steps of:

detecting the occurrence of errors in said audio packets;

upon detecting a first audio packet of a current audio frame containing an error, advancing a write pointer for said buffer by the maximum number of payload bytes (N) contained in one of said fixed length transport packets and designating said current audio frame as being in error;

monitoring the detected audio packets of said current audio frame for the next audio synchronization word after said error has been detected, and if said synchronization word is not received where expected in the audio stream, discarding subsequent audio data while searching for said synchronization word rather than storing the subsequent audio data into said buffer;

resuming the storage of audio data in said buffer upon detection of said next audio synchronization word if said next audio synchronization word is located within N bytes after the commencement of the search therefor; and

if said next audio synchronization word is not located within said N bytes after the commencement of the search therefor, commencing a reacquisition of said synchronization condition.

7. A method in accordance with claim **6** comprising the further step of concealing television audio errors whenever the audio data from which said television audio is being reconstructed is in error.

8. A method in accordance with claim **7** wherein:

a current audio frame is designated as being in error by altering the audio synchronization word for that frame; and

said concealing step is responsive to an altered synchronization word for concealing audio associated with the corresponding audio frame.

9. A method for processing digital audio data from a packetized data stream carrying digital television information in a succession of transport packets having a fixed length of N bytes, each of said packets including a packet identifier (PID), some of said packets containing a program clock reference (PCR) value for synchronizing a decoder system time clock, and some of said packets containing a presentation time stamp (PTS) indicative of a time for commencing the output of associated data for use in reconstructing a television signal, said method comprising the steps of:

monitoring the PID's for the packets carried in said data stream to detect audio packets;

examining the detected audio packets to locate the occurrence of audio synchronization words for use in achieving a synchronization condition, each two consecutive audio synchronization words defining an audio frame therebetween;

monitoring the detected audio packets after said synchronization condition has been achieved to locate an audio PTS;

searching the detected audio packets after locating said audio PTS to locate the next audio synchronization word;

storing audio data following said next audio synchronization word in a buffer;

detecting the occurrence of errors in said audio packets;

upon detecting a first audio packet of a current audio frame containing an error, advancing a write pointer for said buffer by N bytes and designating said current audio frame as being in error;

monitoring the detected audio packets of said current audio frame for the next audio synchronization word after said error has been detected, and if said synchronization word is not received where expected in the audio stream, discarding subsequent audio data while searching for said synchronization word rather than storing the subsequent audio data into said buffer;

resuming the storage of audio data in said buffer upon detection of said next audio synchronization word if said next audio synchronization word is located within N bytes after the commencement of the search therefor; and

if said next audio synchronization word is not located within said N bytes after the commencement of the search therefor, commencing a reacquisition of said synchronization condition.

10. A method in accordance with claim 9 comprising the further step of concealing television audio errors whenever the audio data from which said television audio is being reconstructed is in error.

11. A method in accordance with claim 10 wherein:

a current audio frame is designated as being in error by altering the audio synchronization word for that frame; and

said concealing step is responsive to an altered synchronization word for concealing audio associated with the corresponding audio frame.

12. A method in accordance with claim 9 wherein said audio data includes information indicative of an audio sample rate and audio bit rate, at least one of said audio sample rate and audio bit rate being variable, said method comprising the further step of attempting to maintain synchronization of said audio packets during a rate change indicated by said audio data by:

ignoring a rate change indicated by said audio data on the assumption that the rate has not actually changed;

concealing the audio frame containing the data indicative of an audio sample rate change while attempting to maintain said synchronization condition; and

commencing a reacquisition of said synchronization condition if said condition cannot be maintained.

13. A method in accordance with claim 9 wherein said audio data includes information indicative of an audio sample rate and audio bit rate, at least one of said audio sample rate and audio bit rate being variable, said method comprising the further step of attempting to maintain synchronization of said audio packets during a rate change indicated by said audio data by:

processing said audio data in accordance with a new rate indicated by said audio data in the absence of an error indication pertaining to the audio frame containing the new rate, while attempting to maintain said synchronization condition;

processing said audio data without changing the rate if an error indication pertains to the audio frame containing the new rate, while concealing the audio frame to which said error condition pertains and attempting to maintain said synchronization condition; and

commencing a reacquisition of said synchronization condition if said condition cannot be maintained.

14. Apparatus for acquiring audio information carried by a packetized data stream and processing errors therein, comprising:

means for detecting audio transport packets in said data stream;

means for recovering audio data from said detected audio transport packets for storage in a buffer;

means for locating an audio presentation time stamp (PTS) in said detected audio transport packets;

means responsive to said PTS for commencing the output of audio data from said buffer at a specified time;

means for monitoring the detected audio transport packets after the output of audio data from said buffer has commenced, to locate subsequent audio PTS's;

means for comparing a time derived from a decoder system time clock (STC) to a time derived from the subsequent audio PTS's to determine whether audio data stored in said buffer is too early to decode, too late to decode, or ready to be decoded; and

means responsive to said comparing means for adjusting the time at which said stored audio data is output from said buffer.

15. Apparatus in accordance with claim 14 further comprising:

means for maintaining a PTS pointer with a current PTS value and an address of said buffer identifying where a portion of audio data referred to by said current PTS is stored;

means for replacing said PTS value in said PTS pointer with a new current PTS value after data stored at said address has been output from said buffer, and for replacing said address in said PTS pointer with a new address corresponding to a portion of audio data referred to by said new current PTS value;

means responsive to said PTS pointer for suspending the output of data from said buffer when said new address is reached; and

means for recommencing the output of data from said buffer at a time derived from said new current PTS value.

16. Apparatus in accordance with claim 15 further comprising:

means for concealing error in an audio signal reproduced from data output from said buffer and reestablishing the detection of said audio transport packets if the time derived from said new current PTS value is outside a predetermined range.

17. Apparatus in accordance with claim 14 wherein said audio transport packets each contain a fixed number N of payload bytes, said packets being arranged into successive audio frames commencing with an audio synchronization word, said apparatus further comprising:

means for detecting the occurrence of errors in said audio packets;

means for advancing a write pointer for said buffer by N bytes and designating a current audio frame as being in error upon detecting an error in an audio transport packet of said current audio frame;

means for monitoring the detected audio transport packets of said current audio frame for the next audio synchronization word after said error has been detected, and if said synchronization word is not received where expected in the audio stream, discarding subsequent audio data while searching for said synchronization word rather than storing the subsequent audio data into said buffer;

means for resuming the storage of audio data in said buffer upon detection of said next audio synchronization word if said next audio synchronization word is located within said fixed number N of bytes after the commencement of the search therefor; and

means for reestablishing the detection of said audio transport packets if said next audio synchronization word is not located within said fixed number N of bytes after the commencement of the search therefor.

18. Apparatus in accordance with claim 17 further comprising:

means for concealing error in an audio signal reproduced from data output from said buffer when the data output from said buffer is in error.

19. Apparatus in accordance with claim 18 further comprising:

means for altering the audio synchronization word associated with a current audio frame to designate that frame as being in error;

wherein said concealing means are responsive to altered synchronization words for concealing errors in audio associated with the corresponding audio frame.

20. Apparatus for acquiring audio information carried by a packetized data stream and processing errors therein, comprising:

means for detecting audio transport packets in said data stream, said packets being arranged into successive audio frames commencing with an audio synchronization word;

means responsive to said synchronization words for obtaining a synchronization condition enabling the recovery of audio data from said detected audio transport packets for storage in a buffer;

means for detecting the presence of errors in said audio data;

means responsive to said error detecting means for controlling the flow of data through said buffer when an error is present, to attempt to maintain said synchronization condition while masking said error; and

means for reestablishing the detection of said audio transport packets if said controlling means cannot maintain said synchronization condition.

21. Apparatus in accordance with claim 20 wherein said audio transport packets each contain a fixed number N of payload bytes, and said means responsive to said error detecting means comprise:

means for advancing a write pointer for said buffer by said fixed number N of bytes and designating a current audio frame as being in error upon the detection of an error in an audio transport packet thereof;

means for monitoring the detected audio transport packets of said current audio frame for the next audio synchronization word after said error has been detected, and if said synchronization word is not received where expected in the audio stream, discarding subsequent audio data while searching for said synchronization word rather than storing the subsequent audio data into said buffer; and

means for resuming the storage of audio data in said buffer upon detection of said next audio synchronization word if said next audio synchronization word is located within said fixed number N of bytes after the commencement of the search therefor.

22. Apparatus in accordance with claim 20 further comprising:

means for concealing error in an audio signal reproduced from data output from said buffer when the data output from said buffer is in error.

23. Apparatus in accordance with claim 22 further comprising:

means for altering the audio synchronization word associated with an audio frame containing a data error to designate that frame as being in error;

wherein said concealing means are responsive to altered synchronization words for concealing errors in audio associated with the corresponding audio frame.

24. A method for managing errors in data received in bursts from a packetized data stream carrying digital information in a succession of fixed length transport packets, at least some of said packets containing a presentation time stamp (PTS) indicative of a time for commencing the fixed rate presentation of presentation units from a buffer into which they are temporarily stored upon receipt, said method comprising the steps of:

monitoring received packets to locate associated PTS's, said received packets carrying presentation units to be presented;

synchronizing the presentation of said presentation units from said buffer to a system time clock (STC) associated with the packetized data stream using timing information derived from the PTS's located in said monitoring step; and

identifying discontinuity errors resulting from a loss of one or more transmitted packets between successive ones of the received packets and, if a discontinuity of no more than one packet is identified, advancing a write pointer of said buffer by a suitable number of bits to compensate for the discontinuity, while maintaining the synchronization of said presentation with respect to said STC.

25. A method in accordance with claim 24 wherein said transport packets each contain a fixed number N of payload bytes, said method comprising the further steps of:

advancing said write pointer by said fixed number N of bytes upon the detection of a discontinuity error;

continuing said monitoring step after said discontinuity error has been detected in order to search for a synchronization word, and if said synchronization word is not located where expected, discarding subsequent presentation units while searching for said synchronization word rather than storing said subsequent presentation units in said buffer; and

resuming the storage of presentation units in said buffer upon the detection of said synchronization word if said synchronization word is located within said fixed number N of bytes after the commencement of the search therefor.

* * * * *

US005802054A

United States Patent [19]

Bellenger

[11] **Patent Number: 5,802,054**

[45] **Date of Patent: Sep. 1, 1998**

[54] **ATOMIC NETWORK SWITCH WITH INTEGRATED CIRCUIT SWITCH NODES**

[75] Inventor: **Donald M. Bellenger**, Los Altos Hills, Calif.

[73] Assignee: **3Com Corporation**, Santa Clara, Calif.

[21] Appl. No.: **698,745**

[22] Filed: **Aug. 16, 1996**

[51] **Int. Cl.**[6] **H04L 12/66**

[52] **U.S. Cl.** **370/401**

[58] **Field of Search** 370/351, 400, 370/401, 402, 407, 408, 422

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,947,390	8/1990	Sheehy	370/401
5,047,917	9/1991	Athas et al.	364/200
5,166,931	11/1992	Riddle	370/401
5,321,695	6/1994	Faulk, Jr.	370/401
5,390,173	2/1995	Spinney et al.	370/401
5,477,547	12/1995	Sugiyama	370/401
5,610,905	3/1997	Murthy et al.	370/401
5,657,314	8/1997	McClure et al.	370/401

OTHER PUBLICATIONS

ATOMIC: A Low-Cost, Very High-Speed, Local Communication Architecture, Danny Cohen, Gregory Finn, Robert Felderman, Annette DeSchon, USC/Information Sciences Institute, 1993 International Conference on Parallel Processing.

The Use of Message-Based Multicomputer Components to Construct Gigabit Networks, by D. Cohen, G. Finn, R. Felderman and A. DeSchon, University of Southern California/Information Sciences Institute.

ATOMIC: A High-Speed Local Communication Architecture, by R. Felderman, A. DeSchon, D. Cohen, G. Finn, USC/Information Sciences Institute, Journal of High Speed Networks 1 (1994) pp. 1-28, IOS Press.

ATOMIC: A Local Communication Network Created Through Repeated Application of Multicomputing Components, by D. Cohen, G. Finn, R. Felderman, A. DeSchon.

An Integration of Network Communication and Workstation Architecture, by Gregory G. Finn, USC/Information Sciences Institute, Published Oct. 1991, ACM Computer Communication Review.

(List continued on next page.)

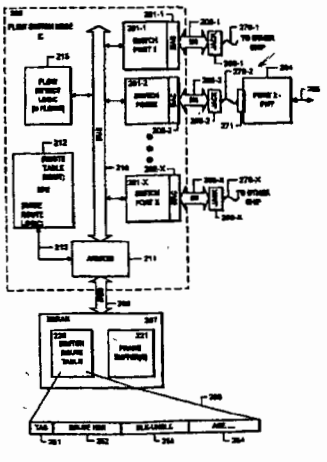
Primary Examiner—Ajit Patel

Attorney, Agent, or Firm—Mark A. Haynes; Kent R. Richardson; Wilson, Sonsini, Goodrich & Rosati

[57] **ABSTRACT**

An atomic type switch mesh is combined with standard local area network links, such as high speed Ethernet, and a bridge-like protocol to provide a high performance scalable network switch. The network switch comprises a plurality of switch nodes, a first set of communication links which are coupled between switch nodes internal to the network switch, and a second set of communication links which comprise network links from switch nodes on the border of the network switch to systems external to the network switch. The respective switch nodes include a set of ports (having more than two members) which are connected to respective communication links in either the first or second set of communication links. Each port in the set comprises a medium access control (MAC) logic unit for a connectionless network protocol, preferably high speed Ethernet. The switch nodes also include a route table memory which has a set of accessible memory locations that store switch route data specifying routes through the plurality of switch nodes within the boundaries of the network switch. Flow detect logic is coupled with the set of ports on the switch node, which monitors frames received by the set of ports and generates an identifying tag for use in accessing the route table memory. Finally, the switch node includes node route logic which is coupled with the flow detect logic, the route table memory and the set of ports. The node route logic monitors frames received by the set of ports to route a received frame for transmission out a port in the set of ports.

56 Claims, 6 Drawing Sheets



OTHER PUBLICATIONS

ATOMIC: A Low–Cost, Very–High–Speed LAN, by D. Cohen, G. Finn, R. Felderman, A. DeSchon.

The Design of the Caltech Mosaic C Multicomputer, C. Seitz, N. Boden, J. Seizovic, and W. Su, Computer Science 256–80, California Institute of Technology.

802.3z Higher Speed Task Force Objectives (Gigabit Ethernet), Apr., 1996.

Netstation Architecture Multi–Gigabit Workstation Network Fabric, G. Finn, P. Mockapetris, USC/Information Sciences Institute.

A Zero–Pass End–to–End Checksum Mechanism for IPv6[1], G. Finn, S. Hotz, C. Rogers, USC/Information Sciences Institute, Dec., 1995.

Network Backplane, G. Finn, USC/Information Sciences Institute, Apr., 1994.

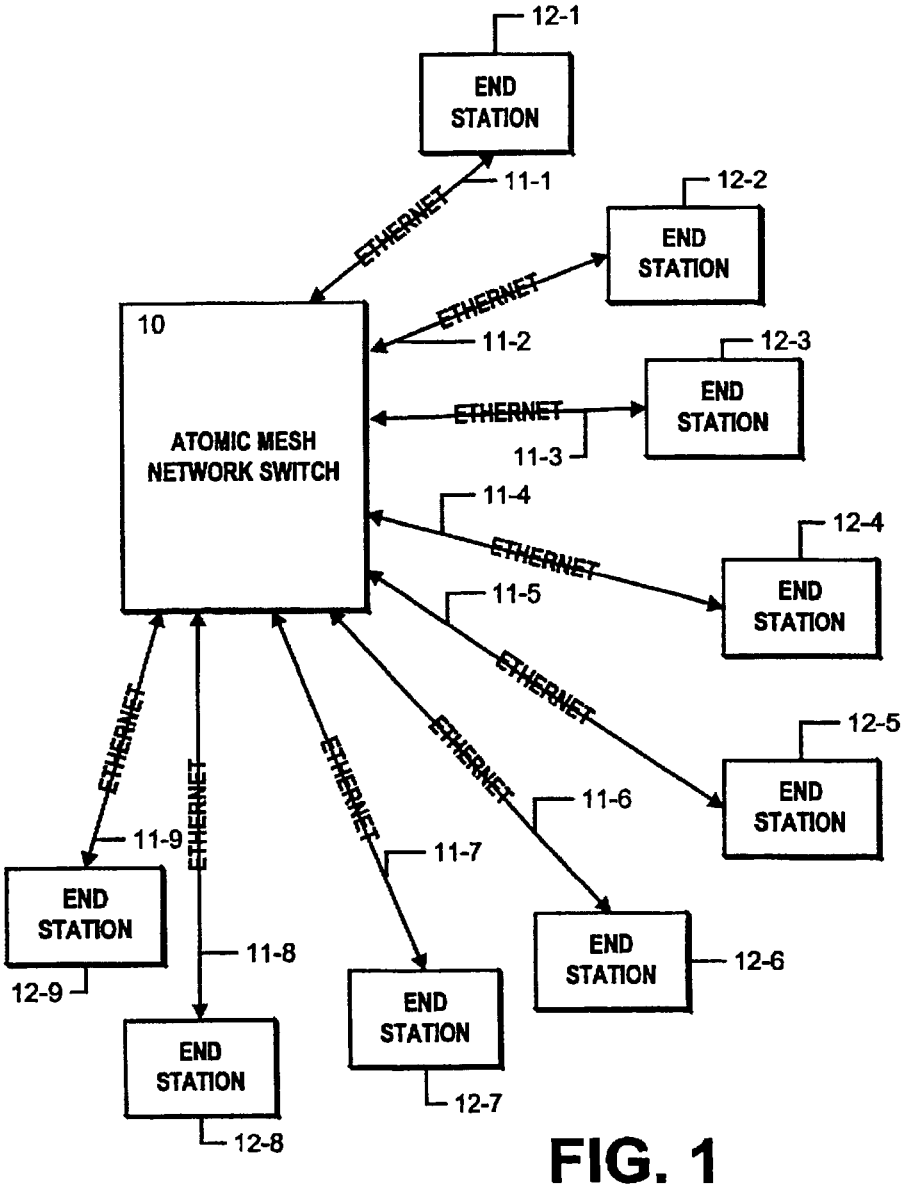

12-1
END STATION
11-1
ETHERNET
12-2
END STATION
ETHERNET
10
ATOMIC MESH NETWORK SWITCH
11-2
12-3
END STATION
ETHERNET
11-3
11-4
ETHERNET
12-4
END STATION
11-5
ETHERNET
12-5
END STATION
ETHERNET
11-6
ETHERNET
11-7
ETHERNET
11-8
ETHERNET
11-9
END STATION
12-9
END STATION
12-8
END STATION
12-7
END STATION
12-6

FIG. 1

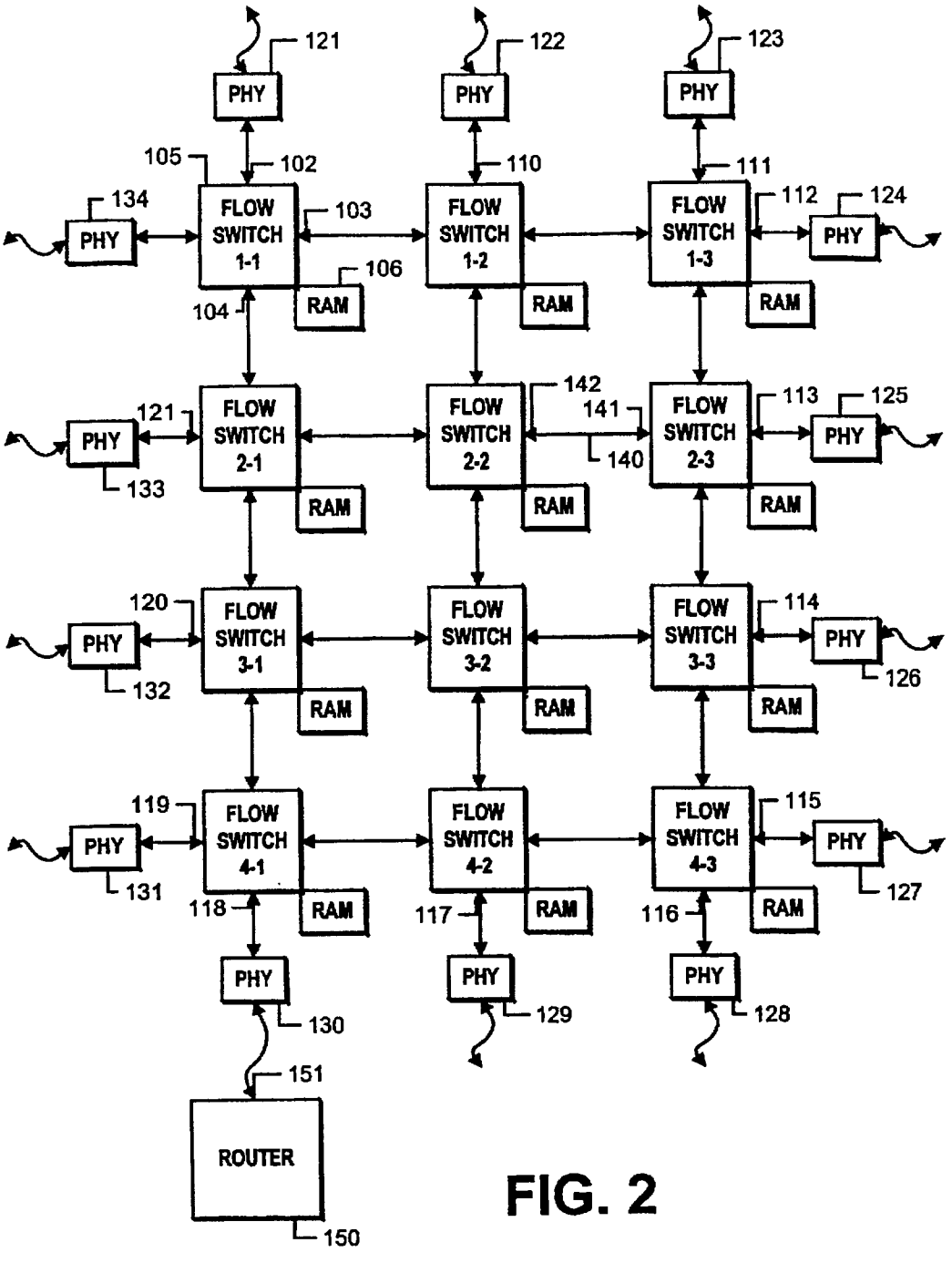

121
122
123
PHY
PHY
PHY
105
102
110
111
134
FLOW SWITCH 1-1
103
FLOW SWITCH 1-2
FLOW SWITCH 1-3
112
124
PHY
106
RAM
RAM
RAM
104
121
142
141
113
125
PHY
FLOW SWITCH 2-1
FLOW SWITCH 2-2
FLOW SWITCH 2-3
PHY
133
140
RAM
RAM
RAM
120
114
PHY
FLOW SWITCH 3-1
FLOW SWITCH 3-2
FLOW SWITCH 3-3
PHY
132
126
RAM
RAM
RAM
119
115
PHY
FLOW SWITCH 4-1
FLOW SWITCH 4-2
FLOW SWITCH 4-3
PHY
131
127
118
RAM
117
RAM
116
RAM
PHY
PHY
PHY
130
129
128
151
ROUTER
150

FIG. 2

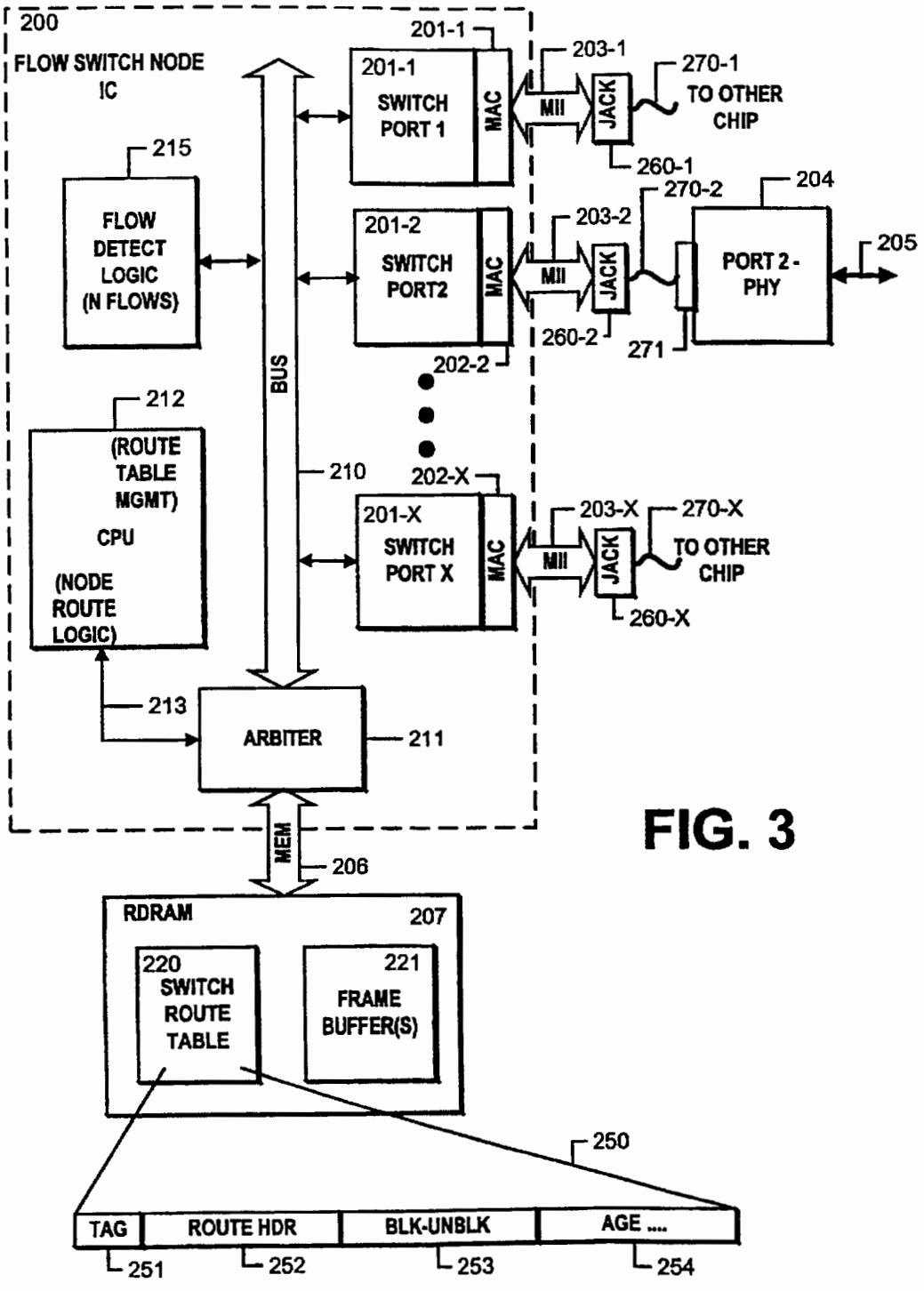
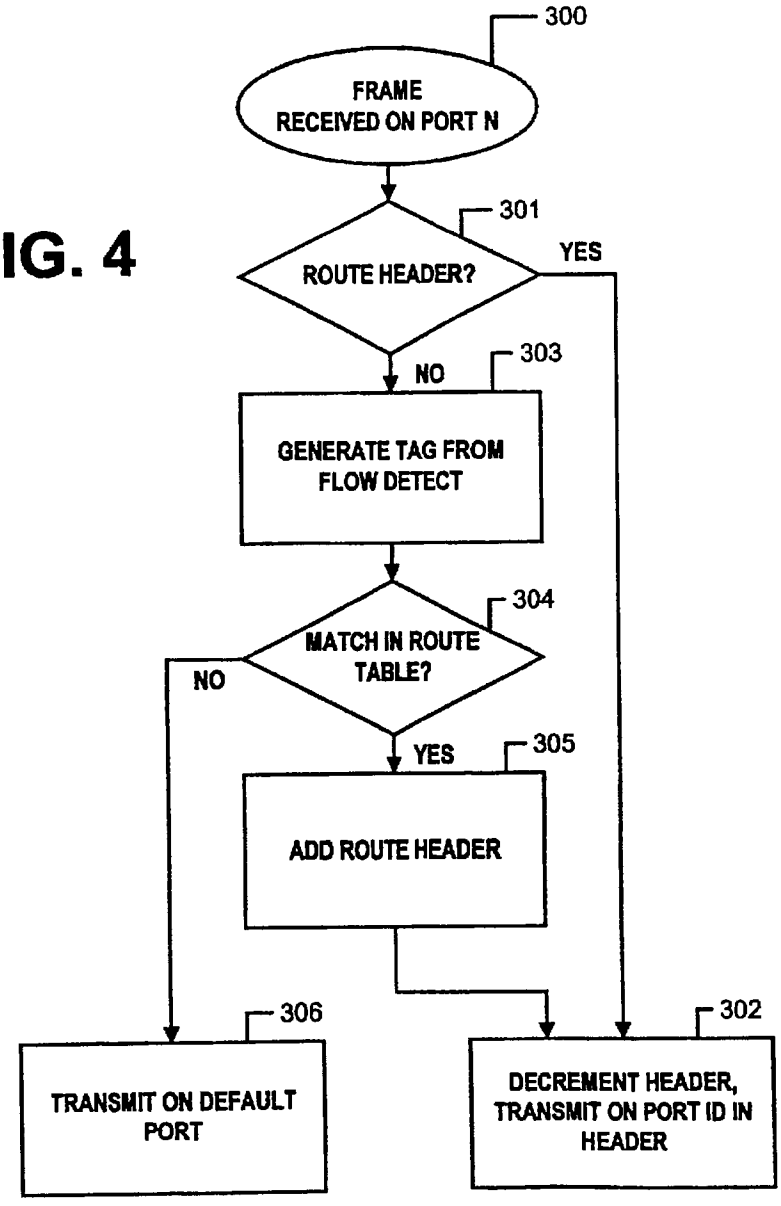

200
FLOW SWITCH NODE IC
201-1
201-1 SWITCH PORT 1
MAC
203-1
MII
JACK
270-1
TO OTHER CHIP
260-1
215
FLOW DETECT LOGIC (N FLOWS)
201-2 SWITCH PORT2
203-2
270-2
204
PORT 2 - PHY
205
260-2
271
202-2
212
(ROUTE TABLE MGMT)
CPU
(NODE ROUTE LOGIC)
BUS
210
202-X
201-X SWITCH PORT X
203-X
270-X
TO OTHER CHIP
260-X
213
ARBITER
211
MEM
206
RDRAM
207
220 SWITCH ROUTE TABLE
221 FRAME BUFFER(S)
250
TAG
ROUTE HDR
BLK-UNBLK
AGE ....
251
252
253
254

FIG. 3

FIG. 4


300
FRAME RECEIVED ON PORT N
301
ROUTE HEADER?
YES
NO
303
GENERATE TAG FROM FLOW DETECT
304
MATCH IN ROUTE TABLE?
NO
YES
305
ADD ROUTE HEADER
306
TRANSMIT ON DEFAULT PORT
302
DECREMENT HEADER, TRANSMIT ON PORT ID IN HEADER

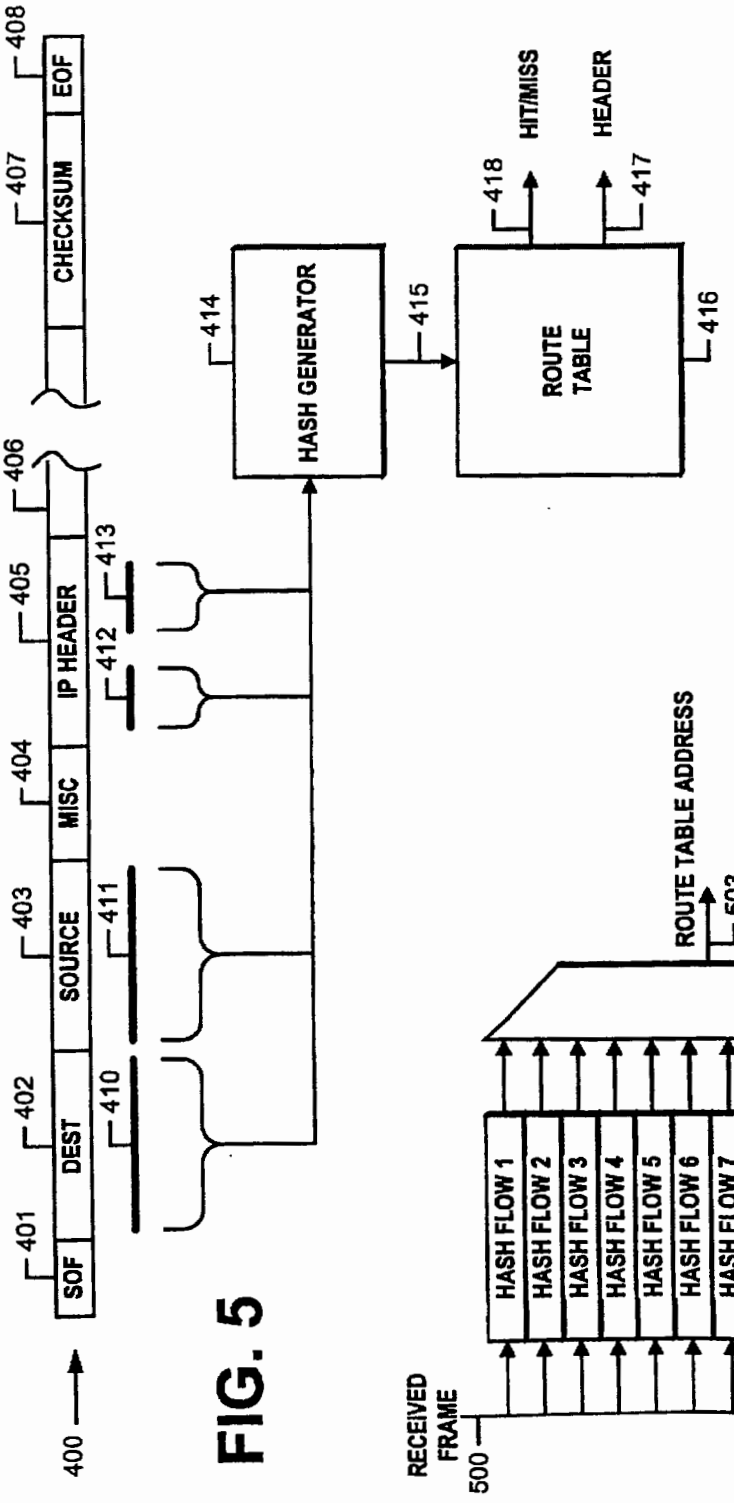

400
SOF
401
DEST
402
SOURCE
403
MISC
404
IP HEADER
405
406
CHECKSUM
407
EOF
408
410
411
412
413
HASH GENERATOR
414
415
ROUTE TABLE
416
HEADER
417
HIT/MISS
418

FIG. 5

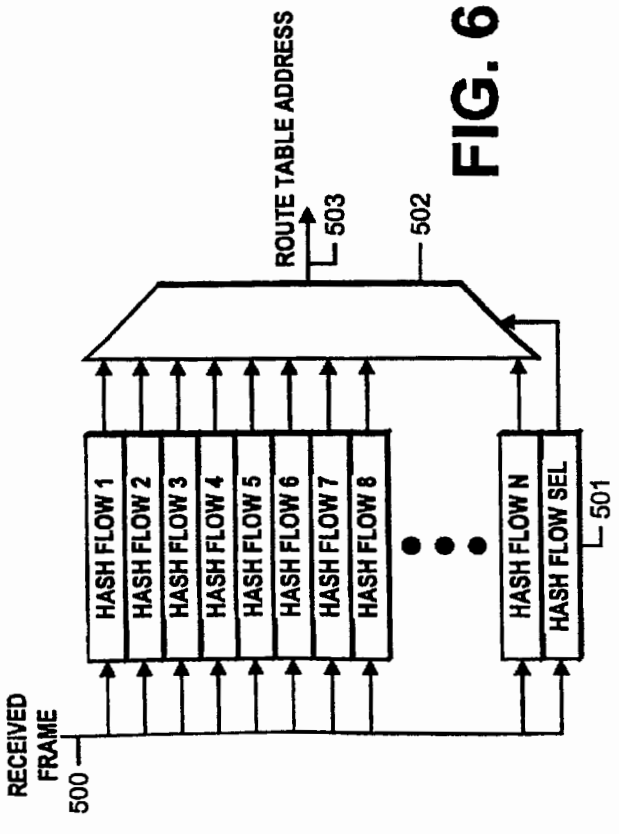

RECEIVED FRAME
500
HASH FLOW 1
HASH FLOW 2
HASH FLOW 3
HASH FLOW 4
HASH FLOW 5
HASH FLOW 6
HASH FLOW 7
HASH FLOW 8
HASH FLOW N
HASH FLOW SEL
501
502
ROUTE TABLE ADDRESS
503

FIG. 6

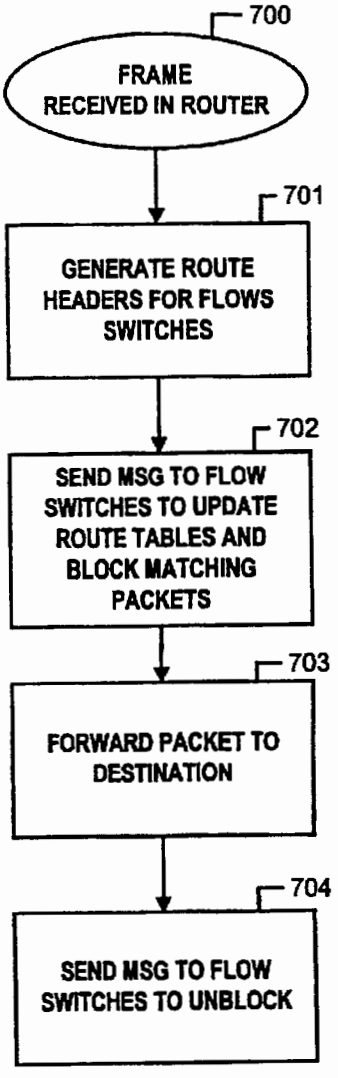

700
FRAME RECEIVED IN ROUTER
701
GENERATE ROUTE HEADERS FOR FLOWS SWITCHES
702
SEND MSG TO FLOW SWITCHES TO UPDATE ROUTE TABLES AND BLOCK MATCHING PACKETS
703
FORWARD PACKET TO DESTINATION
704
SEND MSG TO FLOW SWITCHES TO UNBLOCK

FIG. 7

ATOMIC NETWORK SWITCH WITH INTEGRATED CIRCUIT SWITCH NODES

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to the field of network intermediate devices, and more particularly to high-performance switches for routing data in computer networks.

2. Description of Related Art

Network intermediate systems for interconnecting networks include various classes of devices, including bridges, routers and switches. Systems for the interconnection of multiple networks encounter a variety of problems, including the diversity of network protocols executed in the networks to be interconnected, the high bandwidth required in order to handle the convergence of data from the interconnected networks at one place, and the complexity of the systems being designed to handle these problems. As the bandwidth of local area network protocols increases, with the development of so-called asynchronous transfer mode ("ATM"), 100 megabit per second Ethernet standards, and proposals for gigabit per second Ethernet standards, the problems encountered at network intermediate systems are being multiplied.

One technique which has been the subject of significant research for increasing the throughput of networks is known as the so-called atomic LAN. The atomic LAN is described for example in Cohen, et al., "ATOMIC: A Low-Cost, Very High-Speed, Local Communication Architecture", 1993 International Conference on Parallel Processing. There is a significant amount of published information about the atomic LAN technology. Felderman, et al. "ATOMIC: A High-Speed Local Communication Architecture", *Journal of High Speed Networks*, Vol. 1, 1994, pp. 1–28; Cohen, et al., "ATOMIC: A Local Communication Network Created Through Repeated Application of Multicomputing Components", DARPA Contract No. DABT63-91-C-001, Oct. 1, 1992; Cohen et al., "The Use of Message-Based Multicomputer Components to Construct Gigabyte Networks"; DARPA Contract No. DABT63-91-C-001, published Jun. 1, 1992; Finn, "An Integration of Network Communications with Workstation Architecture", ACM, A Computer Communication Review, October 1991; Cohen et al., "ATOMIC: Low-cost, Very-High-Speed LAN", DARPA Contract No. DABT63-91-C-001 (publication date unknown, downloaded from Internet on or about May 10, 1996).

The atomic LAN is built by repeating simple four port switch integrated circuits in the end stations, based on the well known Mosaic architecture created at the California Institute of Technology. These integrated circuits at the end stations are interconnected in a mesh arrangement to produce a large pool of bandwidth that can cross many ports. The links that interconnect the switches run at 500 megabits per second. Frames are routed among the end stations of the network using a differential source route code adapted for the mesh. One or more end stations in the mesh act "address consultants" to map the mesh and calculate source route codes. All of the links are self timed, and depend on acknowledged signal protocols to coordinate flow across the links to prevent congestion. The routing method for navigating through the mesh, known as "worm hole" routing is designed to reduce the buffering requirements at each node.

The atomic LAN has not achieved commercial application to a significant degree, with an exception possibly in connection with a supercomputer known as Paragon from Intel Corporation of Santa Clara, Calif. Basically it has been only a research demonstration project. Critical limitations of the design include the fact that it is based on grossly non-standard elements which make commercial use impractical. For example, there is no way to interface the switch chips taught according to the atomic LAN project with standard workstations. Each workstation needs a special interface chip to become part of the mesh in order to participate in the LAN. Nonetheless, the ATOMIC LAN project has demonstrated a high throughput and readily extendable architecture for communicating data.

Typical switches and routers in the prior art are based on an architecture requiring a "backplane" having electrical characteristics that are superior to any of the incoming links to be switched. For example, 3Com Corporation of Santa Clara, Calif., produces a product known as NetBuilder2, having a core bus backplane defined which runs at 800 megabits per second. This backplane moves traffic among various local area network external ports.

There are several problems with the backplane approach typical of prior art intermediate systems. First, the backplane must be defined fast enough to handle the largest load that might occur in the intermediate system. Furthermore, the customer must pay for worst case backplane design, regardless of the customer's actual need for the worst case system. Second, the backplane itself is just another communication link. This communication link must be completely supported as a backplane for the network intermediate system, involving intricate and expensive design. The lower volumes for specialized backplane link further increases the cost of network intermediate systems based on the backplane architecture.

In light of the ever increasing complexity and bandwidth requirements of network intermediate systems in commercial settings, it is desirable to apply the atomic LAN principles in practical, easy to implement, and extendable network intermediate systems.

SUMMARY OF THE INVENTION

According to the present invention, the fine scalability of an atomic type LAN mesh, is combined with standard local area network links, such as high speed Ethernet, and a standard routing protocol to provide a high performance scalable network switch. The need for the special purpose backplane bus is removed according to this architecture, while providing scalability, high performance, and simplicity of design.

Accordingly, the present invention can be characterized as a network switch that comprises a plurality of switch nodes arranged in a mesh, a first set of internal communication links which are coupled between switch nodes internal to the network switch, and a second set of external communication links which comprise network links from switch nodes on the border of the network switch to systems external to the network switch. The respective switch nodes include a set of ports (having more than two members) which are connected to respective communication links in one of the first or second sets of communication links. The ports in the set of ports include respective medium access control (MAC) units for transmission and reception of data frames according to a network protocol, preferably a connectionless protocol like high speed Ethernet, and are connectable to a port on another network switch node inside the mesh across an internal communication link, or to a network communication medium outside the mesh which constitutes, or is coupled with, an external communication link.

The switch nodes also include resources to execute a routing process for frames inside the mesh. These resources include a route table memory which has a set of accessible memory locations that store switch route data specifying routes through the plurality of switch nodes inside the mesh of the network switch for specific flows of data frames, or for data frames having specific destination addresses. Flow detect logic is coupled with the set of ports on the switch node, which monitors frames received by the set of ports and generates an identifying tag for use in accessing the route table memory. Example tags consist of a destination address at one of the data link layer or the network layer, a portion of the destination address, or hash values based on one or more fields in control segments of the frame. The tags preferably act as flow signatures to associate a frame with a sequence of frames traversing the switch. For example, when a large file is transferred, a sequence of frames is generated which constitutes a flow of data to a single destination, and frames in the sequence have a single identifying tag. Finally, the switch node includes node route logic which is coupled with the flow detect logic, the route table memory and the set of ports. The node route logic monitors frames received by the set of ports to route a received frame for transmission out a port in the set of ports.

The node route logic determines whether the received frame includes a switch route field that indicates a port in the set of ports to which the frame should be directed for transmission. If the received frame includes a switch route field, that field is updated according to a source route type protocol, and the frame is forwarded with the updated switch route field out the indicated port. If the received frame does not include a switch route field, such as would normally be the case for a frame entering the network switch at a switch node on the border of the network switch, then the identifying tag generated by the flow detect logic is used to access the route table memory. Switch route data is retrieved from the route table memory, if an entry exists for the identifying tag of the current frame. This data is used to generate a switch route field for the frame, and to direct the frame out a port indicated by the data.

The node route logic on the respective switch node also includes logic that forwards a received frame for transmission on a default port in the set of ports, when the route table memory does not include switch route data for the identifying tag. The default port is coupled to a route leading to a processor in the system at which switch route data is generated, such as a multi-protocol network router either internal or external to the network switch. Thus, the node route logic further includes logic to receive switch route data from a remote system for a particular identifying tag. This switch route data is stored in the route table memory in association with the particular identifying tag. When a new entry is made in a switch route table, frames having the particular identifying tag are blocked, with or without buffering, until notification is received that it is clear to forward frames having the particular identifying tag. This blocking technique allows the remote system to which a frame was directed for routing, to forward the frame to its destination, prior to other frames in the same flow sequence being routed to that destination. This preserves the order of transmission of frames in a particular flow. The node route logic begins forwarding frames according to the switch route data stored in the route table memory for a particular tag after it receives notification from the remote system that it is clear to forward frames.

The term frame is used herein, unless stated otherwise, in a generic sense as a unit of data transferred according to a network protocol, intending to include data units called frames, packets, cells, strings, or other names which may have more specific meaning in other contexts.

In the preferred system, all the ports on the switch node execute a single local area network protocol. Preferably this protocol is an Ethernet protocol like the carrier sense, multiple access with collision detection CSMA/CD protocol of the widely used Ethernet standard and variants of it. More preferably, the protocol is specified for operation at 100 megabits per second or higher, more preferably at the emerging one gigabit per second Ethernet standard protocol. For example, half duplex and full duplex "Gigabit" Ethernet (IEEE802.3z) or 100 Megabit Ethernet (802.3u) are used in preferred embodiments.

Flow control between the nodes is handled according to the standard LAN protocol of the ports, such as the Ethernet protocol. Thus, management of the frame flow through the switch is conducted on a frame by frame basis with the format of the frame inside the switch essentially unaltered from the format entering or exiting the switch, with well understood and easily implemented technology.

According to another aspect of the present invention, the flow detect logic on the respective switch nodes comprises logic which computes a plurality of hash values in response to respective sets of control fields in a received frame. The respective sets of control fields correlate with different network frame formats which might be encountered in the network. Logic is also included which determines a particular network frame format for a received frame, and selects one of the plurality of hash values as the identifying tag in response to the particular network frame format that has been detected. The hash values preferably comprise cyclic redundancy codes which are generated with hardware CRC generators. In this manner, the identifying tag for an incoming frame is generated very quickly, allowing for cut through of frames in a switch node so that a transmission of a frame on an outgoing port can begin before the complete frame has been received at the incoming port.

The present invention can also be characterized as individual switch nodes for use in a network switch in the configuration described above. In another aspect, the network switch node comprises an integrated circuit on which the plurality of ports, the flow control logic, and the flow detect logic are incorporated, and interconnected by an embedded high speed bus. A system including any two or more of such integrated circuits combined together to form a mesh, provide a network switch. According to another aspect of the invention, the ports on the integrated circuits are coupled with standard jack connectors, or other standard connector interfaces, allowing users of switch circuits including a plurality of integrated circuits to connect them together using cables in any desired configuration. Thus, a very flexible switch architecture is provided which can be configured for individual installations very easily.

A high performance network switch is provided according to the present invention based on a switch node made with an integrated circuit having 3 or more LAN ports. A frame is routed amongst the nodes in the switch without moving across any intermediate non-LAN bus (excluding the memory interface in each of the nodes used for the frame buffers). A route decision is made in each node based on a switch route header attached to the LAN frame, or on the Ethernet address contained within the frame, or directed to a default route if no route is stored in the route table and the Ethernet address is unknown. The flow control amongst the nodes in the switch is handled based on standard LAN

control signals. In the preferred system, the standard LAN interface amongst the nodes is 100 megabit per second or higher Ethernet, and more preferably the emerging 1 gigabit per second Ethernet protocol.

Other aspects and advantages of the present invention can be seen upon review of the drawings, the detailed description and the claims which follow.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a simplified diagram of a network including an atomic network switch according to the present invention, interconnecting a plurality of standard Ethernet links.

FIG. **2** is a block diagram of a network switch based on a mesh of switch nodes according to the present invention.

FIG. **3** is a block diagram of a switch node according to the present invention.

FIG. **4** is a flow chart illustrating the process executed by the node route logic in the switch node of FIG. **3**.

FIG. **5** is a diagram illustrating the process of generating identifying tags based on cyclic redundancy code hash generators for the flow detect logic of the system of FIG. **3**.

FIG. **6** is a simplified block diagram of the flow detect logic for multiple parallel flows for use in the system of FIG. **3**.

FIG. **7** is a flow chart illustrating the process executed in a router or other network route processor for frames received from the network switch, which do not have entries in the route tables of the network switch.

DETAILED DESCRIPTION

A detailed description of embodiments of the present invention is provided with reference to FIGS. **1** through **7**, where FIG. **1** illustrates the context in which the present invention is utilized. In FIG. **1**, an atomic network switch **10** according to the present invention is connected by standard Ethernet links **11-1** through **11-9** to a plurality of end stations **12-1** through **12-9**. The number of end stations and Ethernet links shown in FIG. **1** is arbitrary. A larger or smaller number of links could be connected to a single atomic switch **10** according to the present invention, as described in detail below. Furthermore, the connections **11-1** through **11-9** from the atomic switch to the respective end stations are all standard network connections, preferably CSMA/CD protocol links, such as the standard full duplex fast Ethernet (IEEE802.3u) specified for 100 megabits per second each way, or the emerging standard full duplex, 1 gigabit per second Ethernet protocol. In the preferred system, all links **11-1** through **11-9** operate according to the same network protocol. However, alternative systems accommodate multiple network protocols on the external ports of switch **10**.

The end stations **12-1** through **12-9** may be personal computers, high performance workstations, multimedia appliances, printers, network intermediate systems coupled to further networks, or other data processing devices as understood in the art.

According to one embodiment of the present invention one of the end stations, such as end station **12-1** includes resources to manage the configuration of the atomic network switch **10**, such as initializing route tables, maintaining the route tables, and providing other functions. Thus, end station **12-1** may include resources to act as a multi-protocol router, such as the NetBuilder2 manufactured by 3Com Corporation of Santa Clara, Calif.

FIG. **2** illustrates the internal architecture of the atomic network switch **10** shown in FIG. **1**. The atomic network switch **10** is comprised of a plurality of switch nodes arranged in rows and columns in FIG. **2**. The switch nodes are labeled in the drawing by column and row numbers. Thus, the switch node in the upper left hand corner is node **1-1**. The switch node at row **1**, column **2** is node **1-2**, and so on throughout the mesh. In a preferred embodiment, each switch node includes an integrated circuit, such as integrated circuit **105** in node **1-1**, coupled to a memory chip, such as chip **106** in node **1-1**. Each of the nodes includes four ports. Thus, node **1-1** includes port **101**, port **102**, port **103**, and port **104**.

The boundary of the network switch in FIG. **2** comprises the nodes **101** and **102** of node **1-1**, port **110** of node **1-2**, port **111** of node **1-3**, **112** of node **1-3**, port **113** of node **2-3**, port **114** of node **3-3**, port **115** of node **4-3**, port **116** of node **4-3**, port **117** of node **4-2**, port **118** of node **4-1**, port **119** of node **4-1**, port **120** of node **3-1**, and port **121** of node **2-1**. Each of the ports **110–121**, **101** and **102** on the boundary of the switch is connected to through a physical layer device, **121–134** to respective physical communication media, such as fiberoptic cables, twisted pair cables, wireless links, such as radio frequency or infrared channels, or other media specified according to standard local area network physical layer specifications. The connection between switch nodes, such as the connection **140** between port **141** on node **2-3** and port **142** on node **2-2**, consist of medium independent interface connections which are defined for connection between MAC logic on a port, and medium dependent components for a port. However, these medium independent connections are connected from MAC logic to MAC logic directly. Preferably all the links between the ports in the network switch execute the same network protocol as the ports on the boundary of the switch. However, alternative systems support multiple protocol types at the boundary.

Management of the configuration of the network switch is accomplished in a router **150** which is connected across link **151** to the physical layer device **130** on the network switch.

The memory chips, such as chip **106** at node **1-1**, in the network switch are used to store route tables, and as frame buffers used in routing of frames amongst the nodes of the switch.

In operation, the network switch receives and transmits standard LAN frames on physical interfaces **121–134**. Preferably, the LAN interconnections comprise CSMA/CD LANs, such as 100 Megabit Ethernet (IEEE802.3 u), or 1 gigabit Ethernet. When a standard frame enters the switch at one physical interface, it is directed out of the switch through another physical interface as indicated by the address data carried by the frame itself. The individual nodes in the switch include a switch routing feature. Each individual node selects a port on which to transmit a received frame based upon the contents of the header of the incoming frame.

There are two internal modes for routing frames inside the switch. In the base mode, each node routes frames using a switch route header attached to the beginning of the regular LAN frame. The switch route header in one example consists of a series of bytes, each byte specifying one or more hops of the route. The top two bits in one byte specify a direction, in the next bits specify the distance. As a frame moves through each node, the header is updated until it reaches the target. Before a frame leaves the mesh, all the switch route bytes are stripped, and the frame has the same format as it had when it entered the mesh or, if required, a format adapted to the network protocol of the exit port.

The nodes of the switch, at least nodes on the boundary of the switch, also have a look up mode. When a frame

enters the switch, with no source route header, the Ethernet addresses, or other fields of the control header of the frame are utilized access the route table. In preferred systems, a CRC-like checksum generator is run over the header of the frame, or over selected fields in the header. At the end of the header, the checksum, or the low order bits of the checksum, are used as a hash code to access a route table stored in the memory associated with the node. Other look up techniques could be utilized for accessing the route table in the memory. For example, the destination address of the incoming frame could be used directly as an address in the table.

If there is an entry in the route table corresponding to the header of the frame, then the switch route data from the table is used to create a switch route header. The header is attached to the frame, and the frame is transmitted at the appropriate port. If no entry is found in the route table, then the frame is routed to a default address, such as the address of a multiprotocol router associated with the switch. The multiprotocol router at the default address also performs management functions such as reporting status, initializing the network, broadcast functions, and managing node route tables. Routing the frame to a default address alternatively involves attachment of a switch route header to direct the frame to the default address, or simply forwarding the frame at a default port in the local node, such that the next node in the mesh to receive the frame also looks it up in its own route table to determine whether the frame is recognized. Either way, the frame reaches the default address and is handled appropriately.

Flow control of the frames in the mesh, and at the boundary of the mesh, is based on the network protocol of the links, such as Ethernet. Therefore, in the preferred Ethernet example, if a port is not available in a target node due to a busy link, a collision on the link, or lack of memory space at the target node, the frame will be refused with a jam signal or a busy signal on the link. The sending node buffers the frame, and retries the transmission later, according to the backoff and retry rules of the protocol or other flow control techniques of the protocol.

The standard higher-speed Ethernet protocols include both half duplex and full duplex embodiments. The 100 Megabit per second Ethernet, defined by IEEE802.3u, clause 31 "MAC Control," defines a frame-based flow control scheme for the full duplex embodiment. Flow control slows down the aggregate rate of packets that a particular port is sending. The method used revolves around control frames distinguished by a unique multicast address and a length/type field in the packet. When a MAC port controller detects that it has received a control frame, the opcode in the control frame is sensed, and transmission of packets is controlled based on the opcode. In existing specifications, a single opcode PAUSE is defined. Thus, in response to the PAUSE opcode, transmission of packets is either enabled or disabled depending on the current state in a Xon/Xoff type mechanism. Thus, this full duplex mode does not depend on the shared media, collision detect techniques of the classic CSMA/CD protocols.

All the proposed standards in the Ethernet family basically use the standard 802.3/Ethernet frame format, conformed to the 802.2 logical link control layer interface, and the 802 functional requirement document with the possible exception of Hamming distance. Also, the minimum and maximum frame size as specified by the current 802.3 standard and by the half or full duplex operational modes is different in the higher rate standards. Thus, the half and full duplex embodiments of the 100 Megabit per second and Gigabit per second Ethernet standards are often referred to as CSMA/CD protocols, even though they may not fit completely within the classic CSMA/CD definition.

FIG. **3** is a simplified block diagram of a single node in the network switch according to the present invention. The node consists of an integrated circuit **200** comprising ports **201-1, 201-2, . . . 201-X**. Each port includes the frame buffer and port management logic normally associated with standard bridges. Also, coupled to each of the ports, is a medium access control MAC unit **202-1, 202-2, . . . 202-X**. The MAC units **202-1** to **202-X** are coupled to medium independent interfaces MII **203-1, 203-2, . . . 203-X**.

In the embodiment of FIG. **3**, each of the medium independent interfaces is connected to a connector jack **260-1, 260-2, 260-X**. The connector jacks comprise a standard connector to which a cable **270-1, 270-2, 270-X** is easily connected by the user. The cable may comprise a coaxial cable for medium independent interfaces based on serial data, or ribbon cables for wider data buses. A variety of mechanical jack configurations can be used as known in the art. For example, coaxial stubs can be mounted on printed circuit boards adjacent each port of the integrated circuits. A short coaxial cable is then connected from stub-to-stub in order to arrange the plurality of integrated circuit chips in a mesh that suits the particular installation. Also, standard ribbon connector jacks can be surface mounted on printed wiring boards adjacent to the integrated circuit. The ribbon cables are connected into the ribbon connector jacks in order to establish the inter-connection.

In alternatives, each of the switches is mounted on a daughter board, with jacks designed to be connected to a mother board in which the data is routed according to the needs of the particular application. In alternative systems, the jacks **260-1** through **260-X** are not included, and the medium independent interfaces are routed in the printed wiring board in a hard-wired configuration, designed for a particular installation.

Medium independent interfaces allow for communication by means of the jacks **260-1** to **260-X** and cables **270-1** to **270-X**, or otherwise, directly with other MAC units on other switch integrated circuits, or to physical layer devices for connection to actual communication media. For example, the MII **203-1** in FIG. **2** is connected directly to a port on another node in the switch. The MII **203-2** in FIG. **2** is connected to a physical layer device **204** for port **2** through jack **271**. The physical layer device **204** is connected to a physical transmission medium **205** for the LAN being utilized. The MII **203-X** in FIG. **2** is coupled directly to another chip within the switch mesh.

According to one embodiment of the present invention, integrated circuit **200** includes a memory interface **206** for connection directly to an external memory, such as a Rambus dynamic random access memory RDRAM **207**. The RDRAM **207** is utilized to store the switch route table **220**, and for frame buffers **221** utilized during the routing of frames through the node.

The internal architecture of the integrated circuit **200** can take on a variety of formats. In one preferred embodiment, the internal architecture is based on a standard bus architecture specified for operation at 1 Gigabit per second, or higher. In one example, a 64 bit-wide bus **210** operating at 100 Megahertz is used, providing 6.4 Gigabits per second as a theoretical maximum. Even higher data rates are achievable with faster clocks. The integrated circuit of FIG. **3** includes bus **210** which is connected to a memory arbiter unit **211**. Arbiter unit **211** connects the bus **210** to a CPU processor **212** across line **213**. The processor **212** is utilized

to execute the route logic for the node. Each of the switch ports **201**-1 to **201**-X is coupled to the bus **210**, and thereby through the arbiter **211** to the CPU **212** and the memory interface **206**. Also, flow detect logic **215** is coupled to the bus **210** for the purpose of monitoring the frame received in the node to detect flows, and to generate identifying tags for the purpose of accessing the switch route table in the RDRAM **207**. The arbiter **211** provides for arbitration amongst the ports, the flow detect logic, the memory, and the CPU for access to the bus, and other management necessary to accomplish the high speed transfer data from the ports to the frame buffers and back out the port.

A representative location **250** of the switch route table is shown. The location **250** includes a field **251** for the identifying tag, a field **252** for the route header, a field **253** for a block-unblock control bit, and a field **254** or fields for information used in the management of the route table, such as the age of the entry. The tag field **251** may be associated with a location by one or more of using the tag or a portion of the tag in the address, by storing all or part of the actual tag data in the addressed location, or by using other memory tag techniques.

The route header in the preferred embodiment consists of a sequence of route bytes. The first field in a route byte includes information identifying a direction, which corresponds to a particular port on the node, and a second field in the byte includes a count indicating the number of steps through the switch from node to node which should be executed in the direction indicated by the first field. For example, an eight bit route byte in a switch having nodes with four ports, includes a two bit direction field, and a six bit count field, specifying up to 63 hops in one of four directions. A sequence of route bytes is used to specify a route through the switch. Thus, the switch route header uses source routing techniques within the switch for the purposes of managing flow frames through the switch. The source route approach may, for example, in a 4 port node include a field for hops to right, hops to the left, hops up and hops down. The first field may carry information indicating left 4 hops, followed by a field indicating down 2 hops, followed by a field indicating left one hop to exit the switch. Thus, a frame would be transmitted out the left and in the right port of 3 nodes, in the right and out the down port of 1 node, in the top and out the down of 1 node, and in the top and out the left of the last node on the boundary of the switch. A standard Ethernet frame format takes over for transmission through the network outside the switches. As the size of the mesh grows, and the bandwidth handled by the mesh increases, more sophisticated routing techniques are available because of the flexible technology utilized. For larger switches, more than one route exists for frames entering one node and leaving on another node. Thus, the switch can be configured to minimize the number of frames which are blocked in passage through the switch, while maintaining optimum utilization of the bandwidth available through the switch.

The block-unblock field **253** is used during the updating of the switch route table by the host CPU **212** to block routing of frames corresponding to new entries, until it is assured that the first frame in the flow to which the entry corresponds, arrives at its destination before the node begins forwarding following frames in the flow to the destination using the route header, in order to preserve the order of transmission of the frames. The age field **254** is used also by the CPU **212** for the purpose of managing the contents of the route table. Thus, entries which have not been utilized for a certain amount of time are deleted, or used according to least-recently-used techniques for the purposes of finding locations for new entries. Other control fields (not shown) include a field for storing a count of the number of packets forwarded by the node using this route, a drop/keep field to indicate packets that will be dropped during overflow conditions, a priority "high/low" field for quality of service algorithms, and additional fields reserved for future use, to be defined according to a particular embodiment.

The frame buffer **221** is preferably large enough to hold several frames of the standard LAN format. Thus, a standard Ethernet frame may comprise 1500 bytes. Preferably, the frame buffer **221** is large enough to hold at least one frame for each of the ports on the flow switch.

The flow switch **200** includes more than 2 ports, and preferably 4 or more ports. All the ports are either connected through the media independent interfaces **203**-1 through **203**-X directly to other chips in the mesh, or to physical layer devices for connection to external communication media.

The router or other management node for the switch may communicate with each of the nodes **200** using well-known management protocols, such as SNMP (simple network management protocol), enhancements of SNMP, or the like. Thus, the RDRAM **207** associated with each node also stores statistics and control data used by the management process in controlling the switch node.

Although in FIG. 3, the RDRAM **207** is shown off the chip **200**, alternative embodiments incorporate memory into the switch integrated circuit **200**, for more integrated design, smaller footprint for the switch, and other classic purposes for higher integration designs.

The CPU **212** executes the node route logic for the node. A simplified flow chart of the node route process executed by CPU **211** is shown in FIG. **4**.

The process begins with the receipt of the frame on a particular port (step **300**). The CPU first determines whether the frame carries a route header (step **301**). This process is executed in parallel with the transferring of the frame being received to the frame buffer of the node. If the frame carries a route header, then the CPU updates the header by decrementing the hop count, or otherwise updating the information to account for a traversed leg of the route according to the particular switch route technique utilized. The CPU transmits the frame (with updated header) on the port identified by the header (step **302**). If at step **301**, no switch route header was detected, the flow detect logic is accessed to determine a tag for the frame (step **303**). The tag is utilized by the CPU to access entries in the route table (step **304**). If a match is found in the route table, then a route header is generated for the frame (step **305**). Then, the header is updated (if required), and the frame is transmitted on the port identified by the data in the table (step **302**). If at step **304**, no match was found in the route table, then the frame is transmitted on a default port (step **306**). An alternative technique to transmitting the frame on a default port, is to add a default route header to the frame, and transmit the frame according to the information in the default route header. In this manner, subsequent nodes in the switch will not be required to perform the look-up operation for the purposes of routing the frame. However, it may be desirable to have each node look up the frame in its own route table, in order to insure that if any node already has data useful in forwarding the frame, then that frame will be forwarded appropriately without requiring processing resources of the management process at the default address.

FIG. **5** illustrates the technique executed by the flow detect logic in generating an identifying tag for the frame

being received. FIG. 5 includes the format of a standard Ethernet (802.3) style frame **400**. The frame includes a start of frame deliminator SOF in field **401**. A destination address is carried in field **402**. A source address is carried in field **403**, and miscellaneous control information is carried in additional fields **404**. A network layer header, such as an Internet protocol header in this example, is found in field **405**. Other style network layer headers could be used depending on the particular frame format. The data field of variable length is found at section **406** of the frame. The end of the frame includes a CRC-type checksum field **407** and an end-of-frame deliminator **408**. The flow detect logic runs a CRC-type hash algorithm over selected fields in the control header of the frame to generate a pseudo-random tag. Thus, the field **410**, the field **411**, the field **412**, and the field **413** are selected for input into a CRC hash generator **414**. The tag generated by the hash generator **414** is supplied on line **415** for use in accessing the route table **416**. The route table either supplies a route header on line **417**, or indicates a miss on line **418**. In this way, the route management software executed by the CPU can make the appropriate decisions.

The embodiment of FIG. 5 selects a particular set of fields within the frame for the purpose of generating the pseudo-random tag. The particular set of fields is selected to correspond to one standard frame format encountered in the network. However, a variety of frame formats may be transmitted within a single Ethernet style of network, although in this example, a CRC-type hash generator is utilized, relying on typical CRC-type algorithms, referred to as polynomial arithmetic, modulo II. This type of arithmetic is also referred to as "binary arithmetic with no carry" or serial shift exclusive-OR feedback. However, a variety of pseudo-random number generation techniques can be utilized, other than CRC-like algorithms. The two primary aspects needed for a suitable pseudo-random hash code are width and chaos, where width is the number of bits in the hash code, which is critical to prevent errors caused by the occurrence of packets which are unrelated but nonetheless result in the same hash being generated, and chaos is based on the ability to produce a number in the hash register that is unrelated to previous values.

Also, according to the present invention, the parsing of the frames incoming for the purposes of producing an address to the look-up table can take other approaches. This parsing can be referred to as circuit identification, because it is intended to generate a number that is unique to the particular path of the incoming frame.

The circuit identification method depends on verifying a match on specific fields of numbers in the incoming frame. There are two common table look-up methods, referred to as binary search and hash coding. The key characteristic of binary search is that the time to locate an entry is proportional to the log base 2 of the number of entries in the table. This look-up time is independent of the number of bits in the comparison, and the time to locate a number is relatively precisely known.

A second, more preferred, method of look-up is based on hash coding. In this technique, a subset of address field or other control fields of the frame are used as a short address to look into the circuit table. If the circuit table contains a match to the rest of the address field, then the circuit has been found. If the table contains a null value, then the address is known not to exist in the table. The hash method has several disadvantages. It requires a mostly empty table to be efficient. The time to find a circuit cannot be guaranteed. The distribution of duplicates may not be uniform, depending on the details of which fields are selected for the initial address generation.

The address degeneracy problem of the hash coding technique is reduced by processing the initial address fragment through a polynomial shift register. This translates the initial address to a uniformly-distributed random number. A typical example of random number generation is the CRC algorithm mentioned above. In a preferred hashing technique, the hardware on the flow switch includes at least a template register, pseudo-random number generation logic and a pseudo-random result register. The template register is loaded to specify bytes of a subject frame to be included in the hash code. The template specifies all protocol-dependent fields for a particular protocol. The fields are not distinguished beyond whether they are included in the hash or not. As the frame is processed, each byte of the initial header is either included in the hash function or it is ignored, based on the template. A hash function is generated based on the incoming packet and the template. The pseudo-random number generator is seeded by the input hash bits selected by the template. The change of a single bit in the input stream should cause a completely unrelated random number to be generated. Most common algorithms for generating pseudo-random numbers are linear-congruential, and polynomial shift methods known in the art. Of course, other pseudo-random number generation techniques are available.

A first field of the pseudo-random number is used as an address for the look-up table. The number of bits in this field depends on the dimensions of the look-up table. For example, if the circuit table has 64,000 possible entries, and the hash number is eight bytes long, the first two bytes are used as an address. The other six bytes are stored as a key in the hash table. If the key in the hash table matches the key in the hash code, then the circuit is identified. The additional bytes in the table for the addressed entry specify the route to be applied. The length of the pseudo-random hash code is critical, to account for the probability that two unrelated frames will result in the same hash number being generated. The required length depends on the size of the routing tables, and the rate of turnover of routes.

The problem with a pure hash code circuit identification technique is that there is a chance of randomly misrouting a packet. The problem arises when you are generating random numbers out of a larger set. There is a chance that two different input patterns will produce the same hash code. Typically, a hash code will be loaded into a table with a known route. Then a second, different, packet will appear that reduces to the same hash code as the one already in the table. The second packet will be falsely identified as having a known route, and will be sent to the wrong address. The exact mechanism of this error can be understood by the well-known statistics of the "birthday problem." The "birthday problem" answers the question, "What is the probability that two people in a group will have the same birthday?" It turns out that the number of people in a group required for there to be a likelihood of two people having the same birthday is quite small. For example, there is a 50% chance that two people out of a group of 23 will have the same birthday.

The probability of a switching error depends on the number of circuits active. For example, if there are no circuits active, then there is no chance that an invalid circuit will be confused with another circuit, since there are no valid circuits. As each circuit is added to the table, it decreases the remaining available space for other numbers by approximately $(\frac{1}{2})^{bits}$, where "bits" is the number of bits in the hash code. If the hash code is 32 bits long, then each entry into the circuit table will reduce the remaining code space by $(\frac{1}{2})^{32}$, which is equal to 2.32×10^{-10}. The cumulative prob-

ability of not making an error in the circuit table is equal to the product of the individual entry errors up to the size of the table. This is $(1)*(1-1/2^{32})*(1-2/2^{32})*(1-3/2^{32}) \ldots *(1-n/2^{32})$, where n is the number of entries in the table. In the case of a 32-bit hash code, and an 8,000-entry circuit table, the probability of making an error in the table would be about 0.7%. With a 64,000-entry circuit table, the probability of an error would be about 39%.

Using a 32-bit hash code and some typical-sized circuit tables indicates that the conventional wisdom is correct. That is, there will be routing errors if only a 32-bit hash code is used. However, if the number of bits in the hash code is increased and probability is recalculated for typical-sized circuit tables, we find that the probability of error quickly approaches zero for hash codes just slightly longer than 32 bits. For example, an 8,000-entry table with a 40-bit hash code will reduce the error rate to 0.003%. A 48-bit hash code will reduce the error to 0.000012%. These calculations show that a pure hash code look-up table can be used if the length of the hash code is longer than 32 bits for typical-size tables.

As a further example, consider the case of a 64-bit hash code. Assuming an 8,000-entry table, the probability of making an error is $2*10^{-12}$. Even if the table is completely replaced with new entries every 24 hours, it would take over one billion years for an error to occur. Using a 64-bit hash code with a 64,000-entry table would give a probability of error of 10^{-10}. Assuming the table turned over every day, it would take about 28 million years for an error to occur. An error might occur sooner, but the rate would be negligible. In all cases, there is no realistic chance of making an error based on this routing technique within the lifetime of typical networking equipment.

In a preferred embodiment, filtering mechanisms are implemented on the flow switch integrated circuit, and multiple filters operate in parallel. The circuit look-up table is implemented with external memory much larger than the number of circuits expected to be simultaneously active. This means that the hash pointer generated either points to a valid key or a miss is assumed. There is no linear search for matching key. When a circuit is not found in the table, the packet is routed to a default address. Normally, this default address directs the packet to a stored program router. The router will then parse the packet using standard methods, and then communicate with the flow switch circuit to update the circuit table with the correct entry. All subsequent packets are directly routed by the switch element without further assistance from the router.

Example template organizations for the bridging embodiment, the IP routing embodiment, and the IPX routing embodiment are set forth below.

Example for bridging:

Basic ethernet packet:	Preamble 64 bits are discarded	
DestinationAddress:	bytes 1–6	Used
SourceAddress:	bytes 7–12	Used
Packet Type:	bytes 13–14	are ignored (802.3 length)
Data bytes:	15 upto 60	are ignored
CRC:	Last 4 bytes	are ignored

The template register is 8 bytes long. Each bit specifies one byte of the header. The first bit corresponds to byte I of the DestinationAddress.

The template for bridging is FF-F0-00-00 00-00-00-00

The selector is: Always TRUE. Hierarchy=1 (default to bridging)

Example for IP:

Preamble 64 bits are discarded		
Destination	bytes 1–6	optional
Source	bytes 7–12	optional
Packet type	bytes 13–14	Ignore (802.3 length)
byte 15:	IP byte 1	= version length = optional
byte 16:	IP byte 2	= service type = Ignore
17–18:	IP 3–4	= length = Ignore
19–22:	IP 5–8	= Ignore
23	IP 9	= TTL = optional
24	IP 10	= Proto = optional
25–26	IP 11–12	= Hdr chksum = Ignore
27–30	IP 13–16	= Source IP address = Used
31–34	IP 17–20	= Destination IP address = Used
35–	IP 21–	= Ignore

Assume that optional fields are included in the pseudo-random hash code.

The template would then be: FF-F2-03-03 FC-00-00-00

The selector is: Bytes 13–15=080045, Hierarchy=2

Example for IPX in an Ethernet frame:

Preamble 64 bits are discarded			
Destination	bytes 1–6		Optional
Source	bytes 7–12		Optional
Type	bytes 13–14		Optional (Selector = 8137)
byte	IPX		
15–16	1–2	Checksum	Ignore
17–18	3–4	Length	Ignore
19	5	Hop count	Optional
20	6	Type	Optional (Selector = 2 or 4)
21–24	7–10	Dest Net	Use
25–30	11–16	Dest Host	Use
31–32	17–18	Dest Socket	Ignore
33–36	19–22	Src Net	Use
37–42	23–28	Src Host	Use
43–	29–		Ignore
Template (with optional fields):			FF-FC-3F-FC FF-C0-00-00
Selector:	Bytes 13–14 = 8137, Hierarchy = 2		

The examples shown are representative, and may not correspond to what would actually be required for any particular application. There are many protocol pattern possibilities. Some combinations may not be resolvable with the hierarchy described in these three examples.

In the embodiment in which there are a number of filters operating in parallel, the flow detect logic includes the template register discussed above, a second register loaded with a template for detecting the specific protocol type represented by the template register. This feeds combinational logic that provides a boolean function, returning a true or false condition based on a string compare of a section of the frame to determine the protocol. A third register is loaded with a hierarchy number, which is used to arbitrate among similar protocols, which might simultaneously appear to be true based on the second protocol detect register. A fourth register is optional, and contains a memory start address which triggers the operation of the filter.

The multiple instantiations of the filters operate in parallel. The filters can be reprogrammed on the fly to support the exact types of traffic encountered. Furthermore, the filters may operate in a pipeline mode along a series of switching nodes. Each protocol returns its hierarchy number when that filter detects the protocol pattern contained in the template. For example, bridging protocol may be defined as true for hierarchy 1 for all frames, If no stronger filter fires, such as an IP or IPX filter, then the bridging filter will be selected as the default.

Thus, the flow detect logic in a preferred system executes a plurality of hash flow analyses in parallel as illustrated by

FIG. 6. Thus in FIG. 6, a received frame is supplied on line 500 in parallel to hash flow logic 1 through hash flow logic N, each flow corresponding to a particular frame format. Also, the received frame is supplied to a hash flow "select" 501 which is used for selecting one of the N flows. The output of flows 1 through N are supplied through multiplexer 502 in FIG. 6, which is controlled by the output of the select flow 501. The output of the select flow 501 causes selection of a single flow on line 503, which is used for accessing the route table by the CPU.

Thus a preferred embodiment of the present invention uses a routing technique base on flow signatures. Individual frames of data move from one of the Ethernet ports to a shared buffer memory at the node. As the data is being moved from the input port to the buffer, a series of hash codes is computed for various sections of the input data stream. Which bits are or are not included in each hash calculation is determined by a stored vector in a vector register corresponding to that calculation. For example, in the most common case of an IP packet, the hash function starts at the 96th bit to find the "0800" code following the link-layer source address, it then includes the "45" code, 32 bits of IP source, 32 bits of IP destination, skips to protocol ID 8 bits, and then at byte 20 takes the source port 16 bits and the destination port 16 bits. The result is a 64 bit random number identifying this particular IP flow.

The hash code is looked up in or used to access a local memory. If the code is found, it means that this flow type has been analyzed previously, and the node will know to apply the same routing as applied to the rest of the flow. If there is no entry corresponding to this hash code, it means that the flow has not been seen lately, and the node will route the frame to a default destination. A least recently used algorithm, or other cache replacement scheme, is used to age flow entries out of the local tables.

In practice, many filters operate simultaneously. For example, filters may be defined for basic bridging, IP routing, sub-variants, Apple Talk, and so on. The actual limit to the number of filters will be determined by the available space on the ASIC. The logic of the filters is basically the same for all the filters. The actual function of each filter is defined by a vector register specifying which bits are detected.

A second feature is the use of multi-level filters. In the common case simultaneously supporting bridging, IP, and IPX; about ten filters operate in parallel. An additional level of coding is used to select which of the other filters is to be used as the relevant hash code. This second level filter would detect whether the flow was IP or IPX for example.

In the case where the flow is not recognized, it is passed to the default route. As the packet passes along the default route, additional nodes may examine the packet and detect its flow type based on different filters or on a different set of flow signatures (hash table entries) stored. This method of cascading filters and tables allows for the total size and speed of the mesh to be expanded by adding nodes. Ultimately, if a packet can not be routed by any of the nodes along the default route, the packet will arrive at the final default router, typically a NetBuilder2. The default router will analyze the packet using standard parsing methods to determine its correct destination. A flow signature will be installed in an appropriate node, or nodes, of the mesh so that subsequent flows of the same signature can be routed autonomously without further intervention.

A flow effectively defines a "circuit" or a "connection"; however, in standard Ethernet design, packets are treated individually without any regard to a connection. Typically a router will analyze every single packet as if it had never seen it before, even though the router might have just processed thousands of identical packets. This is obviously a huge waste of routing resources. The automation of this flow analysis with multiple levels of parallel and cascaded hashing algorithms combined with a default router is believed to be a significant improvement over existing routing methods.

Flow based switching is also critical to ensuring quality of service guarantees for different classes of traffic.

FIG. 7 is a flow chart illustrating the process executed in the router or other management node, whenever a frame is received which does not have a switch route header. Thus, the process of FIG. 7 begins at step 700 where a frame is received in the router, such as the router 150 in FIG. 2. The router applies the multiprotocol routing techniques to determine the destination of the frame. Based on the destination, and other information about the flows within the switch, switch route headers are generated for nodes in the switch (step 701). Thus, a different route header is generated for each node in the switch mesh, and correlated with the tag which would be generated according to the received frame at each node. Next, a message is sent to the nodes in the switch to update the route tables with the new route headers, and to block frames which match the tag of the frame being routed (block 702).

After step 702, the frame is forwarded from the router to its destination (step 703). After the frame has been forwarded to its destination, the router sends a message to all of the nodes in the switch to unblock frames which have a matching tag (step 704). This blocking and unblocking protocol is used to preserve the order in which frames are transmitted through the switch, by making sure that the first frame of a single flow arrives at its destination ahead of following frames.

Logic in the nodes for the purpose of accomplishing the blocking and unblocking operation take a variety of formats. In one example, the entry at each location in the route table includes a field which indicates whether the flow is blocked or not. When an entry is first made in the route table, the blocking field is set. Only after a special instruction is received to unblock the location, is the blocking field cleared, and use of the location allowed at the switch node.

Accordingly, in the preferred system the atomic network switch according to the present invention is based on repeated use of a simple 4-port switch integrated circuit. The integrated circuits are interconnected to create a mesh with a large pool of bandwidth across many ports. The links that interconnect the integrated circuits run according to a LAN protocol, at preferably 100 megabits per second or higher, such as a gigabit per second. Individual ports act as autonomous routers between the boundaries of the switch according to the switch route protocol which is layered on top of the standard frame format. The overall bandwidth of the switch can be arbitrarily increased by adding more atomic nodes to the switch. Using a well-understood and simple interface based on standard Ethernet LAN protocols, vastly simplifies the implementation of each node in the switch, because each is able to rely on well understood MAC logic units and port structures, rather than proprietary complex systems of prior atomic LANS. Furthermore, any node of any switch can be connected to a physical layer device that connects to an Ethernet medium, or can be disconnected from the Ethernet medium and connected to another node switch to readily expand and change the topology of the switch. The fine granularity and scalability of the mesh

architecture, combined with the ability to optimize the topology of the switch for a particular environment allow implementation of a high bandwidth, low cost network switch.

A high bandwidth and very flexible network switch is achievable according to the present invention with a simple, scalable, low-cost architecture.

The foregoing description of a preferred embodiment of the invention has been presented for purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise forms disclosed. Obviously, many modifications and variations will be apparent to practitioners skilled in this art. It is intended that the scope of the invention be defined by the following claims and their equivalents.

What is claimed is:

1. For a network switch including a mesh of interconnected network switch nodes, a network switch node comprising:

a set of ports having more than two members, and the ports in the set including respective medium access control units for transmission and reception of data frames according to a network protocol, the ports in the set of ports being connectable to a port on another network switch node inside the mesh, or to a network communication medium outside the mesh; and

node route logic, coupled with the set of ports, which monitors frames received by the set of ports to route a received frame for transmission according to the network protocol to a selected port in the set of ports, including logic to select the selected port according to rules for navigating through the mesh inside to the network switch, and wherein the node route logic forwards the received frame for transmission to a default location of a multiprotocol router resource associated with the switch when the node route logic cannot otherwise determine a route for the received frame.

2. The network switch node of claim 1, wherein the network protocol comprises a connectionless protocol.

3. The network switch node of claim 1, wherein the network protocol comprises an Ethernet protocol.

4. The network switch node of claim 1, wherein the network protocol comprises an Ethernet, full duplex protocol.

5. The network switch node of claim 1, wherein ports in the set of ports include medium independent interfaces for the network protocol.

6. The network switch node of claim 1, further including:

route table memory, coupled with the node route logic, having a set of accessible locations for storing switch route data;

flow detect logic, coupled with the set of ports, which monitors frames received by the set of ports and generates an identifying tag for use in accessing the route table memory;

wherein the node route logic includes logic which determines whether the received frame includes a switch route field indicating a port in the set of ports, and if the received frame includes a switch route field, updates the switch route field, and forwards the received frame with the updated switch route field to the port indicated by the switch route field, and if the received frame does not include a switch route field, accesses the route table memory using the identifying tag generated in the flow detect logic to retrieve switch route data indicating a port in the set of ports, adds a switch route field to the received frame, and forwards the received frame with the switch route field to the port indicated by the switch route data.

7. The network switch node of claim 6, wherein the default location includes a default port and wherein the node route logic forwards the received frame for transmission on the default port in the set of ports when the switch route table does not include switch route data for the identifying tag.

8. The network switch node of claim 7, wherein the default port is coupled to a route to a multi-protocol, network route processor at which switch route data is generated.

9. The network switch node of claim 6, including logic to receive switch route data from a remote system for a particular identifying tag, to store the switch route data in the route table memory in association with the particular identifying tag, and to block frames having the particular identifying tag until notification is received that it is clear to forward frames having the particular identifying tag, and after notification is received that it is clear to forward frames having the particular identifying tag, forward frames having the particular tag according to the switch route data.

10. The network switch node of claim 6, wherein the default location includes a default port and wherein the node route logic forwards the received frame for transmission on the default port in the set of ports when the route table memory does not include switch route data for the identifying tag; and further including:

logic to receive switch route data from a remote system for a particular identifying tag, to store the switch route data in the route table memory in association with the particular identifying tag, and to block frames having the particular identifying tag until notification is received that it is clear to forward frames having the particular identifying tag, and after notification is received that it is clear to forward frames having the particular identifying tag, forward frames having the particular tag according to the switch route data.

11. The network switch node of claim 10, wherein the default port is coupled to a route to a multi-protocol, network route processor at which switch route data is generated.

12. The network switch node of claim 6, wherein the flow detect logic comprises:

logic which computes a plurality of hash values in response to respective sets of control fields in a received frame, where the respective sets of control fields correlate with respective network frame formats; and

logic which determines a particular network frame format for a received frame, and selects one of the plurality of hash values as the identifying tag in response to the particular network frame format.

13. The network switch node of claim 12, wherein the hash values comprise pseudo-random codes.

14. The network switch node of claim 6, wherein the flow detect logic comprises:

logic which computes a hash value in response to a set of control fields in a received frame, where the set of control fields correlates with a network frame format, and applies the hash value as the identifying tag.

15. The network switch node of claim 14, wherein the hash value comprises a pseudo-random code.

16. The network switch node of claim 1, wherein the network protocol comprises an Ethernet protocol, specified for operation at 100 Megabits per second.

17. The network switch node of claim 16, wherein the Ethernet protocol comprises a full duplex protocol.

18. The network switch node of claim 1, wherein said set of ports and said node route logic comprise elements of a single integrated circuit.

19. The network switch node of claim 18, wherein ports in the set of ports include medium independent interfaces for the network protocol, and the network protocol comprises an Ethernet protocol, specified for operation at 100 Megabits per second or higher.

20. The network switch node of claim 19, wherein the Ethernet protocol comprises a full duplex protocol.

21. The network switch node of claim 1, wherein ports of the set of ports include medium independent interfaces for the network protocol, the medium independent interfaces defining a particular bus configuration, and further including connectors coupled to the medium independent interfaces adapted to receive cables configured according to the particular bus configuration.

22. An integrated circuit, comprising:

a set of ports for access to respective communication media, the set of ports having more than two members, and the ports in the set including respective medium access control logic for a network protocol;

a memory interface for connection to a route table memory having a set of accessible locations for storing switch route data;

flow detect logic, coupled with the set of ports, which monitors frames received by the set of ports and generates an identifying tag for use in accessing the route table memory; and

node route logic, coupled with the flow detect logic, the memory interface and the set of ports, which monitors frames received by the set of ports to route a received frame for transmission to a port in the set of ports, the node route logic determining whether the received frame includes a switch route field indicating a port in the set of ports, and if the received frame includes a switch route field, updates the switch route field, and forwards the received frame with the updated switch route field to the port indicated by the switch route field, and if the received frame does not include a switch route field, accesses the route table memory through the memory interface using the identifying tag generated in the flow detect logic to retrieve switch route data indicating a port in the set of ports, adds a switch route field to the received frame, and forwards the received frame with the switch route field to the port indicated by the switch route data and if the route table memory does not include switch route data for the identifying tag, then forwards the received frame to a default location of a multiprotocol router resource associated with the switch.

23. The integrated circuit of claim 22, wherein the network protocol comprises a connectionless protocol.

24. The integrated circuit of claim 22, wherein the network protocol comprises an Ethernet protocol.

25. The integrated circuit of claim 24, wherein the Ethernet protocol comprises a full duplex protocol.

26. The integrated circuit of claim 22, wherein ports in the set of ports include medium independent interfaces for the network protocol.

27. The integrated circuit of claim 22, wherein the default location includes a default port and wherein the node route logic forwards the received frame for transmission on the default port in the set of ports when the switch route table does not include switch route data for the identifying tag.

28. The integrated circuit of claim 27, including logic to receive switch route data from a remote system for a particular identifying tag, to store the switch route data in the route table memory in association with the particular identifying tag, and to block frames having the particular identifying tag until notification is received that it is clear to forward frames having the particular identifying tag, and after notification is received that it is clear to forward frames having the particular identifying tag, forward frames having the particular identifying tag according to the switch route data.

29. The integrated circuit of claim 22, wherein the default location includes a default port and wherein the node route logic forwards the received frame for transmission on the default port in the set of ports when the route table memory does not include switch route data for the identifying tag; and further including:

logic to receive switch route data from a remote system for a particular identifying tag, to store the switch route data in the route table memory in association with the particular identifying tag, and to block frames having the particular identifying tag until notification is received that it is clear to forward frames having the particular identifying tag, and after notification is received that it is clear to forward frames having the particular identifying tag, forward frames having the particular identifying tag according to the switch route data.

30. The integrated circuit of claim 22, wherein the flow detect logic comprises:

logic which computes a plurality of hash values in response to respective sets of control fields in a received frame, where the respective sets of control fields correlate with respective network frame formats; and

logic which determines a particular network frame format for a received frame, and selects one of the plurality of hash values as the identifying tag in response to the particular network frame format.

31. The integrated circuit of claim 30, wherein the hash values comprise pseudo-random codes.

32. The integrated circuit of claim 22, wherein the flow detect logic comprises:

logic which computes a hash value in response to set of control fields in a received frame, where the set of control fields correlates with a network frame format, and applies the hash value as the identifying tag.

33. The integrated circuit of claim 32, wherein the hash value comprises a pseudo-random code.

34. The integrated circuit of claim 22, including an embedded bus interconnecting the set of ports, the flow detect logic, the node route logic and the memory interface.

35. The integrated circuit of claim 22, wherein the network protocol comprises an Ethernet protocol, specified for operation at 100 Megabits per second or higher.

36. The integrated circuit of claim 35, wherein the Ethernet protocol comprises a full duplex protocol.

37. The integrated circuit of claim 35, including a bi-directional, embedded bus interconnecting the set of ports, the flow detect logic, the node route logic and the memory interface, the embedded bus specified for operation at 1 Gigabit per second or higher.

38. The integrated circuit of claim 22, including the route table memory on the integrated circuit.

39. A network switch, comprising:

a plurality of switch nodes;

a first set of communication links, communication links in the first set coupled between switch nodes in the plurality of switch nodes internal to the network switch;

a second set of communication links, communication links in the second set comprising network links external to the network switch;

the respective switch nodes in the plurality of switch nodes including

a set of ports connected to respective communication links in either the first set of communication links or the second set of communication links, the set of ports having more than two members, and the ports in the set including respective medium access control logic for a network protocol;

route table memory having a set of accessible locations for storing switch route data which specify routes through the plurality of switch nodes;

flow detect logic, coupled with the set of ports, which monitors frames received by the set of ports and generates an identifying tag for use in accessing the route table memory; and

node route logic, coupled with the flow detect logic, the route table memory and the set of ports, which monitors frames received by the set of ports to route a received frame for transmission to a port in the set of ports, the node route logic determining whether the received frame includes a switch route field indicating a port in the set of ports, and if the received frame includes a switch route field, updates the switch route field, and forwards the received frame with the updated switch route field to the port indicated by the switch route field, and if the received frame does not include a switch route field, accesses the route table memory using the identifying tag generated in the flow detect logic to retrieve switch route data indicating a port in the set of ports, adds a switch route field to the received frame, and forwards the received frame with the switch route field to the port indicated by the switch route data, and if the route table memory does not include switch route data corresponding to the identifying tag, then forwarding the received frame to a default location of a multiprotocol router resource associated with the switch.

40. The network switch of claim **39**, wherein the network protocol for ports in the set of ports on the respective switch nodes comprises a connectionless protocol.

41. The network switch of claim **39**, wherein the network protocol for ports in the set of ports on the respective switch nodes comprises an Ethernet protocol.

42. The network switch of claim **41**, wherein the Ethernet protocol comprises a full duplex protocol.

43. The network switch of claim **39**, wherein ports in the set of ports on the respective switch nodes include medium independent interfaces for the network protocol.

44. The network switch of claim **39**, wherein the default location includes a default port and wherein the node route logic on the respective switch nodes forwards the received frame for transmission on the default port in the set of ports when the switch route table does not include switch route data for the identifying tag.

45. The network switch of claim **44**, wherein the default port is coupled to a route to a multi-protocol, network route processor at which switch route data is generated.

46. The network switch of claim **39**, including logic on the respective switch nodes to receive switch route data from a remote system for a particular identifying tag, to store the switch route data in the route table memory in association with the particular identifying tag, and to block frames having the particular identifying tag until notification is received that it is clear to forward frames having the particular identifying tag, and after notification is received that it is clear to forward frames having the particular identifying tag, forward frames having the particular identifying tag according to the switch route data.

47. The network switch of claim **39**, wherein the node route logic on the respective switch nodes forwards the received frame for transmission on a default port in the set of ports when the route table memory does not include switch route data for the identifying tag; and further including:

logic on the respective switch nodes to receive switch route data from a remote system for a particular identifying tag, to store the switch route data in the route table memory in association with the particular identifying tag, and to block frames having the particular identifying tag until notification is received that it is clear to forward frames having the particular identifying tag, and after notification is received that it is clear to forward frames having the particular identifying tag, forward frames having the particular identifying tag according to the switch route data.

48. The network switch of claim **47**, wherein the default port is coupled to a route to a multi-protocol, network route processor at which switch route data is generated.

49. The network switch of claim **39**, wherein the flow detect logic on the respective switch nodes comprises:

logic which computes a plurality of hash values in response to respective sets of control fields in a received frame, where the respective sets of control fields correlate with respective network frame formats; and

logic which determines a particular network frame format for a received frame, and selects one of the plurality of hash values as the identifying tag in response to the particular network frame format.

50. The network switch of claim **49**, wherein the hash values comprise pseudo-random codes.

51. The network switch of claim **39**, wherein the flow detect logic on the respective switch nodes comprises:

logic which computes a hash value in response to set of control fields in a received frame, where the set of control fields correlates with a network frame format, and applies the hash value as the identifying tag.

52. The network switch of claim **51**, wherein the hash value comprises a pseudo-random code.

53. The network switch of claim **39**, wherein the network protocol for ports in the set of ports on the respective switch nodes comprises an Ethernet protocol, specified for operation at 100 Megabits per second or higher.

54. The network switch of claim **53**, wherein the Ethernet protocol comprises a full duplex protocol.

55. The network switch of claim **39**, wherein the MAC logic for ports in the set of ports on the respective switch nodes executes the same network protocol for all ports in the set of ports.

56. The network switch of claim **39**, wherein ports in the set of ports on the respective switch nodes include medium independent interfaces for the network protocol, the medium independent interfaces defining a particular bus configuration, and further include connectors coupled to the medium independent interfaces adapted to receive cables configured according to the particular bus configuration.

* * * * *

2755
2151

O I P E JC42 APR 18 2002 PATENT & TRADEMARK OFFICE

Our Docket/Ref. No.: APPT-00.1 Patent

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant(s): Dietz et al. Serial No.: 09/608237 Filed: June 30, 2000 Title: METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK	Group Art Unit: 2755 Examiner:

RECEIVED
APR 2 2 2002
Technology Center 2100

Commissioner for Patents
Washington, D.C. 20231

TRANSMITTAL: INFORMATION DISCLOSURE STATEMENT

Dear Commissioner:

Transmitted herewith are:

X An Information Disclosure Statement for the above referenced patent application, together with PTO form 1449 and a copy of each reference cited in form 1449.

___ A check for petition fees.

X Return postcard.

X The commissioner is hereby authorized to charge payment of any missing fee associated with this communication or credit any overpayment to Deposit Account 50-0292.
A DUPLICATE OF THIS TRANSMITTAL IS ATTACHED

Date: 30 Mar 2002

Respectfully submitted,

Dov Rosenfeld
Attorney/Agent for Applicant(s)
Reg. No. 38687

Correspondence Address:
Dov Rosenfeld
5507 College Avenue, Suite 2
Oakland, CA 94618
Telephone No.: +1-510-547-3378

Certificate of Mailing under 37 CFR 1.18

I hereby certify that this correspondence is being deposited with the United States Postal Service as first class mail in an envelope addressed to: Commissioner for Patents, Washington, D.C. 20231.

Date of Deposit: 30 Mar 2002 Signature: _____
Dov Rosenfeld, Reg. No. 38,687

UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO	CONFIRMATION NO.
09/608,237	06/30/2000	Russell S. Dietz	APPT-001-1	9993

7590 06/25/2003

Dov Rosenfeld
Suite 2
5507 College Avenue
Oakland, CA 94618

EXAMINER	
MEKY, MOUSTAFA M	
ART UNIT	PAPER NUMBER
2157	6

DATE MAILED: 06/25/2003

Please find below and/or attached an Office communication concerning this application or proceeding.

Office Action Summary	Application No.	Applicant(s)	
	09/608,237	DIETZ ET AL.	
	Examiner	Art Unit	
	Moustafa M Meky	2157	

-- The MAILING DATE of this communication appears on the cover sheet with the correspondence address --

Period for Reply

A SHORTENED STATUTORY PERIOD FOR REPLY IS SET TO EXPIRE 3 MONTH(S) FROM THE MAILING DATE OF THIS COMMUNICATION.

- Extensions of time may be available under the provisions of 37 CFR 1.136(a) In no event, however, may a reply be timely filed after SIX (6) MONTHS from the mailing date of this communication.
- If the period for reply specified above is less than thirty (30) days, a reply within the statutory minimum of thirty (30) days will be considered timely.
- If NO period for reply is specified above, the maximum statutory period will apply and will expire SIX (6) MONTHS from the mailing date of this communication.
- Failure to reply within the set or extended period for reply will, by statute, cause the application to become ABANDONED (35 U.S C. § 133)
- Any reply received by the Office later than three months after the mailing date of this communication, even if timely filed, may reduce any earned patent term adjustment. See 37 CFR 1.704(b).

Status

1)☒ Responsive to communication(s) filed on *18 April 2002* .

2a)☐ This action is **FINAL**. 2b)☒ This action is non-final.

3)☐ Since this application is in condition for allowance except for formal matters, prosecution as to the merits is closed in accordance with the practice under *Ex parte Quayle*, 1935 C.D. 11, 453 O.G. 213.

Disposition of Claims

4)☒ Claim(s) *1-59* is/are pending in the application.

4a) Of the above claim(s) _____ is/are withdrawn from consideration.

5)☒ Claim(s) *1-10* is/are allowed.

6)☒ Claim(s) *11-59* is/are rejected.

7)☐ Claim(s) _____ is/are objected to.

8)☐ Claim(s) _____ are subject to restriction and/or election requirement.

Application Papers

9)☐ The specification is objected to by the Examiner.

10)☐ The drawing(s) filed on _____ is/are: a)☐ accepted or b)☐ objected to by the Examiner.

Applicant may not request that any objection to the drawing(s) be held in abeyance. See 37 CFR 1.85(a).

11)☐ The proposed drawing correction filed on _____ is: a)☐ approved b)☐ disapproved by the Examiner.

If approved, corrected drawings are required in reply to this Office action.

12)☐ The oath or declaration is objected to by the Examiner.

Priority under 35 U.S.C. §§ 119 and 120

13)☐ Acknowledgment is made of a claim for foreign priority under 35 U.S.C. § 119(a)-(d) or (f).

a)☐ All b)☐ Some * c)☐ None of:

1.☐ Certified copies of the priority documents have been received.

2.☐ Certified copies of the priority documents have been received in Application No. _____ .

3.☐ Copies of the certified copies of the priority documents have been received in this National Stage application from the International Bureau (PCT Rule 17.2(a)).

* See the attached detailed Office action for a list of the certified copies not received.

14)☒ Acknowledgment is made of a claim for domestic priority under 35 U.S.C. § 119(e) (to a provisional application).

a) ☐ The translation of the foreign language provisional application has been received.

15)☐ Acknowledgment is made of a claim for domestic priority under 35 U.S.C. §§ 120 and/or 121.

Attachment(s)

1) ☒ Notice of References Cited (PTO-892)
2) ☐ Notice of Draftsperson's Patent Drawing Review (PTO-948)
3) ☒ Information Disclosure Statement(s) (PTO-1449) Paper No(s) *4,5* .
4) ☐ Interview Summary (PTO-413) Paper No(s). _____ .
5) ☐ Notice of Informal Patent Application (PTO-152)
6) ☐ Other: .

NOAC Ex. 1015 Page 211

1. Claims 1-59 are presenting for examination.

2. Claims 1-10 are allowed over the prior art of record.

2.1. The prior art of record taken singularly or in combination does not teach or suggest a packet monitor having a state patterns/operations memory configured to store a set of predefined state transition patters and state operations such that traversing a particular transition pattern as a result of a particular conversational flow-sequence of packets indicates that the particular conversational flow-sequence is associated with the operation of a particular application program and a state processor configured to carry out any state operations in the state patterns/operations memory for the protocol and state of the flow of the packet (claim 1).

3. The following is a quotation of the appropriate paragraphs of 35 U.S.C. 102 that form the basis for the rejections under this section made in this Office action:

> A person shall be entitled to a patent unless -
>
> (e) the invention was described in a patent granted on an application for patent by another filed in the United States before the invention thereof by the applicant for patent, or on an international application by another who has fulfilled the requirements of paragraphs (1), (2), and (4) of section 371© of this title before the invention thereof by the applicant for patent.

The changes made to 35 U.S.C. 102(e) by the American Inventors Protection Act of 1999 (AIPA) do not apply to the examination of this application as the application being examined was not (1) filed on or after November 29, 2000, or (2) voluntarily published under 35 U.S.C. 122(b). Therefore, this application is examined under 35 U.S.C. 102(e) prior to the amendment by the AIPA (pre-AIPA 35 U.S.C. 102(e)).

4. Claims 11-59 are rejected under 35 U.S.C. 102(e) as being anticipated by Muller et al. (US Pat. No. 6,483,804).

5. As to claims 11-12, Muller shows in Fig 1A, a method of examining packets through a connection point (the point connects the network to the NIC of the circuit 100).

Muller discloses the following steps:

* receiving a packet from a packet acquisition device (NIC), see col 6, lines 26-29, lines 54-60, col 8, lines 33-35;

* performing one or more parsing/extraction operations to create a record comprising a function of selected portions of the packet, see col 7, lines 31-44, col 8, lines 50-67, col 9, lines 1-5;

* looking up a flow-entry database 110 to determine if the packet is of an existing flow, see col 9, lines 18-24, col 11, lines 32-45 ;

* if the packet is of an existing flow, classifying the packet as belonging to the found existing flow, see col 11, lines 46-52; and

* if the packet is of a new flow, storing a new flow-entry in the flow-entry database 110, see col 11, lines 46-52.

6. As to claims 13-15, Muller teaches updating the flow-entry of the existing flow including measures selected from the set consisting of the total packet count, see col 7, lines 36-45, col 8, lines 50-54, lines 64-66.

7. As to claim 16, Muller shows that the function of the selected portions of the packet forms a signature (flow key), see col 8, lines 64-67, col 9, lines 1-5, col 11, lines 35-37.

8. As to claims 17-20, Muller shows at least one of the protocols uses source and destination addresses, see col 7, lines 31-40.

9. As to claim 21, Muller shows the looking up of the flow-entry database 110 uses a hash of the selected packet portions, see col 9, lines 18-22.

10. As to claim 22, Muller shows determining a set of one or more protocol from data in the packet, see col 10, lines 63-67, col 11, lines 27-30.

11. As to claim 23, Muller shows obtaining the last encountered state of the existing flow and performing any state operations required for a new flow, see col 9, lines 15-28.

12. As to claim 24, Muller shows identifying of the application program of the flow, see col 8, lines 60-61, col 12, lines 45-47.

13. As to claim 25, Muller shows storing identifying information for future packets, see col 9, lines 26-28.

14. As to claim 26, Muller shows identifying the application program of the flow, see col 8, lines 60-61, col 12, lines 45-47.

15. As to claim 27, Muller shows searching the parser record for the existence of one or more reference strings, see col 9, lines 32-36.

16. As to claim 28, Muller shows the state operations are carried by state processor , see col 9, lines 42-47, col 10, lines 61-63

17. As to claim 29-59, the claims are similar in scope to claims 11-28, and they are rejected under the same rationale.

Therefore, it can be seen from paragraphs 5-17 that Muller anticipates claims 11-59.

18. The prior art made of record and not relied upon is considered pertinent to applicant's disclosure.

19. Any inquiry concerning this communication or earlier communications from the examiner should be directed to Moustafa M. Meky whose telephone number is (703) 305-9697. The examiner can normally be reached on week days from 8:30 am to 4:30 pm.

If attempts to reach the examiner by telephone are unsuccessful, the examiner's supervisor, Ario Etienne, can be reached on (703) 308-7562. The fax phone number for this Group is (703) 308-9052.

Any inquiry of a general nature or relating to the status of this application or proceeding should be directed to the Group receptionist whose telephone number is (703) 305-9600. The fax number for the After-Final correspondence/amendment is (703) 746-7238. The fax number for official correspondence/amendment is (703) 746-7239. The fax number for Non-official draft correspondence/amendment is (703) 746-7240.

M.M.M

June 22, 2003

Moustafa M. Meky

Notice of References Cited	Application/Control No. 09/608,237	Applicant(s)/Patent Under Reexamination DIETZ ET AL.	
	Examiner Moustafa M Meky	Art Unit 2157	Page 1 of 1

U.S. PATENT DOCUMENTS

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Name	Classification
	A	US-6,483,804	11-2002	Muller et al.	370/230
	B	US-6,570,875	05-2003	Hegde	370/389
	C	US-6,452,915	09-2002	Jorgensen	370/338
	D	US-6,466,985	10-2002	Goyal et al.	709/238
	E	US-6,453,360	09-2002	Muller et al.	709/250
	F	US-6,243,667	06-2001	Kerr et al.	703/27
	G	US-6,118,760	09-2000	Zaumen et al.	370/229
	H	US-			
	I	US-			
	J	US-			
	K	US-			
	L	US-			
	M	US-			

FOREIGN PATENT DOCUMENTS

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Country	Name	Classification
	N					
	O					
	P					
	Q					
	R					
	S					
	T					

NON-PATENT DOCUMENTS

*		Include as applicable: Author, Title Date, Publisher, Edition or Volume, Pertinent Pages)
	U	
	V	
	W	
	X	

*A copy of this reference is not being furnished with this Office action. (See MPEP § 707.05(a).)
Dates in MM-YYYY format are publication dates. Classifications may be US or foreign.

US006483804B1

(12) **United States Patent**
Muller et al.

(10) **Patent No.: US 6,483,804 B1**
(45) **Date of Patent: Nov. 19, 2002**

(54) **METHOD AND APPARATUS FOR DYNAMIC PACKET BATCHING WITH A HIGH PERFORMANCE NETWORK INTERFACE**

(75) Inventors: **Shimon Muller**, Sunnyvale, CA (US); **Denton E. Gentry, Jr.**, Fremont, CA (US)

(73) Assignee: **Sun Microsystems, Inc.**, Santa Clara, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/260,324**

(22) Filed: **Mar. 1, 1999**

(51) **Int. Cl.**[7] **H04J 1/16**

(52) **U.S. Cl.** **370/230**; 370/235; 709/225; 709/228

(58) **Field of Search** 370/230, 231, 370/235, 392, 389, 225, 226, 241, 401, 428, 427, 473, 474, 394, 252, 466, 409; 709/225, 226, 235, 241, 228

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,414,704	A	5/1995	Spinney	370/60
5,583,940	A	12/1996	Vidrascu et al.	380/49
5,684,954	A	11/1997	Kaiserswerth et al.	395/200.2
5,748,905	A	5/1998	Hauser et al.	395/200.79
5,758,089	A	5/1998	Gentry et al.	395/200.64
5,778,180	A	7/1998	Gentry et al.	395/200.42
5,778,414	A	7/1998	Winter et al.	711/5
5,787,255	A	7/1998	Parlan et al.	395/200.63
5,793,954	A	8/1998	Baker et al.	395/200.8
5,870,394	A	2/1999	Oprea	370/392
5,920,705	A *	7/1999	Lyon et al.	370/409
6,157,955	A *	12/2000	Narad et al.	709/228

FOREIGN PATENT DOCUMENTS

EP	0 447 725	9/1991	G06F/15/16
EP	0 573 739	12/1993	H04L/12/56
EP	0 853 411	7/1998	H04L/29/06
EP	0 865 180	9/1998	H04L/12/56
WO	WO 95/14269	5/1995	G06F/7/08
WO	WO 97/28505	8/1997	G06F/13/14
WO	WO 99/00737	1/1999	G06F/13/00
WO	WO99/00945	1/1999	H04L/12/46
WO	WO99/00948	1/1999	H04L/12/56
WO	WO 99/00949	1/1999	H04L/12/56

OTHER PUBLICATIONS

Toong Shoon Chan, et al., "Parallel Architecture Support for High-Speed Protocol Processing," Feb. 1, 1997, *Microprocessors And Microsystems*, vol. 20, No. 6, pp. 325–339.

(List continued on next page.)

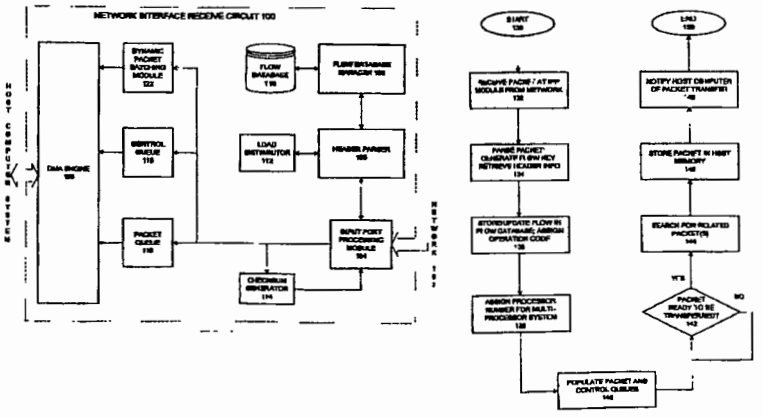
Primary Examiner—Wellington Chin
Assistant Examiner—William Schultz
(74) *Attorney, Agent, or Firm*—Park, Vaughan & Fleming LLP

(57) **ABSTRACT**

A system and method are provided for identifying related packets in a communication flow for the purpose of collectively processing them through a protocol stack comprising one or more protocols under which the packets were transmitted. A packet received at a network interface is parsed to retrieve information from one or more protocol headers. A flow key is generated to identify a communication flow that includes the packet, and is stored in a database of flow keys. When the packet is placed in a queue to be transferred to a host computer, the flow key and/or its flow number (e.g., its index into the database) is stored in a separate queue. Near to the time at which the packet is transferred to the host computer, a dynamic packet batching module searches for a packet that is related to the packet being transferred (i.e., is in the same flow) but which will be transferred later in time. If a related packet is located, the host computer is alerted and, as a result, delays processing the transferred packet until the related packet is also received. By collectively processing the related packets, processor time is more efficiently utilized.

identify flow of the packet

27 Claims, 49 Drawing Sheets



OTHER PUBLICATIONS

Peter Newman, et al., "IP Switching and Gigabit Routers," *IEEE Communications Magazine*, vol. 335, No. 1, Jan. 1997, pp. 64–69.

Francois Le Faucheur, "IETF Multiprotocol Label Switching (MPLS) Architecture," *IEEE International Conference*, Jun. 22, 1998, pp. 6–15.

F. Hallsall, "Data Communications, Computer Networks and Open Systems," *Electronic Systems Engineering Series*, pp. 451–452.

R. Cole, et al., "IP Over ATM: A Framework Document," *IETF Online*, Apr. 1996, pp. 1–31.

Sally Floyd & Van Jacobson, *Random Early Detection Gateways for Congestion Avoidance*, Aug., 1993, IEEE/ACM Transactions on Networking.

U.S. patent application Ser. No. 08/893,862, entitled "Mechanism for Reducing Interrupt Overhead in Device Drivers," filed Jul. 11, 1997, inventor Denton Gentry.

Pending U.S. patent application Ser. No. 09/259,445, entitled "Method and Apparatus for Distributing Network Processing on a Multiprocessor Computer," by Shimon Muller et al., filed Mar. 1, 1999 (Attorney Docket SUN–P3481–JTF).

Pending U.S. patent application Ser. No. 09/260,367, entitled "Method and Apparatus for Suppressing Interrupts in a High–Speed Network Environment," by Denton Gentry, filed Mar. 1, 1999 (Attorney Docket SUN–P3482–JTF).

*Pending U.S. patent application Ser. No. 09/259,736*entitled "Method and Apparatus for Modulating Interrupts in a Network Interface," by Denton Gentry et al., filed Mar. 1, 1999 (Attorney Docket SUN–P3483–JTF).

Pending U.S. patent application Ser. No. 09/259,765, entitled "A High Performance Network Interface," by Shimon Muller et al., filed Mar. 1, 1999 (Attorney Docket SUN–P3485–JTF).

Pending U.S. patent application Ser. No. 09/260,618, entitled "Method and Apparatus for Classifying Network Traffic in a High Performance Network INterface," by Shimon Muller et al., filed Mar. 1, 1999 (Attorney Docket SUN–P3486–JTF).

Pending U.S. patent application Ser. No. 09/259,932, entitled "Method and Apparatus for Managing a Network Flow in a High Performance Network Interface," by Shimon Muller et al., filed Mar. 1, 1999 (Attorney Docket SUN–P3487–JTF).

Pending U.S. patent application Ser. No. 09/258,952, entitled "Method and Apparatus for Early Random Discard of Packets," by Shimon Muller et al., filed Mar. 1, 1999 (Attorney Docket SUN–P3490–JTF).

Pending U.S. patent application Ser. No. 09.260,333, entitled "Method and Apparatus for Data Re–Assembly with a High Performance Network Interface," by Shimon Muller et al., filed Mar. 1, 1999 (Attorney Docket SUN–P3507–JTF).

Pending U.S. patent application Ser. No. 09/258,955, entitled "Dynamic Parsing in a High Performance Network Interface," by Denton Gentry, filed Mar. 1, 1999 (Attorney Docket SUN–P3715–JTF).

Pending U.S. patent application Ser. No. 09/259,936, entitled "Method and Apparatus for Indicating an Interrupt in a Network Interface," by Denton Gentry et al., filed Mar. 1, 1999 (Attorney Docket SUN–P3814–JTF).

* cited by examiner

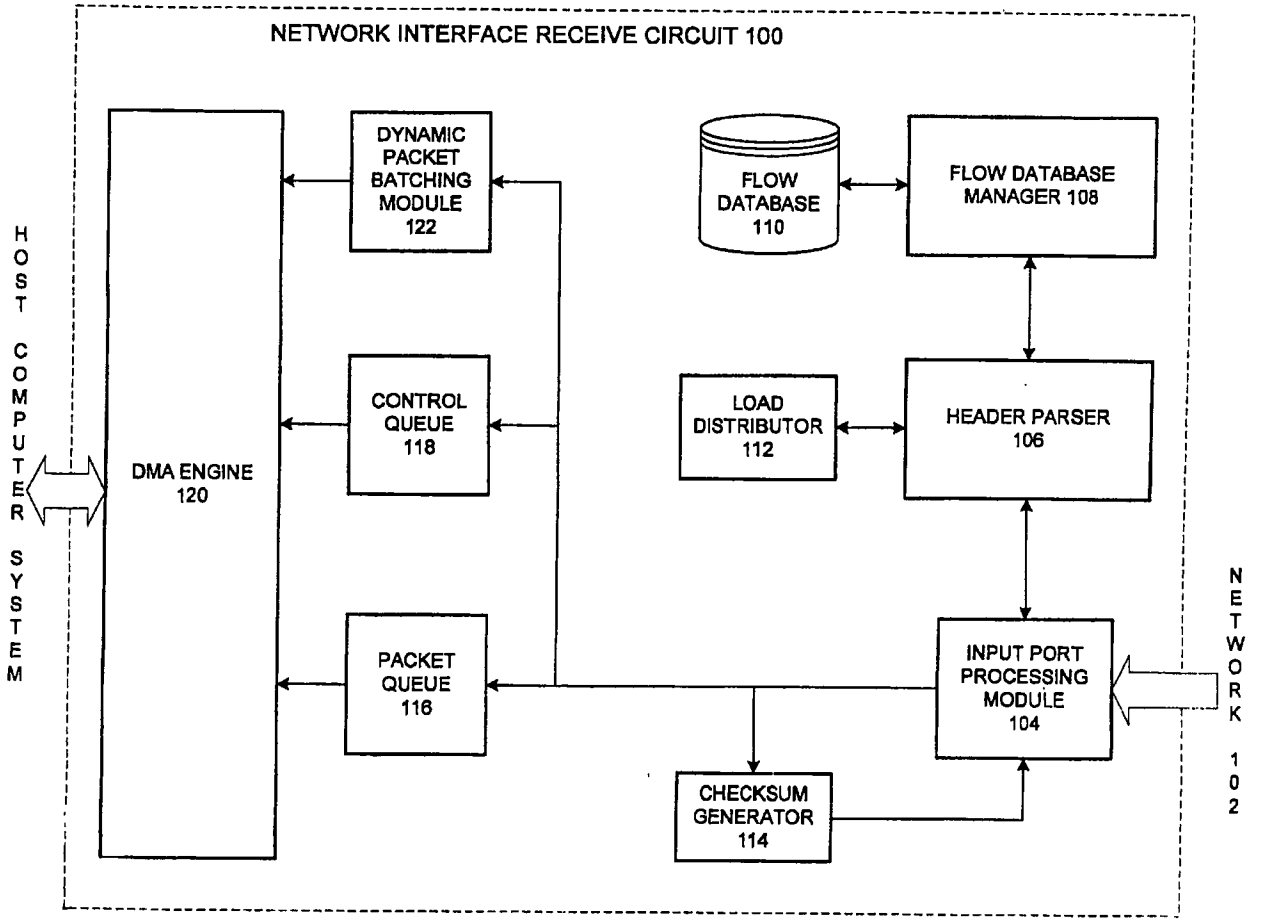

NETWORK INTERFACE RECEIVE CIRCUIT 100
HOST COMPUTER SYSTEM
DMA ENGINE 120
DYNAMIC PACKET BATCHING MODULE 122
CONTROL QUEUE 118
PACKET QUEUE 116
FLOW DATABASE 110
FLOW DATABASE MANAGER 108
LOAD DISTRIBUTOR 112
HEADER PARSER 106
INPUT PORT PROCESSING MODULE 104
CHECKSUM GENERATOR 114
NETWORK 102

FIG. 1A

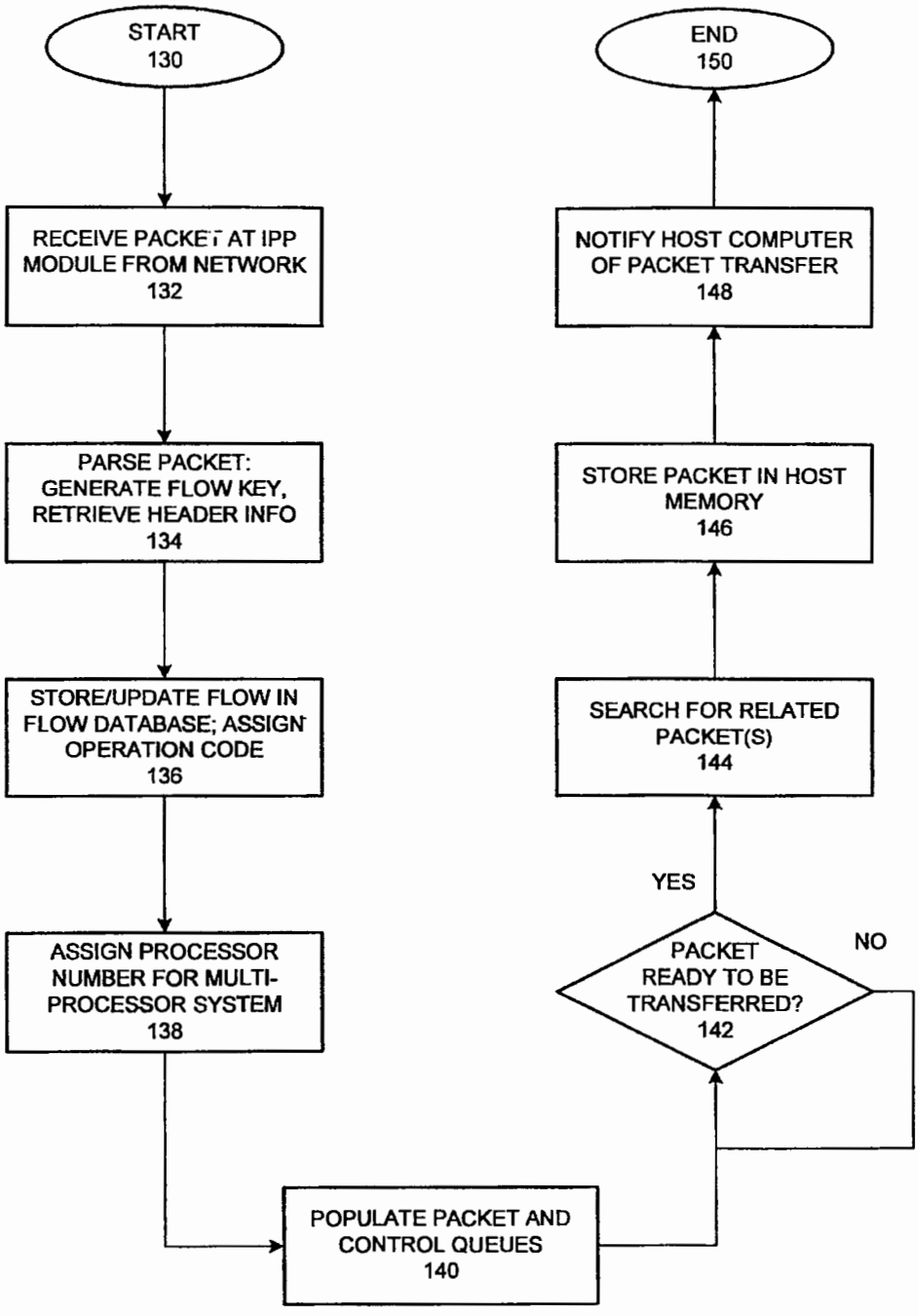

START
130
RECEIVE PACKET AT IPP MODULE FROM NETWORK
132
PARSE PACKET: GENERATE FLOW KEY, RETRIEVE HEADER INFO
134
STORE/UPDATE FLOW IN FLOW DATABASE; ASSIGN OPERATION CODE
136
ASSIGN PROCESSOR NUMBER FOR MULTI-PROCESSOR SYSTEM
138
POPULATE PACKET AND CONTROL QUEUES
140
PACKET READY TO BE TRANSFERRED?
142
YES
NO
SEARCH FOR RELATED PACKET(S)
144
STORE PACKET IN HOST MEMORY
146
NOTIFY HOST COMPUTER OF PACKET TRANSFER
148
END
150

FIG. 1B

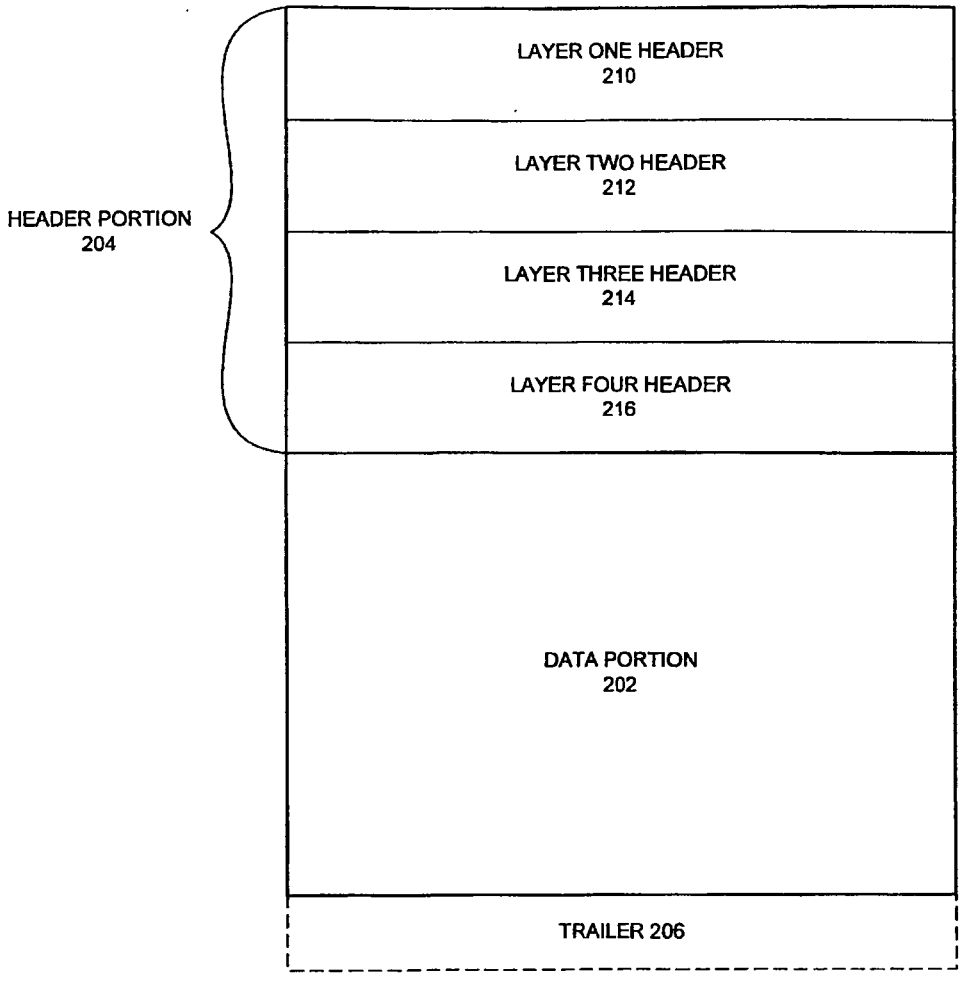

LAYER ONE HEADER
210
LAYER TWO HEADER
212
LAYER THREE HEADER
214
LAYER FOUR HEADER
216
HEADER PORTION
204
DATA PORTION
202
TRAILER 206
PACKET 200

FIG. 2

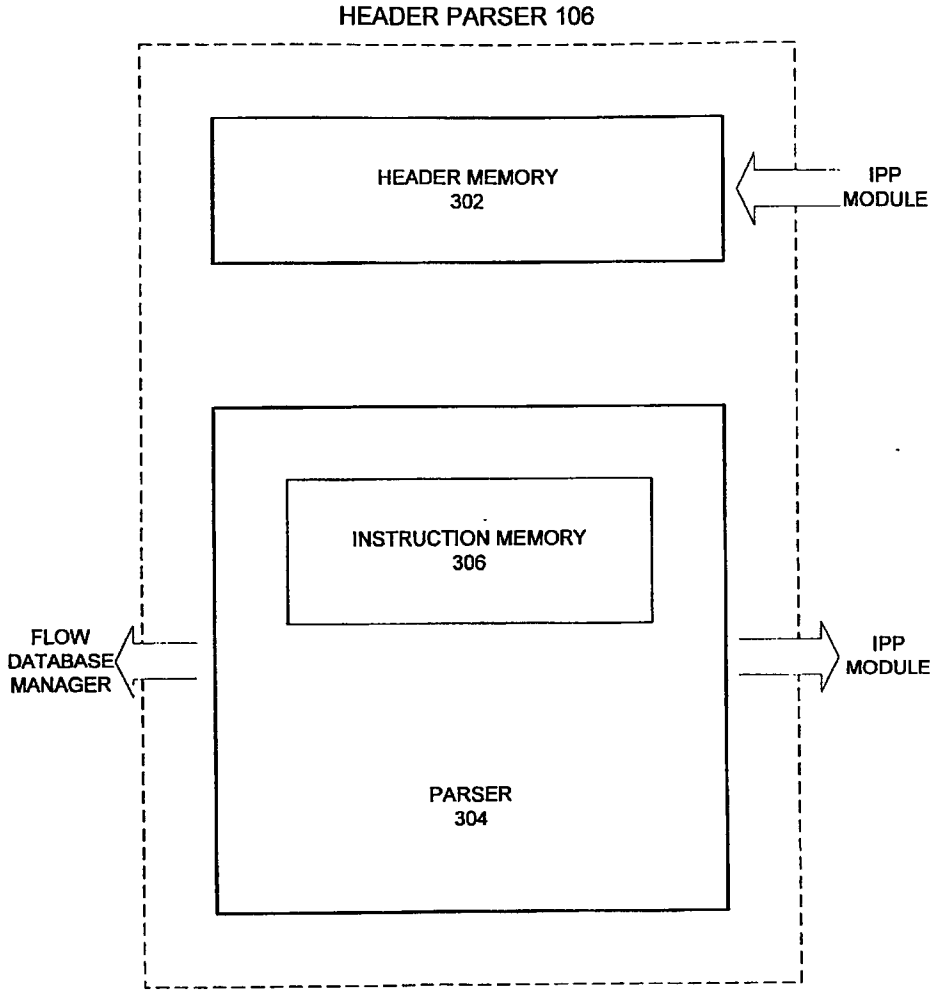

HEADER PARSER 106
HEADER MEMORY
302
IPP
MODULE
INSTRUCTION MEMORY
306
FLOW
DATABASE
MANAGER
IPP
MODULE
PARSER
304

FIG. 3

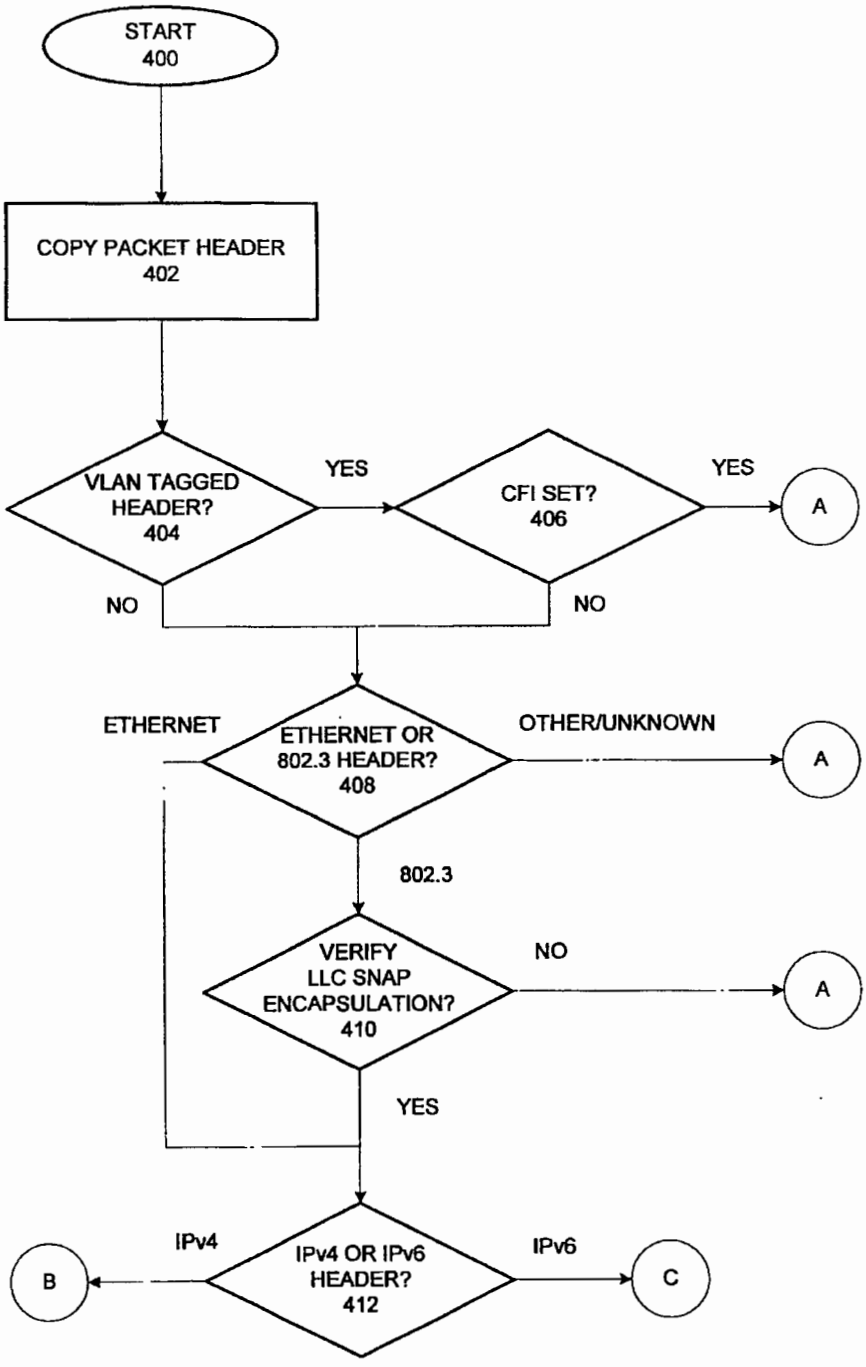

START
400
COPY PACKET HEADER
402
VLAN TAGGED HEADER?
404
YES
CFI SET?
406
YES
A
NO
NO
ETHERNET
ETHERNET OR 802.3 HEADER?
408
OTHER/UNKNOWN
A
802.3
VERIFY LLC SNAP ENCAPSULATION?
410
NO
A
YES
IPv4
IPv4 OR IPv6 HEADER?
412
IPv6
B
C

FIG. 4A

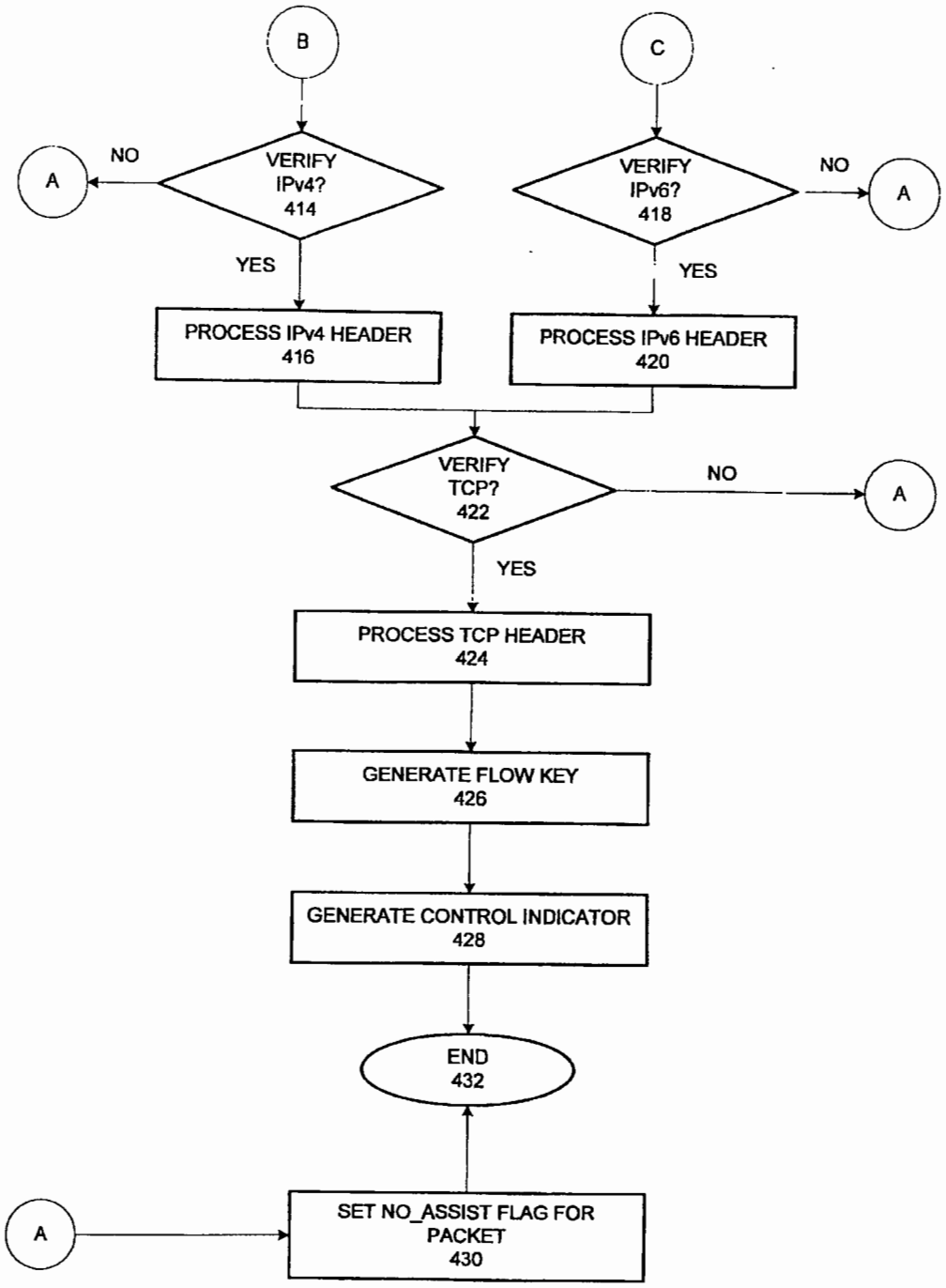

B
C
NO
A
VERIFY IPv4? 414
VERIFY IPv6? 418
NO
A
YES
YES
PROCESS IPv4 HEADER 416
PROCESS IPv6 HEADER 420
VERIFY TCP? 422
NO
A
YES
PROCESS TCP HEADER 424
GENERATE FLOW KEY 426
GENERATE CONTROL INDICATOR 428
END 432
A
SET NO_ASSIST FLAG FOR PACKET 430

FIG. 4B

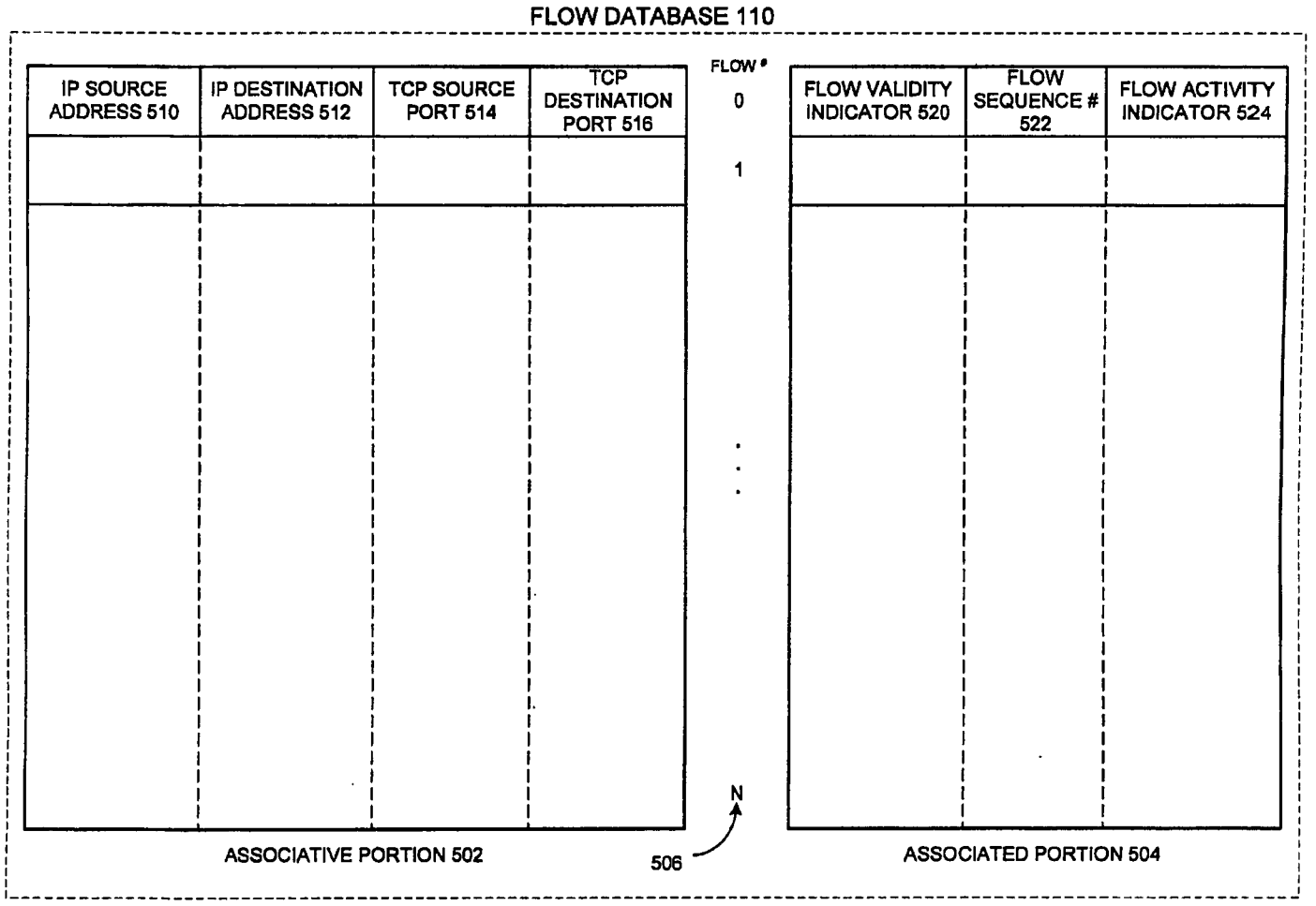

FLOW DATABASE 110
IP SOURCE ADDRESS 510
IP DESTINATION ADDRESS 512
TCP SOURCE PORT 514
TCP DESTINATION PORT 516
FLOW #
0
1
N
506
FLOW VALIDITY INDICATOR 520
FLOW SEQUENCE # 522
FLOW ACTIVITY INDICATOR 524
ASSOCIATIVE PORTION 502
ASSOCIATED PORTION 504

FIG. 5

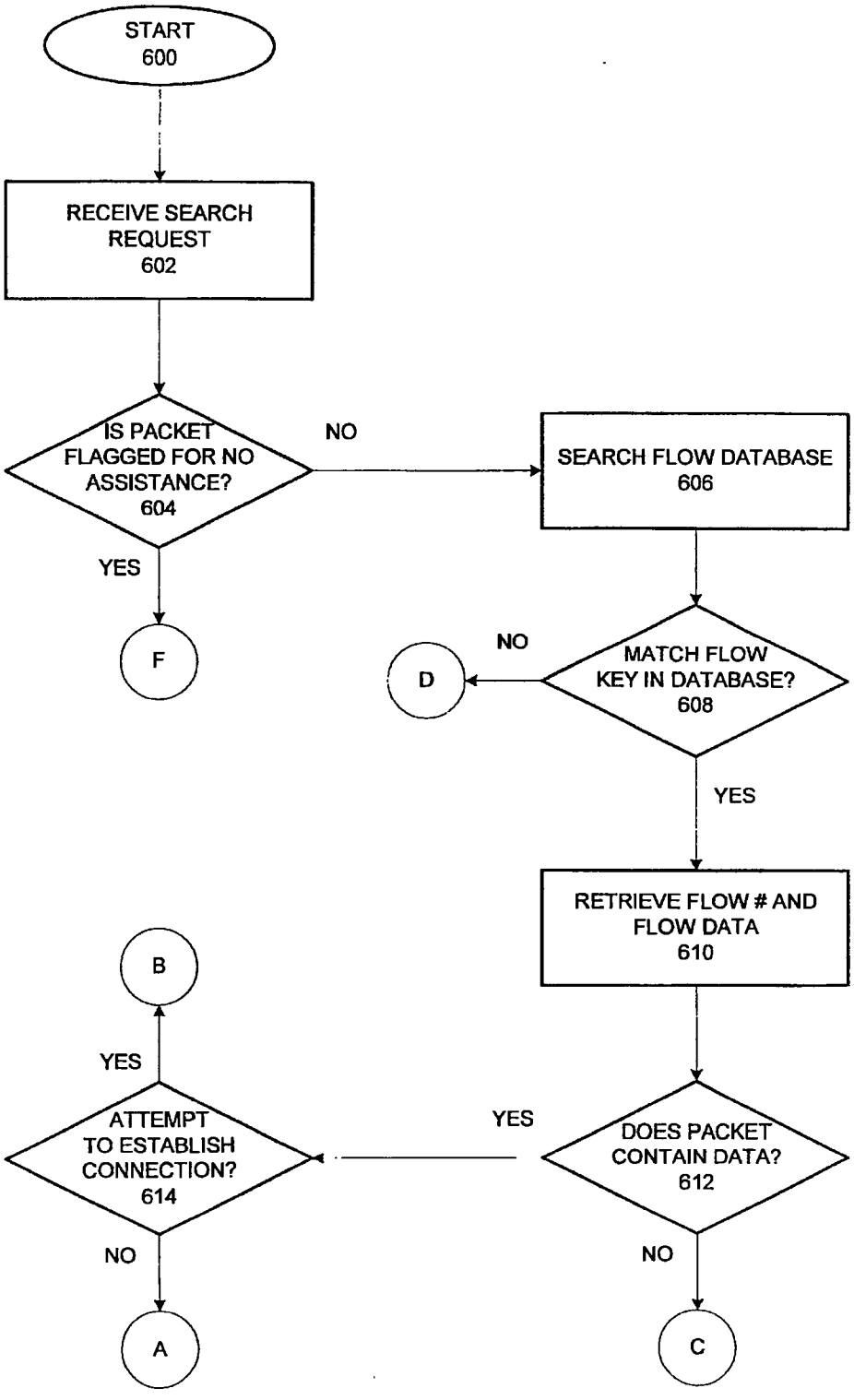

START
600
RECEIVE SEARCH REQUEST
602
IS PACKET FLAGGED FOR NO ASSISTANCE?
604
NO
SEARCH FLOW DATABASE
606
YES
F
NO
D
MATCH FLOW KEY IN DATABASE?
608
YES
RETRIEVE FLOW # AND FLOW DATA
610
B
YES
ATTEMPT TO ESTABLISH CONNECTION?
614
YES
DOES PACKET CONTAIN DATA?
612
NO
NO
A
C

FIG. 6A

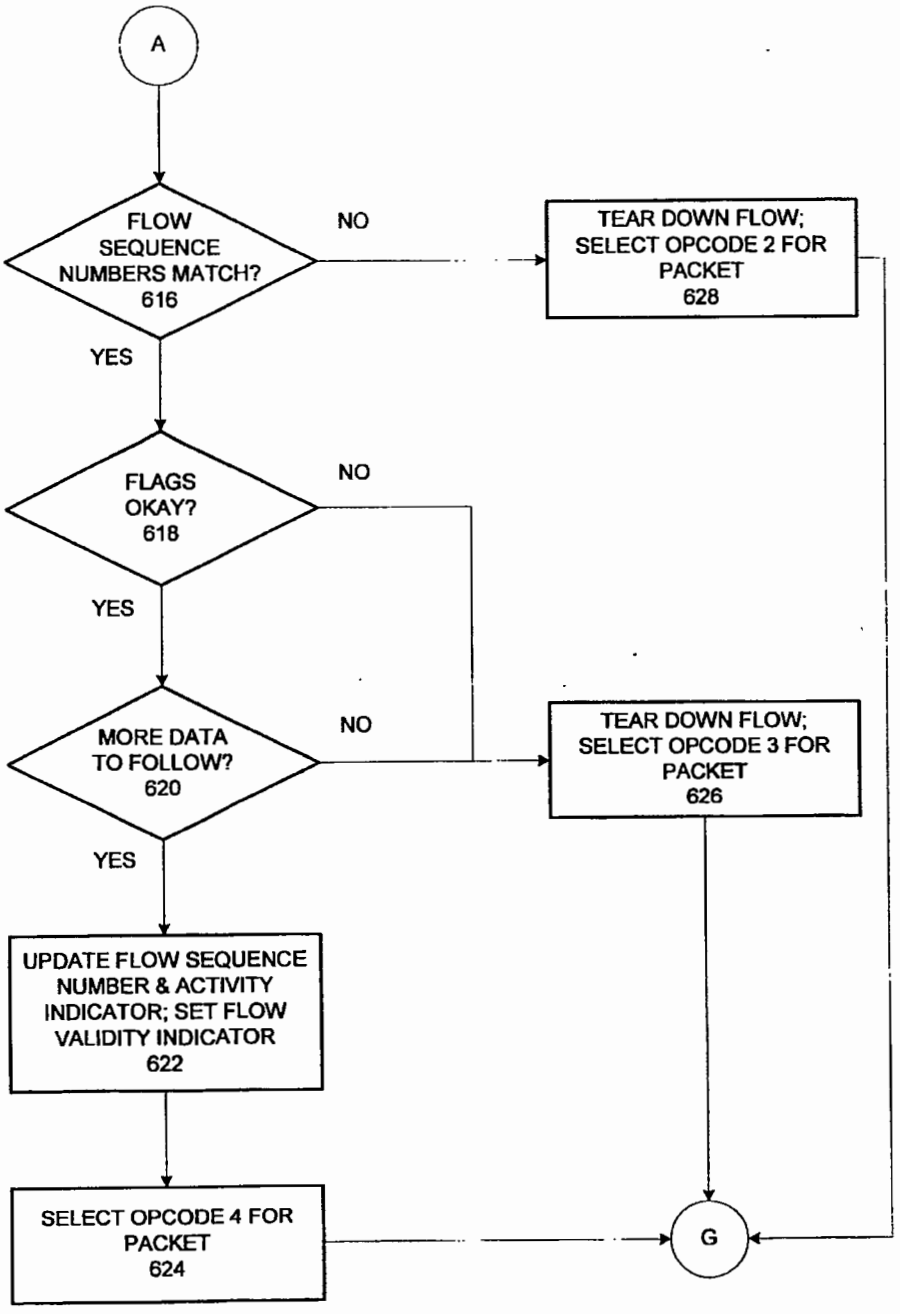

A
FLOW SEQUENCE NUMBERS MATCH? 616
NO
TEAR DOWN FLOW; SELECT OPCODE 2 FOR PACKET 628
YES
FLAGS OKAY? 618
NO
YES
MORE DATA TO FOLLOW? 620
NO
TEAR DOWN FLOW; SELECT OPCODE 3 FOR PACKET 626
YES
UPDATE FLOW SEQUENCE NUMBER & ACTIVITY INDICATOR; SET FLOW VALIDITY INDICATOR 622
SELECT OPCODE 4 FOR PACKET 624
G

FIG. 6B

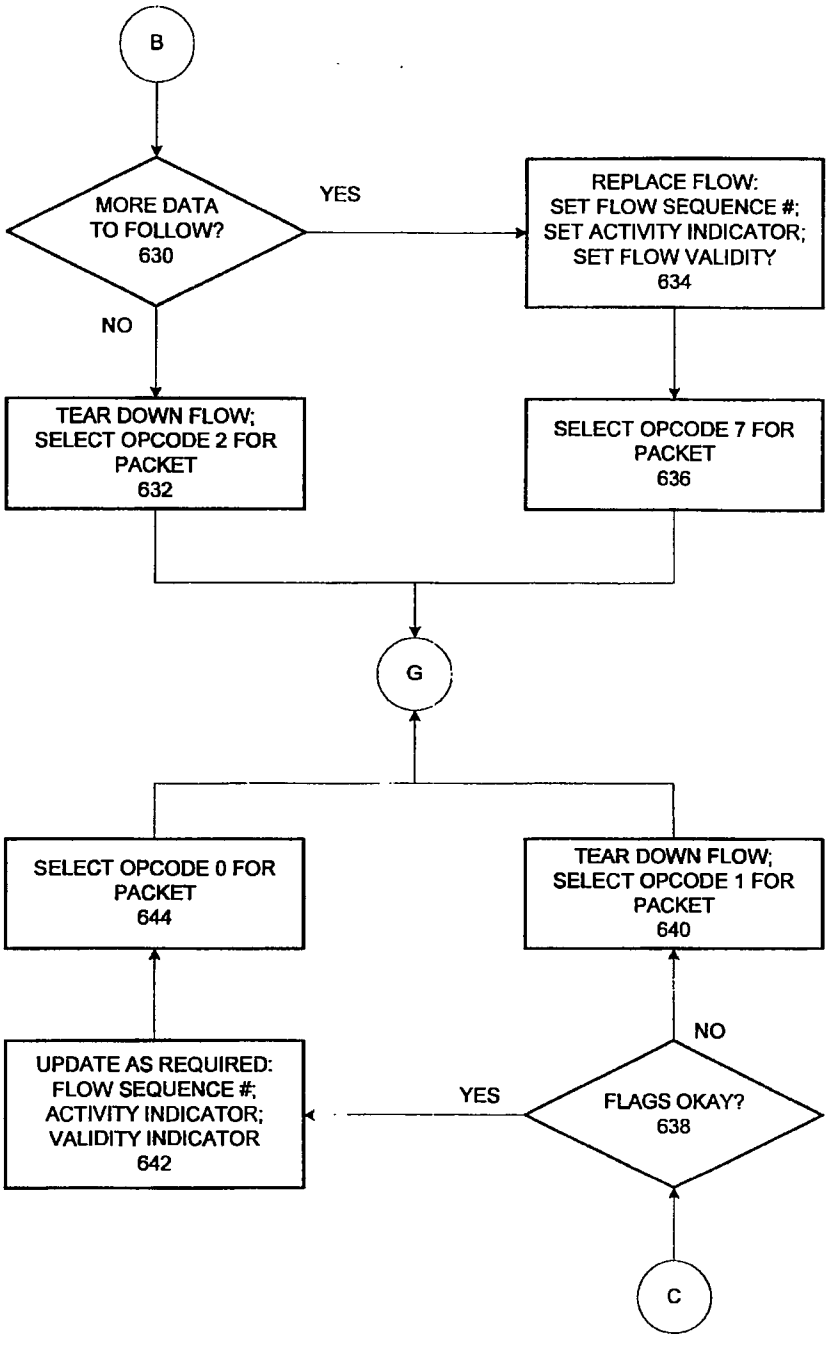

B
MORE DATA TO FOLLOW? 630
YES
NO
REPLACE FLOW: SET FLOW SEQUENCE #; SET ACTIVITY INDICATOR; SET FLOW VALIDITY 634
TEAR DOWN FLOW; SELECT OPCODE 2 FOR PACKET 632
SELECT OPCODE 7 FOR PACKET 636
G
SELECT OPCODE 0 FOR PACKET 644
TEAR DOWN FLOW; SELECT OPCODE 1 FOR PACKET 640
UPDATE AS REQUIRED: FLOW SEQUENCE #; ACTIVITY INDICATOR; VALIDITY INDICATOR 642
YES
NO
FLAGS OKAY? 638
C

FIG. 6C

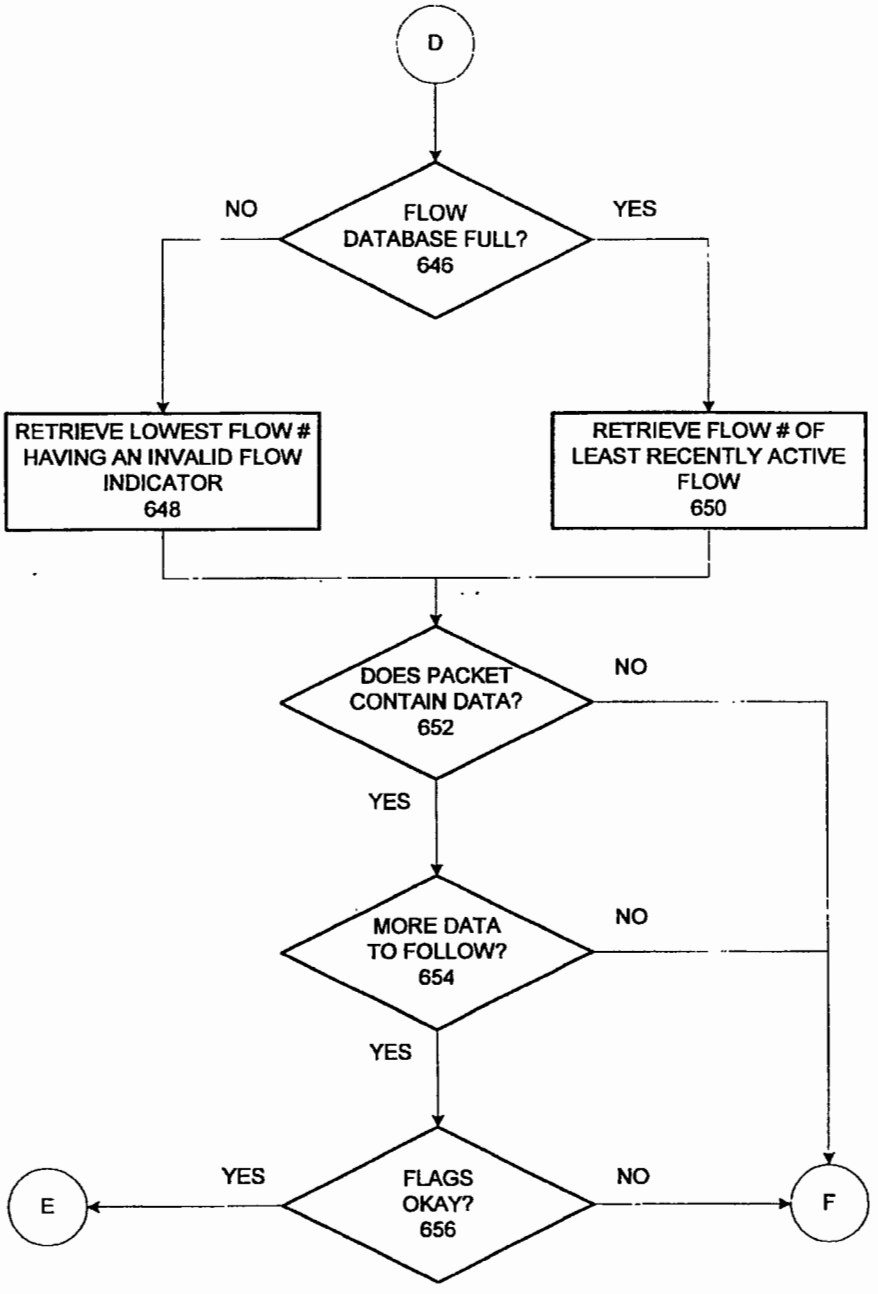

D
FLOW DATABASE FULL? 646
NO
YES
RETRIEVE LOWEST FLOW # HAVING AN INVALID FLOW INDICATOR 648
RETRIEVE FLOW # OF LEAST RECENTLY ACTIVE FLOW 650
DOES PACKET CONTAIN DATA? 652
NO
YES
MORE DATA TO FOLLOW? 654
NO
YES
FLAGS OKAY? 656
YES
NO
E
F

FIG. 6D

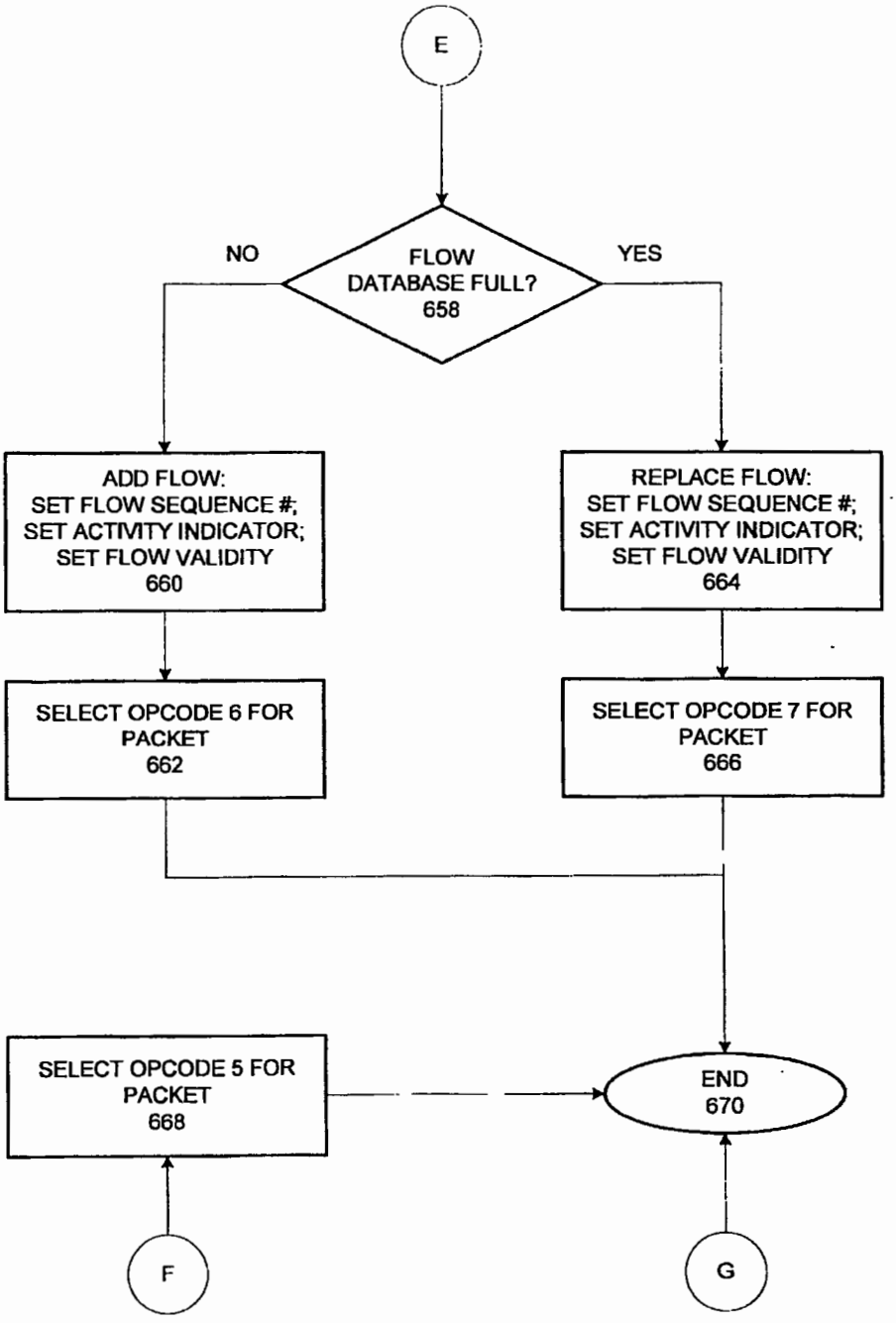

E
FLOW DATABASE FULL? 658
NO
YES
ADD FLOW: SET FLOW SEQUENCE #; SET ACTIVITY INDICATOR; SET FLOW VALIDITY 660
REPLACE FLOW: SET FLOW SEQUENCE #; SET ACTIVITY INDICATOR; SET FLOW VALIDITY 664
SELECT OPCODE 6 FOR PACKET 662
SELECT OPCODE 7 FOR PACKET 666
SELECT OPCODE 5 FOR PACKET 668
END 670
F
G

FIG. 6E

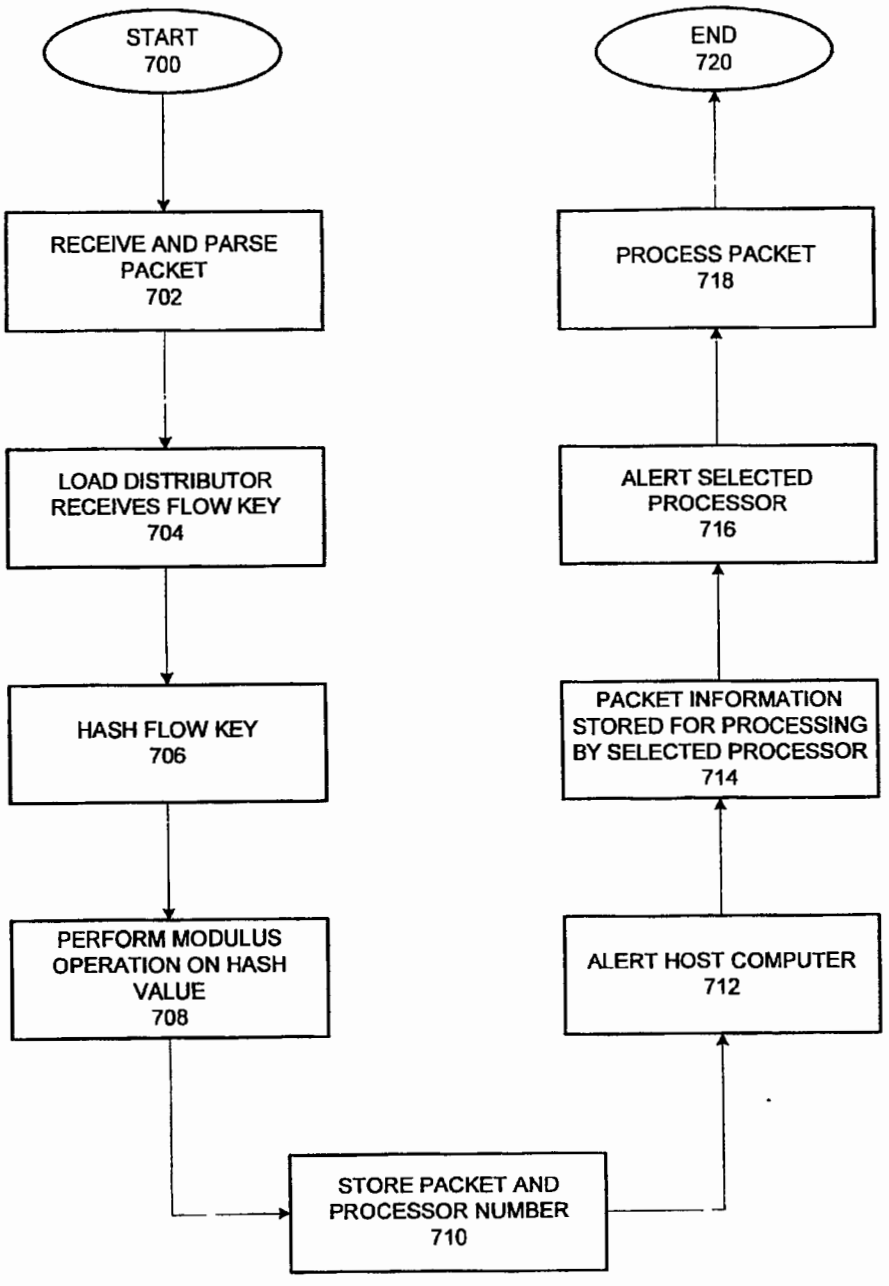

START
700
RECEIVE AND PARSE PACKET
702
LOAD DISTRIBUTOR RECEIVES FLOW KEY
704
HASH FLOW KEY
706
PERFORM MODULUS OPERATION ON HASH VALUE
708
STORE PACKET AND PROCESSOR NUMBER
710
ALERT HOST COMPUTER
712
PACKET INFORMATION STORED FOR PROCESSING BY SELECTED PROCESSOR
714
ALERT SELECTED PROCESSOR
716
PROCESS PACKET
718
END
720

FIG. 7

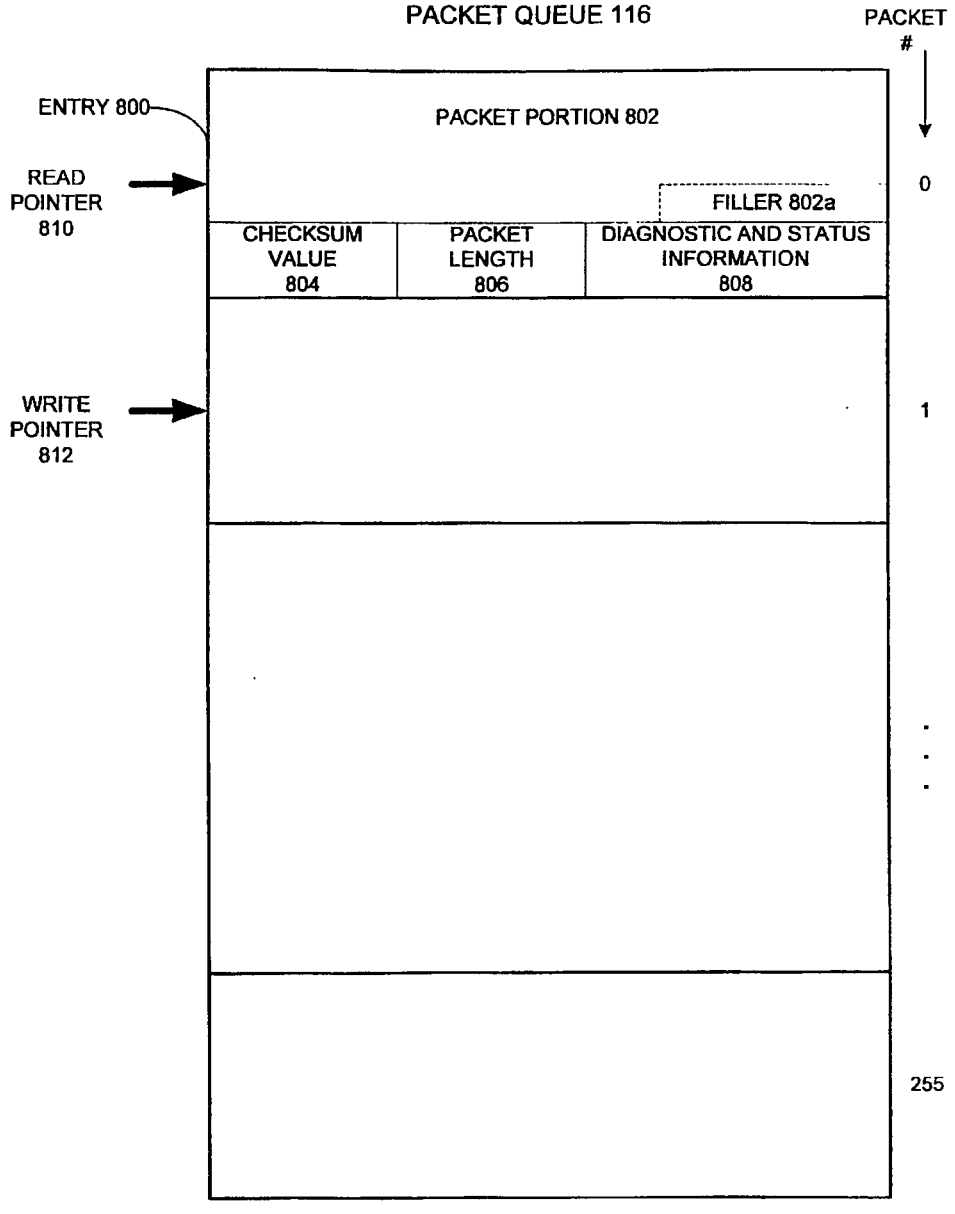

PACKET QUEUE 116
PACKET #
ENTRY 800
PACKET PORTION 802
READ POINTER 810
FILLER 802a
0
CHECKSUM VALUE 804
PACKET LENGTH 806
DIAGNOSTIC AND STATUS INFORMATION 808
WRITE POINTER 812
1
255

FIG. 8

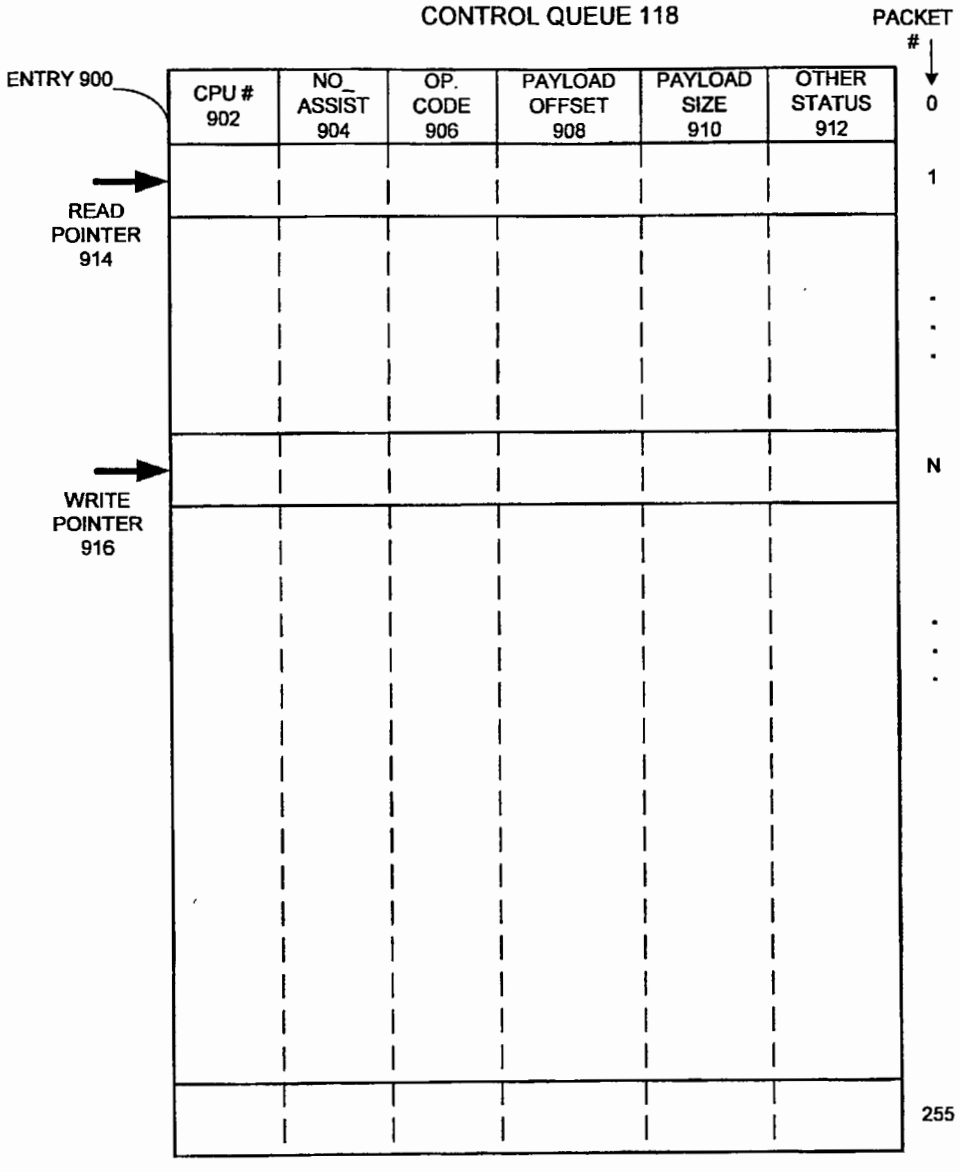

CONTROL QUEUE 118
PACKET #
ENTRY 900
CPU # 902
NO_ ASSIST 904
OP. CODE 906
PAYLOAD OFFSET 908
PAYLOAD SIZE 910
OTHER STATUS 912
0
1
READ POINTER 914
N
WRITE POINTER 916
255

FIG. 9

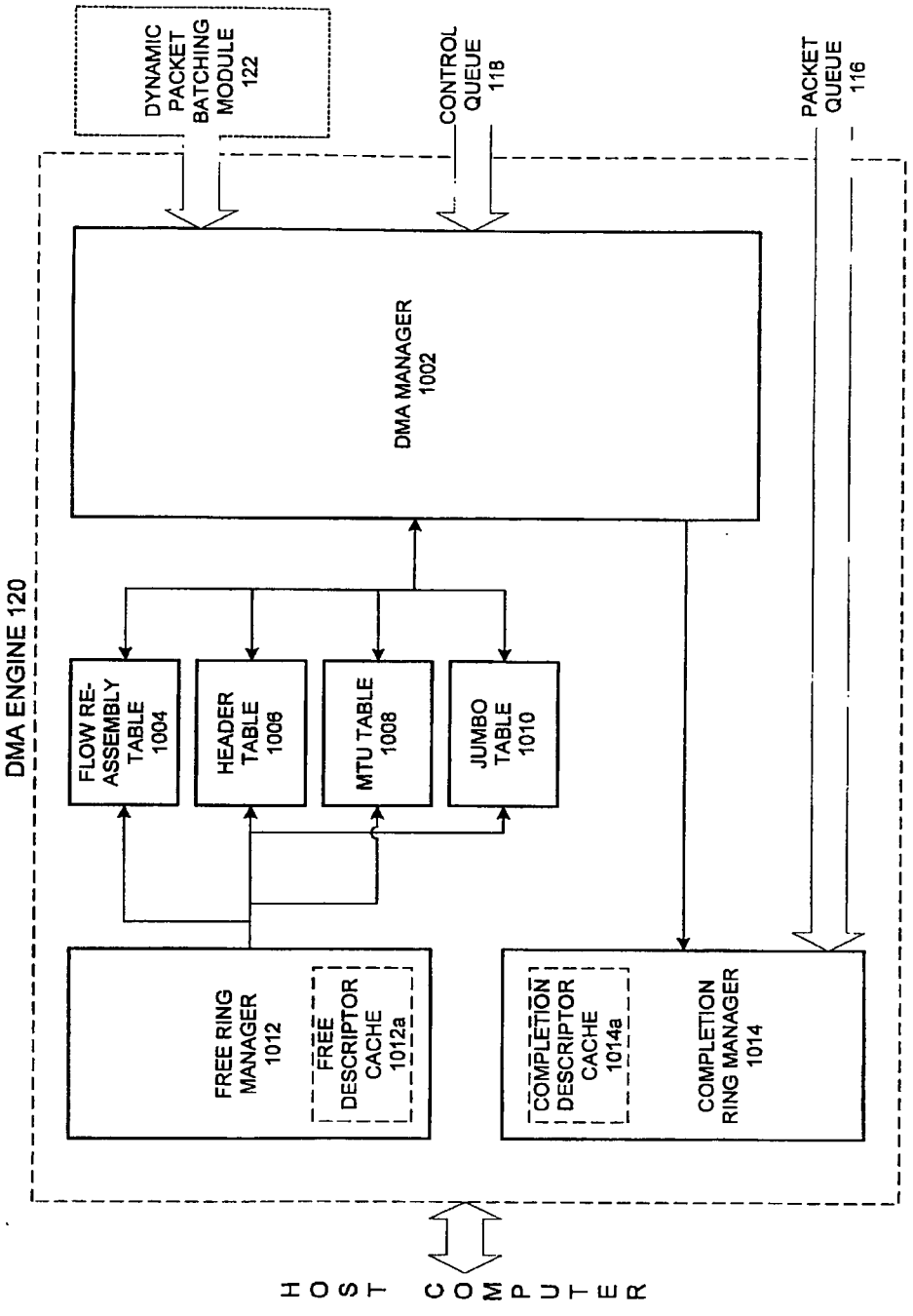

DYNAMIC PACKET BATCHING MODULE 122
CONTROL QUEUE 118
PACKET QUEUE 116
DMA MANAGER 1002
DMA ENGINE 120
FLOW RE-ASSEMBLY TABLE 1004
HEADER TABLE 1006
MTU TABLE 1008
JUMBO TABLE 1010
FREE RING MANAGER 1012
FREE DESCRIPTOR CACHE 1012a
COMPLETION DESCRIPTOR CACHE 1014a
COMPLETION RING MANAGER 1014
HOST COMPUTER

FIG. 10

FLOW RE-ASSEMBLY TABLE 1004

VALIDITY INDICATOR 1106	NEXT ADDRESS 1104	FLOW RE-ASSEMBLY BUFFER INDEX 1102

PACKET # ↓ 0 . . . 63

HEADER TABLE 1006

VALIDITY INDICATOR 1116	NEXT ADDRESS 1114	HEADER BUFFER INDEX 1112

MTU TABLE 1008

VALIDITY INDICATOR 1126	NEXT ADDRESS 1124	MTU BUFFER INDEX 1122

JUMBO TABLE 1010

VALIDITY INDICATOR 1136	NEXT ADDRESS 1134	JUMBO BUFFER INDEX 1132

FIG. 11

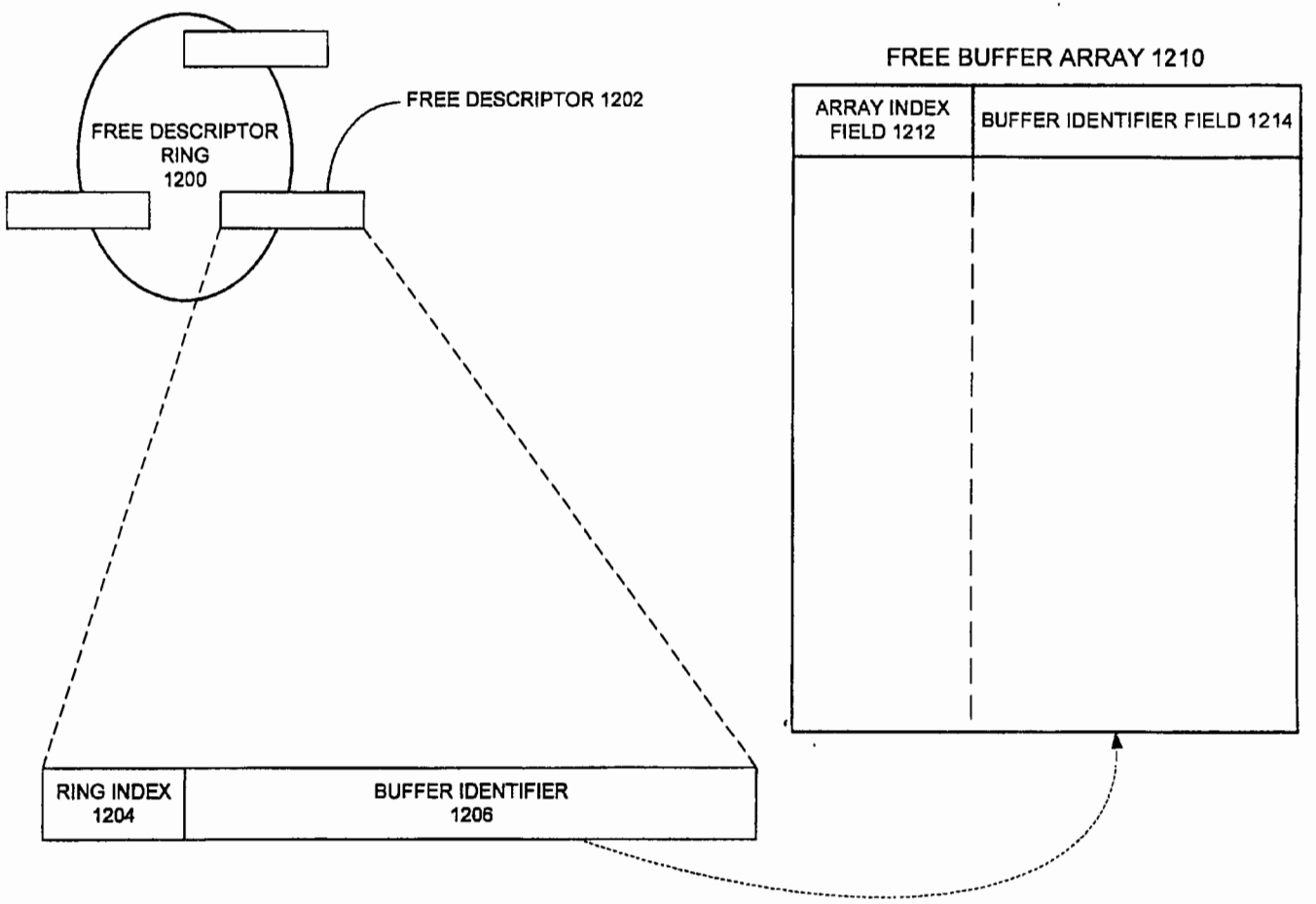

FREE DESCRIPTOR RING 1200
FREE DESCRIPTOR 1202
FREE BUFFER ARRAY 1210
ARRAY INDEX FIELD 1212
BUFFER IDENTIFIER FIELD 1214
RING INDEX 1204
BUFFER IDENTIFIER 1206

FIG. 12A

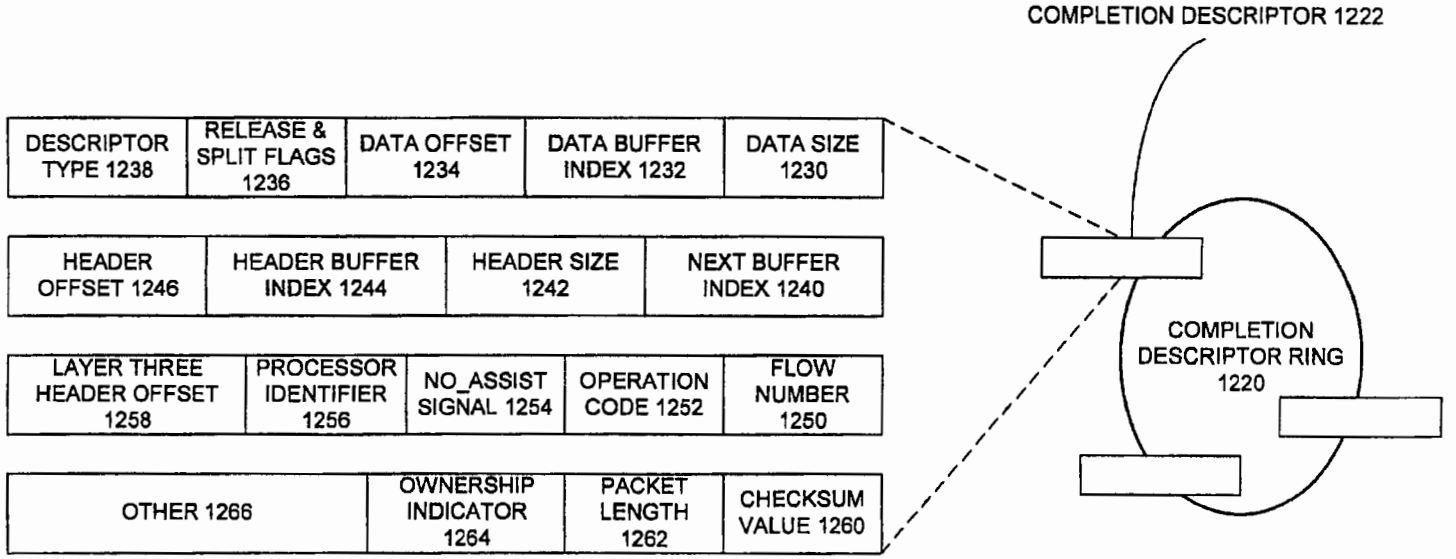

COMPLETION DESCRIPTOR 1222
DESCRIPTOR TYPE 1238
RELEASE & SPLIT FLAGS 1236
DATA OFFSET 1234
DATA BUFFER INDEX 1232
DATA SIZE 1230
HEADER OFFSET 1246
HEADER BUFFER INDEX 1244
HEADER SIZE 1242
NEXT BUFFER INDEX 1240
LAYER THREE HEADER OFFSET 1258
PROCESSOR IDENTIFIER 1256
NO_ASSIST SIGNAL 1254
OPERATION CODE 1252
FLOW NUMBER 1250
OTHER 1266
OWNERSHIP INDICATOR 1264
PACKET LENGTH 1262
CHECKSUM VALUE 1260
COMPLETION DESCRIPTOR RING 1220

FIG. 12B

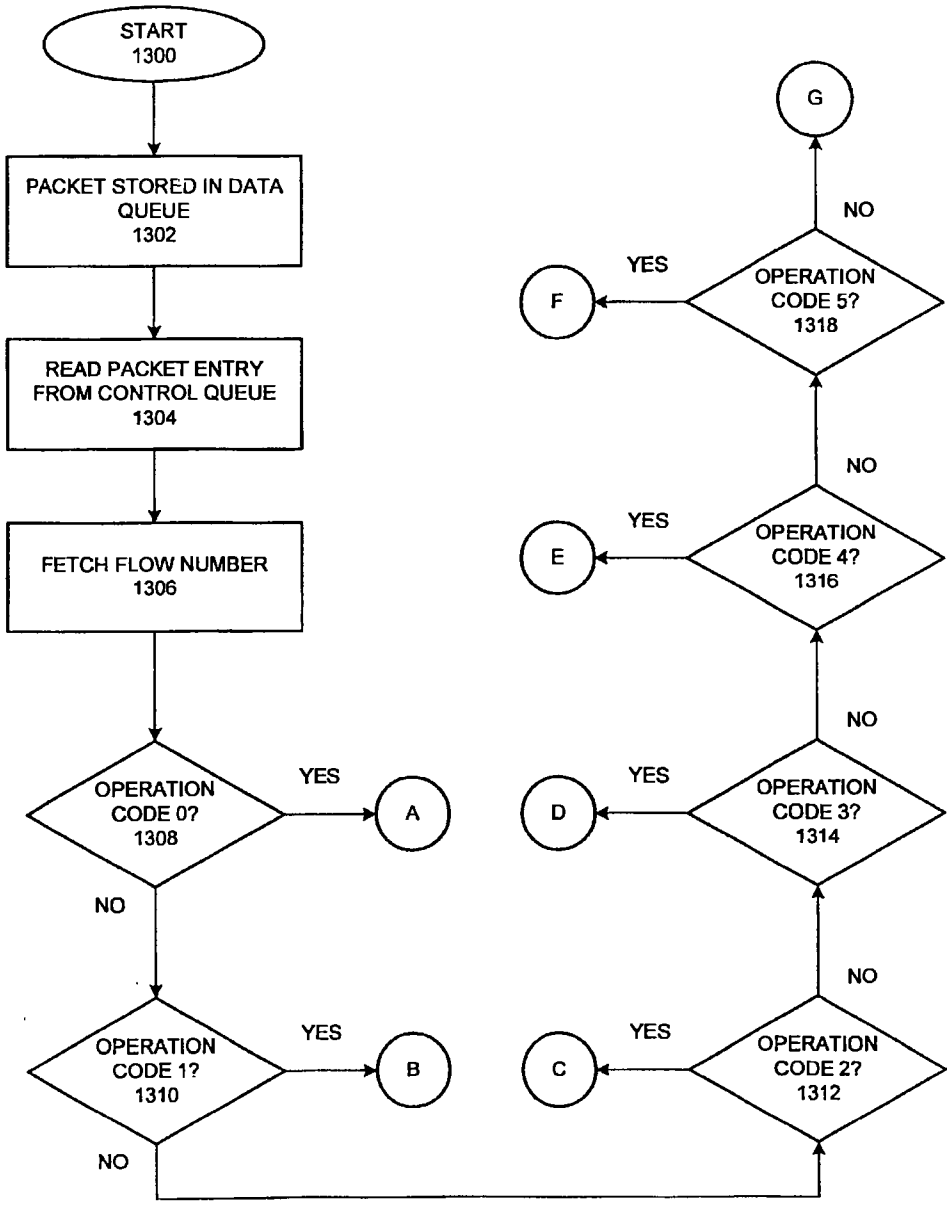

START
1300
PACKET STORED IN DATA QUEUE
1302
READ PACKET ENTRY FROM CONTROL QUEUE
1304
FETCH FLOW NUMBER
1306
OPERATION CODE 0?
1308
YES
A
NO
OPERATION CODE 1?
1310
YES
B
NO
OPERATION CODE 2?
1312
YES
C
NO
OPERATION CODE 3?
1314
YES
D
NO
OPERATION CODE 4?
1316
YES
E
NO
OPERATION CODE 5?
1318
YES
F
NO
G

FIG. 13

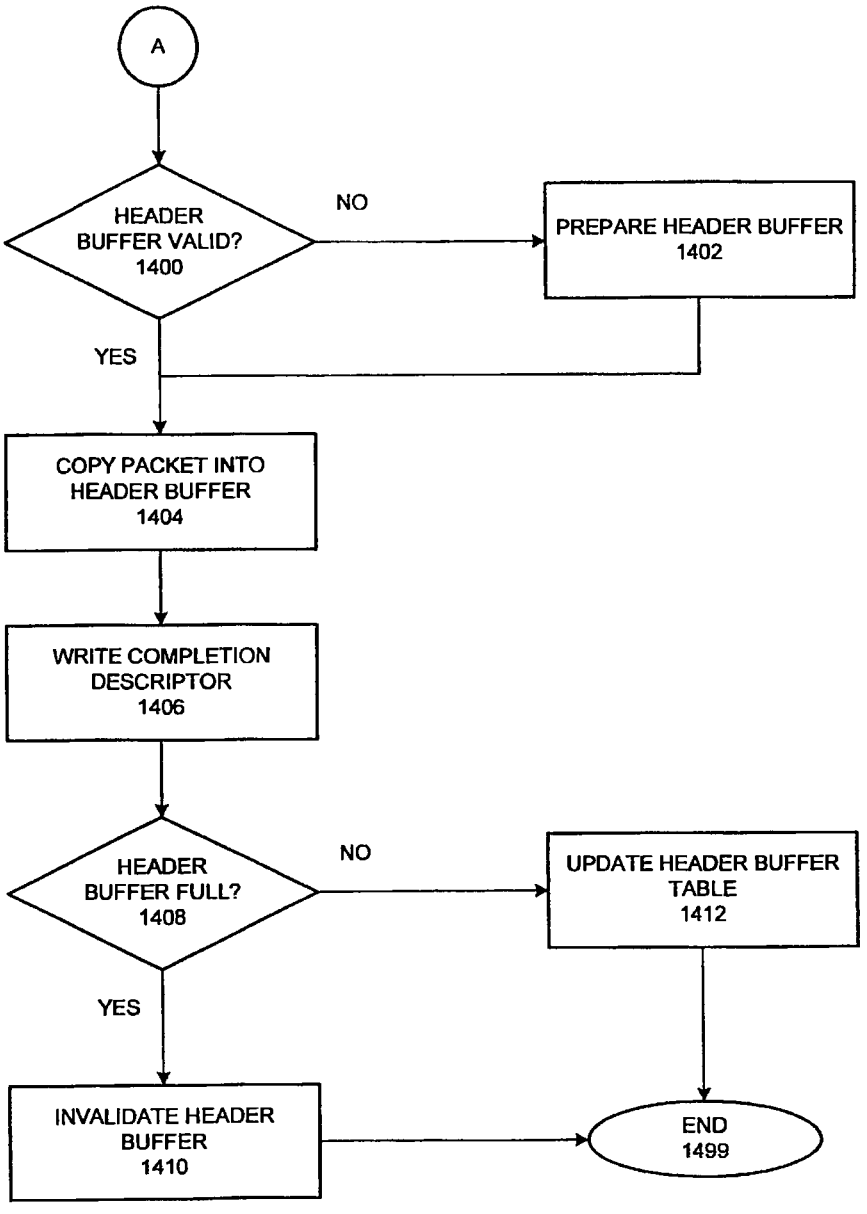

A
HEADER BUFFER VALID? 1400
NO
PREPARE HEADER BUFFER 1402
YES
COPY PACKET INTO HEADER BUFFER 1404
WRITE COMPLETION DESCRIPTOR 1406
HEADER BUFFER FULL? 1408
NO
UPDATE HEADER BUFFER TABLE 1412
YES
INVALIDATE HEADER BUFFER 1410
END 1499

FIG. 14

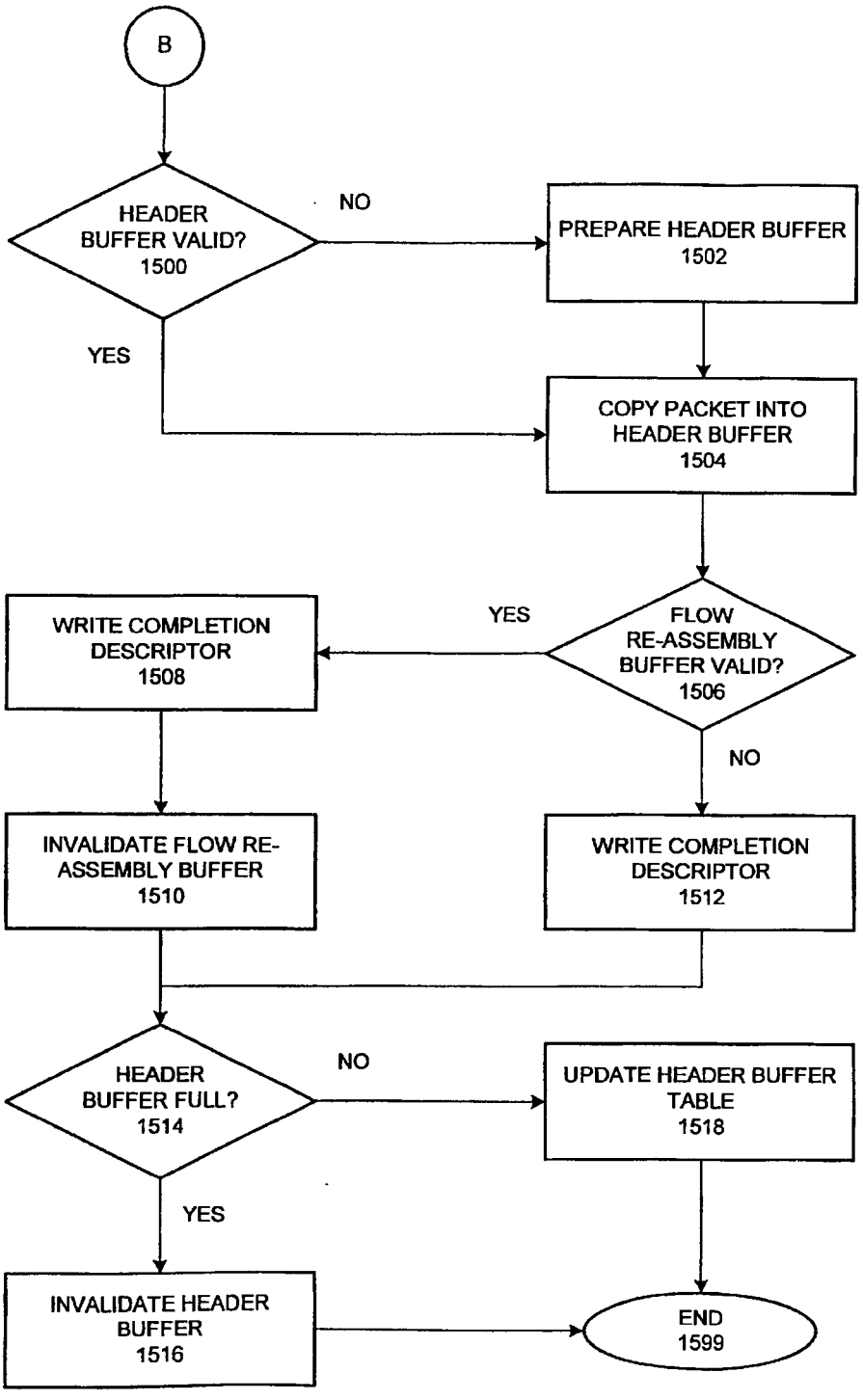

B
HEADER BUFFER VALID? 1500
NO
PREPARE HEADER BUFFER 1502
YES
COPY PACKET INTO HEADER BUFFER 1504
FLOW RE-ASSEMBLY BUFFER VALID? 1506
YES
WRITE COMPLETION DESCRIPTOR 1508
NO
INVALIDATE FLOW RE-ASSEMBLY BUFFER 1510
WRITE COMPLETION DESCRIPTOR 1512
HEADER BUFFER FULL? 1514
NO
UPDATE HEADER BUFFER TABLE 1518
YES
INVALIDATE HEADER BUFFER 1516
END 1599

FIG. 15

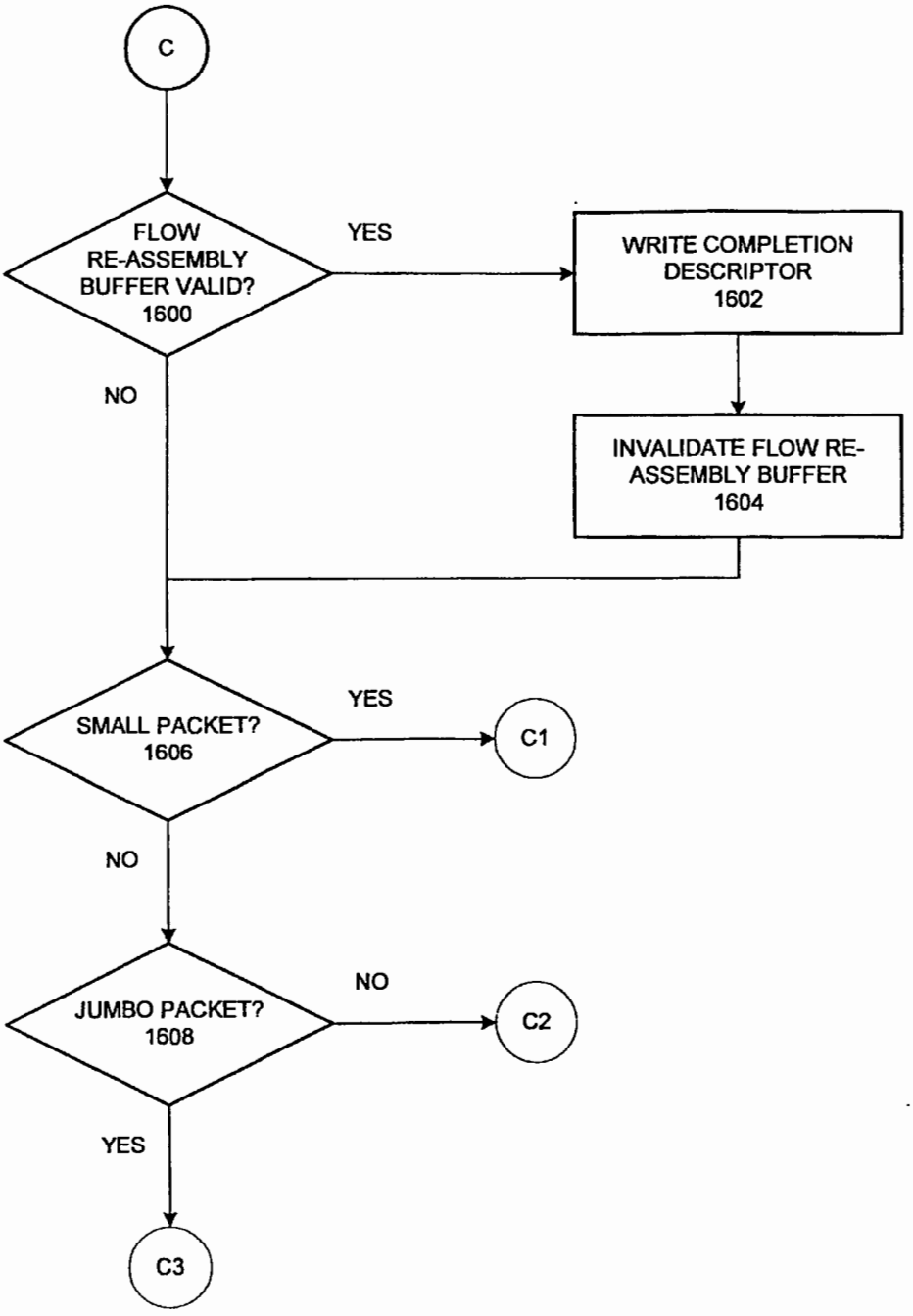

C
FLOW RE-ASSEMBLY BUFFER VALID? 1600
YES
WRITE COMPLETION DESCRIPTOR 1602
NO
INVALIDATE FLOW RE-ASSEMBLY BUFFER 1604
SMALL PACKET? 1606
YES
C1
NO
JUMBO PACKET? 1608
NO
C2
YES
C3

FIG. 16A

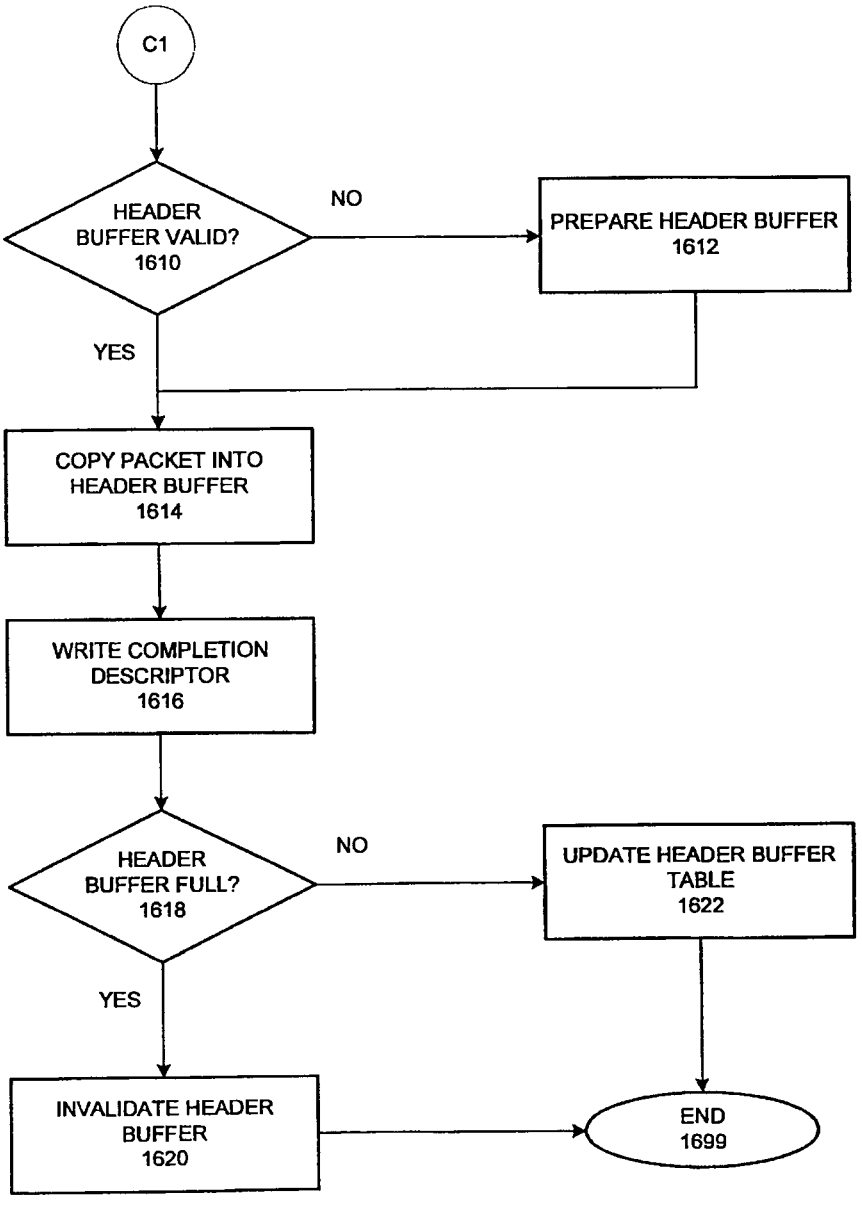

C1
HEADER BUFFER VALID? 1610
NO
PREPARE HEADER BUFFER 1612
YES
COPY PACKET INTO HEADER BUFFER 1614
WRITE COMPLETION DESCRIPTOR 1616
HEADER BUFFER FULL? 1618
NO
UPDATE HEADER BUFFER TABLE 1622
YES
INVALIDATE HEADER BUFFER 1620
END 1699

FIG. 16B

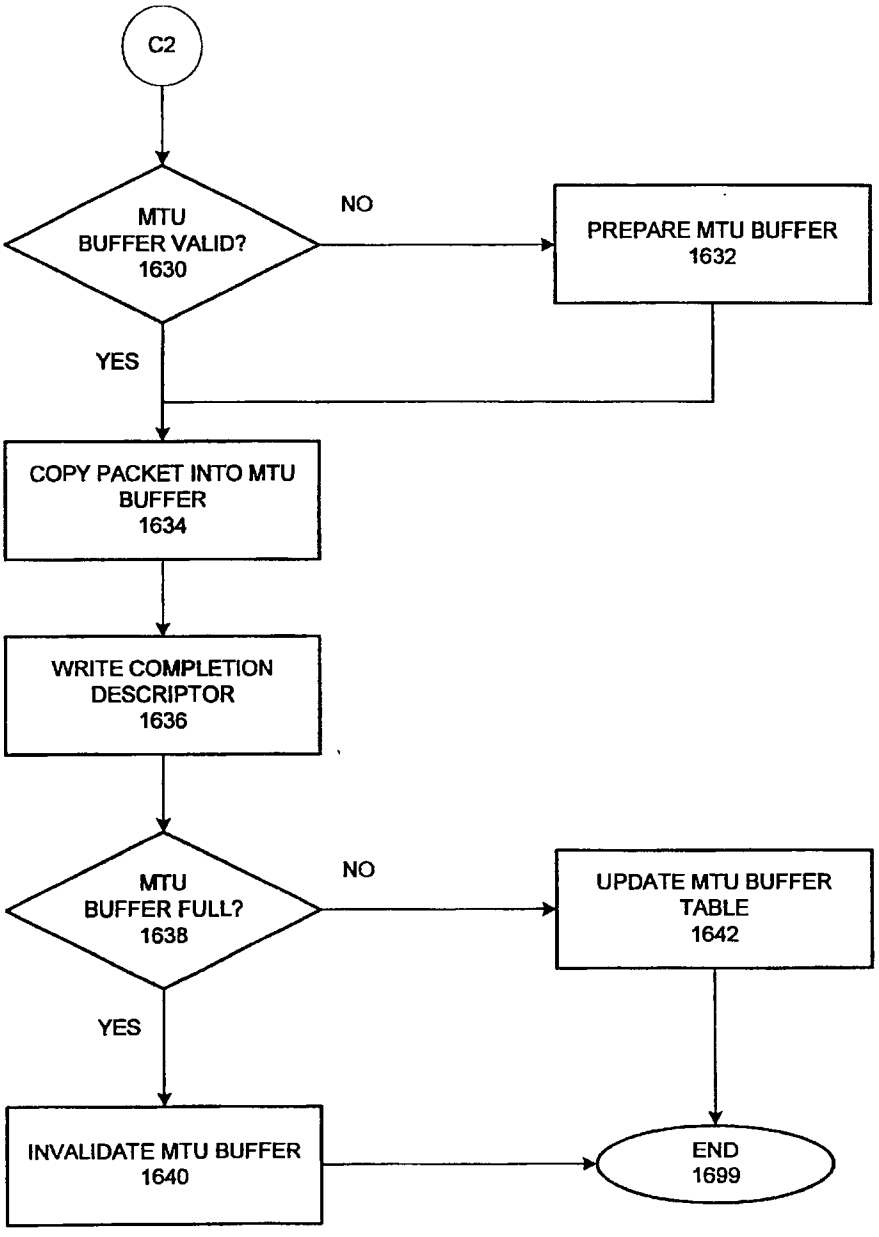

C2
MTU BUFFER VALID? 1630
NO
PREPARE MTU BUFFER 1632
YES
COPY PACKET INTO MTU BUFFER 1634
WRITE COMPLETION DESCRIPTOR 1636
MTU BUFFER FULL? 1638
NO
UPDATE MTU BUFFER TABLE 1642
YES
INVALIDATE MTU BUFFER 1640
END 1699

FIG. 16C

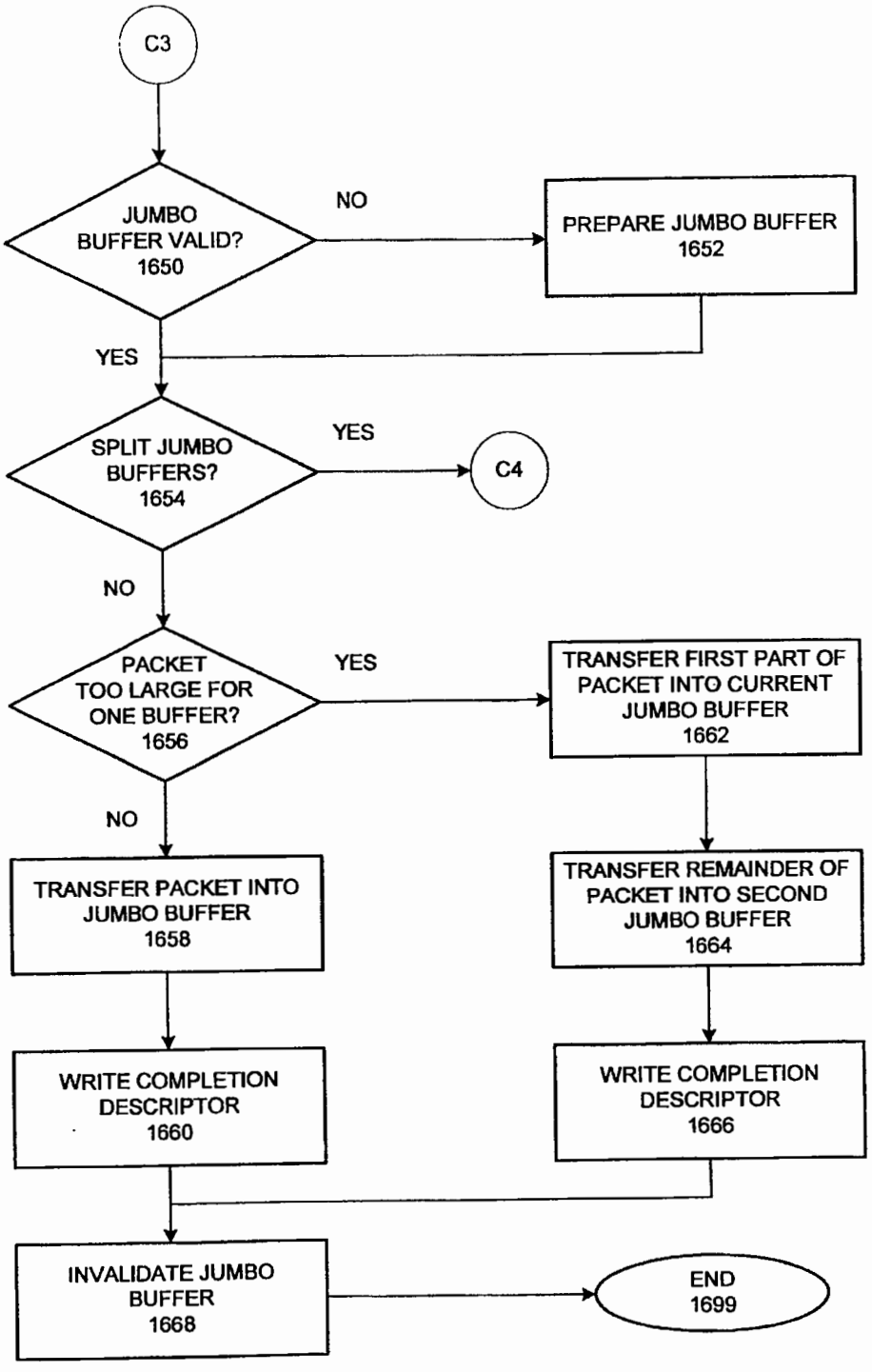

C3
JUMBO BUFFER VALID? 1650
NO
PREPARE JUMBO BUFFER 1652
YES
SPLIT JUMBO BUFFERS? 1654
YES
C4
NO
PACKET TOO LARGE FOR ONE BUFFER? 1656
YES
TRANSFER FIRST PART OF PACKET INTO CURRENT JUMBO BUFFER 1662
NO
TRANSFER PACKET INTO JUMBO BUFFER 1658
TRANSFER REMAINDER OF PACKET INTO SECOND JUMBO BUFFER 1664
WRITE COMPLETION DESCRIPTOR 1660
WRITE COMPLETION DESCRIPTOR 1666
INVALIDATE JUMBO BUFFER 1668
END 1699

FIG. 16D

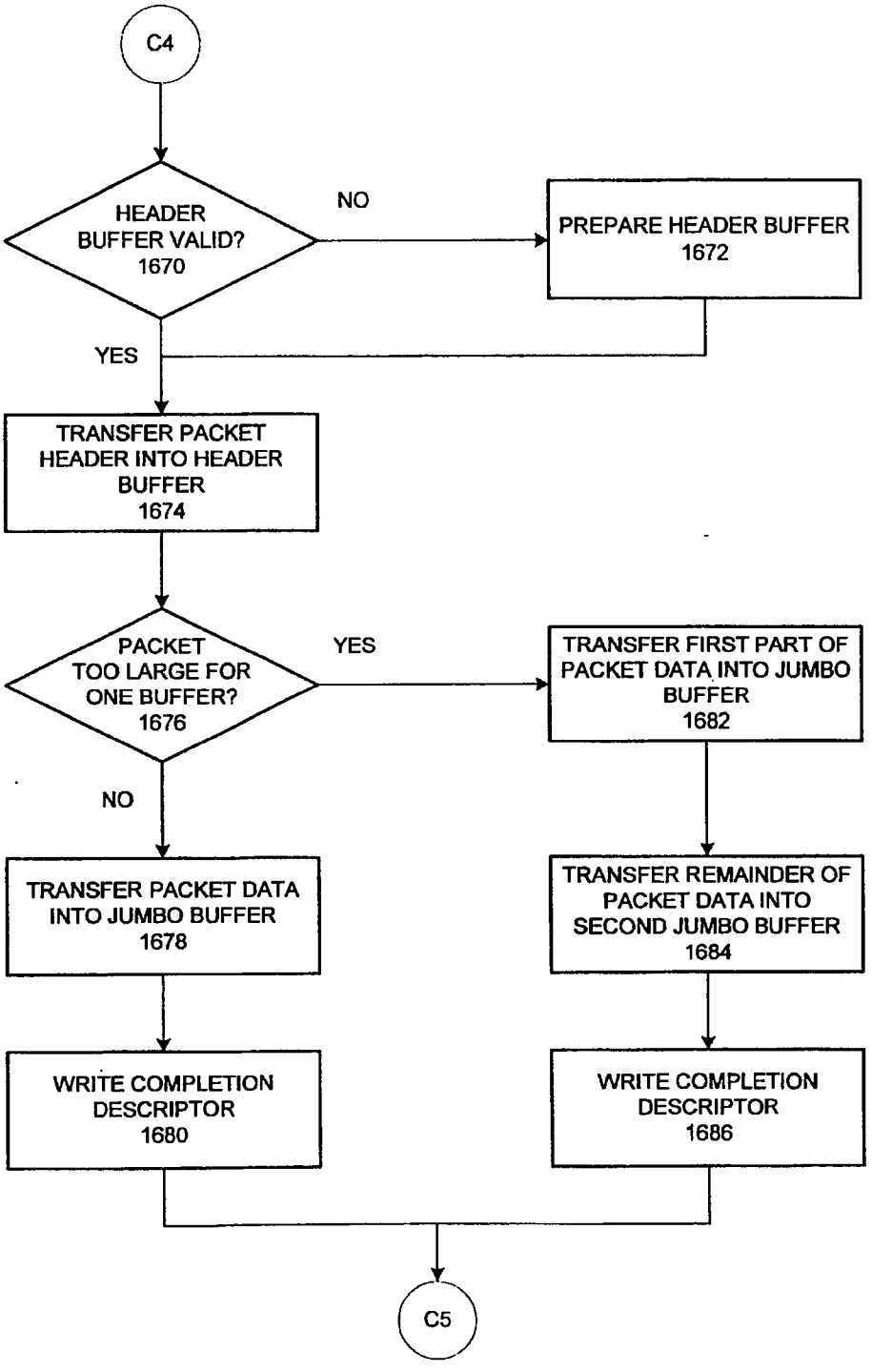

C4
HEADER BUFFER VALID? 1670
NO
PREPARE HEADER BUFFER 1672
YES
TRANSFER PACKET HEADER INTO HEADER BUFFER 1674
PACKET TOO LARGE FOR ONE BUFFER? 1676
YES
TRANSFER FIRST PART OF PACKET DATA INTO JUMBO BUFFER 1682
NO
TRANSFER PACKET DATA INTO JUMBO BUFFER 1678
TRANSFER REMAINDER OF PACKET DATA INTO SECOND JUMBO BUFFER 1684
WRITE COMPLETION DESCRIPTOR 1680
WRITE COMPLETION DESCRIPTOR 1686
C5

FIG. 16E

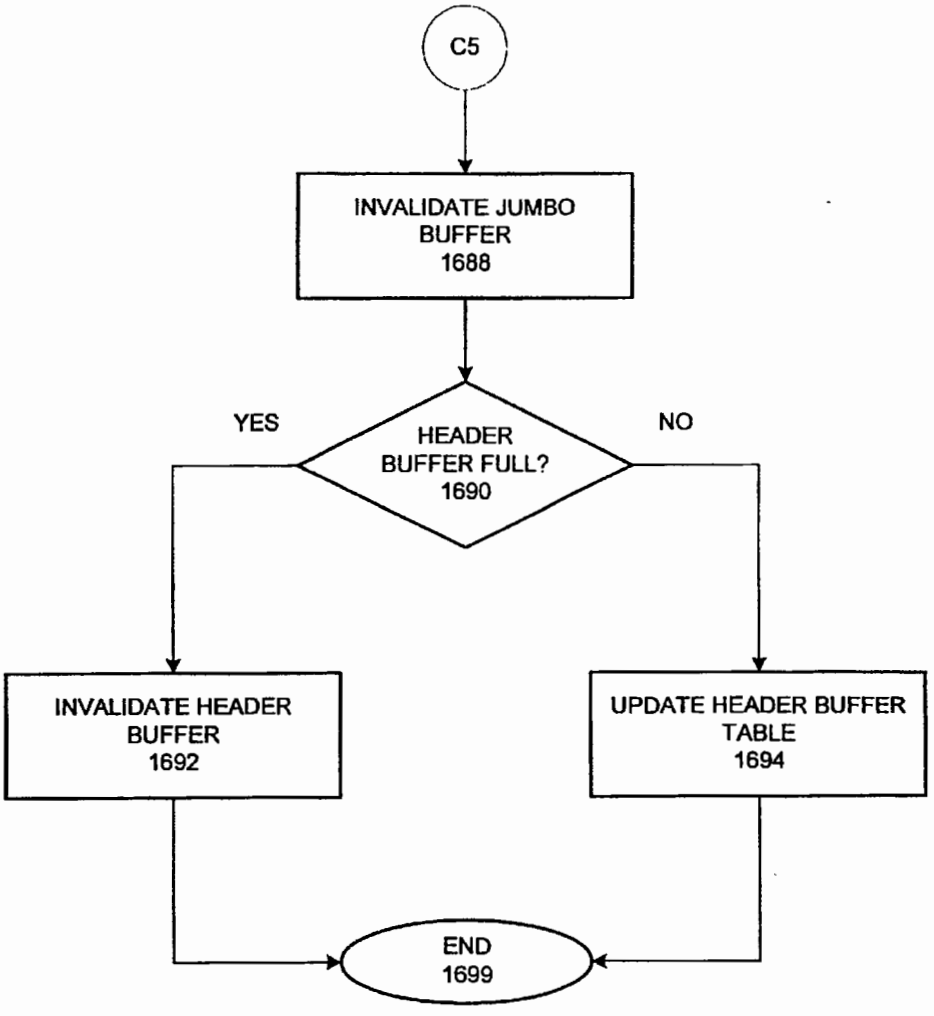

C5
INVALIDATE JUMBO BUFFER 1688
YES
HEADER BUFFER FULL? 1690
NO
INVALIDATE HEADER BUFFER 1692
UPDATE HEADER BUFFER TABLE 1694
END 1699

FIG. 16F

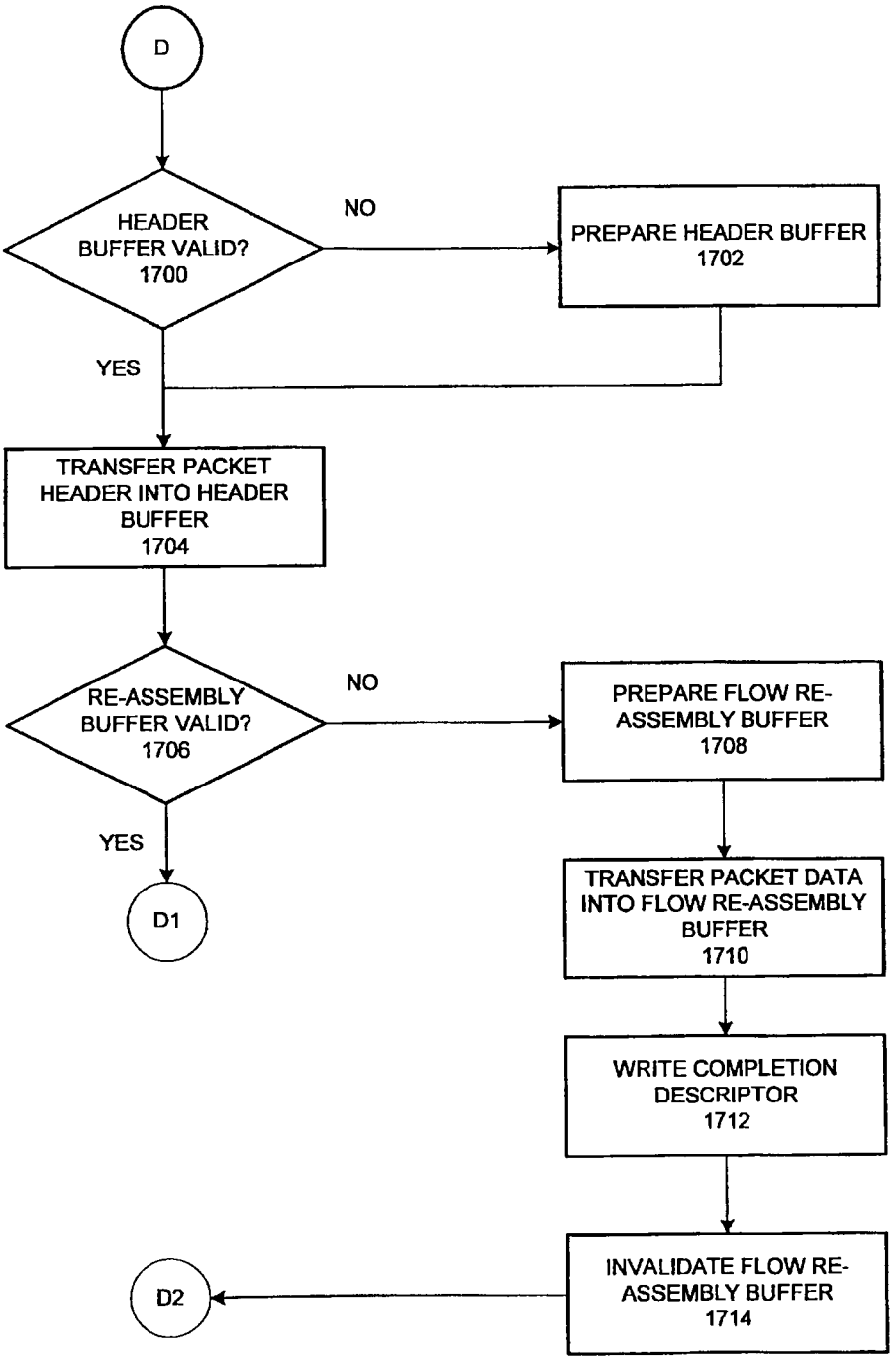

D
HEADER BUFFER VALID? 1700
NO
PREPARE HEADER BUFFER 1702
YES
TRANSFER PACKET HEADER INTO HEADER BUFFER 1704
RE-ASSEMBLY BUFFER VALID? 1706
NO
PREPARE FLOW RE-ASSEMBLY BUFFER 1708
YES
D1
TRANSFER PACKET DATA INTO FLOW RE-ASSEMBLY BUFFER 1710
WRITE COMPLETION DESCRIPTOR 1712
INVALIDATE FLOW RE-ASSEMBLY BUFFER 1714
D2

FIG. 17A

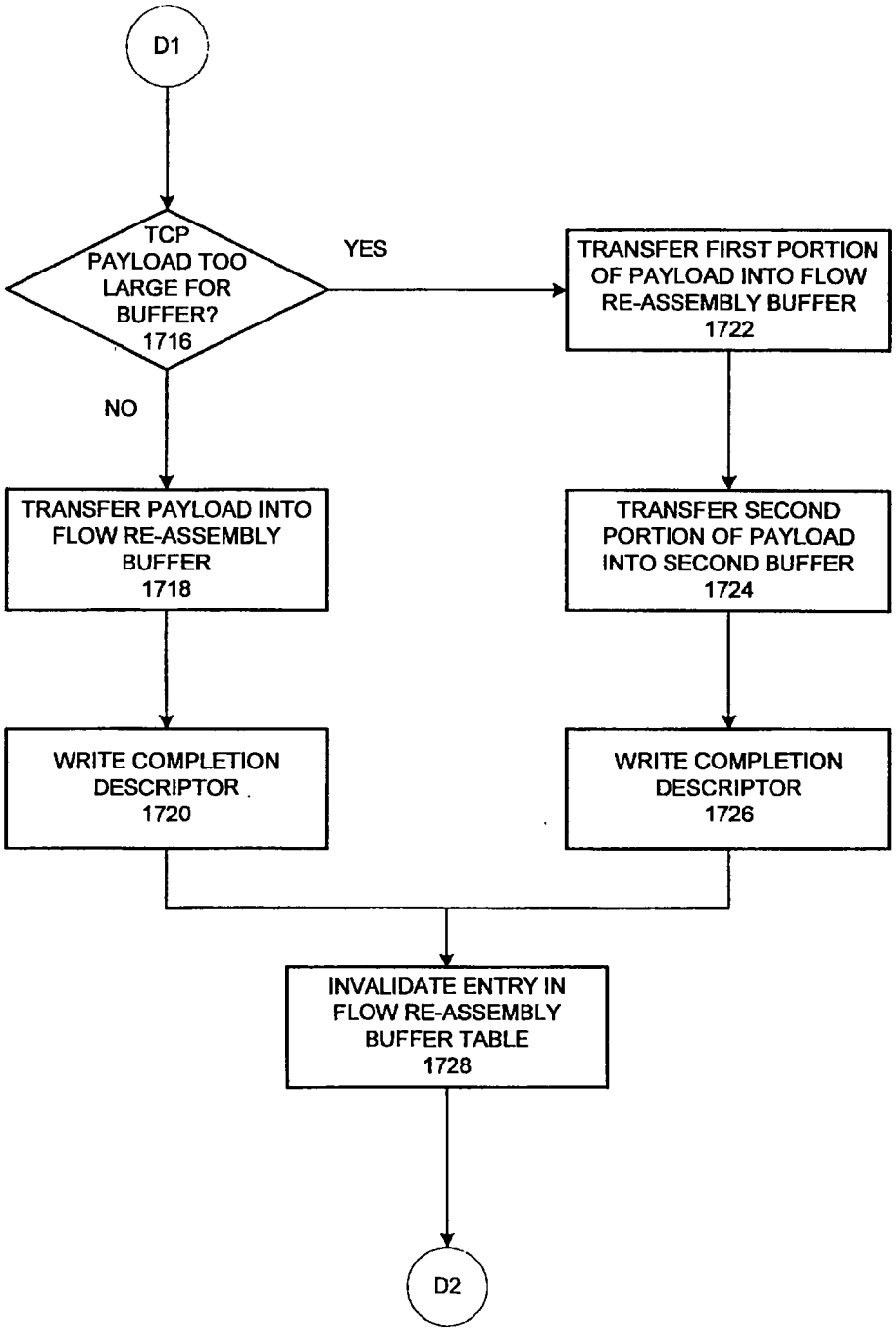

D1
TCP PAYLOAD TOO LARGE FOR BUFFER? 1716
YES
NO
TRANSFER FIRST PORTION OF PAYLOAD INTO FLOW RE-ASSEMBLY BUFFER 1722
TRANSFER PAYLOAD INTO FLOW RE-ASSEMBLY BUFFER 1718
TRANSFER SECOND PORTION OF PAYLOAD INTO SECOND BUFFER 1724
WRITE COMPLETION DESCRIPTOR 1720
WRITE COMPLETION DESCRIPTOR 1726
INVALIDATE ENTRY IN FLOW RE-ASSEMBLY BUFFER TABLE 1728
D2

FIG. 17B

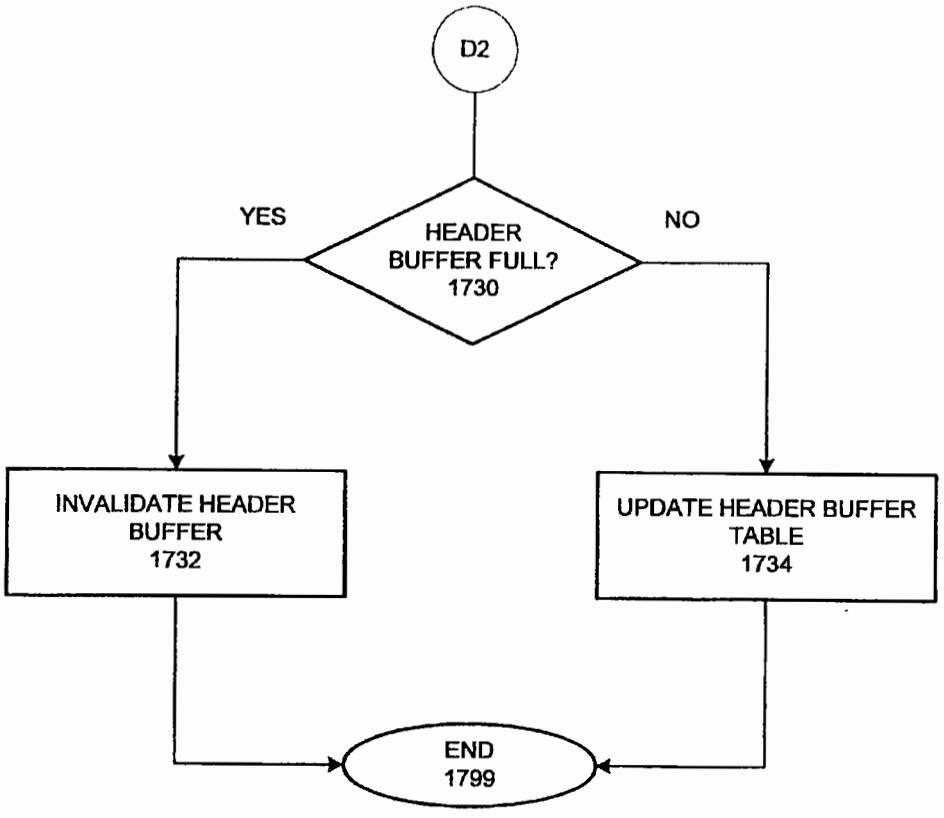

D2
HEADER BUFFER FULL? 1730
YES
NO
INVALIDATE HEADER BUFFER 1732
UPDATE HEADER BUFFER TABLE 1734
END 1799

FIG. 17C

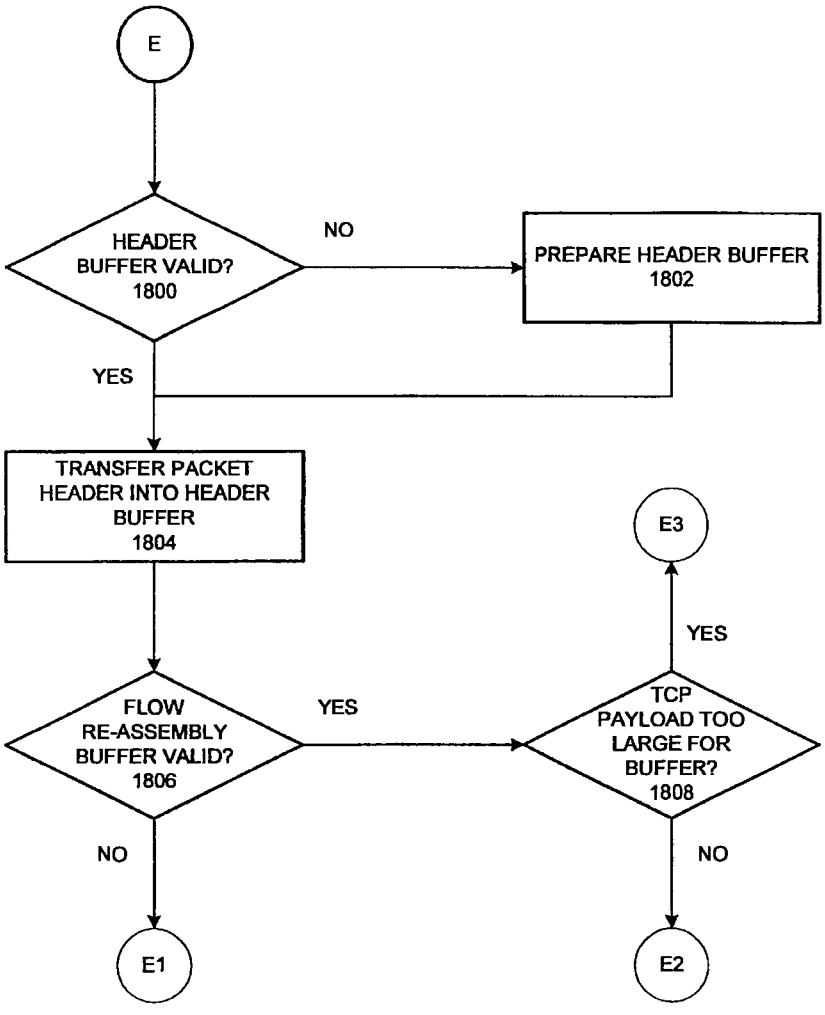

E
HEADER BUFFER VALID? 1800
NO
PREPARE HEADER BUFFER 1802
YES
TRANSFER PACKET HEADER INTO HEADER BUFFER 1804
FLOW RE-ASSEMBLY BUFFER VALID? 1806
YES
TCP PAYLOAD TOO LARGE FOR BUFFER? 1808
YES
NO
NO
E1
E2
E3

FIG. 18A

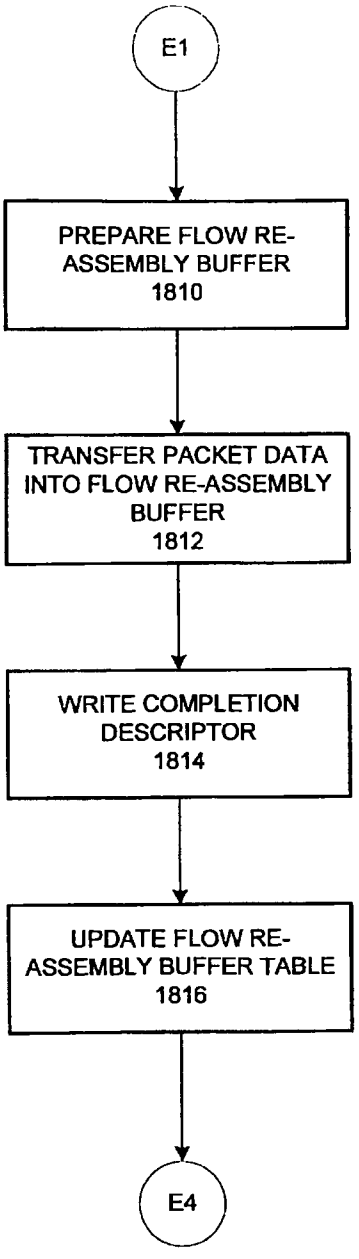

E1
PREPARE FLOW RE-ASSEMBLY BUFFER 1810
TRANSFER PACKET DATA INTO FLOW RE-ASSEMBLY BUFFER 1812
WRITE COMPLETION DESCRIPTOR 1814
UPDATE FLOW RE-ASSEMBLY BUFFER TABLE 1816
E4

FIG. 18B

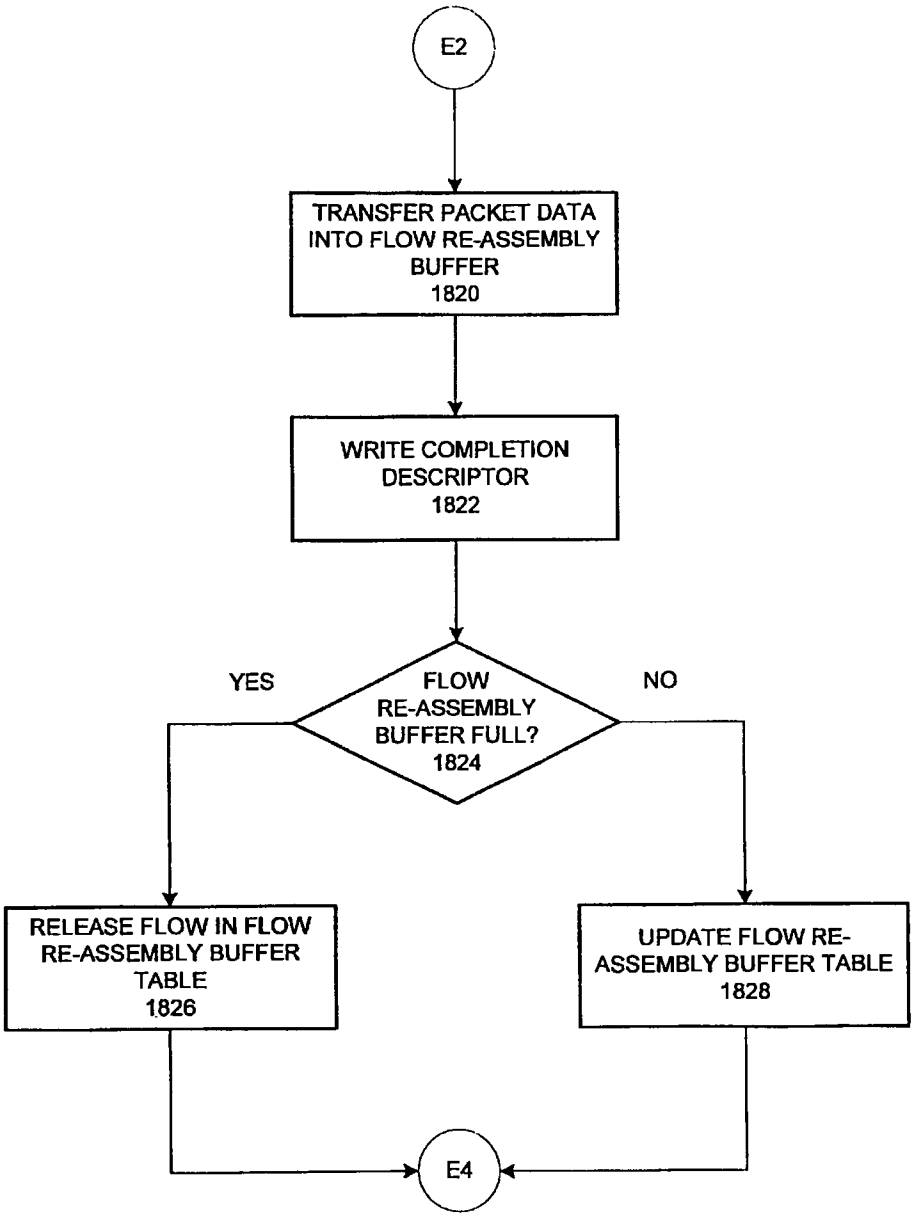

E2
TRANSFER PACKET DATA INTO FLOW RE-ASSEMBLY BUFFER 1820
WRITE COMPLETION DESCRIPTOR 1822
FLOW RE-ASSEMBLY BUFFER FULL? 1824
YES
NO
RELEASE FLOW IN FLOW RE-ASSEMBLY BUFFER TABLE 1826
UPDATE FLOW RE-ASSEMBLY BUFFER TABLE 1828
E4

FIG. 18C

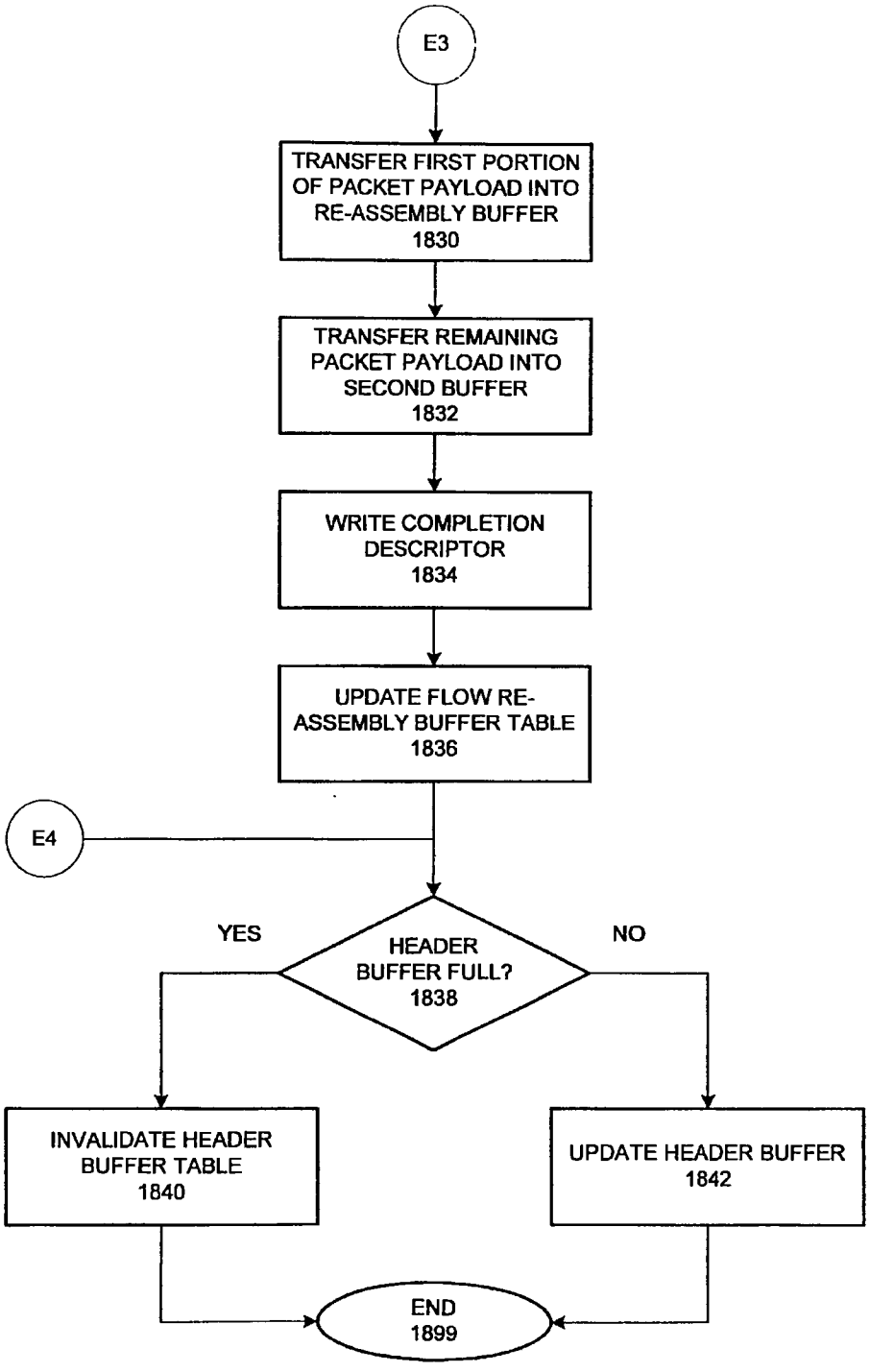

E3
TRANSFER FIRST PORTION OF PACKET PAYLOAD INTO RE-ASSEMBLY BUFFER 1830
TRANSFER REMAINING PACKET PAYLOAD INTO SECOND BUFFER 1832
WRITE COMPLETION DESCRIPTOR 1834
UPDATE FLOW RE-ASSEMBLY BUFFER TABLE 1836
E4
YES
HEADER BUFFER FULL? 1838
NO
INVALIDATE HEADER BUFFER TABLE 1840
UPDATE HEADER BUFFER 1842
END 1899

FIG. 18D

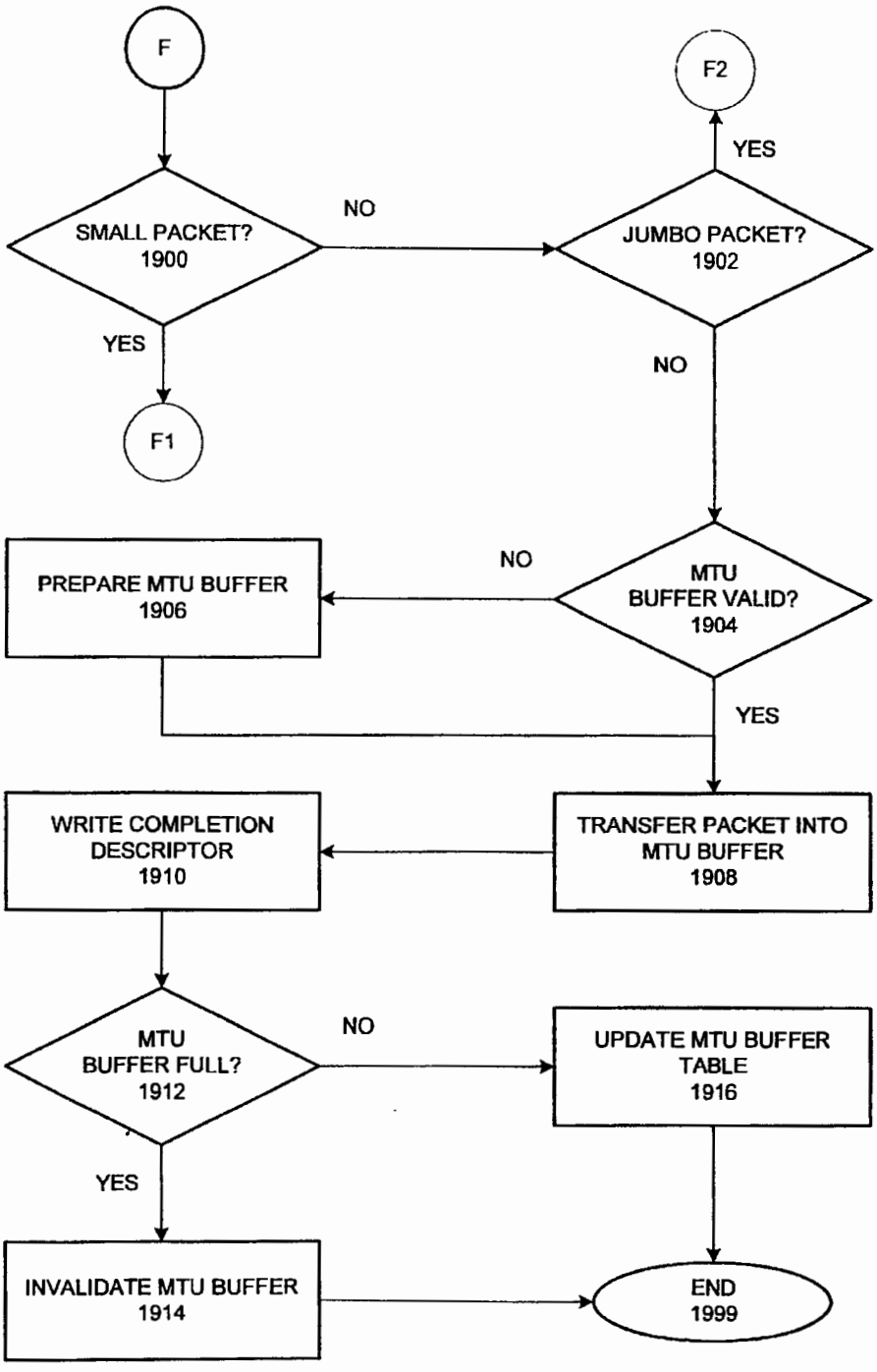

F
F2
YES
SMALL PACKET?
1900
NO
JUMBO PACKET?
1902
YES
NO
F1
PREPARE MTU BUFFER
1906
NO
MTU
BUFFER VALID?
1904
YES
WRITE COMPLETION
DESCRIPTOR
1910
TRANSFER PACKET INTO
MTU BUFFER
1908
MTU
BUFFER FULL?
1912
NO
UPDATE MTU BUFFER
TABLE
1916
YES
INVALIDATE MTU BUFFER
1914
END
1999

FIG. 19A

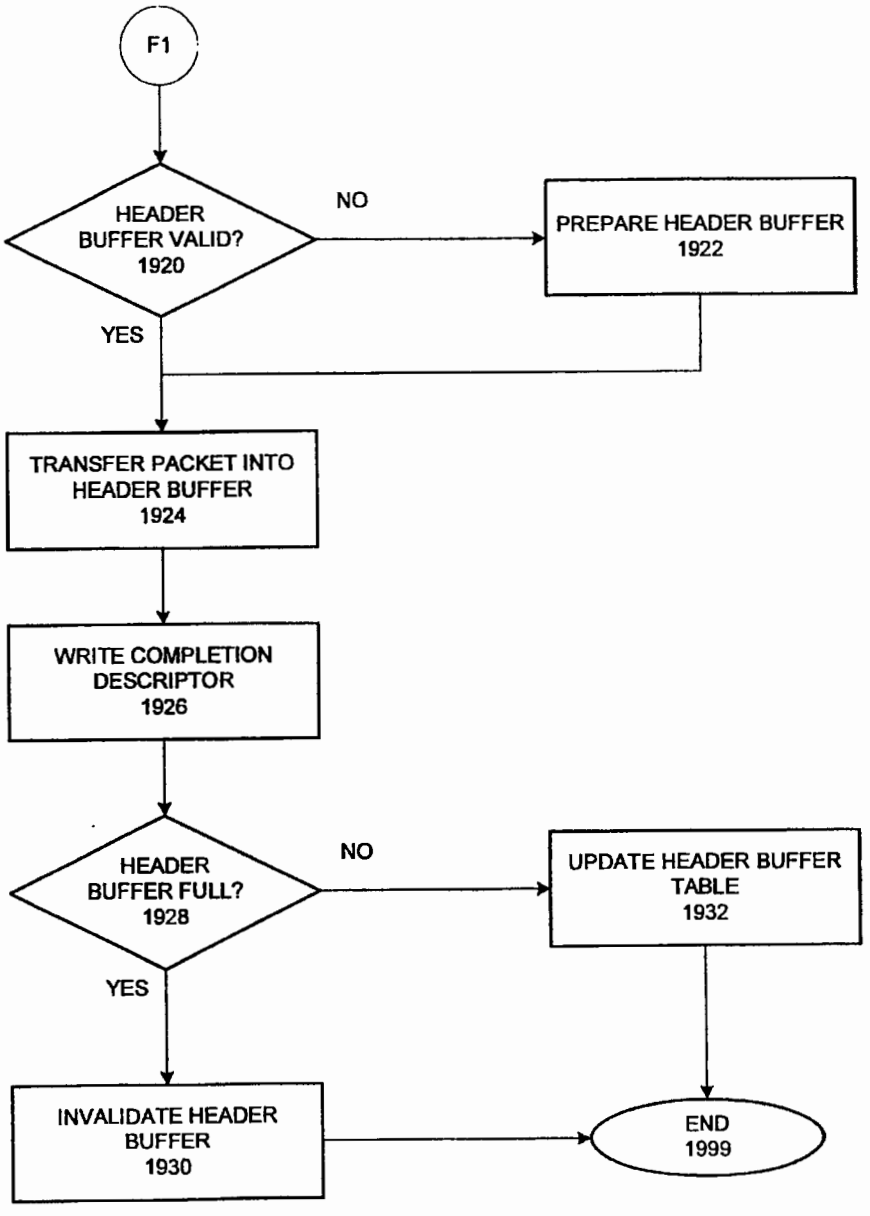

F1
HEADER BUFFER VALID? 1920
NO
PREPARE HEADER BUFFER 1922
YES
TRANSFER PACKET INTO HEADER BUFFER 1924
WRITE COMPLETION DESCRIPTOR 1926
HEADER BUFFER FULL? 1928
NO
UPDATE HEADER BUFFER TABLE 1932
YES
INVALIDATE HEADER BUFFER 1930
END 1999

FIG. 19B

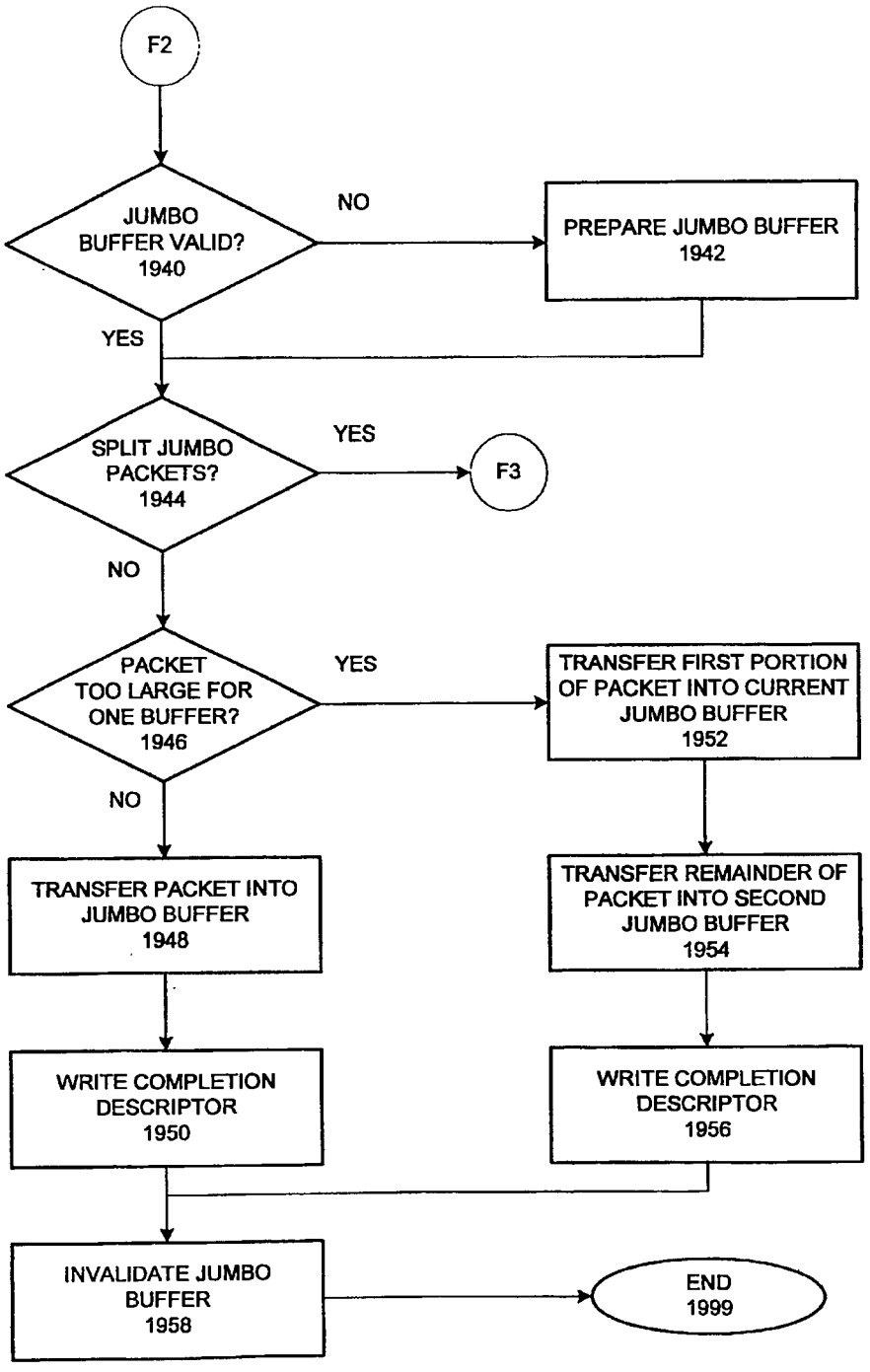

F2
JUMBO BUFFER VALID? 1940
NO
PREPARE JUMBO BUFFER 1942
YES
SPLIT JUMBO PACKETS? 1944
YES
F3
NO
PACKET TOO LARGE FOR ONE BUFFER? 1946
YES
TRANSFER FIRST PORTION OF PACKET INTO CURRENT JUMBO BUFFER 1952
NO
TRANSFER PACKET INTO JUMBO BUFFER 1948
TRANSFER REMAINDER OF PACKET INTO SECOND JUMBO BUFFER 1954
WRITE COMPLETION DESCRIPTOR 1950
WRITE COMPLETION DESCRIPTOR 1956
INVALIDATE JUMBO BUFFER 1958
END 1999

FIG. 19C

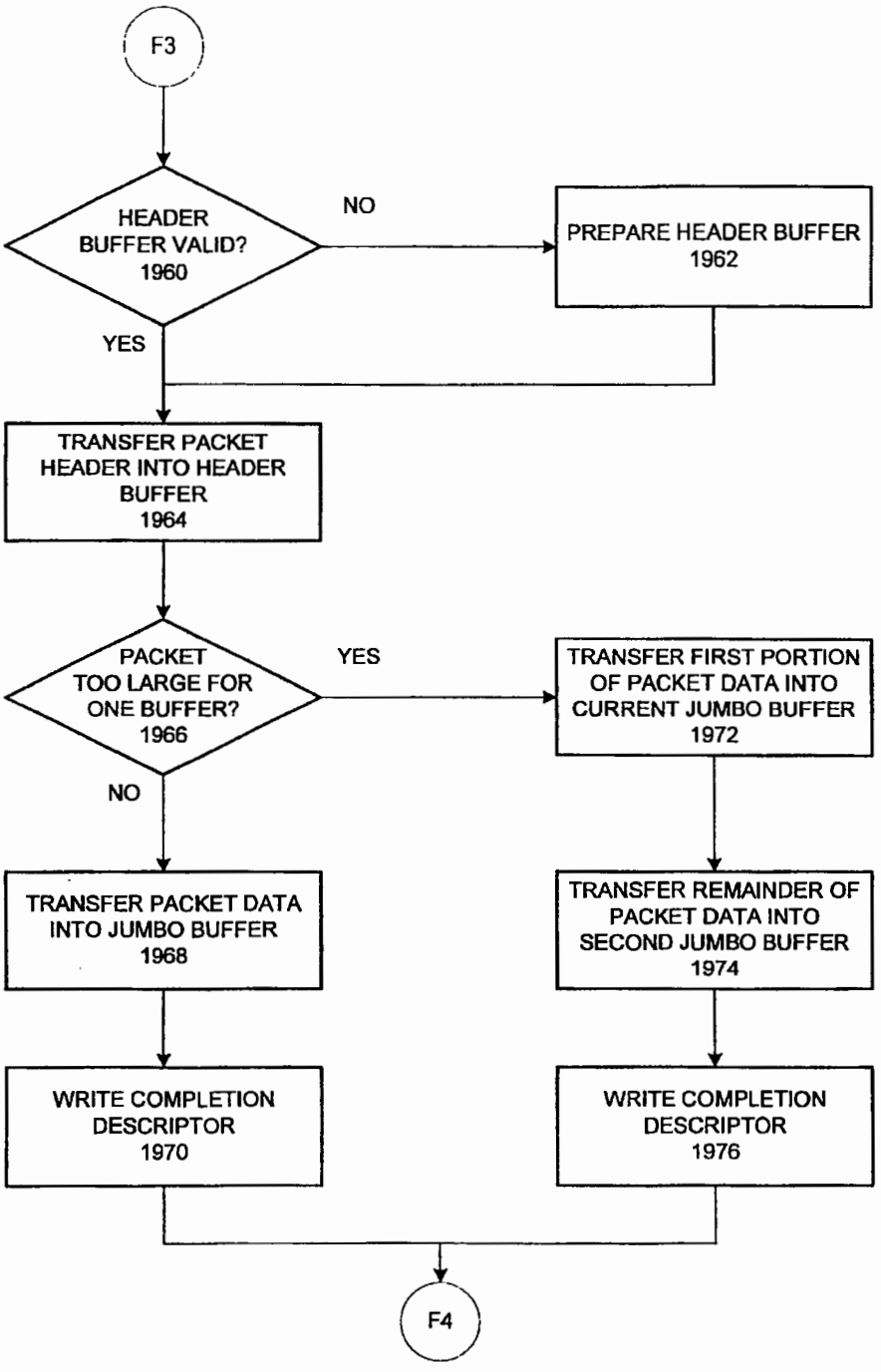

F3
HEADER BUFFER VALID? 1960
NO
PREPARE HEADER BUFFER 1962
YES
TRANSFER PACKET HEADER INTO HEADER BUFFER 1964
PACKET TOO LARGE FOR ONE BUFFER? 1966
YES
TRANSFER FIRST PORTION OF PACKET DATA INTO CURRENT JUMBO BUFFER 1972
NO
TRANSFER PACKET DATA INTO JUMBO BUFFER 1968
TRANSFER REMAINDER OF PACKET DATA INTO SECOND JUMBO BUFFER 1974
WRITE COMPLETION DESCRIPTOR 1970
WRITE COMPLETION DESCRIPTOR 1976
F4

FIG. 19D

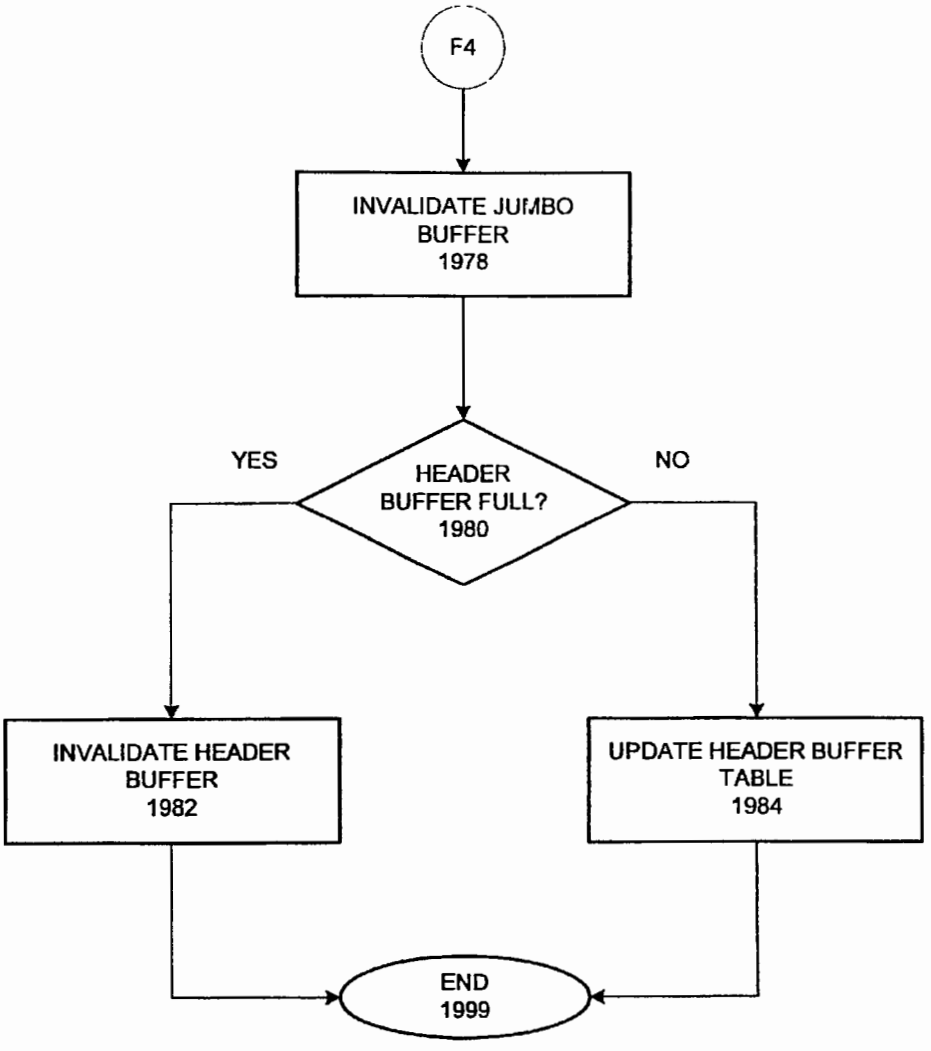

F4
INVALIDATE JUMBO BUFFER 1978
HEADER BUFFER FULL? 1980
YES
NO
INVALIDATE HEADER BUFFER 1982
UPDATE HEADER BUFFER TABLE 1984
END 1999

FIG. 19E

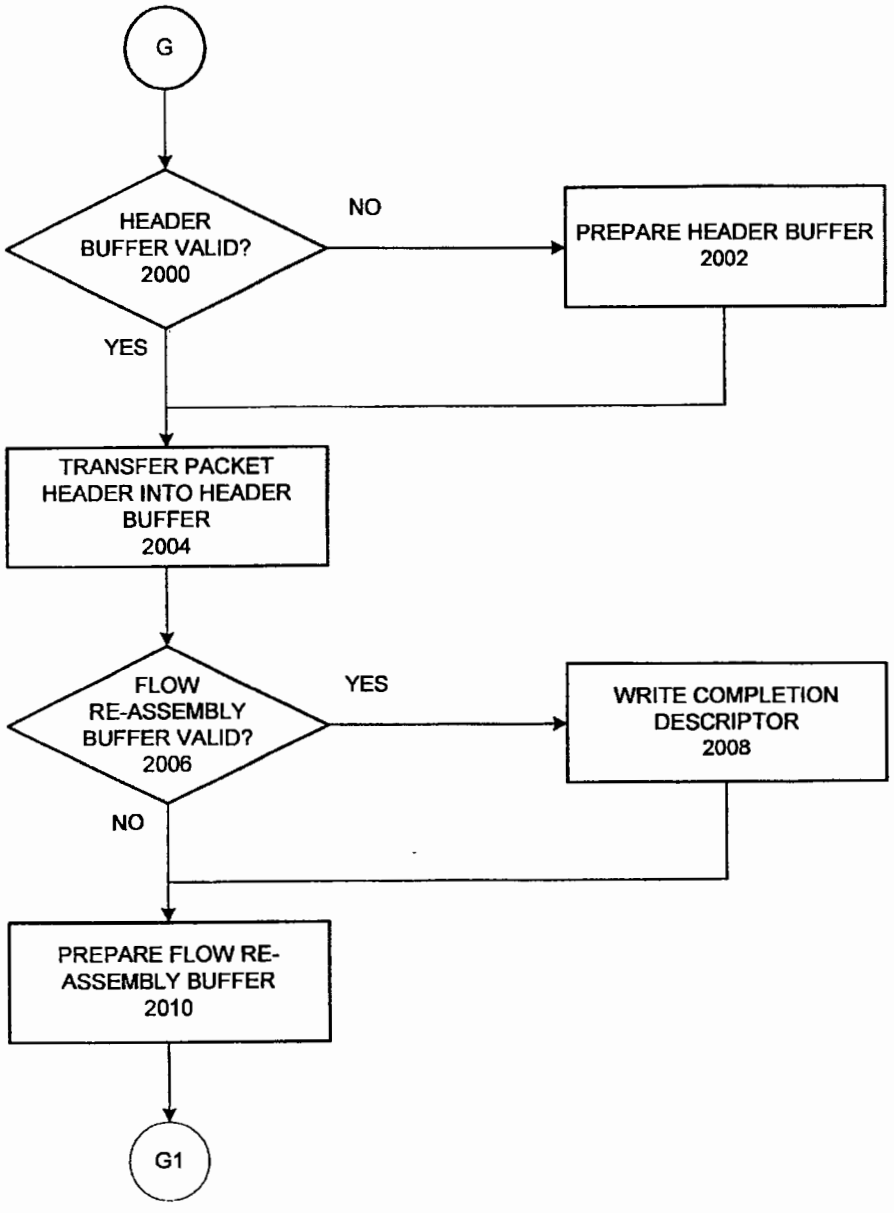

G
HEADER BUFFER VALID? 2000
NO
PREPARE HEADER BUFFER 2002
YES
TRANSFER PACKET HEADER INTO HEADER BUFFER 2004
FLOW RE-ASSEMBLY BUFFER VALID? 2006
YES
WRITE COMPLETION DESCRIPTOR 2008
NO
PREPARE FLOW RE-ASSEMBLY BUFFER 2010
G1

FIG. 20A

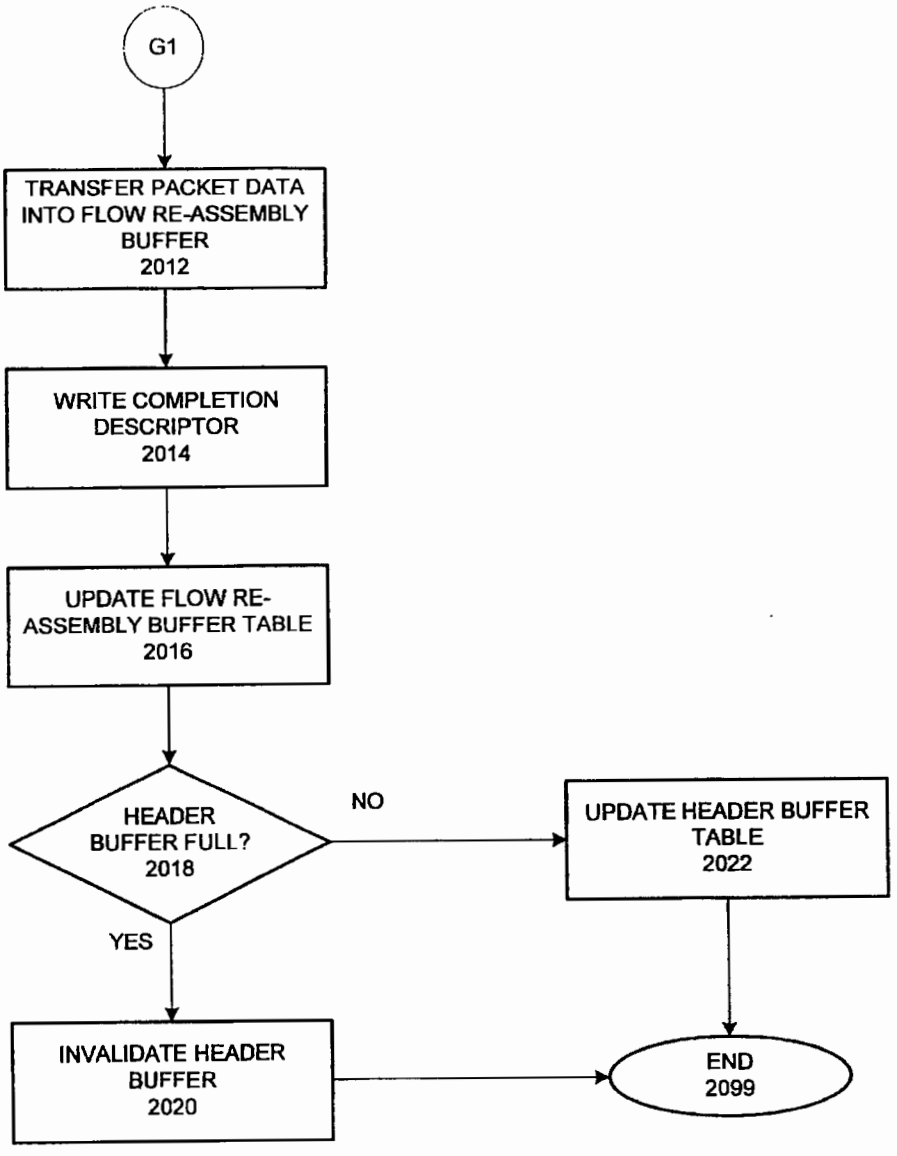

G1
TRANSFER PACKET DATA INTO FLOW RE-ASSEMBLY BUFFER 2012
WRITE COMPLETION DESCRIPTOR 2014
UPDATE FLOW RE-ASSEMBLY BUFFER TABLE 2016
HEADER BUFFER FULL? 2018
NO
UPDATE HEADER BUFFER TABLE 2022
YES
INVALIDATE HEADER BUFFER 2020
END 2099

FIG. 20B

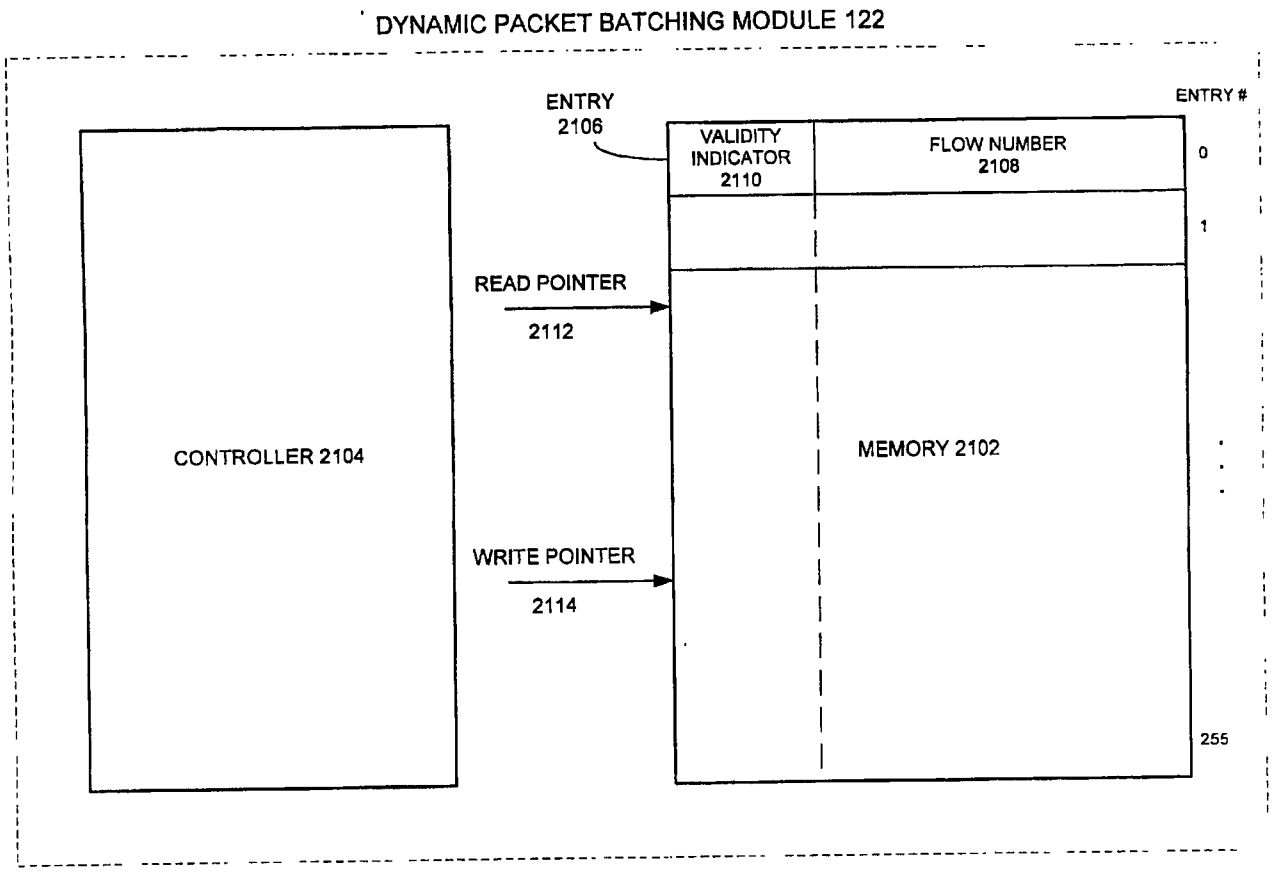

DYNAMIC PACKET BATCHING MODULE 122
ENTRY
2106
ENTRY #
VALIDITY INDICATOR 2110
FLOW NUMBER 2108
0
1
READ POINTER
2112
CONTROLLER 2104
MEMORY 2102
WRITE POINTER
2114
255

FIG. 21

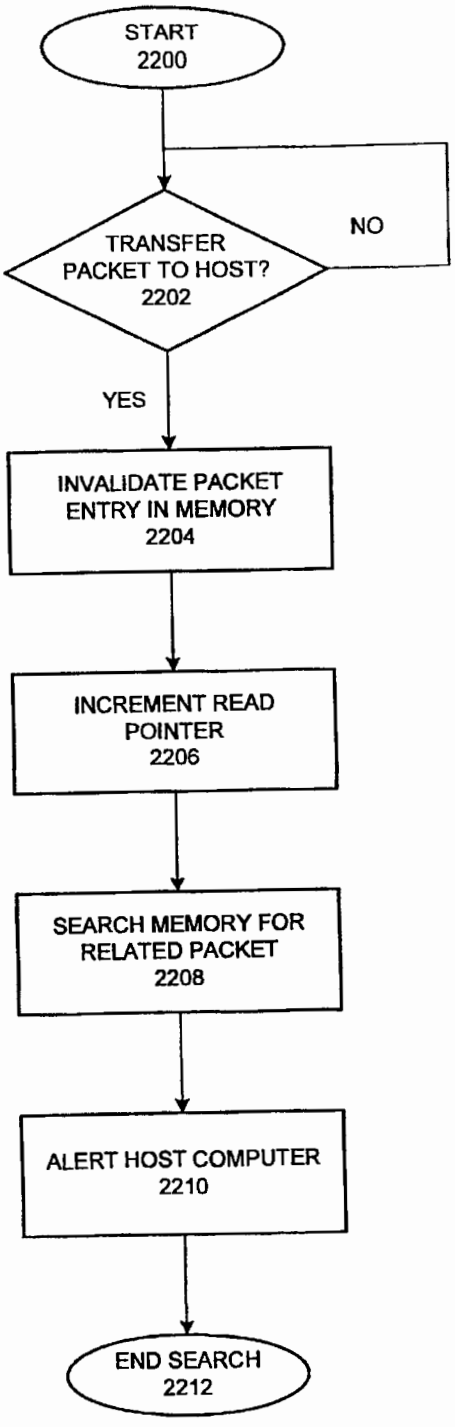

START
2200
TRANSFER PACKET TO HOST?
2202
NO
YES
INVALIDATE PACKET ENTRY IN MEMORY
2204
INCREMENT READ POINTER
2206
SEARCH MEMORY FOR RELATED PACKET
2208
ALERT HOST COMPUTER
2210
END SEARCH
2212

FIG. 22A

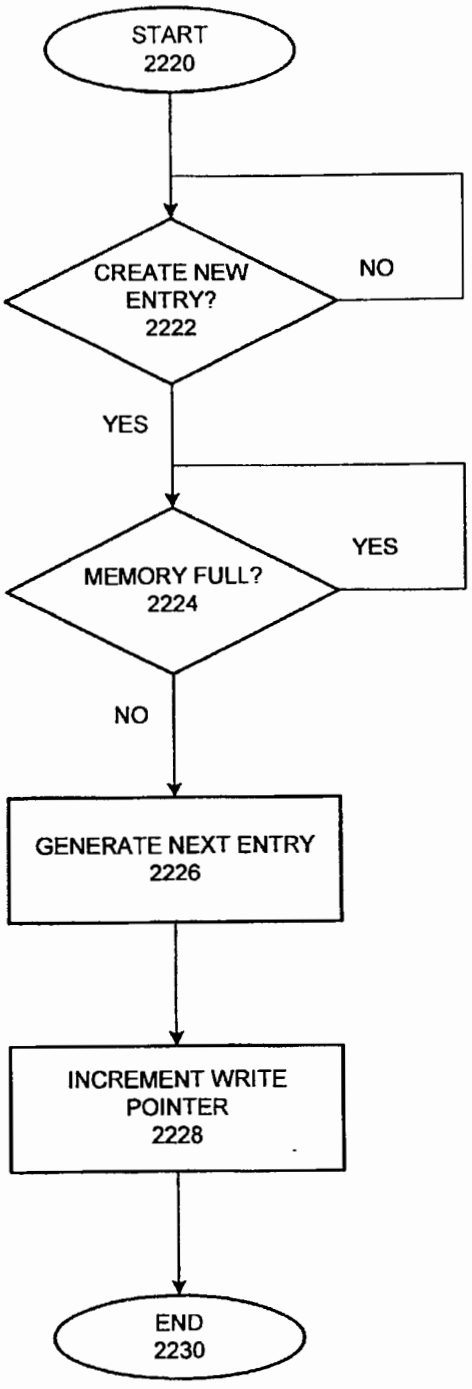

START
2220
CREATE NEW ENTRY?
2222
NO
YES
MEMORY FULL?
2224
YES
NO
GENERATE NEXT ENTRY
2226
INCREMENT WRITE POINTER
2228
END
2230

FIG. 22B

INSTR. NO. 2302	INSTR. NAME 2304	INSTRUCTION CONTENT 2306 (EXTRACTION MASK, COMPARE VALUE, OPERATOR, SUCCESS OFFSET, SUCCESS INSTRUCTION, FAILURE OFFSET, FAILURE INSTRUCTION, OUTPUT OPERATION, OPERATION ARGUMENT, OPERATION ENABLER, SHIFT, OUTPUT MASK)
0	WAIT	0xFFFF, 0x0000, NP, 6, VLAN, 0, WAIT, CLR_REG, 0x3FF, 1, 0, 0x0000
1	VLAN	0xFFFF, 0x8100, EQ, 1, CFI, 0, 802.3, IM_CTL, 0x00A, 3, 0, 0xFFFF
2	CFI	0x1000, 0x1000, EQ, 0, DONE, 1, 802.3, NONE, 0x000, 0, 0, 0x0000
3	802.3	0xFFFF, 0x0600, LT, 1, LLC_1, 0, IPV4_1, NONE, 0x000, 0, 0, 0x0000
4	LLC_1	0xFFFF, 0xAAAA, EQ, 1, LLC_2, 0, DONE, NONE, 0x000, 0, 0, 0x0000
5	LLC_2	0xFF00, 0x0300, EQ, 2, IPV4_1, 0, DONE, NONE, 0x000, 0, 0, 0x0000
6	IPV4_1	0xFFFF, 0x0800, EQ, 1, IPV4_2, 0, IPV6_1, LD_SAP, 0x100, 3, 0, 0xFFFF
7	IPV4_2	0xFF00, 0x4500, EQ, 3, IPV4_3, 0, DONE, LD_SUM, 0x00A, 1, 0, 0x0000
8	IPV4_3	0x3FFF, 0x0000, EQ, 1, IPV4_4, 0, DONE, LD_LEN, 0x03E, 1, 0, 0xFFFF
9	IPV4_4	0x00FF, 0x0006, EQ, 7, TCP_1, 0, DONE, LD_FID, 0x182, 1, 0, 0xFFFF
10	IPV6_1	0xFFFF, 0x86DD, EQ, 1, IPV6_2, 0, DONE, LD_SUM, 0x015, 1, 0x0000
11	IPV6_2	0xF000, 0x6000, EQ, 0, IPV6_3, 0, DONE, IM_R1, 0x114, 1, 0, 0xFFFF
12	IPV6_3	0x0000, 0x0000, EQ, 3, IPV6_4, 0, DONE, LD_FID, 0x484, 1, 0, 0xFFFF
13	IPV6_4	0xFF00, 0x0600, EQ, 18, TCP_1, 0, DONE, LD_LEN, 0x03F, 1, 0xFFFF
14	TCP_1	0x0000, 0x0000, EQ, 0, TCP_2, 4, TCP_2, LD_SEQ, 0x081, 3, 0, 0xFFFF
15	TCP_2	0x0000, 0x0000, EQ, 0, TCP_3, 0, TCP_3, ST_FLAG, 0x145, 3, 0, 0x002F
16	TCP_3	0x0000, 0x0000, EQ, 0, TCP_4, 0, TCP_4, LD_R1, 0x205, 3, 0xB, 0xF000
17	TCP_4	0x0000, 0x0000, EQ, 0, WAIT, 0, WAIT, LD_HDR, 0x0FF, 3, 0, 0xFFFF
18	DONE	0x0000, 0x0000, EQ, 0, WAIT, 0, WAIT, IM_CTL, 0x001, 3, 0x0000

PROGRAM 2300

FIG. 23

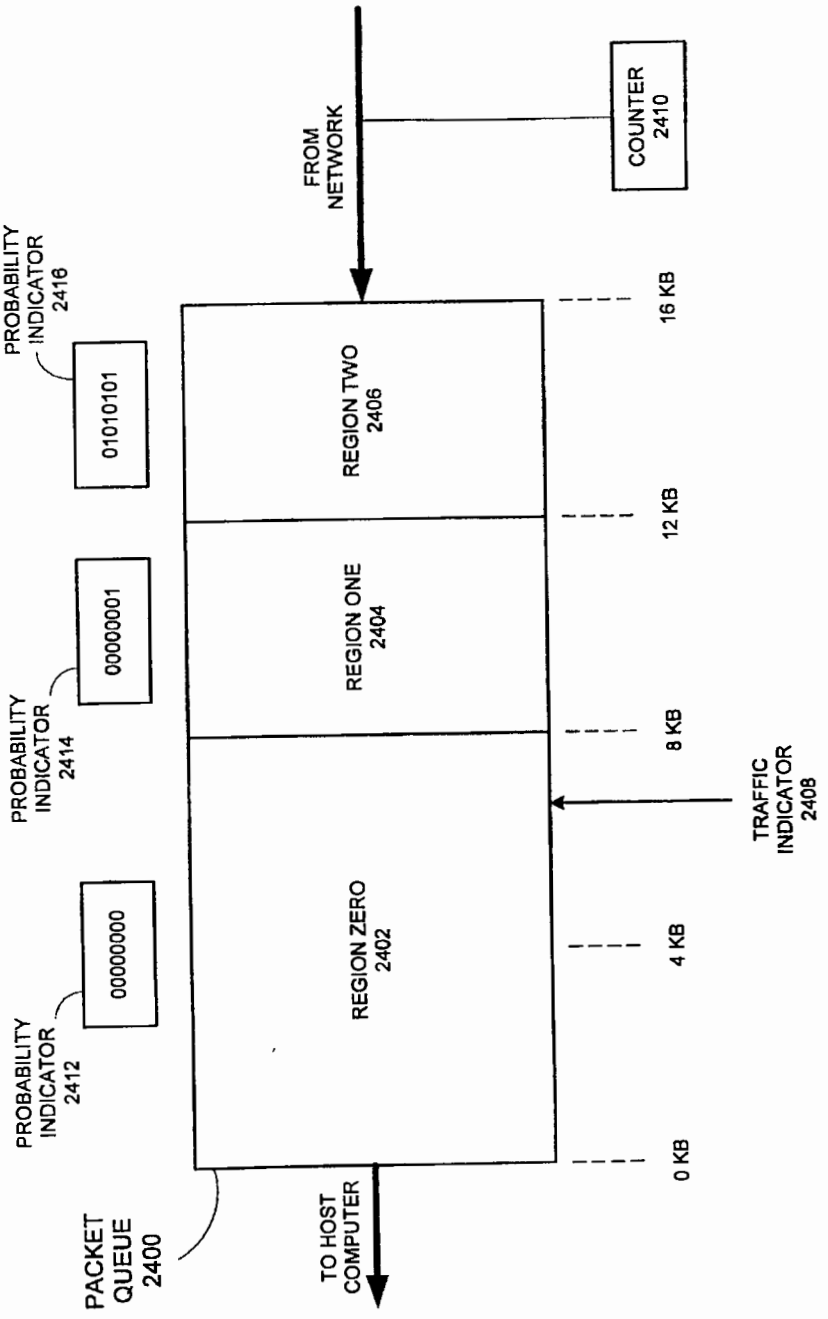

FROM NETWORK
COUNTER 2410
PROBABILITY INDICATOR 2416
01010101
REGION TWO 2406
16 KB
12 KB
PROBABILITY INDICATOR 2414
00000001
REGION ONE 2404
8 KB
TRAFFIC INDICATOR 2408
PROBABILITY INDICATOR 2412
00000000
REGION ZERO 2402
4 KB
0 KB
PACKET QUEUE 2400
TO HOST COMPUTER

FIG. 24

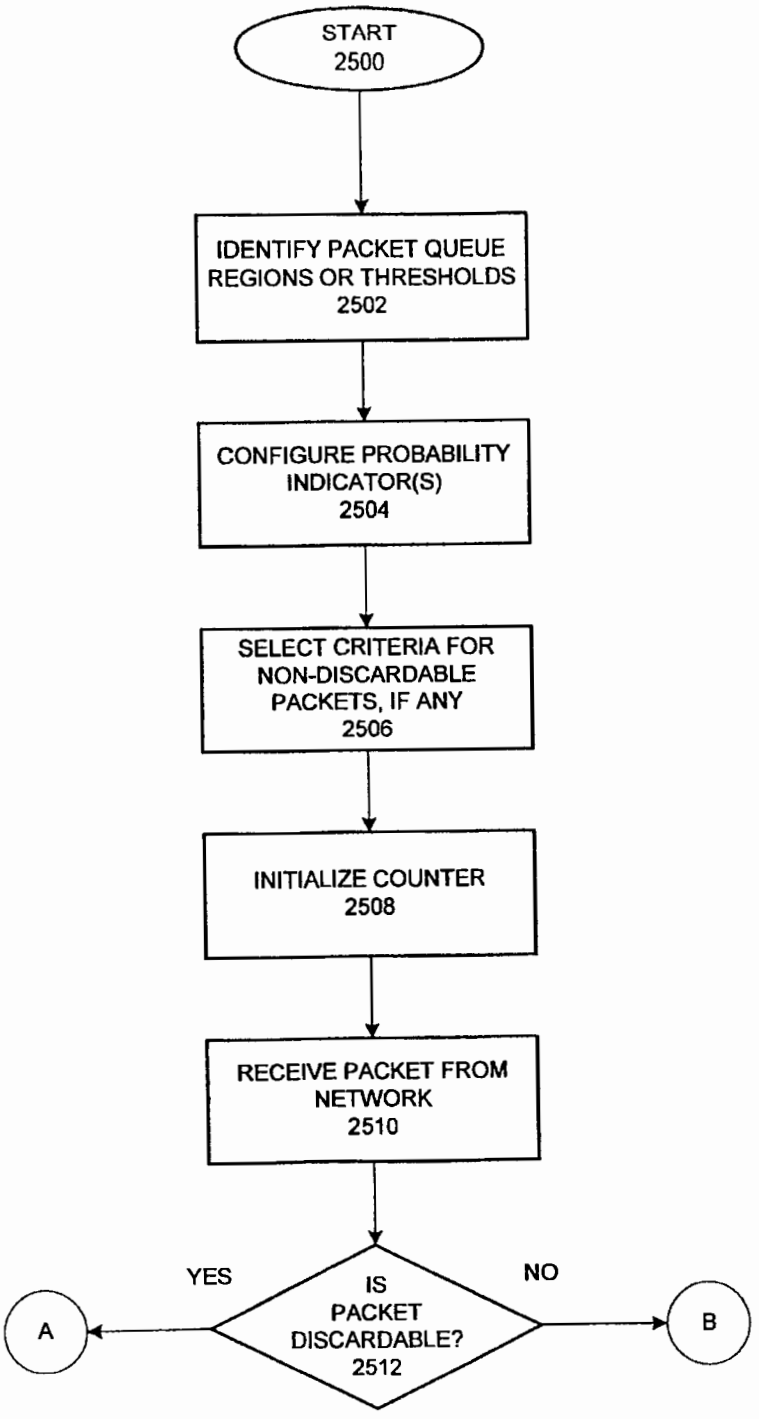

START
2500
IDENTIFY PACKET QUEUE REGIONS OR THRESHOLDS
2502
CONFIGURE PROBABILITY INDICATOR(S)
2504
SELECT CRITERIA FOR NON-DISCARDABLE PACKETS, IF ANY
2506
INITIALIZE COUNTER
2508
RECEIVE PACKET FROM NETWORK
2510
IS PACKET DISCARDABLE?
2512
YES
NO
A
B

FIG. 25A

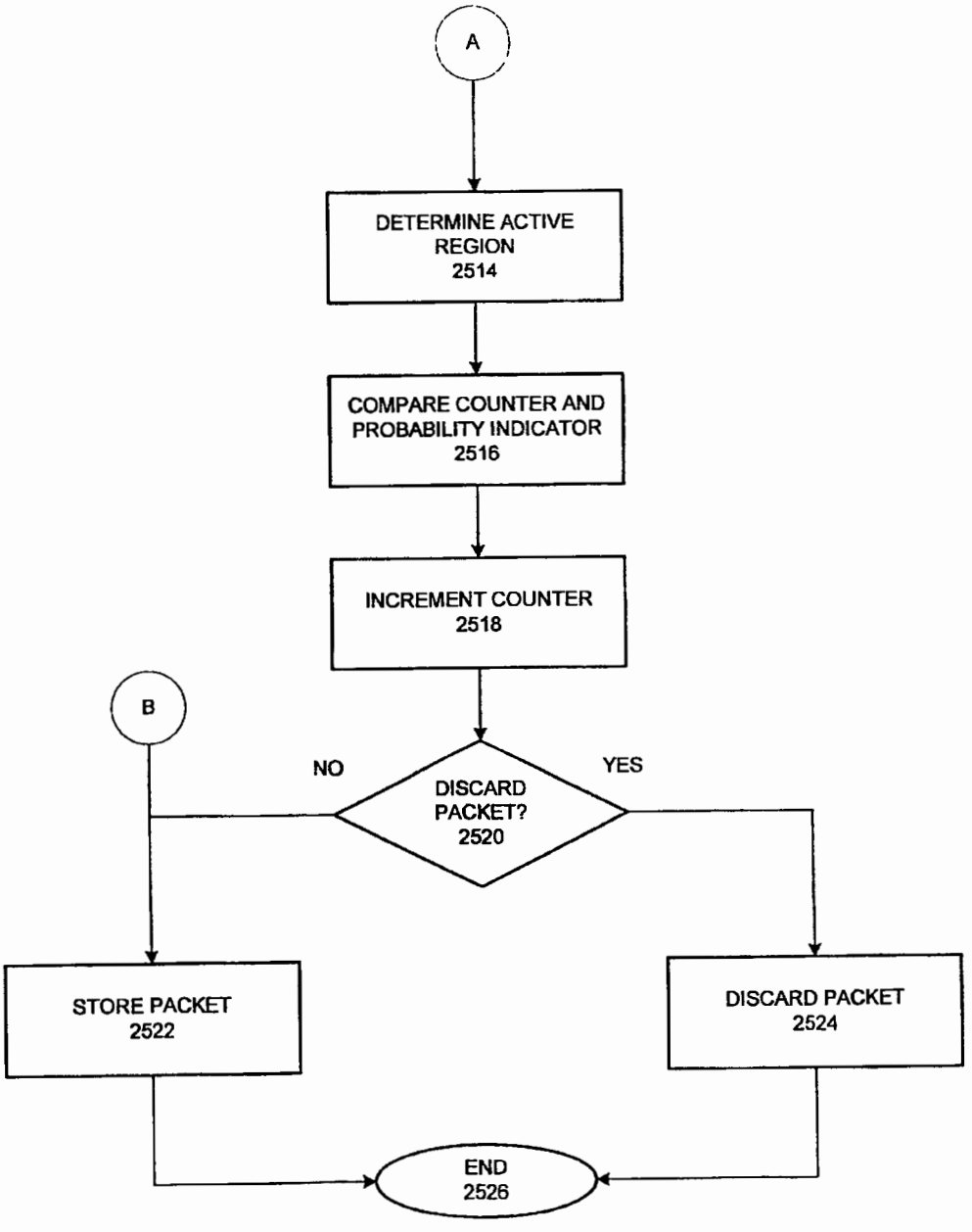

A
DETERMINE ACTIVE REGION 2514
COMPARE COUNTER AND PROBABILITY INDICATOR 2516
INCREMENT COUNTER 2518
B
NO
DISCARD PACKET? 2520
YES
STORE PACKET 2522
DISCARD PACKET 2524
END 2526

FIG. 25B

METHOD AND APPARATUS FOR DYNAMIC PACKET BATCHING WITH A HIGH PERFORMANCE NETWORK INTERFACE

TABLE OF CONTENTS

BACKGROUND
SUMMARY
BRIEF DESCRIPTION OF THE FIGURES
DETAILED DESCRIPTION
Introduction
One Embodiment of a High Performance Network Interface Circuit
An Illustrative Packet
One Embodiment of a Header Parser
Dynamic Header Parsing Instructions in One Embodiment of the Invention
One Embodiment of a Flow Database
One Embodiment of a Flow Database Manager
One Embodiment of a Load Distributor
One Embodiment of a Packet Queue
One Embodiment of a Control Queue
One Embodiment of a DMA Engine
Methods of Transferring a Packet Into a Memory Buffer by a DMA Engine
A Method of Transferring a Packet with Operation Code 0
A Method of Transferring a Packet with Operation Code 1
A Method of Transferring a Packet with Operation Code 2
A Method of Transferring a Packet with Operation Code 3
A Method of Transferring a Packet with Operation Code 4
A Method of Transferring a Packet with Operation Code 5
A Method of Transferring a Packet with Operation Code 6 or 7
One Embodiment of a Dynamic Packet Batching Module
Early Random Packet Discard in One Embodiment of the Invention
CLAIMS

BACKGROUND

This invention relates to the fields of computer systems and computer networks. In particular, the present invention relates to a Network Interface Circuit (NIC) for processing communication packets exchanged between a computer network and a host computer system.

The interface between a computer and a network is often a bottleneck for communications passing between the computer and the network. While computer performance (e.g., processor speed) has increased exponentially over the years and computer network transmission speeds have undergone similar increases, inefficiencies in the way network interface circuits handle communications have become more and more evident. With each incremental increase in computer or network speed, it becomes ever more apparent that the interface between the computer and the network cannot keep pace. These inefficiencies involve several basic problems in the way communications between a network and a computer are handled.

Today's most popular forms of networks tend to be packet-based. These types of networks, including the Internet and many local area networks, transmit information in the form of packets. Each packet is separately created and transmitted by an originating endstation and is separately received and processed by a destination endstation. In addition, each packet may, in a bus topology network for example, be received and processed by numerous stations located between the originating and destination endstations.

One basic problem with packet networks is that each packet must be processed through multiple protocols or protocol levels (known collectively as a "protocol stack") on both the origination and destination endstations. When data transmitted between stations is longer than a certain minimal length, the data is divided into multiple portions, and each portion is carried by a separate packet. The amount of data that a packet can carry is generally limited by the network that conveys the packet and is often expressed as a maximum transfer unit (MTU). The original aggregation of data is sometimes known as a "datagram," and each packet carrying part of a single datagram is processed very similarly to the other packets of the datagram.

Communication packets are generally processed as follows. In the origination endstation, each separate data portion of a datagram is processed through a protocol stack. During this processing multiple protocol headers (e.g., TCP, IP, Ethernet) are added to the data portion to form a packet that can be transmitted across the network. The packet is received by a network interface circuit, which transfers the packet to the destination endstation or a host computer that serves the destination endstation. In the destination endstation, the packet is processed through the protocol stack in the opposite direction as in the origination endstation. During this processing the protocol headers are removed in the opposite order in which they were applied. The data portion is thus recovered and can be made available to a user, an application program, etc.

Several related packets (e.g., packets carrying data from one datagram) thus undergo substantially the same process in a serial manner (i.e., one packet at a time). The more data that must be transmitted, the more packets must be sent, with each one being separately handled and processed through the protocol stack in each direction. Naturally, the more packets that must be processed, the greater the demand placed upon an endstation's processor. The number of packets that must be processed is affected by factors other than just the amount of data being sent in a datagram. For example, as the amount of data that can be encapsulated in a packet increases, fewer packets need to be sent. As stated above, however, a packet may have a maximum allowable size, depending on the type of network in use (e.g., the maximum transfer unit for standard Ethernet traffic is approximately 1,500 bytes). The speed of the network also affects the number of packets that a NIC may handle in a given period of time. For example, a gigabit Ethernet network operating at peak capacity may require a NIC to receive approximately 1.48 million packets per second. Thus, the number of packets to be processed through a protocol stack may place a significant burden upon a computer's processor. The situation is exacerbated by the need to process each packet separately even though each one will be processed in a substantially similar manner.

A related problem to the disjoint processing of packets is the manner in which data is moved between "user space" (e.g., an application program's data storage) and "system space" (e.g., system memory) during data transmission and receipt. Presently, data is simply copied from one area of

memory assigned to a user or application program into another area of memory dedicated to the processor's use. Because each portion of a datagram that is transmitted in a packet may be copied separately (e.g., one byte at a time), there is a nontrivial amount of processor time required and frequent transfers can consume a large amount of the memory bus' bandwidth. Illustratively, each byte of data in a packet received from the network may be read from the system space and written to the user space in a separate copy operation, and vice versa for data transmitted over the network. Although system space generally provides a protected memory area (e.g., protected from manipulation by user programs), the copy operation does nothing of value when seen from the point of view of a network interface circuit. Instead, it risks over-burdening the host processor and retarding its ability to rapidly accept additional network traffic from the NIC. Copying each packet's data separately can therefore be very inefficient, particularly in a high-speed network environment.

In addition to the inefficient transfer of data (e.g., one packet's data at a time), the processing of headers from packets received from a network is also inefficient. Each packet carrying part of a single datagram generally has the same protocol headers (e.g., Ethernet, IP and TCP), although there may be some variation in the values within the packets' headers for a particular protocol. Each packet, however, is individually processed through the same protocol stack, thus requiring multiple repetitions of identical operations for related packets. Successively processing unrelated packets through different protocol stacks will likely be much less efficient than progressively processing a number of related packets through one protocol stack at a time.

Another basic problem concerning the interaction between present network interface circuits and host computer systems is that the combination often fails to capitalize on the increased processor resources that are available in multi-processor computer systems. In other words, present attempts to distribute the processing of network packets (e.g., through a protocol stack) among a number of protocols in an efficient manner are generally ineffective. In particular, the performance of present NICs does not come close to the expected or desired linear performance gains one may expect to realize from the availability of multiple processors. In some multi-processor systems, little improvement in the processing of network traffic is realized from the use of more than 4–6 processors, for example.

In addition, the rate at which packets are transferred from a network interface circuit to a host computer or other communication device may fail to keep pace with the rate of packet arrival at the network interface. One element or another of the host computer (e.g., a memory bus, a processor) may be over-burdened or otherwise unable to accept packets with sufficient alacrity. In this event one or more packets may be dropped or discarded. Dropping packets may cause a network entity to re-transmit some traffic and, if too many packets are dropped, a network connection may require re-initialization. Further, dropping one packet or type of packet instead of another may make a significant difference in overall network traffic. If, for example, a control packet is dropped, the corresponding network connection may be severely affected and may do little to alleviate the packet saturation of the network interface circuit because of the typically small size of a control packet. Therefore, unless the dropping of packets is performed in a manner that distributes the effect among many network connections or that makes allowance for certain types of packets, network traffic may be degraded more than necessary.

Thus, present NICs fail to provide adequate performance to interconnect today's high-end computer systems and high-speed networks. In addition, a network interface circuit that cannot make allowance for an over-burdened host computer may degrade the computer's performance.

SUMMARY

In one embodiment of the invention a system and method are provided for identifying a packet within a particular communication flow through a communication device such as a network interface. In particular, the communication flow may include a first packet transferred from the network interface to a host computer. Based on an identifier of the flow, another packet in the same flow may be identified to the host computer. To increase the efficiency of handling network traffic, the flow packets may then be collectively processed through a protocol stack on a host computer.

In this embodiment, a high performance network interface of a host computer receives a packet from a network. Information within a header portion of the packet is assembled to generate a flow key to identify the communication flow, connection or circuit that includes the packet. Illustratively, the flow key includes identifiers of the source and destination entities that are exchanging the packet. In one embodiment of the invention flow keys from one or more communication flows are stored in a flow database, which is indexed by a flow number and which may be managed by a flow database management module. If the database does not already include the flow key of the received packet, then the received packet's communication flow may be a new flow at the network interface. In this case the flow is registered in the database by storing its flow key and, possibly, other information concerning the flow. Thus, a packet's flow may be identified by its flow key and/or its flow number.

The packet is stored in a packet memory (e.g., a queue) to await transfer to the host computer, and the packet's flow number is stored in a flow memory of a dynamic packet batching module. When the packet is transferred or is about to be transferred, the flow memory is searched to determine whether another packet stored in the packet memory is part of the same communication flow (e.g., has the same flow number or flow key).

In this embodiment, if another packet has the same flow number then the host computer is alerted by storing an indicator in a host memory, such as a descriptor. In another embodiment of the invention, if no other packet is found with the same flow number then a different indicator is stored in a host memory. A different indicator may be stored, for example, if the packet is determined to be the last packet of its communication flow. Depending on the indicator that is stored, the host computer may delay processing the packet to await another packet having the same flow number.

The dynamic packet batching module also includes a controller in a present embodiment of the invention. The controller attempts to populate the flow memory with information associated with or derived from packets stored in the packet memory. Illustratively, each entry in the flow memory in this embodiment stores a packet's flow number and an indicator of whether the entry is valid. An entry may be invalidated when its packet is transferred to the host computer, at which time it may be replaced with another entry.

In one embodiment of the invention, only packets that conform to one or more of a set of pre-selected protocols are eligible for dynamic packet batching. In this embodiment, a

header parser module may be configured to determine whether a received packet is formatted in accordance with one of the protocols. If compatible with the pre-selected protocols, the received packet may also receive the benefit of other processing efficiencies, such as re-assembling data from multiple packets in one flow or distributing the processing of packets among processors in a multi-processor system.

BRIEF DESCRIPTION OF THE FIGURES

FIG. **1A** is a block diagram depicting a network interface circuit (NIC) for receiving a packet from a network in accordance with an embodiment of the present invention.

FIG. **1B** is a flow chart demonstrating one method of operating the NIC of FIG. **1A** to transfer a packet received from a network to a host computer in accordance with an embodiment of the invention.

FIG. **2** is a diagram of a packet transmitted over a network and received at a network interface circuit in one embodiment of the invention.

FIG. **3** is a block diagram depicting a header parser of a network interface circuit for parsing a packet in accordance with an embodiment of the invention.

FIGS. **4A–4B** comprise a flow chart demonstrating one method of parsing a packet received from a network at a network interface circuit in accordance with an embodiment of the present invention.

FIG. **5** is a block diagram depicting a network interface circuit flow database in accordance with an embodiment of the invention.

FIGS. **6A–6E** comprise a flowchart illustrating one method of managing a network interface circuit flow database in accordance with an embodiment of the invention.

FIG. **7** is a flow chart demonstrating one method of distributing the processing of network packets among multiple processors on a host computer in accordance with an embodiment of the invention.

FIG. **8** is a diagram of a packet queue for a network interface circuit in accordance with an embodiment of the invention.

FIG. **9** is a diagram of a control queue for a network interface circuit in accordance with an embodiment of the invention.

FIG. **10** is a block diagram of a DMA engine for transferring a packet received from a network to a host computer in accordance with an embodiment of the invention.

FIG. **11** includes diagrams of data structures for managing the storage of network packets in host memory buffers in accordance with an embodiment of the invention.

FIGS. **12A–12B** are diagrams of a free descriptor, a completion descriptor and a free buffer array in accordance with an embodiment of the invention.

FIGS. **13–20** are flow charts demonstrating methods of transferring a packet received from a network to a buffer in a host computer memory in accordance with an embodiment of the invention.

FIG. **21** is a diagram of a dynamic packet batching module in accordance with an embodiment of the invention.

FIGS. **22A–22B** comprise a flow chart demonstrating one method of dynamically searching a memory containing information concerning packets awaiting transfer to a host computer in order to locate a packet in the same communication flow as a packet being transferred, in accordance with an embodiment of the invention.

FIG. **23** depicts one set of dynamic instructions for parsing a packet in accordance with an embodiment of the invention.

FIG. **24** depicts a system for randomly discarding a packet from a network interface in accordance with an embodiment of the invention.

FIGS. **25A–25B** comprise a flow chart demonstrating one method of discarding a packet from a network interface in accordance with an embodiment of the invention.

DETAILED DESCRIPTION

The following description is presented to enable any person skilled in the art to make and use the invention, and is provided in the context of particular applications of the invention and their requirements. Various modifications to the disclosed embodiments will be readily apparent to those skilled in the art and the general principles defined herein may be applied to other embodiments and applications without departing from the spirit and scope of the present invention. Thus, the present invention is not intended to be limited to the embodiments shown, but is to be accorded the widest scope consistent with the principles and features disclosed herein.

In particular, embodiments of the invention are described below in the form of a network interface circuit (NIC) receiving communication packets formatted in accordance with certain communication protocols compatible with the Internet. One skilled in the art will recognize, however, that the present invention is not limited to communication protocols compatible with the Internet and may be readily adapted for use with other protocols and in communication devices other than a NIC.

The program environment in which a present embodiment of the invention is executed illustratively incorporates a general-purpose computer or a special purpose device such a hand-held computer. Details of such devices (e.g., processor, memory, data storage, input/output ports and display) are well known and are omitted for the sake of clarity.

It should also be understood that the techniques of the present invention might be implemented using a variety of technologies. For example, the methods described herein may be implemented in software running on a programmable microprocessor, or implemented in hardware utilizing either a combination of microprocessors or other specially designed application specific integrated circuits, programmable logic devices, or various combinations thereof. In particular, the methods described herein may be implemented by a series of computer-executable instructions residing on a storage medium such as a carrier wave, disk drive, or other computer-readable medium.

Introduction

In one embodiment of the present invention, a network interface circuit (NIC) is configured to receive and process communication packets exchanged between a host computer system and a network such as the Internet. In particular, the NIC is configured to receive and manipulate packets formatted in accordance with a protocol stack (e.g., a combination of communication protocols) supported by a network coupled to the NIC.

A protocol stack may be described with reference to the seven-layer ISO-OSI (International Standards Organization—Open Systems Interconnection) model framework. Thus, one illustrative protocol stack includes the Transport Control Protocol (TCP) at layer four, Internet Protocol (IP) at layer three and Ethernet at layer two. For

purposes of discussion, the term "Ethernet" may be used herein to refer collectively to the standardized IEEE (Institute of Electrical and Electronics Engineers) 802.3 specification as well as version two of the non-standardized form of the protocol. Where different forms of the protocol need to be distinguished, the standard form may be identified by including the "802.3" designation.

Other embodiments of the invention are configured to work with communications adhering to other protocols, both known (e.g., AppleTalk, IPX (Internetwork Packet Exchange), etc.) and unknown at the present time. One skilled in the art will recognize that the methods provided by this invention are easily adaptable for new communication protocols.

In addition, the processing of packets described below may be performed on communication devices other than a NIC. For example, a modem, switch, router or other communication port or device (e.g., serial, parallel, USB, SCSI) may be similarly configured and operated.

In embodiments of the invention described below, a NIC receives a packet from a network on behalf of a host computer system or other communication device. The NIC analyzes the packet (e.g., by retrieving certain fields from one or more of its protocol headers) and takes action to increase the efficiency with which the packet is transferred or provided to its destination entity. Equipment and methods discussed below for increasing the efficiency of processing or transferring packets received from a network may also be used for packets moving in the reverse direction (i.e., from the NIC to the network).

One technique that may be applied to incoming network traffic involves examining or parsing one or more headers of an incoming packet (e.g., headers for the layer two, three and four protocols) in order to identify the packet's source and destination entities and possibly retrieve certain other information. Using identifiers of the communicating entities as a key, data from multiple packets may be aggregated or re-assembled. Typically, a datagram sent to one destination entity from one source entity is transmitted via multiple packets. Aggregating data from multiple related packets (e.g., packets carrying data from the same datagram) thus allows a datagram to be re-assembled and collectively transferred to a host computer. The datagram may then be provided to the destination entity in a highly efficient manner. For example, rather than providing data from one packet at a time (and one byte at a time) in separate "copy" operations, a "page-flip" operation may be performed. In a page-flip, an entire memory page of data may be provided to the destination entity, possibly in exchange for an empty or unused page.

In another technique, packets received from a network are placed in a queue to await transfer to a host computer. While awaiting transfer, multiple related packets may be identified to the host computer. After being transferred, they may be processed as a group by a host processor rather than being processed serially (e.g., one at a time).

Yet another technique involves submitting a number of related packets to a single processor of a multi-processor host computer system. By distributing packets conveyed between different pairs of source and destination entities among different processors, the processing of packets through their respective protocol stacks can be distributed while still maintaining packets in their correct order.

The techniques discussed above for increasing the efficiency with which packets are processed may involve a combination of hardware and software modules located on a network interface and/or a host computer system. In one particular embodiment, a parsing module on a host computer's NIC parses header portions of packets. Illustratively, the parsing module comprises a microsequencer operating according to a set of replaceable instructions stored as micro-code. Using information extracted from the packets, multiple packets from one source entity to one destination entity may be identified. A hardware re-assembly module on the NIC may then gather the data from the multiple packets. Another hardware module on the NIC is configured to recognize related packets awaiting transfer to the host computer so that they may be processed through an appropriate protocol stack collectively, rather than serially. The re-assembled data and the packet's headers may then be provided to the host computer so that appropriate software (e.g., a device driver for the NIC) may process the headers and deliver the data to the destination entity.

Where the host computer includes multiple processors, a load distributor (which may also be implemented in hardware on the NIC) may select a processor to process the headers of the multiple packets through a protocol stack.

In another embodiment of the invention, a system is provided for randomly discarding a packet from a NIC when the NIC is saturated or nearly saturated with packets awaiting transfer to a host computer.

One Embodiment of a High Performance Network Interface Circuit

FIG. 1A depicts NIC **100** configured in accordance with an illustrative embodiment of the invention. A brief description of the operation and interaction of the various modules of NIC **100** in this embodiment follows. Descriptions incorporating much greater detail are provided in subsequent sections.

A communication packet may be received at NIC **100** from network **102** by a medium access control (MAC) module (not shown in FIG. 1A). The MAC module performs low-level processing of the packet such as reading the packet from the network, performing some error checking, detecting packet fragments, detecting over-sized packets, removing the layer one preamble, etc.

Input Port Processing (IPP) module **104** then receives the packet. The IPP module stores the entire packet in packet queue **116**, as received from the MAC module or network, and a portion of the packet is copied into header parser **106**. In one embodiment of the invention IPP module **104** may act as a coordinator of sorts to prepare the packet for transfer to a host computer system. In such a role, IPP module **104** may receive information concerning a packet from various modules of NIC **100** and dispatch such information to other modules.

Header parser **106** parses a header portion of the packet to retrieve various pieces of information that will be used to identify related packets (e.g., multiple packets from one same source entity for one destination entity) and that will affect subsequent processing of the packets. In the illustrated embodiment, header parser **106** communicates with flow database manager (FDBM) **108**, which manages flow database (FDB) **110**. In particular, header parser **106** submits a query to FDBM **108** to determine whether a valid communication flow (described below) exists between the source entity that sent a packet and the destination entity. The destination entity may comprise an application program, a communication module, or some other element of a host computer system that is to receive the packet.

In the illustrated embodiment of the invention, a communication flow comprises one or more datagram packets from one source entity to one destination entity. A flow may be identified by a flow key assembled from source and desti-

nation identifiers retrieved from the packet by header parser **106**. In one embodiment of the invention a flow key comprises address and/or port information for the source and destination entities from the packet's layer three (e.g., IP) and/or layer four (e.g., TCP) protocol headers.

For purposes of the illustrated embodiment of the invention, a communication flow is similar to a TCP end-to-end connection but is generally shorter in duration. In particular, in this embodiment the duration of a flow may be limited to the time needed to receive all of the packets associated with a single datagram passed from the source entity to the destination entity.

Thus, for purposes of flow management, header parser **106** passes the packet's flow key to flow database manager **108**. The header parser may also provide the flow database manager with other information concerning the packet that was retrieved from the packet (e.g., length of the packet).

Flow database manager **108** searches FDB **110** in response to a query received from header parser **106**. Illustratively, flow database **110** stores information concerning each valid communication flow involving a destination entity served by NIC **100**. Thus, FDBM **108** updates FDB **110** as necessary, depending upon the information received from header parser **106**. In addition, in this embodiment of the invention FDBM **108** associates an operation or action code with the received packet. An operation code may be used to identify whether a packet is part of a new or existing flow, whether the packet includes data or just control information, the amount of data within the packet, whether the packet data can be re-assembled with related data (e.g., other data in a datagram sent from the source entity to the destination entity), etc. FDBM **108** may use information retrieved from the packet and provided by header parser **106** to select an appropriate operation code. The packet's operation code is then passed back to the header parser, along with an index of the packet's flow within FDB **110**.

In one embodiment of the invention the combination of header parser **106**, FDBM **108** and FDB **110**, or a subset of these modules, may be known as a traffic classifier due to their role in classifying or identifying network traffic received at NIC **100**.

In the illustrated embodiment, header parser **106** also passes the packet's flow key to load distributor **112**. In a host computer system having multiple processors, load distributor **112** may determine which processor an incoming packet is to be routed to for processing through the appropriate protocol stack. For example, load distributor **112** may ensure that related packets are routed to a single processor. By sending all packets in one communication flow or end-to-end connection to a single processor, the correct ordering of packets can be enforced. Load distributor **112** may be omitted in one alternative embodiment of the invention. In another alternative embodiment, header parser **106** may also communicate directly with other modules of NIC **100** besides the load distributor and flow database manager.

Thus, after header parser **106** parses a packet FDBM **108** alters or updates FDB **110** and load distributor **112** identifies a processor in the host computer system to process the packet. After these actions, the header parser passes various information back to IPP module **104**. Illustratively, this information may include the packet's flow key, an index of the packet's flow within flow database **110**, an identifier of a processor in the host computer system, and various other data concerning the packet (e.g., its length, a length of a packet header).

Now the packet may be stored in packet queue **116**, which holds packets for manipulation by DMA (Direct Memory Access) engine **120** and transfer to a host computer. In addition to storing the packet in a packet queue, a corresponding entry for the packet is made in control queue **118** and information concerning the packet's flow may also be passed to dynamic packet batching module **122**. Control queue **118** contains related control information for each packet in packet queue **116**.

Packet batching module **122** draws upon information concerning packets in packet queue **116** to enable the batch (i.e., collective) processing of headers from multiple related packets. In one embodiment of the invention packet batching module **122** alerts the host computer to the availability of headers from related packets so that they may be processed together.

Although the processing of a packet's protocol headers is performed by a processor on a host computer system in one embodiment of the invention, in another embodiment the protocol headers may be processed by a processor located on NIC **100**. In the former embodiment, software on the host computer (e.g., a device driver for NIC **100**) can reap the advantages of additional memory and a replaceable or upgradeable processor (e.g., the memory may be supplemented and the processor may be replaced by a faster model).

During the storage of a packet in packet queue **116**, checksum generator **114** may perform a checksum operation. The checksum may be added to the packet queue as a trailer to the packet. Illustratively, checksum generator **114** generates a checksum from a portion of the packet received from network **102**. In one embodiment of the invention, a checksum is generated from the TCP portion of a packet (e.g., the TCP header and data). If a packet is not formatted according to TCP, a checksum may be generated on another portion of the packet and the result may be adjusted in later processing as necessary. For example, if the checksum calculated by checksum generator **114** was not calculated on the correct portion of the packet, the checksum may be adjusted to capture the correct portion. This adjustment may be made by software operating on a host computer system (e.g., a device driver). Checksum generator **114** may be omitted or merged into another module of NIC **100** in an alternative embodiment of the invention.

From the information obtained by header parser **106** and the flow information managed by flow database manager **108**, the host computer system served by NIC **100** in the illustrated embodiment is able to process network traffic very efficiently. For example, data portions of related packets may be re-assembled by DMA engine **120** to form aggregations that can be more efficiently manipulated. And, by assembling the data into buffers the size of a memory page, the data can be more efficiently transferred to a destination entity through "page-flipping," in which an entire memory page filled by DMA engine **120** is provided at once. One page-flip can thus take the place of multiple copy operations. Meanwhile, the header portions of the re-assembled packets may similarly be processed as a group through their appropriate protocol stack.

As already described, in another embodiment of the invention the processing of network traffic through appropriate protocol stacks may be efficiently distributed in a multi-processor host computer system. In this embodiment, load distributor **112** assigns or distributes related packets (e.g., packets in the same communication flow) to the same processor. In particular, packets having the same source and destination addresses in their layer three protocol (e.g., IP) headers and/or the same source and destination ports in their layer four protocol (e.g., TCP) headers may be sent to a single processor.

In the NIC illustrated in FIG. 1A, the processing enhancements discussed above (e.g., re-assembling data, batch processing packet headers, distributing protocol stack processing) are possible for packets received from network **102** that are formatted according to one or more pre-selected protocol stacks. In this embodiment of the invention network **102** is the Internet and NIC **100** is therefore configured to process packets using one of several protocol stacks compatible with the Internet. Packets not configured according to the pre-selected protocols are also processed, but may not receive the benefits of the full suite of processing efficiencies provided to packets meeting the pre-selected protocols.

For example, packets not matching one of the pre-selected protocol stacks may be distributed for processing in a multi-processor system on the basis of the packets' layer two (e.g., medium access control) source and destination addresses rather than their layer three or layer four addresses. Using layer two identifiers provides less granularity to the load distribution procedure, thus possibly distributing the processing of packets less evenly than if layer three/four identifiers were used.

FIG. 1B depicts one method of using NIC **100** of FIG. 1A to receive one packet from network **102** and transfer it to a host computer. State **130** is a start state, possibly characterized by the initialization or resetting of NIC **100**.

In state **132**, a packet is received by NIC **100** from network **102**. As already described, the packet may be formatted according to a variety of communication protocols. The packet may be received and initially manipulated by a MAC module before being passed to an IPP module.

In state **134**, a portion of the packet is copied and passed to header parser **106**. Header parser **106** then parses the packet to extract values from one or more of its headers and/or its data. A flow key is generated from some of the retrieved information to identify the communication flow that includes the packet. The degree or extent to which the packet is parsed may depend upon its protocols, in that the header parser may be configured to parse headers of different protocols to different depths. In particular, header parser **106** may be optimized (e.g., its operating instructions configured) for a specific set of protocols or protocol stacks. If the packet conforms to one or more of the specified protocols it may be parsed more fully than a packet that does not adhere to any of the protocols.

In state **136**, information extracted from the packet's headers is forwarded to flow database manager **108** and/or load distributor **112**. The FDBM uses the information to set up a flow in flow database **110** if one does not already exist for this communication flow. If an entry already exists for the packet's flow, it may be updated to reflect the receipt of a new flow packet. Further, FDBM **108** generates an operation code to summarize one or more characteristics or conditions of the packet. The operation code may be used by other modules of NIC **100** to handle the packet in an appropriate manner, as described in subsequent sections. The operation code is returned to the header parser, along with an index (e.g., a flow number) of the packet's flow in the flow database.

In state **138**, load distributor **112** assigns a processor number to the packet, if the host computer includes multiple processors, and returns the processor number to the header processor. Illustratively, the processor number identifies which processor is to conduct the packet through its protocol stack on the host computer. State **138** may be omitted in an alternative embodiment of the invention, particularly if the host computer consists of only a single processor.

In state **140**, the packet is stored in packet queue **116**. As the contents of the packet are placed into the packet queue, checksum generator **114** may compute a checksum. The checksum generator may be informed by IPP module **104** as to which portion of the packet to compute the checksum on. The computed checksum is added to the packet queue as a trailer to the packet. In one embodiment of the invention, the packet is stored in the packet queue at substantially the same time that a copy of a header portion of the packet is provided to header parser **106**.

Also in state **140**, control information for the packet is stored in control queue **118** and information concerning the packet's flow (e.g., flow number, flow key) may be provided to dynamic packet batching module **122**.

In state **142**, NIC **100** determines whether the packet is ready to be transferred to host computer memory. Until it is ready to be transferred, the illustrated procedure waits.

When the packet is ready to be transferred (e.g., the packet is at the head of the packet queue or the host computer receives the packet ahead of this packet in the packet queue), in state **144** dynamic packet batching module **122** determines whether a related packet will soon be transferred. If so, then when the present packet is transferred to host memory the host computer is alerted that a related packet will soon follow. The host computer may then process the packets (e.g., through their protocol stack) as a group.

In state **146**, the packet is transferred (e.g., via a direct memory access operation) to host computer memory. And, in state **148**, the host computer is notified that the packet was transferred. The illustrated procedure then ends at state **150**.

One skilled in the art of computer systems and networking will recognize that the procedure described above is just one method of employing the modules of NIC **100** to receive a single packet from a network and transfer it to a host computer system. Other suitable methods are also contemplated within the scope of the invention.

An Illustrative Packet

FIG. **2** is a diagram of an illustrative packet received by NIC **100** from network **102**. Packet **200** comprises data portion **202** and header portion **204**, and may also contain trailer portion **206**. Depending upon the network environment traversed by packet **200**, its maximum size (e.g., its maximum transfer unit or MTU) may be limited.

In the illustrated embodiment, data portion **202** comprises data being provided to a destination or receiving entity within a computer system (e.g., user, application program, operating system) or a communication subsystem of the computer. Header portion **204** comprises one or more headers prefixed to the data portion by the source or originating entity or a computer system comprising the source entity. Each header normally corresponds to a different communication protocol.

In a typical network environment, such as the Internet, individual headers within header portion **204** are attached (e.g., prepended) as the packet is processed through different layers of a protocol stack (e.g., a set of protocols for communicating between entities) on the transmitting computer system. For example, FIG. **2** depicts protocol headers **210**, **212**, **214** and **216**, corresponding to layers one through four, respectively, of a suitable protocol stack. Each protocol header contains information to be used by the receiving computer system as the packet is received and processed through the protocol stack. Ultimately, each protocol header is removed and data portion **202** is retrieved.

As described in other sections, in one embodiment of the invention a system and method are provided for parsing

packet **200** to retrieve various bits of information. In this embodiment, packet **200** is parsed in order to identify the beginning of data portion **202** and to retrieve one or more values for fields within header portion **204**. Illustratively, however, layer one protocol header or preamble **210** corresponds to a hardware-level specification related to the coding of individual bits. Layer one protocols are generally only needed for the physical process of sending or receiving the packet across a conductor. Thus, in this embodiment of the invention layer one preamble **210** is stripped from packet **200** shortly after being received by NIC **100** and is therefore not parsed.

The extent to which header portion **204** is parsed may depend upon how many, if any, of the protocols represented in the header portion match a set of pre-selected protocols. For example, the parsing procedure may be abbreviated or aborted once it is determined that one of the packet's headers corresponds to an unsupported protocol.

In particular, in one embodiment of the invention NIC **100** is configured primarily for Internet traffic. Thus, in this embodiment packet **200** is extensively parsed only when the layer two protocol is Ethernet (either traditional Ethernet or 802.3 Ethernet, with or without tagging for Virtual Local Area Networks), the layer three protocol is IP (Internet Protocol) and the layer four protocol is TCP (Transport Control Protocol). Packets adhering to other protocols may be parsed to some (e.g., lesser) extent. NIC **100** may, however, be configured to support and parse virtually any communication protocol's header. Illustratively, the protocol headers that are parsed, and the extent to which they are parsed, are determined by the configuration of a set of instructions for operating header parser **106**.

As described above, the protocols corresponding to headers **212**, **214** and **216** depend upon the network environment in which a packet is sent. The protocols also depend upon the communicating entities. For example, a packet received by a network interface may be a control packet exchanged between the medium access controllers for the source and destination computer systems. In this case, the packet would be likely to include minimal or no data, and may not include layer three protocol header **214** or layer four protocol header **216**. Control packets are typically used for various purposes related to the management of individual connections.

Another communication flow or connection could involve two application programs. In this case, a packet may include headers **212**, **214** and **216**, as shown in FIG. **2**, and may also include additional headers related to higher layers of a protocol stack (e.g., session, presentation and application layers in the ISO-OSI model). In addition, some applications may include headers or header-like information within data portion **202**. For example, for a Network File System (NFS) application, data portion **202** may include NFS headers related to individual NFS datagrams. A datagram may be defined as a collection of data sent from one entity to another, and may comprise data transmitted in multiple packets. In other words, the amount of data constituting a datagram may be greater than the amount of data that can be included in one packet.

One skilled in the art will appreciate that the methods for parsing a packet that are described in the following section are readily adaptable for packets formatted in accordance with virtually any communication protocol.

One Embodiment of a Header Parser

FIG. **3** depicts header parser **106** of FIG. **1A** in accordance with a present embodiment of the invention. Illustratively, header parser **106** comprises header memory **302** and parser **304**, and parser **304** comprises instruction memory **306**. Although depicted as distinct modules in FIG. **3**, in an alternative embodiment of the invention header memory **302** and instruction memory **306** are contiguous.

In the illustrated embodiment, parser **304** parses a header stored in header memory **302** according to instructions stored in instruction memory **306**. The instructions are designed for the parsing of particular protocols or a particular protocol stack, as discussed above. In one embodiment of the invention, instruction memory **306** is modifiable (e.g., the memory is implemented as RAM, EPROM, EEPROM or the like), so that new or modified parsing instructions may be downloaded or otherwise installed. Instructions for parsing a packet are further discussed in the following section.

In FIG. **3**, a header portion of a packet stored in IPP module **104** (shown in FIG. **1A**) is copied into header memory **302**. Illustratively, a specific number of bytes (e.g., **114**) at the beginning of the packet are copied. In an alternative embodiment of the invention, the portion of a packet that is copied may be of a different size. The particular amount of a packet copied into header memory **302** should be enough to capture one or more protocol headers, or at least enough information (e.g., whether included in a header or data portion of the packet) to retrieve the information described below. The header portion stored in header memory **302** may not include the layer one header, which may be removed prior to or in conjunction with the packet being processed by IPP module **104**.

After a header portion of the packet is stored in header memory **302**, parser **304** parses the header portion according to the instructions stored in instruction memory **306**. In the presently described embodiment, instructions for operating parser **304** apply the formats of selected protocols to step through the contents of header memory **302** and retrieve specific information. In particular, specifications of communication protocols are well known and widely available. Thus, a protocol header may be traversed byte by byte or some other fashion by referring to the protocol specifications. In a present embodiment of the invention the parsing algorithm is dynamic, with information retrieved from one field of a header often altering the manner in which another part is parsed.

For example, it is known that the Type field of a packet adhering to the traditional, form of Ethernet (e.g., version two) begins at the thirteenth byte of the (layer two) header. By comparison, the Type field of a packet following the IEEE 802.3 version of Ethernet begins at the twenty-first byte of the header. The Type field is in yet other locations if the packet forms part of a Virtual Local Area Network (VLAN) communication (which illustratively involves tagging or encapsulating an Ethernet header). Thus, in a present embodiment of the invention, the values in certain fields are retrieved and tested in order to ensure that the information needed from a header is drawn from the correct portion of the header. Details concerning the form of a VLAN packet may be found in specifications for the IEEE 802.3p and EEE 802.3q forms of the Ethernet protocol.

The operation of header parser **106** also depends upon other differences between protocols, such as whether the packet uses version four or version six of the Internet Protocol, etc. Specifications for versions four and six of IP may be located in IETF (Internet Engineering Task Force) RFCs (Request for Comment) 791 and 2460, respectively.

The more protocols that are "known" by parser **304**, the more protocols a packet may be tested for, and the more complicated the parsing of a packet's header portion may become. One skilled in the art will appreciate that the protocols that may be parsed by parser **304** are limited only

by the instructions according to which it operates. Thus, by augmenting or replacing the parsing instructions stored in instruction memory **306**, virtually all known protocols may be handled by header parser **106** and virtually any information may be retrieved from a packet's headers.

If, of course, a packet header does not conform to an expected or suspected protocol, the parsing operation may be terminated. In this case, the packet may not be suitable for one more of the efficiency enhancements offered by NIC **100** (e.g., data re-assembly, packet batching, load distribution).

Illustratively, the information retrieved from a packet's headers is used by other portions of NIC **100** when processing that packet. For example, as a result of the packet parsing performed by parser **304** a flow key is generated to identify the communication flow or communication connection that comprises the packet. Illustratively, the flow key is assembled by concatenating one or more addresses corresponding to one or more of the communicating entities. In a present embodiment, a flow key is formed from a combination of the source and destination addresses drawn from the IP header and the source and destination ports taken from the TCP header. Other indicia of the communicating entities may be used, such as the Ethernet source and destination addresses (drawn from the layer two header), NFS file handles or source and destination identifiers for other application datagrams drawn from the data portion of the packet.

One skilled in the art will appreciate that the communicating entities may be identified with greater resolution by using indicia drawn from the higher layers of the protocol stack associated with a packet. Thus, a combination of IP and TCP indicia may identify the entities with greater particularity than layer two information.

Besides a flow key, parser **304** also generates a control or status indicator to summarize additional information concerning the packet. In one embodiment of the invention a control indicator includes a sequence number (e.g., TCP sequence number drawn from a TCP header) to ensure the correct ordering of packets when re-assembling their data. The control indicator may also reveal whether certain flags in the packet's headers are set or cleared, whether the packet contains any data, and, if the packet contains data, whether the data exceeds a certain size. Other data are also suitable for inclusion in the control indicator, limited only by the information that is available in the portion of the packet parsed by parser **304**.

In one embodiment of the invention, header parser **106** provides the flow key and all or a portion of the control indicator to flow database manager **108**. As discussed in a following section, FDBM **108** manages a database or other data structure containing information relevant to communication flows passing through NIC **100**.

In other embodiments of the invention, parser **304** produces additional information derived from the header of a packet for use by other modules of NIC **100**. For example, header parser **106** may report the offset, from the beginning of the packet or from some other point, of the data or payload portion of a packet received from a network. As described above, the data portion of a packet typically follows the header portion and may be followed by a trailer portion. Other data that header parser **106** may report include the location in the packet at which a checksum operation should begin, the location in the packet at which the layer three and/or layer four headers begin, diagnostic data, payload information, etc. The term "payload" is often used to refer to the data portion of a packet. In particular, in one embodiment of the invention header parser **106** provides a payload offset and payload size to control queue **118**.

In appropriate circumstances, header parser **106** may also report (e.g., to IPP module **104** and/or control queue **118**) that the packet is not formatted in accordance with the protocols that parser **304** is configured to manipulate. This report may take the form of a signal (e.g., the No_Assist signal described below), alert, flag or other indicator. The signal may be raised or issued whenever the packet is found to reflect a protocol other than the pre-selected protocols that are compatible with the processing enhancements described above (e.g., data re-assembly, batch processing of packet headers, load distribution). For example, in one embodiment of the invention parser **304** may be configured to parse and efficiently process packets using TCP at layer four, IP at layer three and Ethernet at layer two. In this embodiment, an IPX (Internetwork Packet Exchange) packet would not be considered compatible and IPX packets therefore would not be gathered for data re-assembly and batch processing.

At the conclusion of parsing in one embodiment of the invention, the various pieces of information described above are disseminated to appropriate modules of NIC **100**. After this (and as described in a following section), flow database manager **108** determines whether an active flow is associated with the flow key derived from the packet and sets an operation code to be used in subsequent processing. In addition, IPP module **104** transmits the packet to packet queue **116**. IPP module **104** may also receive some of the information extracted by header parser **106**, and pass it to another module of NIC **100**.

In the embodiment of the invention depicted in FIG. **3**, an entire header portion of a received packet to be parsed is copied and then parsed in one evolution, after which the header parser turns its attention to another packet. However, in an alternative embodiment multiple copy and/or parsing operations may be performed on a single packet. In particular, an initial header portion of the packet may be copied into and parsed by header parser **106** in a first evolution, after which another header portion may be copied into header parser **106** and parsed in a second evolution. A header portion in one evolution may partially or completely overlap the header portion of another evolution. In this manner, extensive headers may be parsed even if header memory **302** is of limited size. Similarly, it may require more than one operation to load a full set of instructions for parsing a packet into instruction memory **306**. Illustratively, a first portion of the instructions may be loaded and executed, after which other instructions are loaded.

With reference now to FIGS. **4A-4B**, a flow chart is presented to illustrate one method by which a header parser may parse a header portion of a packet received at a network interface circuit from a network. In this implementation, the header parser is configured, or optimized, for parsing packets conforming to a set of pre-selected protocols (or protocol stacks). For packets meeting these criteria, various information is retrieved from the header portion to assist in the re-assembly of the data portions of related packets (e.g., packets comprising data from a single datagram). Other enhanced features of the network interface circuit may also be enabled.

The information generated by the header parser includes, in particular, a flow key with which to identify the communication flow or communication connection that comprises the received packet. In one embodiment of the invention, data from packets having the same flow key may be identified and re-assembled to form a datagram. In addition, headers of packets having the same flow key may be processed collectively through their protocol stack (e.g., rather than serially).

In another embodiment of the invention, information retrieved by the header parser is also used to distribute the processing of network traffic received from a network. For example, multiple packets having the same flow key may be submitted to a single processor of a multi-processor host computer system.

In the method illustrated in FIGS. **4A–4B**, the set of pre-selected protocols corresponds to communication protocols frequently transmitted via the Internet. In particular, the set of protocols that may be extensively parsed in this method include the following. At layer two: Ethernet (traditional version), 802.3 Ethernet, Ethernet VLAN (Virtual Local Area Network) and 802.3 Ethernet VLAN. At layer three: IPv4 (with no options) and IPv6 (with no options). Finally, at layer four, only TCP protocol headers (with or without options) are parsed in the illustrated method. Header parsers in alternative embodiments of the invention parse packets formatted through other protocol stacks. In particular, a NIC may be configured in accordance with the most common protocol stacks in use on a given network, which may or may not include the protocols compatible with the header parser method illustrated in FIGS. **4A–4B**.

As described below, a received packet that does not correspond to the protocols parsed by a given method may be flagged and the parsing algorithm terminated for that packet. Because the protocols under which a packet has been formatted can only be determined, in the present method, by examining certain header field values, the determination that a packet does not conform to the selected set of protocols may be made at virtually any time during the procedure. Thus, the illustrated parsing method has as one goal the identification of packets not meeting the formatting criteria for re-assembly of data.

Various protocol header fields appearing in headers for the selected protocols are discussed below. Communication protocols that may be compatible with an embodiment of the present invention (e.g., protocols that may be parsed by a header parser) are well known to persons skilled in the art and are described with great particularity in a number of references. They therefore need not be visited in minute detail herein. In addition, the illustrated method of parsing a header portion of a packet for the selected protocols is merely one method of gathering the information described below. Other parsing procedures capable of doing so are equally suitable.

In a present embodiment of the invention, the illustrated procedure is implemented as a combination of hardware and software. For example, updateable micro-code instructions for performing the procedure may be executed by a microsequencer. Alternatively, such instructions may be fixed (e.g., stored in read-only memory) or may be executed by a processor or microprocessor.

In FIGS. **4A–4B**, state **400** is a start state during which a packet is received by NIC **100** (shown in FIG. **1A**) and initial processing is performed. NIC **100** is coupled to the Internet for purposes of this procedure. Initial processing may include basic error checking and the removal of the layer one preamble. After initial processing, the packet is held by IPP module **104** (also shown in FIG. **1A**). In one embodiment of the invention, state **400** comprises a logical loop in which the header parser remains in an idle or wait state until a packet is received.

In state **402**, a header portion of the packet is copied into memory (e.g., header memory **302** of FIG. **3**). In a present embodiment of the invention a predetermined number of bytes at the beginning (e.g., 114 bytes) of the packet are copied. Packet portions of different sizes are copied in alternative embodiments of the invention, the sizes of which are guided by the goal of copying enough of the packet to capture and/or identify the necessary header information. Illustratively, the full packet is retained by IPP module **104** while the following parsing operations are performed, although the packet may, alternatively, be stored in packet queue **116** prior to the completion of parsing.

Also in state **402**, a pointer to be used in parsing the packet may be initialized. Because the layer one preamble was removed, the header portion copied to memory should begin with the layer two protocol header. Illustratively, therefore, the pointer is initially set to point to the twelfth byte of the layer two protocol header and the two-byte value at the pointer position is read. As one skilled in the art will recognize, these two bytes may be part of a number of different fields, depending upon which protocol constitutes layer two of the packet's protocol stack. For example, these two bytes may comprise the Type field of a traditional Ethernet header, the Length field of an 802.3 Ethernet header or the TPID (Tag Protocol IDentifier) field of a VLAN-tagged header.

In state **404**, a first examination is made of the layer two header to determine if it comprises a VLAN-tagged layer two protocol header. Illustratively, this determination depends upon whether the two bytes at the pointer position store the hexadecimal value 8100. If so, the pointer is probably located at the TPID field of a VLAN-tagged header. If not a VLAN header, the procedure proceeds to state **408**.

If, however, the layer two header is a VLAN-tagged header, in state **406** the CFI (Canonical Format Indicator) bit is examined. If the CFI bit is set (e.g., equal to one), the illustrated procedure jumps to state **430**, after which it exits. In this embodiment of the invention the CFI bit, when set, indicates that the format of the packet is not compatible with (i.e., does not comply with) the pre-selected protocols (e.g., the layer two protocol is not Ethernet or 802.3 Ethernet). If the CFI bit is clear (e.g., equal to zero), the pointer is incremented (e.g., by four bytes) to position it at the next field that must be examined.

In state **408**, the layer two header is further tested. Although it is now known whether this is or is not a VLAN-tagged header, depending upon whether state **408** was reached through state **406** or directly from state **404**, respectively, the header may reflect either the traditional Ethernet format or the 802.3 Ethernet format. At the beginning of state **408**, the pointer is either at the twelfth or sixteenth byte of the header, either of which may correspond to a Length field or a Type field. In particular, if the two-byte value at the position identified by the pointer is less than 0600 (hexadecimal), then the packet corresponds to 802.3 Ethernet and the pointer is understood to identify a Length field. Otherwise, the packet is a traditional (e.g., version two) Ethernet packet and the pointer identifies a Type field.

If the layer two protocol is 802.3 Ethernet, the procedure continues at state **410**. If the layer two protocol is traditional Ethernet, the Type field is tested for the hexadecimal values of 0800 and 08DD. If the tested field has one of these values, then it has also been determined that the packet's layer three protocol is the Internet Protocol. In this case the illustrated procedure continues at state **412**. Lastly, if the field is a Type field having a value other than 0800 or 86DD (hexadecimal), then the packet's layer three protocol does not match the pre-selected protocols according to which the header parser was configured. Therefore, the procedure continues at state **430** and then ends.

In one embodiment of the invention the packet is examined in state **408** to determine if it is a jumbo Ethernet frame. This determination would likely be made prior to deciding whether the layer two header conforms to Ethernet or 802.3 Ethernet. Illustratively, the jumbo frame determination may be made based on the size of the packet, which may be reported by IPP module **104** or a MAC module. If the packet is a jumbo frame, the procedure may continue at state **410**; otherwise, it may resume at state **412**.

In state **410**, the procedure verifies that the layer two protocol is 802.3 Ethernet with LLC SNAP encapsulation. In particular, the pointer is advanced (e.g., by two bytes) and the six-byte value following the Length field in the layer two header is retrieved and examined. If the header is an 802.3 Ethernet header, the field is the LLC_SNAP field and should have a value of AAAA03000000 (hexadecimal). The original specification for an LLC SNAP header may be found in the specification for IEEE 802.2. If the value in the packet's LLC_SNAP field matches the expected value the pointer is incremented another six bytes, the two-byte 802.3 Ethernet Type field is read and the procedure continues at state **412**. If the values do not match, then the packet does not conform to the specified protocols and the procedure enters state **430** and then ends.

In state **412**, the pointer is advanced (e.g., another two bytes) to locate the beginning of the layer three protocol header. This pointer position may be saved for later use in quickly identifying the beginning of this header. The packet is now known to conform to an accepted layer two protocol (e.g., traditional Ethernet, Ethernet with VLAN tagging, or 802.3 Ethernet with LLC SNAP) and is now checked to ensure that the packet's layer three protocol is IP. As discussed above, in the illustrated embodiment only packets conforming to the IP protocol are extensively processed by the header parser.

Illustratively, if the value of the Type field in the layer two header (retrieved in state **402** or state **410**) is 0800 (hexadecimal), the layer three protocol is expected to be IP, version four. If the value is 86DD (hexadecimal), the layer three protocol is expected to be IP, version six. Thus, the Type field is tested in state **412** and the procedure continues at state **414** or state **418**, depending upon whether the hexadecimal value is 0800 or 86DD, respectively.

In state **414**, the layer three header's conformity with version four of IP is verified. In one embodiment of the invention the Version field of the layer three header is tested to ensure that it contains the hexadecimal value 4, corresponding to version four of IP. If in state **414** the layer three header is confirmed to be IP version four, the procedure continues at state **416**; otherwise, the procedure proceeds to state **430** and then ends at state **432**.

In state **416**, various pieces of information from the IP header are saved. This information may include the IHL (IP Header Length), Total Length, Protocol and/or Fragment Offset fields. The IP source address and the IP destination addresses may also be stored. The source and destination address values are each four bytes long in version four of IP. These addresses are used, as described above, to generate a flow key that identifies the communication flow in which this packet was sent. The Total Length field stores the size of the IP segment of this packet, which illustratively comprises the IP header, the TCP header and the packet's data portion. The TCP segment size of the packet (e.g., the size of the TCP header plus the size of the data portion of the packet) may be calculated by subtracting twenty bytes (the size of the IP version four header) from the Total Length value. After state **416**, the illustrated procedure advances to state **422**.

In state **418**, the layer three header's conformity with version six of IP is verified by testing the Version field for the hexadecimal value 6. If the Version field does not contain this value, the illustrated procedure proceeds to state **430**.

In state **420**, the values of the Payload Length (e.g., the size of the TCP segment) and Next Header field are saved, plus the IP source and destination addresses. Source and destination addresses are each sixteen bytes long in version six of IP.

In state **422** of the illustrated procedure, it is determined whether the IP header (either version four or version six) indicates that the layer four header is TCP. Illustratively, the Protocol field of a version four IP header is tested while the Next Header field of a version six header is tested. In either case, the value should be 6 (hexadecimal). The pointer is then incremented as necessary (e.g., twenty bytes for IP version four, forty bytes for IP version six) to reach the beginning of the TCP header. If it is determined in state **422** that the layer four header is not TCP, the procedure advances to state **430** and ends at end state **432**.

In one embodiment of the invention, other fields of a version four IP header may be tested in state **422** to ensure that the packet meets the criteria for enhanced processing by NIC **100**. For example, an IHL field value other than 5 (hexadecimal) indicates that IP options are set for this packet, in which case the parsing operation is aborted. A fragmentation field value other than zero indicates that the IP segment of the packet is a fragment, in which case parsing is also aborted. In either case, the procedure jumps to state **430** and then ends at end state **432**.

In state **424**, the packet's TCP header is parsed and various data are collected from it. In particular, the TCP source port and destination port values are saved. The TCP sequence number, which is used to ensure the correct re-assembly of data from multiple packets, is also saved. Further, the values of several components of the Flags field—illustratively, the URG (urgent), PSH (push), RST (reset), SYN (synch) and FIN (finish) bits—are saved. As will be seen in a later section, in one embodiment of the invention these flags signal various actions to be performed or statuses to be considered in the handling of the packet.

Other signals or statuses may be generated in state **424** to reflect information retrieved from the TCP header. For example, the point from which a checksum operation is to begin may be saved (illustratively, the beginning of the TCP header); the ending point of a checksum operation may also be saved (illustratively, the end of the data portion of the packet). An offset to the data portion of the packet may be identified by multiplying the value of the Header Length field of the TCP header by four. The size of the data portion may then be calculated by subtracting the offset to the data portion from the size of the entire TCP segment.

In state **426**, a flow key is assembled by concatenating the IP source and destination addresses and the TCP source and destination ports. As already described, the flow key may be used to identify a communication flow or communication connection, and may be used by other modules of NIC **100** to process network traffic more efficiently. Although the sizes of the source and destination addresses differ between IP versions four and six (e.g., four bytes each versus sixteen bytes each, respectively), in the presently described embodiment of the invention all flow keys are of uniform size. In particular, in this embodiment they are thirty-six bytes long, including the two-byte TCP source port and two-byte TCP destination port. Flow keys generated from IP, version four, packet headers are padded as necessary (e.g., with twenty-four clear bytes) to fill the flow key's allocated space.

In state **428**, a control or status indicator is assembled to provide various information to one or more modules of NIC **100**. In one embodiment of the invention a control indicator includes the packet's TCP sequence number, a flag or identifier (e.g., one or more bits) indicating whether the packet contains data (e.g., whether the TCP payload size is greater than zero), a flag indicating whether the data portion of the packet exceeds a pre-determined size, and a flag indicating whether certain entries in the TCP Flags field are equivalent to pre-determined values. The latter flag may, for example, be used to inform another module of NIC **100** that components of the Flags field do or do not have a particular configuration. After state **428**, the illustrated procedure ends with state **432**.

State **430** may be entered at several different points of the illustrated procedure. This state is entered, for example, when it is determined that a header portion that is being parsed by a header parser does not conform to the pre-selected protocol stacks identified above. As a result, much of the information described above is not retrieved. A practical consequence of the inability to retrieve this information is that it then cannot be provided to other modules of NIC **100** and the enhanced processing described above and in following sections may not be performed for this packet. In particular, and as discussed previously, in a present embodiment of the invention one or more enhanced operations may be performed on parsed packets to increase the efficiency with which they are processed. Illustrative operations that may be applied include the re-assembly of data from related packets (e.g., packets containing data from a single datagram), batch processing of packet headers through a protocol stack, load distribution or load sharing of protocol stack processing, efficient transfer of packet data to a destination entity, etc.

In the illustrated procedure, in state **430** a flag or signal (illustratively termed No_Assist) is set or cleared to indicate that the packet presently held by IPP module **104** (e.g., which was just processed by the header parser) does not conform to any of the pre-selected protocol stacks. This flag or signal may be relied upon by another module of NIC **100** when deciding whether to perform one of the enhanced operations.

Another flag or signal may be set or cleared in state **430** to initialize a checksum parameter indicating that a checksum operation, if performed, should start at the beginning of the packet (e.g., with no offset into the packet). Illustratively, incompatible packets cannot be parsed to determine a more appropriate point from which to begin the checksum operation. After state **430**, the procedure ends with end state **432**.

After parsing a packet, the header parser may distribute information generated from the packet to one or more modules of NIC **100**. For example, in one embodiment of the invention the flow key is provided to flow database manager **108**, load distributor **112** and one or both of control queue **118** and packet queue **116**. Illustratively, the control indicator is provided to flow database manager **108**. This and other control information, such as TCP payload size, TCP payload offset and the No_Assist signal may be returned to IPP module **104** and provided to control queue **118**. Yet additional control and/or diagnostic information, such as offsets to the layer three and/or layer four headers, may be provided to IPP module **104**, packet queue **116** and/or control queue **118**. Checksum information (e.g., a starting point and either an ending point or other means of identifying a portion of the packet from which to compute a checksum) may be provided to checksum generator **114**.

As discussed in a following section, although a received packet is parsed on NIC **100** (e.g., by header parser **106**), the packets are still processed (e.g., through their respective protocol stacks) on the host computer system in the illustrated embodiment of the invention. However, after parsing a packet in an alternative embodiment of the invention, NIC **100** also performs one or more subsequent processing steps. For example, NIC **100** may include one or more protocol processors for processing one or more of the packet's protocol headers.

Dynamic Header Parsing Instructions in One Embodiment of the Invention

In one embodiment of the present invention, header parser **106** parses a packet received from a network according to a dynamic sequence of instructions. The instructions may be stored in the header parser's instruction memory (e.g., RAM, SRAM, DRAM, flash) that is re-programmable or that can otherwise be updated with new or additional instructions. In one embodiment of the invention software operating on a host computer (e.g., a device driver) may download a set of parsing instructions for storage in the header parser memory.

The number and format of instructions stored in a header parser's instruction memory may be tailored to one or more specific protocols or protocol stacks. An instruction set configured for one collection of protocols, or a program constructed from that instruction set, may therefore be updated or replaced by a different instruction set or program. For packets received at the network interface that are formatted in accordance with the selected protocols (e.g., "compatible" packets), as determined by analyzing or parsing the packets, various enhancements in the handling of network traffic become possible as described in the following sections. In particular, packets from one datagram that are configured according to a selected protocol may be re-assembled for efficient transfer in a host computer. In addition, header portions of such packets may be processed collectively rather than serially. And, the processing of packets from different datagrams by a multi-processor host computer may be shared or distributed among the processors. Therefore, one objective of a dynamic header parsing operation is to identify a protocol according to which a received packet has been formatted or determine whether a packet header conforms to a particular protocol.

FIG. **23**, discussed in detail shortly, presents an illustrative series of instructions for parsing the layer two, three and four headers of a packet to determine if they are Ethernet, IP and TCP, respectively. The illustrated instructions comprise one possible program or microcode for performing a parsing operation. As one skilled in the art will recognize, after a particular set of parsing instructions is loaded into a parser memory, a number of different programs may be assembled. FIG. **23** thus presents merely one of a number of programs that may be generated from the stored instructions. The instructions presented in FIG. **23** may be performed or executed by a microsequencer, a processor, a microprocessor or other similar module located within a network interface circuit.

In particular, other instruction sets and other programs may be derived for different communication protocols, and may be expanded to other layers of a protocol stack. For example, a set of instructions could be generated for parsing NFS (Network File System) packets. Illustratively, these instructions would be configured to parse layer five and six headers to determine if they are Remote Procedure Call (RPC) and External Data Representation (XDR), respectively. Other instructions could be configured to parse a portion of the packet's data (which may be considered layer seven). An NFS header may be considered a part of a packet's layer six protocol header or part of the packet's data.

One type of instruction executed by a microsequencer may be designed to locate a particular field of a packet (e.g., at a specific offset within the packet) and compare the value stored at that offset to a value associated with that field in a particular communication protocol. For example, one instruction may require the microsequencer to examine a value in a packet header at an offset that would correspond to a Type field of an Ethernet header. By comparing the value actually stored in the packet with the value expected for the protocol, the microsequencer can determine if the packet appears to conform to the Ethernet protocol. Illustratively, the next instruction applied in the parsing program depends upon whether the previous comparison was successful. Thus, the particular instructions applied by the microsequencer, and the sequence in which applied, depend upon which protocols are represented by the packet's headers.

The microsequencer may test one or more field values within each header included in a packet. The more fields that are tested and that are found to comport with the format of a known protocol, the greater the certainty that the packet conforms to that protocol. As one skilled in the art will appreciate, one communication protocol may be quite different than another protocol, thus requiring examination of different parts of packet headers for different protocols. Illustratively, the parsing of one packet may end in the event of an error or because it was determined that the packet being parsed does or does not conform to the protocol(s) the instructions are designed for.

Each instruction in FIG. **23** may be identified by a number and/or a name. A particular instruction may perform a variety of tasks other than comparing a header field to an expected value. An instruction may, for example, call another instruction to examine another portion of a packet header, initialize, load or configure a register or other data structure, prepare for the arrival and parsing of another packet, etc. In particular, a register or other storage structure may be configured in anticipation of an operation that is performed in the network interface after the packet is parsed. For example, a program instruction in FIG. **23** may identify an output operation that may or may not be performed, depending upon the success or failure of the comparison of a value extracted from a packet with an expected value. An output operation may store a value in a register, configure a register (e.g., load an argument or operator) for a post-parsing operation, clear a register to await a new packet, etc.

A pointer may be employed to identify an offset into a packet being parsed. In one embodiment, such a pointer is initially located at the beginning of the layer two protocol header. In another embodiment, however, the pointer is situated at a specific location within a particular header (e.g., immediately following the layer two destination and/or source addresses) when parsing commences. Illustratively, the pointer is incremented through the packet as the parsing procedure executes. In one alternative embodiment, however, offsets to areas of interest in the packet may be computed from one or more known or computed locations.

In the parsing program depicted in FIG. **23**, a header is navigated (e.g., the pointer is advanced) in increments of two bytes (e.g., sixteen-bit words). In addition, where a particular field of a header is compared to a known or expected value, up to two bytes are extracted at a time from the field. Further, when a value or header field is copied for storage in a register or other data structure, the amount of data that may be copied in one operation may be expressed in multiples of two-byte units or in other units altogether (e.g., individual bytes). This unit of measurement (e.g., two bytes) may be increased or decreased in an alternative embodiment of the invention. Altering the unit of measurement may alter the precision with which a header can be parsed or a header value can be extracted.

In the embodiment of the invention illustrated in FIG. **23**, a set of instructions loaded into the header parser's instruction memory comprises a number of possible operations to be performed while testing a packet for compatibility with selected protocols. Program **2300** is generated from the instruction set. Program **2300** is thus merely one possible program, microcode or sequence of instructions that can be formed from the available instruction set.

In this embodiment, the loaded instruction set enables the following sixteen operations that may be performed on a packet that is being parsed. Specific implementations of these operations in program **2300** are discussed in additional detail below. These instructions will be understood to be illustrative in nature and do not limit the composition of instruction sets in other embodiments of the invention. In addition, any subset of these operations may be employed in a particular parsing program or microcode. Further, multiple instructions may employ the same operation and have different effects.

A CLR_REG operation allows the selective initialization of registers or other data structures used in program **2300** and, possibly, data structures used in functions performed after a packet is parsed. Initialization may comprise storing the value zero. A number of illustrative registers that may be initialized by a CLR_REG operation are identified in the remaining operations.

A LD_FID operation copies a variable amount of data from a particular offset within the packet into a register configured to store a packet's flow key or other flow identifier. This register may be termed a FLOWID register. The effect of an LD_FID operation is cumulative. In other words, each time it is invoked for one packet the generated data is appended to the flow key data stored previously.

A LD_SEQ operation copies a variable amount of data from a particular offset within the packet into a register configured to store a packet's sequence number (e.g., a TCP sequence number). This register may be assigned the label SEQNO. This operation is also cumulative—the second and subsequent invocations of this operation for the packet cause the identified data to be appended to data stored previously.

A LD_CTL operation loads a value from a specified offset in the packet into a CONTROL register. The CONTROL register may comprise a control indicator discussed in a previous section for identifying whether a packet is suitable for data re-assembly, packet batching, load distribution or other enhanced functions of NIC **100**. In particular, a control indicator may indicate whether a No_Assist flag should be raised for the packet, whether the packet includes any data, whether the amount of packet data is larger than a predetermined threshold, etc. Thus, the value loaded into a CONTROL register in a LD_CTL operation may affect the post-parsing handling of the packet.

A LD_SAP operation loads a value into the CONTROL register from a variable offset within the packet. The loaded value may comprise the packet's ethertype. In one option that may be associated with a LD_SAP operation, the offset of the packet's layer three header may also be stored in the CONTROL register or elsewhere. As one skilled in the art will recognize, a packet's layer three header may immediately follow its layer two ethertype field if the packet conforms to the Ethernet and IP protocols.

A LD_R1 operation may be used to load a value into a temporary register (e.g., named R1) from a variable offset

within the packet. A temporary register may be used for a variety of tasks, such as accumulating values to determine the length of a header or other portion of the packet. A LD_R1 operation may also cause a value from another variable offset to be stored in a second temporary register (e.g., named R2). The values stored in the R1 and/or R2 registers during the parsing of a packet may or may not be cumulative.

A LD_L3 operation may load a value from the packet into a register configured to store the location of the packet's layer three header. This register may be named L3OFFSET. In one optional method of invoking this operation, it may be used to load a fixed value into the L3OFFSET register. As another option, the LD_L3 operation may add a value stored in a temporary register (e.g., R1) to the value being stored in the L3OFFSET register.

A LD_SUM operation stores the starting point within the packet from which a checksum should be calculated. The register in which this value is stored may be named a CSUMSTART register. In one alternative invocation of this operation, a fixed or predetermined value is stored in the register. As another option, the LD_SUM operation may add a value stored in a temporary register (e.g., R1) to the value being stored in the CSUMSTART register.

A LD_HDR operation loads a value into a register configured to store the location within the packet at which the header portion may be split. The value that is stored may, for example, be used during the transfer of the packet to the host computer to store a data portion of the packet in a separate location than the header portion. The loaded value may thus identify the beginning of the packet data or the beginning of a particular header. In one invocation of a LD_HDR operation, the stored value may be computed from a present position of a parsing pointer described above. In another invocation, a fixed or predetermined value may be store. As yet another alternative, a value stored in a temporary register (e.g., R1) and/or a constant may be added to the loaded value.

A LD_LEN operation stores the length of the packet's payload into a register (e.g., a PAYLOADLEN register).

An IM_FID operation appends or adds a fixed or predetermined value to the existing contents of the FLOWID register described above.

An IM_SEQ operation appends or adds a fixed or predetermined value to the contents of the SEQNO register described above.

An IM_SAP operation loads or stores a fixed or predetermined value in the CSUMSTART register described above.

An IM_R1 operation may add or load a predetermined value in one or more temporary registers (e.g., R1, R2).

An IM_CTL operation loads or stores a fixed or predetermined value in the CONTROL register described above.

A ST_FLAG operation loads a value from a specified offset in the packet into a FLAGS register. The loaded value may comprise one or more fields or flags from a packet header.

One skilled in the art will recognize that the labels assigned to the operations and registers described above and elsewhere in this section are merely illustrative in nature and in no way limit the operations and parsing instructions that may be employed in other embodiments of the invention.

Instructions in program **2300** comprise instruction number field **2302**, which contains a number of an instruction within the program, and instruction name field **2304**, which contains a name of an instruction. In an alternative embodiment of the invention instruction number and instruction name fields may be merged or one of them may be omitted.

Instruction content field **2306** includes multiple portions for executing an instruction. An "extraction mask" portion of an instruction is a two-byte mask in hexadecimal notation. An extraction mask identifies a portion of a packet header to be copied or extracted, starting from the current packet offset (e.g., the current position of the parsing pointer). Illustratively, each bit in the packet's header that corresponds to a one in the hexadecimal value is copied for comparison to a comparison or test value. For example, a value of 0xFF00 in the extraction mask portion of an instruction signifies that the entire first byte at the current packet offset is to be copied and that the contents of the second byte are irrelevant. Similarly, an extraction mask of 0x3FFF signifies that all but the two most significant bits of the first byte are to be copied. A two-byte value is constructed from the extracted contents, using whatever was copied from the packet. Illustratively, the remainder of the value is padded with zeros. One skilled in the art will appreciate that the format of an extraction mask (or an output mask, described below) may be adjusted as necessary to reflect little endian or big endian representation.

One or more instructions in a parsing program may not require any data extracted from the packet at the pointer location to be able to perform its output operation. These instructions may have an extraction mask value of 0x0000 to indicate that although a two-byte value is still retrieved from the pointer position, every bit of the value is masked off. Such an extraction mask thus yields a definite value of zero. This type of instruction may be used when, for example, an output operation needs to be performed before another substantive portion of header data is extracted with an extraction mask other than 0x0000.

A "compare value" portion of an instruction is a two-byte hexadecimal value with which the extracted packet contents are to be compared. The compare value may be a value known to be stored in a particular field of a specific protocol header. The compare value may comprise a value that the extracted portion of the header should match or have a specified relationship to in order for the packet to be considered compatible with the pre-selected protocols.

An "operator" portion of an instruction identifies an operator signifying how the extracted and compare values are to be compared. Illustratively, EQ signifies that they are tested for equality, NE signifies that they are tested for inequality, LT signifies that the extracted value must be less than the compare value for the comparison to succeed, GE signifies that the extracted value must be greater than or equal to the compare value, etc. An instruction that awaits arrival of a new packet to be parsed may employ an operation of NP. Other operators for other functions may be added and the existing operators may be assigned other monikers.

A "success offset" portion of an instruction indicates the number of two-byte units that the pointer is to advance if the comparison between the extracted and test values succeeds. A "success instruction" portion of an instruction identifies the next instruction in program **2300** to execute if the comparison is successful.

Similarly, "failure offset" and "failure instruction" portions indicate the number of two-byte units to advance the pointer and the next instruction to execute, respectively, if the comparison fails. Although offsets are expressed in units of two bytes (e.g., sixteen-bit words) in this embodiment of the invention, in an alternative embodiment of the invention they may be smaller or larger units. Further, as mentioned above an instruction may be identified by number or name.

Not all of the instructions in a program are necessarily used for each packet that is parsed. For example, a program

may include instructions to test for more than one type or version of a protocol at a particular layer. In particular, program **2300** tests for either version four or six of the IP protocol at layer three. The instructions that are actually executed for a given packet will thus depend upon the format of the packet. Once a packet has been parsed as much as possible with a given program or it has been determined that the packet does or does not conform to a selected protocol, the parsing may cease or an instruction for halting the parsing procedure may be executed. Illustratively, a next instruction portion of an instruction (e.g., "success instruction" or "failure instruction") with the value "DONE" indicates the completion of parsing of a packet. A DONE, or similar, instruction may be a dummy instruction. In other words, "DONE" may simply signify that parsing to be terminated for the present packet. Or, like instruction eighteen of program **2300**, a DONE instruction may take some action to await a new packet (e.g., by initializing a register).

The remaining portions of instruction content field **2306** are used to specify and complete an output or other data storage operation. In particular, in this embodiment an "output operation" portion of an instruction corresponds to the operations included in the loaded instruction set. Thus, for program **2300**, the output operation portion of an instruction identifies one of the sixteen operations described above. The output operations employed in program **2300** are further described below in conjunction with individual instructions.

An "operation argument" portion of an instruction comprises one or more arguments or fields to be stored, loaded or otherwise used in conjunction with the instruction's output operation. Illustratively, the operation argument portion takes the form of a multi-bit hexadecimal value. For program **2300**, operation arguments are eleven bits in size. An argument or portion of an argument may have various meanings, depending upon the output operation. For example, an operation argument may comprise one or more numerical values to be stored in a register or to be used to locate or delimit a portion of a header. Or, an argument bit may comprise a flag to signal an action or status. In particular, one argument bit may specify that a particular register is to be reset; a set of argument bits may comprise an offset into a packet header to a value to be stored in a register, etc. Illustratively, the offset specified by an operation argument is applied to the location of the parsing pointer position before the pointer is advanced as specified by the applicable success offset or failure offset. The operation arguments used in program **2300** are explained in further detail below.

An "operation enabler" portion of an instruction content field specifies whether or when an instruction's output operation is to be performed. In particular, in the illustrated embodiment of the invention an instruction's output operation may or may not be performed, depending on the result of the comparison between a value extracted from a header and the compare value. For example, an output enabler may be set to a first value (e.g., zero) if the output operation is never to be performed. It may take different values if it is to be performed only when the comparison does or does not satisfy the operator (e.g., one or two, respectively). An operation enabler may take yet another value (e.g., three) if it is always to be performed.

A "shift" portion of an instruction comprises a value indicating how an output value is to be shifted. A shift may be necessary because different protocols sometime require values to be formatted differently. In addition, a value indicating a length or location of a header or header field may require shifting in order to reflect the appropriate magnitude represented by the value. For example, because program **2300** is designed to use two-byte units, a value may need to be shifted if it is to reflect other units (e.g., bytes). A shift value in a present embodiment indicates the number of positions (e.g., bits) to right-shift an output value. In another embodiment of the invention a shift value may represent a different shift type or direction.

Finally, an "output mask" specifies how a value being stored in a register or other data structure is to be formatted. As stated above, an output operation may require an extracted, computed or assembled value to be stored. Similar to the extraction mask, the output mask is a two-byte hexadecimal value. For every position in the output mask that contains a one, in this embodiment of the invention the corresponding bit in the two-byte value identified by the output operation and/or operation argument is to be stored. For example, a value of 0xFFFF indicates that the specified two-byte value is to be stored as is. Illustratively, for every position in the output mask that contains a zero, a zero is stored. Thus, a value of 0xF000 indicates that the most significant four bits of the first byte are to be stored, but the rest of the stored value is irrelevant, and may be padded with zeros.

An output operation of "NONE" may be used to indicate that there is no output operation to be performed or stored, in which case other instruction portions pertaining to output may be ignored or may comprise specified values (e.g., all zeros). In the program depicted in FIG. **23**, however, a CLR_REG output operation, which allows the selective re-initialization of registers, may be used with an operation argument of zero to effectively perform no output. In particular, an operation argument of zero for the CLR_REG operation indicates that no registers are to be reset. In an alternative embodiment of the invention the operation enabler portion of an instruction could be set to a value (e.g., zero) indicating that the output operation is never to be performed.

The format and sequence of instructions in FIG. **23** will be understood to represent just one method of parsing a packet to determine whether it conforms to a particular communication protocol. In particular, the instructions are designed to examine one or more portions of one or more packet headers for comparison to known or expected values and to configure or load a register or other storage location as necessary. As one skilled in the art will appreciate, instructions for parsing a packet may take any of a number of forms and be performed in a variety of sequences without exceeding the scope of the invention.

With reference now to FIG. **23**, instructions in program **2300** may be described in detail. Prior to execution of the program depicted in FIG. **23**, a parsing pointer is situated at the beginning of a packet's layer two header. The position of the parsing pointer may be stored in a register for easy reference and update during the parsing procedure. In particular, the position of the parsing pointer as an offset (e.g., from the beginning of the layer two header) may be used in computing the position of a particular position within a header.

Program **2300** begins with a WAIT instruction (e.g., instruction zero) that waits for a new packet (e.g., indicated by operator NP) and, when one is received, sets a parsing pointer to the twelfth byte of the layer two header. This offset to the twelfth byte is indicated by the success offset portion of the instruction. Until a packet is received, the WAIT instruction loops on itself. In addition, a CLR_REG operation is conducted, but the operation enabler setting indicates that it is only conducted when the comparison succeeds (e.g., when a new packet is received).

The specified CLR_REG operation operates according to the WAIT instruction's operation argument (i.e., 0x3FF). In this embodiment, each bit of the argument corresponds to a register or other data structure. The registers initialized in this operation may include the following: ADDR (e.g., to store the parsing pointer's address or location), FLOWID (e.g., to store the packet's flow key), SEQNO (e.g., to store a TCP sequence number), SAP (e.g., the packet's ethertype) and PAYLOADLEN (e.g., payload length). The following registers configured to store certain offsets may also be reset: FLOWOFF (e.g., offset within FLOWID register), SEQOFF (e.g., offset within SEQNO register), L3OFFSET (e.g., offset of the packet's layer three header), HDRSPLIT (e.g., location to split packet) and CSUMSTART (e.g., starting location for computing a checksum). Also, one or more status or control indicators (e.g., CONTROL or FLAGS register) for reporting the status of one or more flags of a packet header may be reset. In addition, one or more temporary registers (e.g., R1, R2) or other data structures may also be initialized. These registers are merely illustrative of the data structures that may be employed in one embodiment of the invention. Other data structures may be employed in other embodiments for the same or different output operations.

Temporary registers such as R1 and/or R2 may be used in program **2300** to track various headers and header fields. One skilled in the art will recognize the number of possible combinations of communication protocols and the effect of those various combinations on the structure and format of a packet's headers. More information may need to be examined or gathered from a packet conforming to one protocol or set of protocols than from a packet conforming to another protocol or set of protocols. For example, if extension headers are used with an Internet Protocol header, values from those extension headers and/or their lengths may need to be stored, which values are not needed if extension headers are not used. When calculating a particular offset, such as an offset to the beginning of a packet's data portion for example, multiple registers may need to be maintained and their values combined or added. In this example, one register or temporary register may track the size or format of an extension header, while another register tracks the base IP header.

Instruction VLAN (e.g., instruction one) examines the two-byte field at the parsing pointer position (possibly a Type, Length or TPID field) for a value indicating a VLAN-tagged header (e.g., 8100 in hexadecimal). If the header is VLAN-tagged, the pointer is incremented a couple of bytes (e.g., one two-byte unit) and execution continues with instruction CFI; otherwise, execution continues with instruction 802.3. In either event, the instruction's operation enabler indicates that an IM_CTL operation is always to be performed.

As described above, an IM_CTL operation causes a control register or other data structure to be populated with one or more flags to report the status or condition of a packet. As described in the previous section, a control indicator may indicate whether a packet is suitable for enhanced processing (e.g., whether a No_Assist signal should be generated for the packet), whether a packet includes any data and, if so, whether the size of the data portion exceeds a specified threshold. The operation argument 0x00 A for instruction VLAN comprises the value to be stored in the control register, with individual bits of the argument corresponding to particular flags. Illustratively, flags associated with the conditions just described may be set to one, or true, in this IM_CTL operation.

Instruction CFI (e.g., instruction two) examines the CFI bit or flag in a layer two header. If the CFI bit is set, then the packet is not suitable for the processing enhancements described in other sections and the parsing procedure ends by calling instruction DONE (e.g., instruction eighteen). If the CFI bit is not set, then the pointer is incremented another couple of bytes and execution continues with instruction 802.3. As explained above, a null output operation (e.g., "NONE") indicates that no output operation is performed. In addition, the output enabler value (e.g., zero) further ensures that no output operation is performed.

In instruction 802.3 (e.g., instruction three), a Type or Length field (depending on the location of the pointer and format of the packet) is examined to determine if the packet's layer two format is traditional Ethernet or 802.3 Ethernet. If the value in the header field appears to indicate 802.3 Ethernet (e.g., contains a hexadecimal value less than 0600), the pointer is incremented two bytes (to what should be an LLC SNAP field) and execution continues with instruction LLC_1. Otherwise, the layer two protocol may be considered traditional Ethernet and execution continues with instruction IPV4 _1. Instruction 802.3 in this embodiment of the invention does not include an output operation.

In instructions LLC_1 and LLC_2 (e.g., instructions four and five), a suspected layer two LLC SNAP field is examined to ensure that the packet conforms to the 802.3 Ethernet protocol. In instruction LLC_1, a first part of the field is tested and, if successful, the pointer is incremented two bytes and a second part is tested in instruction LLC_2. If instruction LLC_2 succeeds, the parsing pointer is advanced four bytes to reach what should be a Type field and execution continues with instruction IPV4 _1. If either test fails, however, the parsing procedure exits. In the illustrated embodiment of the invention, no output operation is performed while testing the LLC SNAP field.

In instruction IPV4_1 (e.g., instruction six), the parsing pointer should be at an Ethernet Type field. This field is examined to determine if the layer three protocol appears to correspond to version four of the Internet Protocol. If this test is successful (e.g., the Type field contains a hexadecimal value of 0800), the pointer is advanced two bytes to the beginning of the layer three header and execution of program **2300** continues with instruction IPV4_2. If the test is unsuccessful, then execution continues with instruction IPV6_1. Regardless of the test results, the operation enabler value (e.g., three) indicates that the specified LD_SAP output operation is always performed.

As described previously, in a LD_SAP operation a packet's ethertype (or Service Access Point) is stored in a register. Part of the operation argument of 0x100, in particular the right-most six bits (e.g., zero) constitute an offset to a two-byte value comprising the ethertype. The offset in this example is zero because, in the present context, the parsing pointer is already at the Type field that contains the ethertype. In the presently described embodiment, the remainder of the operation argument constitutes a flag specifying that the starting position of the layer three header (e.g., an offset from the beginning of the packet) is also to be saved (e.g., in the L3OFFSET register). In particular, the beginning of the layer three header is known to be located immediately after the two-byte Type field.

Instruction IPV4_2 (e.g., instruction seven) tests a suspected layer three version field to ensure that the layer three protocol is version four of IP. In particular, a specification for version four of IP specifies that the first four bits of the layer three header contain a value of 0x4. If the test fails, the parsing procedure ends with instruction DONE. If the test

succeeds, the pointer advances six bytes and instruction IPV4_3 is called.

The specified LD_SUM operation, which is only performed if the comparison in instruction IPV4 _2 succeeds, indicates that an offset to the beginning of a point from which a checksum may be calculated should be stored. In particular, in the presently described embodiment of the invention a checksum should be calculated from the beginning of the TCP header (assuming that the layer four header is TCP). The value of the operation argument (e.g., 0x00A) indicates that the checksum is located twenty bytes (e.g., ten two-byte increments) from the current pointer. Thus, a value of twenty bytes is added to the parsing pointer position and the result is stored in a register or other data structure (e.g., the CSUMSTART register).

Instruction IPV4_3 (e.g., instruction eight) is designed to determine whether the packet's IP header indicates IP fragmentation. If the value extracted from the header in accordance with the extraction mask does not equal the comparison value, then the packet indicates fragmentation. If fragmentation is detected, the packet is considered unsuitable for the processing enhancements described in other sections and the procedure exits (e.g., through instruction DONE). Otherwise, the pointer is incremented two bytes and instruction IPV4_4 is called after performing a LD_LEN operation.

In accordance with the LD_LEN operation, the length of the IP segment is saved. The illustrated operation argument (e.g., 0x03E) comprises an offset to the Total Length field where this value is located. In particular, the least-significant six bits constitute the offset. Because the pointer has already been advanced past this field, the operation argument comprises a negative value. One skilled in the art will recognize that this binary value (e.g., 111110) may be used to represent the decimal value of negative two. Thus, the present offset of the pointer, minus four bytes (e.g., two two-byte units), is saved in a register or other data structure (e.g., the PAYLOADLEN register). Any other suitable method of representing a negative offset may be used. Or, the IP segment length may be saved while the pointer is at a location preceding the Total Length field (e.g., during a previous instruction).

In instruction IPV4_4 (e.g., instruction nine), a one-byte Protocol field is examined to determine whether the layer four protocol appears to be TCP. If so, the pointer is advanced fourteen bytes and execution continues with instruction TCP_1; otherwise the procedure ends.

The specified LD_FID operation, which is only performed when the comparison in instruction IPV4_4 succeeds, involves retrieving the packet's flow key and storing it in a register or other location (e.g., the FLOWID register). One skilled in the art will appreciate that in order for the comparison in instruction IPV4_4 to be successful, the packet's layer three and four headers must conform to IP (version four) and TCP, respectively. If so, then the entire flow key (e.g., IP source and destination addresses plus TCP source and destination port numbers) is stored contiguously in the packet's header portion. In particular, the flow key comprises the last portion of the IP header and the initial portion of the TCP header and may be extracted in one operation. The operation argument (e.g., 0x182) thus comprises two values needed to locate and delimit the flow key. Illustratively, the right-most six bits of the argument (e.g., 0x02) identify an offset from the pointer position, in two-byte units, to the beginning of the flow key. The other five bits of the argument (e.g., 0x06) identify the size of the flow key, in two-byte units, to be stored.

In instruction IPV6_1 (e.g., instruction ten), which follows the failure of the comparison performed by instruction IPV4_1, the parsing pointer should be at a layer two Type field. If this test is successful (e.g., the Type field holds a hexadecimal value of 86DD), instruction IPV6_2 is executed after a LD_SUM operation is performed and the pointer is incremented two bytes to the beginning of the layer three protocol. If the test is unsuccessful, the procedure exits.

The indicated LD_SUM operation in instruction IPV6_1 is similar to the operation conducted in instruction IPV4_2 but utilizes a different argument. Again, the checksum is to be calculated from the beginning of the TCP header (assuming the layer four header is TCP). The specified operation argument (e.g., 0x015) thus comprises an offset to the beginning of the TCP header—twenty-one two-byte steps ahead. The indicated offset is added to the present pointer position and saved in a register or other data structure (e.g., the CSUMSTART register).

Instruction IPV6 _2 (e.g., instruction eleven) tests a suspected layer three version field to further ensure that the layer three protocol is version six of IP. If the comparison fails, the parsing procedure ends with the invocation of instruction DONE. If it succeeds, instruction IPV6_3 is called. Operation IM_R1, which is performed only when the comparison succeeds in this embodiment, saves the length of the IP header from a Payload Length field. As one skilled in the art will appreciate, the Total Length field (e.g., IP segment size) of an IP, version four, header includes the size of the version four header. However, the Payload Length field (e.g., IP segment size) of an IP, version six, header does not include the size of the version six header. Thus, the size of the version six header, which is identified by the right-most eight bits of the output argument (e.g., 0x14, indicating twenty two-byte units) is saved. Illustratively, the remainder of the argument identifies the data structure in which to store the header length (e.g., temporary register R1). Because of the variation in size of layer three headers between protocols, in one embodiment of the invention the header size is indicated in different units to allow greater precision. In particular, in one embodiment of the invention the size of the header is specified in bytes in instruction IPV6_2, in which case the output argument could be 0x128.

Instruction IPV6_3 (e.g., instruction twelve) in this embodiment does not examine a header value. In this embodiment, the combination of an extraction mask of 0x0000 with a comparison value of 0x0000 indicates that an output operation is desired before the next examination of a portion of a header. After the LD_FID operation is performed, the parsing pointer is advanced six bytes to a Next Header field of the version six IP header. Because the extraction mask and comparison values are both 0x0000, the comparison should never fail and the failure branch of instruction should never be invoked.

As described previously, a LD_FID operation stores a flow key in an appropriate register or other data structure (e.g., the FLOWID register). Illustratively, the operation argument of 0x484 comprises two values for identifying and delimiting the flow key. In particular, the right-most six bits (e.g., 0x04) indicates that the flow key portion is located at an offset of eight bytes (e.g., four two-byte increments) from the current pointer position. The remainder of the operation argument (e.g., 0x12) indicates that thirty-six bytes (e.g., the decimal equivalent of 0x12 two-byte units) are to be copied from the computed offset. In the illustrated embodiment of the invention the entire flow key is copied intact, including

the layer three source and destination addresses and layer four source and destination ports.

In instruction IPV6_4 (e.g., instruction thirteen), a suspected Next Header field is examined to determine whether the layer four protocol of the packet's protocol stack appears to be TCP. If so, the procedure advances thirty-six bytes (e.g., eighteen two-byte units) and instruction TCP_1 is called; otherwise the procedure exits (e.g., through instruction DONE). Operation LD_LEN is performed if the value in the Next Header field is 0x06. As described above, this operation stores the IP segment size. Once again the argument (e.g., 0x03F) comprises a negative offset, in this case negative one. This offset indicates that the desired Payload Length field is located two bytes before the pointer's present position. Thus, the negative offset is added to the present pointer offset and the result saved in an appropriate register or other data structure (e.g., the PAYLOADLEN register).

In instructions TCP_1, TCP_2, TCP_3 and TCP_4 (e.g., instructions fourteen through seventeen), no header values—other than certain flags specified in the instruction's output operations—are examined, but various data from the packet's TCP header are saved. In the illustrated embodiment, the data that is saved includes a TCP sequence number, a TCP header length and one or more flags. For each instruction, the specified operation is performed and the next instruction is called. As described above, a comparison between the comparison value of 0x0000 and a null extraction value, as used in each of these instructions, will never fail. After instruction TCP_4, the parsing procedure returns to instruction WAIT to await a new packet.

For operation LD_SEQ in instruction TCP_1, the operation argument (e.g., 0x081) comprises two values to identify and extract a TCP sequence number. The right-most six bits (e.g., 0x01) indicate that the sequence number is located two bytes from the pointer's current position. The rest of the argument (e.g., 0x2) indicates the number of two-byte units that must be copied from that position in order to capture the sequence number. Illustratively, the sequence number is stored in the SEQNO register.

For operation ST_FLAG in instruction TCP_2, the operation argument (e.g., 0x145) is used to configure a register (e.g., the FLAGS register) with flags to be used in a post-parsing task. The right-most six bits (e.g., 0x05) constitute an offset, in two-byte units, to a two-byte portion of the TCP header that contains flags that may affect whether the packet is suitable for post-parsing enhancements described in other sections. For example, URG, PSH, RST, SYN and FIN flags may be located at the offset position and be used to configure the register. The output mask (e.g., 0x002F) indicates that only particular portions (e.g., bits) of the TCP header's Flags field are stored.

Operation LD_R1 of instruction TCP_3 is similar to the operation conducted in instruction IPV6_2. Here, an operation argument of 0x205 includes a value (e.g., the least-significant six bits) identifying an offset of five two-byte units from the current pointer position. That location should include a Header Length field to be stored in a data structure identified by the remainder of the argument (e.g., temporary register R1). The output mask (e.g., 0xF000) indicates that only the first four bits are saved (e.g., the Header Length field is only four bits in size).

As one skilled in the art may recognize, the value extracted from the Header Length field may need to be adjusted in order to reflect the use of two-byte units (e.g., sixteen bit words) in the illustrated embodiment. Therefore, in accordance with the shift portion of instruction TCP_3, the value extracted from the field and configured by the output mask (e.g., 0xF000) is shifted to the right eleven positions when stored in order to simplify calculations.

Operation LD_HDR of instruction TCP_4 causes the loading of an offset to the first byte of packet data following the TCP header. As described in a later section, packets that are compatible with a pre-selected protocol stack may be separated at some point into header and data portions. Saving an offset to the data portion now makes it easier to split the packet later. Illustratively, the right-most seven bits of the 0x0FF operation argument comprise a first element of the offset to the data. One skilled in the art will recognize the bit pattern (e.g., 1111111) as equating to negative one. Thus, an offset value equal to the current parsing pointer (e.g., the value in the ADDR register) minus two bytes—which locates the beginning of the TCP header—is saved. The remainder of the argument signifies that the value of a temporary data structure (e.g., temporary register R1) is to be added to this offset. In this particular context, the value saved in the previous instruction (e.g., the length of the TCP header) is added. These two values combine to form an offset to the beginning of the packet data, which is stored in an appropriate register or other data structure (e.g., the HDRSPLIT register).

Finally, and as mentioned above, instruction DONE (e.g., instruction eighteen) indicates the end of parsing of a packet when it is determined that the packet does not conform to one or more of the protocols associated with the illustrated instructions. This may be considered a "clean-up" instruction. In particular, output operation LD_CTL, with an operation argument of 0x001 indicates that a No_Assist flag is to be set (e.g., to one) in the control register described above in conjunction with instruction VLAN. The No_Assist flag, as described elsewhere, may be used to inform other modules of the network interface that the present packet, is unsuitable for one or more processing enhancements described elsewhere.

It will be recognized by one skilled in the art that the illustrated program or microcode merely provides one method of parsing a packet. Other programs, comprising the same instructions in a different sequence or different instructions altogether, with similar or dissimilar formats, may be employed to examine and store portions of headers and to configure registers and other data structures.

The efficiency gains to be realized from the application of the enhanced processing described in following sections more than offset the time required to parse a packet with the illustrated program. Further, even though a header parser parses a packet on a NIC in a current embodiment of the invention, the packet may still need to be processed through its protocol stack (e.g., to remove the protocol headers) by a processor on a host computer. Doing so avoids burdening the communication device (e.g., network interface) with such a task.

One Embodiment of a Flow Database

FIG. 5 depicts flow database (FDB) **110** according to one embodiment of the invention. Illustratively FDB **110** is implemented as a CAM (Content Addressable Memory) using a re-writeable memory component (e.g., RAM, SRAM, DRAM). In this embodiment, FDB **110** comprises associative portion **502** and associated portion **504**, and may be indexed by flow number **506**.

The scope of the invention does not limit the form or structure of flow database **110**. In alternative embodiments of the invention virtually any form of data structure may be employed (e.g., database, table, queue, list, array), either monolithic or segmented, and may be implemented in hardware or software. The illustrated form of FDB **110** is merely

one manner of maintaining useful information concerning communication flows through NIC **100**. As one skilled in the art will recognize, the structure of a CAM allows highly efficient and fast associative searching.

In the illustrated embodiment of the invention, the information stored in FDB **110** and the operation of flow database manager (FDBM) **108** (described below) permit functions such as data re-assembly, batch processing of packet headers, and other enhancements. These functions are discussed in detail in other sections but may be briefly described as follows.

One form of data re-assembly involves the re-assembly or combination of data from multiple related packets (e.g., packets from a single communication flow or a single datagram). One method for the batch processing of packet headers entails processing protocol headers from multiple related packets through a protocol stack collectively rather than one packet at a time. Another illustrative function of NIC **100** involves the distribution or sharing of such protocol stack processing (and/or other functions) among processors in a multi-processor host computer system. Yet another possible function of NIC **100** is to enable the transfer of re-assembled data to a destination entity (e.g., an application program) in an efficient aggregation (e.g., a memory page), thereby avoiding piecemeal and highly inefficient transfers of one packet's data at a time. Thus, in this embodiment of the invention, one purpose of FDB **110** and FDBM **108** is to generate information for the use of NIC **100** and/or a host computer system in enabling, disabling or performing one or more of these functions.

Associative portion **502** of FDB **110** in FIG. **5** stores the flow key of each valid flow destined for an entity served by NIC **100**. Thus, in one embodiment of the invention associative portion **502** includes IP source address **510**, IP destination address **512**, TCP source port **514** and TCP destination port **516**. As described in a previous section these fields may be extracted from a packet and provided to FDBM **108** by header parser **106**.

Although each destination entity served by NIC **100** may participate in multiple communication flows or end-to-end TCP connections, only one flow at a time will exist between a particular source entity and a particular destination entity. Therefore, each flow key in associative portion **502** that corresponds to a valid flow should be unique from all other valid flows. In alternative embodiments of the invention, associative portion **502** is composed of different fields, reflecting alternative flow key forms, which may be determined by the protocols parsed by the header parser and the information used to identify communication flows.

Associated portion **504** in the illustrated embodiment comprises flow validity indicator **520**, flow sequence number **522** and flow activity indicator **524**. These fields provide information concerning the flow identified by the flow key stored in the corresponding entry in associative portion **502**. The fields of associated portion **504** may be retrieved and/or updated by FDBM **108** as described in the following section.

Flow validity indicator **520** in this embodiment indicates whether the associated flow is valid or invalid. Illustratively, the flow validity indicator is set to indicate a valid flow when the first packet of data in a flow is received, and may be reset to reassert a flow's validity every time a portion of a flow's datagram (e.g., a packet) is correctly received.

Flow validity indicator **520** may be marked invalid after the last packet of data in a flow is received. The flow validity indicator may also be set to indicate an invalid flow whenever a flow is to be torn down (e.g., terminated or aborted) for some reason other than the receipt of a final data packet. For example, a packet may be received out of order from other packets of a datagram, a control packet indicating that a data transfer or flow is being aborted may be received, an attempt may be made to re-establish or re-synchronize a flow (in which case the original flow is terminated), etc. In one embodiment of the invention flow validity indicator **520** is a single bit, flag or value.

Flow sequence number **522** in the illustrated embodiment comprises a sequence number of the next portion of data that is expected in the associated flow. Because the datagram being sent in a flow is typically received via multiple packets, the flow sequence number provides a mechanism to ensure that the packets are received in the correct order. For example, in one embodiment of the invention NIC **100** re-assembles data from multiple packets of a datagram. To perform this re-assembly in the most efficient manner, the packets need to be received in order. Thus, flow sequence number **522** stores an identifier to identify the next packet or portion of data that should be received.

In one embodiment of the invention, flow sequence number **522** corresponds to the TCP sequence number field found in TCP protocol headers. As one skilled in the art will recognize, a packet's TCP sequence number identifies the position of the packet's data relative to other data being sent in a datagram. For packets and flows involving protocols other than TCP, an alternative method of verifying or ensuring the receipt of data in the correct order may be employed.

Flow activity indicator **524** in the illustrated embodiment reflects the recency of activity of a flow or, in other words, the age of a flow. In this embodiment of the invention flow activity indicator **524** is associated with a counter, such as a flow activity counter (not depicted in FIG. **5**). The flow activity counter is updated (e.g., incremented) each time a packet is received as part of a flow that is already stored in flow database **110**. The updated counter value is then stored in the flow activity indicator field of the packet's flow. The flow activity counter may also be incremented each time a first packet of a new flow that is being added to the database is received. In an alternative embodiment, a flow activity counter is only updated for packets containing data (e.g., it is not updated for control packets). In yet another alternative embodiment, multiple counters are used for updating flow activity indicators of different flows.

Because it can not always be determined when a communication flow has ended (e.g., the final packet may have been lost), the flow activity indicator may be used to identify flows that are obsolete or that should be torn down for some other reason. For example, if flow database **110** appears to be fully populated (e.g., flow validity indicator **520** is set for each flow number) when the first packet of a new flow is received, the flow having the lowest flow activity indicator may be replaced by the new flow.

In the illustrated embodiment of the invention, the size of fields in FDB **110** may differ from one entry to another. For example, IP source and destination addresses are four bytes large in version four of the protocol, but are sixteen bytes large in version six. In one alternative embodiment of the invention, entries for a particular field may be uniform in size, with smaller entries being padded as necessary.

In another alternative embodiment of the invention, fields within FDB **110** may be merged. In particular, a flow's flow key may be stored as a single entity or field instead of being stored as a number of separate fields as shown in FIG. **5**. Similarly, flow validity indicator **520**, flow sequence number **522** and flow activity indicator **524** are depicted as separate entries in FIG. **5**. However, in an alternative embodiment of

the invention one or more of these entries may be combined. In particular, in one alternative embodiment flow validity indicator **520** and flow activity indicator **524** comprise a single entry having a first value (e.g., zero) when the entry's associated flow is invalid. As long as the flow is valid, however, the combined entry is incremented as packets are received, and is reset to the first value upon termination of the flow.

In one embodiment of the invention FDB **110** contains a maximum of sixty-four entries, indexed by flow number **506**, thus allowing the database to track sixty-four valid flows at a time. In alternative embodiments of the invention, more or fewer entries may be permitted, depending upon the size of memory allocated for flow database **110**. In addition to flow number **506**, a flow may be identifiable by its flow key (stored in associative portion **502**).

In the illustrated embodiment of the invention, flow database **110** is empty (e.g., all fields are filled with zeros) when NIC **100** is initialized. When the first packet of a flow is received header parser **106** parses a header portion of the packet. As described in a previous section, the header parser assembles a flow key to identify the flow and extracts other information concerning the packet and/or the flow. The flow key, and other information, is passed to flow database manager **108**. FDBM **108** then searches FDB **110** for an active flow associated with the flow key. Because the database is empty, there is no match.

In this example, the flow key is therefore stored (e.g., as flow number zero) by copying the IP source address, IP destination address, TCP source port and TCP destination port into the corresponding fields. Flow validity indicator **520** is then set to indicate a valid flow, flow sequence number **522** is derived from the TCP sequence number (illustratively provided by the header parser), and flow activity indicator **524** is set to an initial value (e.g., one), which may be derived from a counter. One method of generating an appropriate flow sequence number, which may be used to verify that the next portion of data received for the flow is received in order, is to add the TCP sequence number and the size of the packet's data. Depending upon the configuration of the packet (e.g., whether the SYN bit in a Flags field of the packet's TCP header is set), however, the sum may need to be adjusted (e.g., by adding one) to correctly identify the next expected portion of data.

As described above, one method of generating an appropriate initial value for a flow activity indicator is to copy a counter value that is incremented for each packet received as part of a flow. For example, for the first packet received after NIC **100** is initialized, a flow activity counter may be incremented to the value of one. This value may then be stored in flow activity indicator **524** for the associated flow. The next packet received as part of the same (or a new) flow causes the counter to be incremented to two, which value is stored in the flow activity indicator for the associated flow. In this example, no two flows should have the same flow activity indicator except at initialization, when they may all equal zero or some other predetermined value.

Upon receipt and parsing of a later packet received at NIC **100**, the flow database is searched for a valid flow matching that packet's flow key. Illustratively, only the flow keys of active flows (e.g., those flows for which flow validity indicator **520** is set) are searched. Alternatively, all flow keys (e.g., all entries in associative portion **502**) may be searched but a match is only reported if its flow validity indicator indicates a valid flow. With a CAM such as FDB **110** in FIG. **5**, flow keys and flow validity indicators may be searched in parallel.

If a later packet contains the next portion of data for a previous flow (e.g., flow number zero), that flow is updated appropriately. In one embodiment of the invention this entails updating flow sequence number **522** and incrementing flow activity indicator **524** to reflect its recent activity. Flow validity indicator **520** may also be set to indicate the validity of the flow, although it should already indicate that the flow is valid.

As new flows are identified, they are added to FDB **110** in a similar manner to the first flow. When a flow is terminated or torn down, the associated entry in FDB **110** is invalidated. In one embodiment of the invention, flow validity indicator **520** is merely cleared (e.g., set to zero) for the terminated flow. In another embodiment, one or more fields of a terminated flow are cleared or set to an arbitrary or predetermined value. Because of the bursty nature of network packet traffic, all or most of the data from a datagram is generally received in a short amount of time. Thus, each valid flow in FDB **110** normally only needs to be maintained for a short period of time, and its entry can then be used to store a different flow.

Due to the limited amount of memory available for flow database **110** in one embodiment of the invention, the size of each field may be limited. In this embodiment, sixteen bytes are allocated for IP source address **510** and sixteen bytes are allocated for IP destination address **512**. For IP addresses shorter than sixteen bytes in length, the extra space may be padded with zeros. Further, TCP source port **514** and TCP destination port **516** are each allocated two bytes. Also in this embodiment, flow validity indicator **520** comprises one bit, flow sequence number **522** is allocated four bytes and flow activity indicator **524** is also allocated four bytes.

As one skilled in the art will recognize from the embodiments described above, a flow is similar, but not identical, to an end-to-end TCP connection. A TCP connection may exist for a relatively extended period of time, sufficient to transfer multiple datagrams from a source entity to a destination entity. A flow, however, may exist only for one datagram. Thus, during one end-to-end TCP connection, multiple flows may be set up and torn down (e.g., once for each datagram). As described above, a flow may be set up (e.g., added to FDB **110** and marked valid) when NIC **100** detects the first portion of data in a datagram and may be torn down (e.g., marked invalid in FDB **110**) when the last portion of data is received. Illustratively, each flow set up during a single end-to-end TCP connection will have the same flow key because the layer three and layer four address and port identifiers used to form the flow key will remain the same.

In the illustrated embodiment, the size of flow database **110** (e.g., the number of flow entries) determines the maximum number of flows that may be interleaved (e.g., simultaneously active) at one time while enabling the functions of data re-assembly and batch processing of protocol headers. In other words, in the embodiment depicted in FIG. **5**, NIC **100** can set up sixty-four flows and receive packets from up to sixty-four different datagrams (i.e., sixty-four flows may be active) without tearing down a flow. If a maximum number of flows through NIC **100** were known, flow database **110** could be limited to the corresponding number of entries.

The flow database may be kept small because a flow only lasts for one datagram in the presently described embodiment and, because of the bursty nature of packet traffic, a datagram's packets are generally received in a short period of time. The short duration of a flow compensates for a limited number of entries in the flow database. In one embodiment of the invention, if FDB **110** is filled with active

flows and a new flow is commenced (i.e., a first portion of data in a new datagram), the oldest (e.g., the least recently active) flow is replaced by the new one.

In an alternative embodiment of the invention, flows may be kept active for any number of datagrams (or other measure of network traffic) or for a specified length or range of time. For example, when one datagram ends its flow in FDB **110** may be kept "open" (i.e., not torn down) if the database is not full (e.g., the flow's entry is not needed for a different flow). This scheme may further enhance the efficient operation of NIC **100** if another datagram having the same flow key is received. In particular, the overhead involved in setting up another flow is avoided and more data re-assembly and packet batching (as described below) may be performed. Advantageously, a flow may be kept open in flow database **110** until the end-to-end TCP connection that encompasses the flow ends.

One Embodiment of a Flow Database Manager

FIGS. **6A–6E** depict one method of operating a flow database manager (FDBM), such as flow database manager **108** of FIG. **1A**, for managing flow database (FDB) **110**. Illustratively, FDBM **108** stores and updates flow information stored in flow database **110** and generates an operation code for a packet received by NIC **100**. FDBM **108** also tears down a flow (e.g., replaces, removes or otherwise invalidates an entry in FDB **110**) when the flow is terminated or aborted.

In one embodiment of the invention a packet's operation code reflects the packet's compatibility with predetermined criteria for performing one or more functions of NIC **100** (e.g., data re-assembly, batch processing of packet headers, load distribution). In other words, depending upon a packet's operation code, other modules of NIC **100** may or may not perform one of these functions, as described in following sections.

In another embodiment of the invention, an operation code indicates a packet status. For example, an operation code may indicate that a packet: contains no data, is a control packet, contains more than a specified amount of data, is the first packet of a new flow, is the last packet of an existing flow, is out of order, contains a certain flag (e.g., in a protocol header) that does not have an expected value (thus possibly indicating an exceptional circumstance), etc.

The operation of flow database manager **108** depends upon packet information provided by header parser **106** and data drawn from flow database **110**. After FDBM **108** processes the packet information and/or data, control information (e.g., the packet's operation code) is stored in control queue **118** and FDB **110** may be altered (e.g., a new flow may be entered or an existing one updated or torn down).

With reference now to FIGS. **6A–6E**, state **600** is a start state in which FDBM **108** awaits information drawn from a packet received by NIC **100** from network **102**. In state **602**, header parser **106** or another module of NIC **100** notifies FDBM **108** of a new packet by providing the packet's flow key and some control information. Receipt of this data may be interpreted as a request to search FDB **110** to determine whether a flow having this flow key already exists.

In one embodiment of the invention the control information passed to FDBM **108** includes a sequence number (e.g., a TCP sequence number) drawn from a packet header. The control information may also indicate the status of certain flags in the packet's headers, whether the packet includes data and, if so, whether the amount of data exceeds a certain size. In this embodiment, FDBM **108** also receives a No_Assist signal for a packet if the header parser determines that the packet is not formatted according to one of the pre-selected protocol stacks (i.e., the packet is not "compatible"), as discussed in a previous section. Illustratively, the No_Assist signal indicates that one or more functions of NIC **100** (e.g., data re-assembly, batch processing, load-balancing) may not be provided for the packet.

In state **604**, FDBM **108** determines whether a No_Assist signal was asserted for the packet. If so, the procedure proceeds to state **668** (FIG. **6E**). Otherwise, FDBM **108** searches FDB **110** for the packet's flow key in state **606**. In one embodiment of the invention only valid flow entries in the flow database are searched. As discussed above, a flow's validity may be reflected by a validity indicator such as flow validity indicator **520** (shown in FIG. **5**). If, in state **608**, it is determined that the packet's flow key was not found in the database, or that a match was found but the associated flow is not valid, the procedure advances to state **646** (FIG. **6D**).

If a valid match is found in the flow database, in state **610** the flow number (e.g., the flow database index for the matching entry) of the matching flow is noted and flow information stored in FDB **110** is read. Illustratively, this information includes flow validity indicator **520**, flow sequence number **522** and flow activity indicator **524** (shown in FIG. **5**).

In state **612**, FDBM **108** determines from information received from header parser **106** whether the packet contains TCP payload data. If not, the illustrated procedure proceeds to state **638** (FIG. **6C**); otherwise the procedure continues to state **614**.

In state **614**, the flow database manager determines whether the packet constitutes an attempt to reset a communication connection or flow. Illustratively, this may be determined by examining the state of a SYN bit in one of the packet's protocol headers (e.g., a TCP header). In one embodiment of the invention the value of one or more control or flag bits (such as the SYN bit) are provided to the FDBM by the header parser. As one skilled in the art will recognize, one TCP entity may attempt to reset a communication flow or connection with another entity (e.g., because of a problem on one of the entity's host computers) and send a first portion of data along with the re-connection request. This is the situation the flow database manager attempts to discern in state **614**. If the packet is part of an attempt to re-connect or reset a flow or connection, the procedure continues at state **630** (FIG. **6C**).

In state **616**, flow database manager **108** compares a sequence number (e.g., a TCP sequence number) extracted from a packet header with a sequence number (e.g., flow sequence number **522** of FIG. **5**) of the next expected portion of data for this flow. As discussed in a previous section, these sequence numbers should correlate if the packet contains the flow's next portion of data. If the sequence numbers do not match, the procedure continues at state **628**.

In state **618**, FDBM **108** determines whether certain flags extracted from one or more of the packet's protocol headers match expected values. For example, in one embodiment of the invention the URG, PSH, RST and FIN flags from the packet's TCP header are expected to be clear (i.e., equal to zero). If any of these flags are set (e.g., equal to one) an exceptional condition may exist, thus making it possible that one or more of the functions (e.g., data re-assembly, batch processing, load distribution) offered by NIC **100** should not be performed for this packet. As long as the flags are clear, the procedure continues at state **620**; otherwise the procedure continues at state **626**.

In state **620**, the flow database manager determines whether more data is expected during this flow. As discussed

above, a flow may be limited in duration to a single datagram. Therefore, in state **620** the FDBM determines if this packet appears to be the final portion of data for this flow's datagram. Illustratively, this determination is made on the basis of the amount of data included with the present packet. As one skilled in the art will appreciate, a datagram comprising more data than can be carried in one packet is sent via multiple packets. The typical manner of disseminating a datagram among multiple packets is to put as much data as possible into each packet. Thus, each packet except the last is usually equal or nearly equal in size to the maximum transfer unit (MTU) allowed for the network over which the packets are sent. The last packet will hold the remainder, usually causing it to be smaller than the MTU.

Therefore, one manner of identifying the final portion of data in a flow's datagram is to examine the size of each packet and compare it to a figure (e.g., MTU) that a packet is expected to exceed except when carrying the last data portion. It was described above that control information is received by FDBM **108** from header parser **106**. An indication of the size of the data carried by a packet may be included in this information. In particular, header parser **106** in one embodiment of the invention is configured to compare the size of each packet's data portion to a pre-selected value. In one embodiment of the invention this value is programmable. This value is set, in the illustrated embodiment of the invention, to the maximum amount of data a packet can carry without exceeding MTU. In one alternative embodiment, the value is set to an amount somewhat less than the maximum amount of data that can be carried.

Thus, in state **620**, flow database manager **108** determines whether the received packet appears to carry the final portion of data for the flow's datagram. If not, the procedure continues to state **626**.

In state **622**, it has been ascertained that the packet is compatible with pre-selected protocols and is suitable for one or more functions offered by NIC **100**. In particular, the packet has been formatted appropriately for one or more of the functions discussed above. FDBM **108** has determined that the received packet is part of an existing flow, is compatible with the pre-selected protocols and contains the next portion of data for the flow (but not the final portion). Further, the packet is not part of an attempt to re-set a flow/connection, and important flags have their expected values. Thus, flow database **110** can be updated as follows.

The activity indicator (e.g., flow activity indicator **524** of FIG. **5**) for this flow is modified to reflect the recent flow activity. In one embodiment of the invention flow activity indicator **524** is implemented as a counter, or is associated with a counter, that is incremented each time data is received for a flow. In another embodiment of the invention, an activity indicator or counter is updated every time a packet having a flow key matching a valid flow (e.g., whether or not the packet includes data) is received.

In the illustrated embodiment, after a flow activity indicator or counter is incremented it is examined to determine if it "rolled over" to zero (i.e., whether it was incremented past its maximum value). If so, the counter and/or the flow activity indicators for each entry in flow database **110** are set to zero and the current flow's activity indicator is once again incremented. Thus, in one embodiment of the invention the rolling over of a flow activity counter or indicator causes the re-initialization of the flow activity mechanism for flow database **110**. Thereafter, the counter is incremented and the flow activity indicators are again updated as described previously. One skilled in the art will recognize that there are many other suitable methods that may be applied in an embodiment of the present invention to indicate that one flow was active more recently than another was.

Also in state **622**, flow sequence number **522** is updated. Illustratively, the new flow sequence number is determined by adding the size of the newly received data to the existing flow sequence number. Depending upon the configuration of the packet (e.g., values in its headers), this sum may need to be adjusted. For example, this sum may indicate simply the total amount of data received thus far for the flow's datagram. Therefore, a value may need to be added (e.g., one byte) in order to indicate a sequence number of the next byte of data for the datagram. As one skilled in the art will recognize, other suitable methods of ensuring that data is received in order may be used in place of the scheme described here.

Finally, in state **622** in one embodiment of the invention, flow validity indicator **520** is set or reset to indicate the flow's validity.

Then, in state **624**, an operation code is associated with the packet. In the illustrated embodiment of the invention, operation codes comprise codes generated by flow database manager **108** and stored in control queue **118**. In this embodiment, an operation code is three bits in size, thus allowing for eight operation codes. Operation codes may have a variety of other forms and ranges in alternative embodiments. For the illustrated embodiment of the invention, TABLE 1 describes each operation code in terms of the criteria that lead to each code's selection and the ramifications of that selection. For purposes of TABLE 1, setting up a flow comprises inserting a flow into flow database **110**. Tearing down a flow comprises removing or invalidating a flow in flow database **110**. The re-assembly of data is discussed in a following section describing DMA engine **120**.

In the illustrated embodiment of the invention, operation code 4 is selected in state **624** for packets in the present context of the procedure (e.g., compatible packets carrying the next, but not last, data portion of a flow). Thus, the existing flow is not torn down and there is no need to set up a new flow. As described above, a compatible packet in this embodiment is a packet conforming to one or more of the pre-selected protocols. By changing or augmenting the pre-selected protocols, virtually any packet may be compatible in an alternative embodiment of the invention.

Returning now to FIGS. **6A–6E**, after state **624** the illustrated procedure ends at state **670**.

In state **626** (reached from state **618** or state **620**), operation code 3 is selected for the packet. Illustratively, operation code 3 indicates that the packet is compatible and matches a valid flow (e.g., the packet's flow key matches the flow key of a valid flow in FDB **110**). Operation code 3 may also signify that the packet contains data, does not constitute an attempt to re-synchronize or reset a communication flow/connection and the packet's sequence number matches the expected sequence number (from flow database **110**). But, either an important flag (e.g., one of the TCP flags URG, PSH, RST or FIN) is set (determined in state **618**) or the packet's data is less than the threshold value described above (in state **620**), thus indicating that no more data is likely to follow this packet in this flow. Therefore, the existing flow is torn down but no new flow is created. Illustratively, the flow may be torn down by clearing the flow's validity indicator (e.g., setting it to zero). After state **626**, the illustrated procedure ends at state **670**.

In state **628** (reached from state **616**), operation code 2 is selected for the packet. In the present context, operation code 2 may indicate that the packet is compatible, matches

a valid flow (e.g., the packet's flow key matches the flow key of a valid flow in FDB **110**), contains data and does not constitute an attempt to re-synchronize or reset a communication flow/connection. However, the sequence number extracted from the packet (in state **616**) does not match the expected sequence number from flow database **110**. This may occur, for example, when a packet is received out of order. Thus, the existing flow is torn down but no new flow is established. Illustratively, the flow may be torn down by clearing the flow's validity indicator (e.g., setting it to zero). After state **628**, the illustrated procedure ends at state **670**.

State **630** is entered from state **614** when it is determined that the received packet constitutes an attempt to reset a communication flow or connection (e.g., the TCP SYN bit is set). In state **630**, flow database manager **108** determines wbether more data is expected to follow. As explained in conjunction with state **620**, this determination may be made on the basis of control information received by the flow database manager from the header parser. If more data is expected (e.g., the amount of data in the packet equals or exceeds a threshold value), the procedure continues at state **634**.

In state **632**, operation code 2 is selected for the packet. Operation code 2 was also selected in state **628** in a different context. In the present context, operation code 2 may indicate that the packet is compatible, matches a valid flow and contains data. Operation code 2 may also signify in this context that the packet constitutes an attempt to re-synchronize or reset a communication flow or connection, but that no more data is expected once the flow/connection is reset. Therefore, the existing flow is torn down and no new flow is established. Illustratively, the flow may be torn down by clearing the flow's validity indicator (e.g., setting it to zero). After state **632**, the illustrated procedure ends at state **670**.

In state **634**, flow database manager **108** responds to an attempt to reset or re-synchronize a communication flow/connection whereby additional data is expected. Thus, the existing flow is torn down and replaced as follows. The existing flow may be identified by the flow number retrieved in state **610** or by the packet's flow key. The flow's sequence number (e.g., flow sequence number **522** in FIG. **5**) is set to the next expected value. Illustratively, this value depends upon the sequence number (e.g., TCP sequence number) retrieved from the packet (e.g., by header parser **106**) and the amount of data included in the packet. In one embodiment of the invention these two values are added to determine a new flow sequence number. As discussed previously, this sum may need to be adjusted (e.g., by adding one). Also in state **634**, the flow activity indicator is updated (e.g., incremented). As explained in conjunction with state **622**, if the flow activity indicator rolls over, the activity indicators for all flows in the database are set to zero and the present flow is again incremented. Finally, the flow validity indicator is set to indicate that the flow is valid.

In state **636**, operation code 7 is selected for the packet. In the present context, operation code 7 indicates that the packet is compatible, matches a valid flow and contains data. Operation code 7 may further signify, in this context, that the packet constitutes an attempt to re-synchronize or reset a communication flow/connection and that additional data is expected once the flow/connection is reset. In effect, therefore, the existing flow is torn down and a new one (with the same flow key) is stored in its place. After state **636**, the illustrated procedure ends at end state **670**.

State **638** is entered after state **612** when it is determined that the received packet contains no data. This often indicates that the packet is a control packet. In state **638**, flow database manager **108** determines whether one or more flags extracted from the packet by the header parser match expected or desired values. For example, in one embodiment of the invention the TCP flags URG, PSH, RST and FIN must be clear in order for DMA engine **120** to re-assemble data from multiple related packets (e.g., packets having an identical flow key). As discussed above, the TCP SYN bit may also be examined. In the present context (e.g., a packet with no data), the SYN bit is also expected to be clear (e.g., to store a value of zero). If the flags (and SYN bit) have their expected values the procedure continues at state **642**. If, however, any of these flags are set, an exceptional condition may exist, thus making it possible that one or more functions offered by NIC **100** (e.g., data re-assembly, batch processing, load distribution) are unsuitable for this packet, in which case the procedure proceeds to state **640**.

In state **640**, operation code 1 is selected for the packet. Illustratively, operation code 1 indicates that the packet is compatible and matches a valid flow, but does not contain any data and one or more important flags or bits in the packet's header(s) are set. Thus, the existing flow is torn down and no new flow is established. Illustratively, the flow may be torn down by clearing the flow's validity indicator (e.g., setting it to zero). After state **640**, the illustrated procedure ends at end state **670**.

In state **642**, the flow's activity indicator is updated (e.g., incremented) even though the packet contains no data. As described above in conjunction with state **622**, if the activity indicator rolls over, in a present embodiment of the invention all flow activity indicators in the database are set to zero and the current flow is again incremented. The flow's validity indicator may also be reset, as well as the flow's sequence number.

In state **644**, operation code 0 is selected for the packet. Illustratively, operation code 0 indicates that the packet is compatible, matches a valid flow, and that the packet does not contain any data. The packet may, for example, be a control packet. Operation code 0 further indicates that none of the flags checked by header parser **106** and described above (e.g., URG, PSH, RST and FIN) are set. Thus, the existing flow is not torn down and no new flow is established. After state **644**, the illustrated procedure ends at end state **670**.

State **646** is entered from state **608** if the packet's flow key does not match any of the flow keys of valid flows in the flow database. In state **646**, FDBM **108** determines whether flow database **110** is full and may save some indication of whether the database is full. In one embodiment of the invention the flow database is considered full when the validity indicator (e.g., flow validity indicator **520** of FIG. **5**) is set for every flow number (e.g., for every flow in the database). If the database is full, the procedure continues at state **650**, otherwise it continues at state **648**.

In state **648**, the lowest flow number of an invalid flow (e.g., a flow for which the associated flow validity indicator is equal to zero) is determined. Illustratively, this flow number is where a new flow will be stored if the received packet warrants the creation of a new flow. After state **648**, the procedure continues at state **652**.

In state **650**, the flow number of the least recently active flow is determined. As discussed above, in the illustrated embodiment of the invention a flow's activity indicator (e.g., flow activity indicator **524** of FIG. **5**) is updated (e.g., incremented) each time data is received for a flow. Therefore, in this embodiment the least recently active flow can be identified as the flow having the least recently

updated (e.g., lowest) flow activity indicator. Illustratively, if multiple flows have flow activity indicators set to a common value (e.g., zero), one flow number may be chosen from them at random or by some other criteria. After state **650**, the procedure continues at state **652**.

In state **652**, flow database manager **108** determines whether the packet contains data. Illustratively, the control information provided to FDBM **108** by the header parser indicates whether the packet has data. If the packet does not include data (e.g., the packet is a control packet), the illustrated procedure continues at state **668**.

In state **654**, flow database manager **108** determines whether the data received with the present packet appears to contain the final portion of data for the associated datagram/flow. As described in conjunction with state **620**, this determination may be made on the basis of the amount of data included with the packet. If the amount of data is less than a threshold value (a programmable value in the illustrated embodiment), then no more data is expected and this is likely to be the only data for this flow. In this case the procedure continues at state **668**. If, however, the data meets or exceeds the threshold value, in which case more data may be expected, the procedure proceeds to state **656**.

In state **656**, the values of certain flags are examined. These flags may include, for example, the URG, PSH, RST, FIN bits of a TCP header. If any of the examined flags do not have their expected or desired values (e.g., if any of the flags are set), an exceptional condition may exist making one or more of the functions of NIC **100** (e.g., data re-assembly, batch processing, load distribution) unsuitable for this packet. In this case the procedure continues at state **668**; otherwise the procedure proceeds to state **658**.

In state **658**, the flow database manager retrieves the information stored in state **646** concerning whether flow database **110** is full. If the database is full, the procedure continues at state **664**; otherwise the procedure continues at state **660**.

In state **660**, a new flow is added to flow database **110** for the present packet. Illustratively, the new flow is stored at the flow number identified or retrieved in state **648**. The addition of a new flow may involve setting a sequence number (e.g., flow sequence number **522** from FIG. **5**). Flow sequence number **522** may be generated by adding a sequence number (e.g., TCP sequence number) retrieved from the packet and the amount of data included in the packet. As discussed above, this sum may need to be adjusted (e.g., by adding one).

Storing a new flow may also include initializing an activity indicator (e.g., flow activity indicator **524** of FIG. **5**). In one embodiment of the invention this initialization involves storing a value retrieved from a counter that is incremented each time data is received for a flow. Illustratively, if the counter or a flow activity indicator is incremented past its maximum storable value, the counter and all flow activity indicators are cleared or reset. Also in state **660**, a validity indicator (e.g., flow validity indicator **520** of FIG. **5**) is set to indicate that the flow is valid. Finally, the packet's flow key is also stored in the flow database, in the entry corresponding to the assigned flow number.

In state **662**, operation code 6 is selected for the packet. Illustratively, operation code 6 indicates that the packet is compatible, did not match any valid flows and contains the first portion of data for a new flow. Further, the packet's flags have their expected or necessary values, additional data is expected in the flow and the flow database is not full. Thus, operation code 6 indicates that there is no existing flow to tear down and that a new flow has been stored in the flow database. After state **662**, the illustrated procedure ends at state **670**.

new flow

In state **664**, an existing entry in the flow database is replaced so that a new flow, initiated by the present packet, can be stored. Therefore, the flow number of the least recently active flow, identified in state **650**, is retrieved. This flow may be replaced as follows. The sequence number of the existing flow (e.g., flow sequence number **522** of FIG. **5**) is replaced with a value derived by combining a sequence number extracted from the packet (e.g., TCP sequence number) with the size of the data portion of the packet. This sum may need to be adjusted (e.g., by adding one). Then the existing flow's activity indicator (e.g., flow activity indicator **524**) is replaced. For example, the value of a flow activity counter may be copied into the flow activity indicator, as discussed above. The flow's validity indicator (e.g., flow validity indicator **520** of FIG. **5**) is then set to indicate that the flow is valid. Finally, the flow key of the new flow is stored.

In state **666**, operation code 7 is selected for the packet. Operation code 7 was also selected in state **636**. In the present context, operation code 7 may indicate that the packet is compatible, did not match the flow key of any valid flows and contains the first portion of data for a new flow. Further, the packet's flags have compatible values and additional data is expected in the flow. Lastly, however, in this context operation code 7 indicates that the flow database is full, so an existing entry was torn down and the new one stored in its place. After state **666**, the illustrated procedure ends at end state **670**.

In state **668**, operation code 5 is selected for the packet. State **668** is entered from various states and operation code 5 thus represents a variety of possible conditions or situations. For example, operation code 5 may be selected when a No_Assist signal is detected (in state **604**) for a packet. As discussed above, the No_Assist signal may indicate that the corresponding packet is not compatible with a set of pre-selected protocols. In this embodiment of the invention, incompatible packets are ineligible for one or more of the various functions of NIC **100** (e.g., data re-assembly, batch processing, load distribution).

State **668** may also be entered, and operation code 5 selected, from state **652**, in which case the code may indicate that the received packet does not match any valid flow keys and, further, contains no data (e.g., it may be a control packet).

State **668** may also be entered from state **654**. In this context operation code 5 may indicate that the packet does not match any valid flow keys. It may further indicate that the packet contains data, but that the size of the data portion is less than the threshold discussed in conjunction with state **654**. In this context, it appears that the packet's data is complete (e.g., comprises all of the data for a datagram), meaning that there is no other data to re-assemble with this packet's data and therefore there is no reason to make a new entry in the database for this one-packet flow.

Finally, state **668** may also be entered from state **656**. In this context, operation code 5 may indicate that the packet does not match any valid flow keys, contains data, and more data is expected, but at least one flag in one or more of the packet's protocol headers does not have its expected value. For example, in one embodiment of the invention the TCP flags URG, PSH, RST and FIN are expected to be clear. If any of these flags are set an exceptional condition may exist, thus making it possible that one of the functions offered by NIC **100** is unsuitable for this packet.

As TABLE 1 reflects, there is no flow to tear down and no new flow is established when operation code 5 is selected. Following state **668**, the illustrated procedure ends at state **670**.

One skilled in the art will appreciate that the procedure illustrated in FIGS. **6A–6E** and discussed above is but one suitable procedure for maintaining and updating a flow database and for determining a packet's suitability for certain processing functions. In particular, different operation codes may be utilized or may be implemented in a different manner, a goal being to produce information for later processing of the packet through NIC **100**.

Although operation codes are assigned for all packets by a flow database manager in the illustrated procedure, in an alternative procedure an operation code assigned by the FDBM may be replaced or changed by another module of NIC **100**. This may be done to ensure a particular method of treating certain types of packets. For example, in one embodiment of the invention IPP module **104** assigns a predetermined operation code (e.g., operation code 2 of TABLE 1) to jumbo packets (e.g., packets greater in size than MTU) so that DMA engine **120** will not re-assemble them. In particular, the IPP module may independently determine that the packet is a jumbo packet (e.g., from information provided by a MAC module) and therefore assign the predetermined code. Illustratively, header parser **106** and FDBM **108** perform their normal functions for a jumbo packet and IPP module **104** receives a first operation code assigned by the FDBM. However, the IPP module replaces that code before storing the jumbo packet and information concerning the packet. In one alternative embodiment header parser **106** and/or flow database manager **108** may be configured to recognize a particular type of packet (e.g., jumbo) and assign a predetermined operation code.

The operation codes applied in the embodiment of the invention illustrated in FIGS. **6A–6E** are presented and explained in the following TABLE 1. TABLE 1 includes illustrative criteria used to select each operation code and illustrative results or effects of each code.

TABLE 1

Op. Code	Criteria for Selection	Result of Operation Code
0	Compatible control packet with clear flags; a flow was previously established for this flow key.	Do not set up a new flow; Do not tear down existing flow; Do not re-assemble data (packet contains no data).
1	Compatible control packet with at least one flag or SYN bit set; a flow was previously established.	Do not set up a new flow; Tear down existing flow; Do no re-assemble data (packet contains no data).
2	Compatible packet whose sequence number does not match sequence number in flow database, or SYN bit is set (indicating attempt to re-establish a connection) but there is no more data to come; a flow was previously established. -- Or -- Jumbo packet.	Do not set up a new flow; Tear down existing flow; Do not re-assemble packet data.
3	A compatible packet carrying a final portion of flow data, or a flag is set (but packet is in sequence, unlike operation code 2); a flow was previously established.	Do not set up a new flow; Tear down existing flow; Re-assemble data with previous packets.
4	Receipt of next compatible packet in sequence; a flow was previously established.	Do not set up a new flow; Do not tear down existing flow; Re-assemble data with other packets.
5	Packet cannot be re-assembled because: incompatible, a flag is set, packet contains no data or there is no more data to come. No flow was previously established.	Do not set up a flow; There is no flow to tear down; Do not re-assemble.
6	First compatible packet of a new flow; no flow was previously established.	Set up a new flow; There is no flow to tear down; Re-assemble data with packets to follow.
7	First compatible packet of a new flow, but flow database is full; no flow was previously established. – Or – Compatible packet, SYN bit is set and additional data will follow; a flow was previously established.	Replace existing flow; Re-assemble data with packets to follow.

One Embodiment of a Load Distributor

In one embodiment of the invention, load distributor **112** enables the processing of packets through their protocol stacks to be distributed among a number of processors. Illustratively, load distributor **112** generates an identifier (e.g., a processor number) of a processor to which a packet is to be submitted. The multiple processors may be located within a host computer system that is served by NIC **100**. In one alternative embodiment, one or more processors for manipulating packets through a protocol stack are located on NIC **100**.

Without an effective method of sharing or distributing the processing burden, one processor could become overloaded if it were required to process all or most network traffic received at NIC **100**, particularly in a high-speed network environment. The resulting delay in processing network traffic could deteriorate operations on the host computer system as well as other computer systems communicating with the host system via the network.

As one skilled in the art will appreciate, simply distributing packets among processors in a set of processors (e.g., such as in a round-robin scheme) may not be an efficient plan. Such a plan could easily result in packets being processed out of order. For example, if two packets from one communication flow or connection that are received at a network interface in the correct order were submitted to two different processors, the second packet may be processed before the first. This could occur, for example, if the processor that received the first packet could not immediately process the packet because it was busy with another task. When packets are processed out of order a recovery scheme must generally be initiated, thus introducing even more inefficiency and more delay.

Therefore, in a present embodiment of the invention packets are distributed among multiple processors based upon their flow identities. As described above, a header parser may generate a flow key from layer three (e.g., IP) and layer four (e.g., TCP) source and destination identifiers retrieved from a packet's headers. The flow key may be used to identify the communication flow to which the packet belongs. Thus, in this embodiment of the invention all packets having an identical flow key are submitted to a single processor. As long as the packets are received in order by NIC **100**, they should be provided to the host computer and processed in order by their assigned processor.

Illustratively, multiple packets sent from one source entity to one destination entity will have the same flow key even if the packets are part of separate datagrams, as long as their layer three and layer four identifiers remain the same. As discussed above, separate flows are set up and torn down for

each datagram within one TCP end-to-end connection. Therefore, just as all packets within one flow are sent to one processor, all packets within a TCP end-to-end connection will also be sent to the same processor. This helps ensure the correct ordering of packets for the entire connection, even between datagrams.

Depending upon the network environment in which NIC **100** operates (e.g., the protocols supported by network **102**), the flow key may be too large to use as an identifier of a processor. In one embodiment of the invention described above, for example, a flow key measures 288 bits. Meanwhile, the number of processors participating in the load-balancing scheme may be much smaller. For example, in the embodiment of the invention described below in conjunction with FIG. **7**, a maximum of sixty-four processors is supported. Thus, in this embodiment only a six-bit number is needed to identify the selected processor. The larger flow key may therefore be mapped or hashed into a smaller range of values.

FIG. **7** depicts one method of generating an identifier (e.g., a processor number) to specify a processor to process a packet received by NIC **100**, based on the packet's flow key. In this embodiment of the invention, network **102** is the Internet and a received packet is formatted according to a compatible protocol stack (e.g., Ethernet at layer two, IP at layer three and TCP at layer four).

State **700** is a start state. In state **702** a packet is received by NIC **100** and a header portion of the packet is parsed by header parser **106** (a method of parsing a packet is described in a previous section). In state **704**, load distributor **112** receives the packet's flow key that was generated by header parser **106**.

Because a packet's flow key is 288 bits wide in this embodiment, in state **706** a hashing function is performed to generate a value that is smaller in magnitude. The hash operation may, for example, comprise a thirty-two bit CRC (cyclic redundancy check) function such as ATM (Asynchronous Transfer Mode) Adaptation Layer 5 (AAL5). AAL5 generates thirty-two bit numbers that are fairly evenly distributed among the 2^{32} possible values. Another suitable method of hashing is the standard Ethernet CRC-32 function. Other hash functions that are capable of generating relatively small numbers from relatively large flow keys, where the numbers generated are well distributed among a range of values, are also suitable.

With the resulting hash value, in state **708** a modulus operation is performed over the number of processors available for distributing or sharing the processing. Illustratively, software executing on the host computer (e.g., a device driver for NIC **100**) programs or stores the number of processors such that it may be read or retrieved by load distributor **112** (e.g., in a register). The number of processors available for load balancing may be all or a subset of the number of processors installed on the host computer system. In the illustrated embodiment, the number of processors available in a host computer system is programmable, with a maximum value of sixty-four. The result of the modulus operation in this embodiment, therefore, is the number of the processor (e.g., from zero to sixty-three) to which the packet is to be submitted for processing. In this embodiment of the invention, load distributor **112** is implemented in hardware, thus allowing rapid execution of the hashing and modulus functions. In an alternative embodiment of the invention, virtually any number of processors may be accommodated.

In state **710**, the number of the processor that will process the packet through its protocol stack is stored in the host computer's memory. Illustratively, state **710** is performed in parallel with the storage of the packet in a host memory buffer. As described in a following section, in one embodiment of the invention a descriptor ring in the host computer's memory is constructed to hold the processor number and possibly other information concerning the packet (e.g., a pointer to the packet, its size, its TCP checksum).

A descriptor ring in this embodiment is a data structure comprising a number of entries, or "descriptors," for storing information to be used by a network interface circuit's host computer system. In the illustrated embodiment, a descriptor temporarily stores packet information after the packet has been received by NIC **100**, but before the packet is processed by the host computer system. The information stored in a descriptor may be used, for example, by the device driver for NIC **100** or for processing the packet through its protocol stack.

In state **712**, an interrupt or other alert is issued to the host computer to inform it that a new packet has been delivered from NIC **100**. In an embodiment of the invention in which NIC **100** is coupled to the host computer by a PCI (Peripheral Component Interconnect) bus, the INTA signal may be asserted across the bus. A PCI controller in the host receives the signal and the host operating system is alerted (e.g., via an interrupt).

In state **714**, software operating on the host computer (e.g., a device driver for NIC **100**) is invoked (e.g., by the host computer's operating system interrupt handler) to act upon a newly received packet. The software gathers information from one or more descriptors in the descriptor ring and places information needed to complete the processing of each new packet into a queue for the specified processor (i.e., according to the processor number stored in the packet's descriptor). Illustratively, each descriptor corresponds to a separate packet. The information stored in the processor queue for each packet may include a pointer to a buffer containing the packet, the packet's TCP checksum, offsets of one or more protocol headers, etc. In addition, each processor participating in the load distribution scheme may have an associated queue for processing network packets. In an alternative embodiment of the invention, multiple queues may be used (e.g., for multiple priority levels or for different protocol stacks).

Illustratively, one processor on the host computer system is configured to receive all alerts and/or interrupts associated with the receipt of network packets from NIC **100** and to alert the appropriate software routine or device driver. This initial processing may, alternatively, be distributed among multiple processors. In addition, in one embodiment of the invention a portion of the retrieval and manipulation of descriptor contents is performed as part of the handling of the interrupt that is generated when a new packet is stored in the descriptor ring. The processor selected to process the packet will perform the remainder of the retrieval/manipulation procedure.

In state **716**, the processor designated to process a new packet is alerted or woken. In an embodiment of the invention operating on a Solaris™ workstation, individual processes executed by the processor are configured as "threads." A thread is a process running in a normal mode (e.g., not at an interrupt level) so as to have minimal impact on other processes executing on the workstation. A normal mode process may, however, execute at a high priority. Alternatively, a thread may run at a relatively low interrupt level.

A thread responsible for processing an incoming packet may block itself when it has no packets to process, and awaken when it has work to do. A "condition variable" may

be used to indicate whether the thread has a packet to process. Illustratively, the condition variable is set to a first value when the thread is to process a packet (e.g., when a packet is received for processing by the processor) and is set to a second value when there are no more packets to process. In the illustrated embodiment of the invention, one condition variable may be associated with each processor's queue.

In an alternative embodiment, the indicated processor is alerted in state **716** by a "cross-processor call." A cross-processor call is one way of communicating among processors whereby one processor is interrupted remotely by another processor. Other methods by which one processor alerts, or dispatches a process to, another processor may be used in place of threads and cross-processor calls.

In state **718**, a thread or other process on the selected processor begins processing the packet that was stored in the processor's queue. Methods of processing a packet through its protocol stack are well known to those skilled in the art and need not be described in detail. The illustrated procedure then ends with end state **720**.

In one alternative embodiment of the invention, a high-speed network interface is configured to receive and process ATM (Asynchronous Transfer Mode) traffic. In this embodiment, a load distributor is implemented as a set of instructions (e.g., as software) rather than as a hardware module. As one skilled in the art is aware, ATM traffic is connection-oriented and may be identified by a virtual connection identifier (VCI), which corresponds to a virtual circuit established between the packet's source and destination entities. Each packet that is part of a virtual circuit includes the VCI in its header.

Advantageously, a VCI is relatively small in size (e.g., sixteen bits). In this alternative embodiment, therefore, a packet's VCI may be used in place of a flow key for the purpose of distributing or sharing the burden of processing packets through their protocol stacks. Illustratively, traffic from different VCIs is sent to different processors, but, to ensure correct ordering of packets, all packets having the same VCI are sent to the same processor. When an ATM packet is received at a network interface, the VCI is retrieved from its header and provided to the load distributor. The modulus of the VCI over the number of processors that are available for load distribution is then computed. Similar to the illustrated embodiment, the packet and its associated processor number are then provided to the host computer.

As described above, load distribution in a present embodiment of the invention is performed on the basis of a packet's layer three and/or layer four source and destination entity identifiers. In an alternative embodiment of the invention, however, load distribution may be performed on the basis of layer two addresses. In this alternative embodiment, packets having the same Ethernet source and destination addresses, for example, are sent to a single processor.

As one of skill in the art will recognize, however, this may result in a processor receiving many more packets than it would if layer three and/or layer four identifiers were used. For example, if a large amount of traffic is received through a router situated near (in a logical sense) to the host computer, the source Ethernet address for all of the traffic may be the router's address even though the traffic is from a multitude of different end users and/or computers. In contrast, if the host computer is on the same Ethernet segment as all of the end users/computers, the layer two source addresses will show greater variety and allow more effective load sharing.

Other methods of distributing the processing of packets received from a network may differ from the embodiment illustrated in FIG. **7** without exceeding the scope of the invention. In particular, one skilled in the art will appreciate that many alternative procedures for assigning a flow's packets to a processor and delivering those packets to the processor may be employed.

One Embodiment of a Packet Queue

As described above, packet queue **116** stores packets received from IPP module **104** prior to their re-assembly by DMA engine **120** and their transfer to the host computer system. FIG. **8** depicts packet queue **116** according to one embodiment of the invention.

In the illustrated embodiment, packet queue **116** is implemented as a FIFO (First-In First-Out) queue containing up to 256 entries. Each packet queue entry in this embodiment stores one packet plus various information concerning the packet. For example, entry **800** includes packet portion **802** plus a packet status portion. Because packets of various sizes are stored in packet queue **116**, packet portion **802** may include filler **802***a* to supplement the packet so that the packet portion ends at an appropriate boundary (e.g., byte, word, double word).

Filler **802***a* may comprise random data or data having a specified pattern. Filler **802** a may be distinguished from the stored packet by the pattern of the filler data or by a tag field.

Illustratively, packet status information includes TCP checksum value **804** and packet length **806** (e.g., length of the packet stored in packet portion **802**). Storing the packet length may allow the packet to be easily identified and retrieved from packet portion **802**. Packet status information may also include diagnostic/status information **808**. Diagnostic/status information **808** may include a flag indicating that the packet is bad (e.g., incomplete, received with an error), an indicator that a checksum was or was not computed for the packet, an indicator that the checksum has a certain value, an offset to the portion of the packet on which the checksum was computed, etc. Other flags or indicators may also be included for diagnostics, filtering, or other purposes. In one embodiment of the invention, the packet's flow key (described above and used to identify the flow comprising the packet) and/or flow number (e.g., the corresponding index of the packet's flow in flow database **110**) are included in diagnostic/status information **808**. In another embodiment, a tag field to identify or delimit filler **802***a* is included in diagnostic/status information **808**.

In one alternative embodiment of the invention, any or all of the packet status information described above is stored in control queue **118** rather than packet queue **116**.

In the illustrated embodiment of the invention packet queue **116** is implemented in hardware (e.g., as random access memory). In this embodiment, checksum value **804** is sixteen bits in size and may be stored by checksum generator **114**. Packet length **806** is fourteen bits large and may be stored by header parser **106**. Finally, portions of diagnostic/status information **808** may be stored by one or more of IPP module **104**, header parser **106**, flow database manager **108**, load distributor **112** and checksum generator **114**.

Packet queue **116** in FIG. **8** is indexed with two pointers. Read pointer **810** identifies the next entry to be read from the queue, while write pointer **812** identifies the entry in which the next received packet and related information is to be stored. As explained in a subsequent section, the packet stored in packet portion **802** of an entry is extracted from packet queue **116** when its data is to be-reassembled by DMA engine **120** and/or transferred to the host computer system.

One Embodiment of a Control Queue

In one embodiment of the invention, control queue **118** stores control and status information concerning a packet

received by NIC **100**. In this embodiment, control queue **118** retains information used to enable the batch processing of protocol headers and/or the re-assembly of data from multiple related packets. Control queue **118** may also store information to be used by the host computer or a series of instructions operating on a host computer (e.g., a device driver for NIC **100**). The information stored in control queue **118** may supplement or duplicate information stored in packet queue **116**.

FIG. **9** depicts control queue **118** in one embodiment of the invention. The illustrated control queue contains one entry for each packet stored in packet queue **116** (e.g., up to 256 entries). In one embodiment of the invention each entry in control queue **118** corresponds to the entry (e.g., packet) in packet queue **116** having the same number. FIG. **9** depicts entry **900** having various fields, such as CPU number **902**, No_Assist signal **904**, operation code **906**, payload offset **908**, payload size **910** and other status information **912**. An entry may also include other status or control information (not shown in FIG. **9**). Entries in control queue **118** in alternative embodiments of the invention may comprise different information.

CPU (or processor) number **902**, discussed in a previous section, indicates which one of multiple processors on the host computer system should process the packet's protocol headers. Illustratively, CPU number **902** is six bits in size. No_Assist signal **904**, also described in a preceding section, indicates whether the packet is compatible with (e.g., is formatted according to) any of a set of pre-selected protocols that may be parsed by header parser **106**. No_Assist signal **904** may comprise a single flag (e.g. one bit). In one embodiment of the invention the state or value of No_Assist signal **904** may be used by flow database manager **108** to determine whether a packet's data is re-assembleable and/or whether its headers may be processed with those of related packets. In particular, the FDBM may use the No_Assist signal in determining which operation code to assign to the packet.

Operation code **906** provides information to DMA engine **120** to assist in the re-assembly of the packet's data. As described in a previous section, an operation code may indicate whether a packet includes data or whether a packet's data is suitable for re-assembly. Illustratively, operation code **906** is three bits in size. Payload offset **908** and payload size **910** correspond to the offset and size of the packet's TCP payload (e.g., TCP data), respectively. These fields may be seven and fourteen bits large, respectively.

In the illustrated embodiment, other status information **912** includes diagnostic and/or status information concerning the packet. Status information **912** may include a starting position for a checksum calculation (which may be seven bits in size), an offset of the layer three (e.g., IP) protocol header (which may also be seven bits in size), etc. Status information **912** may also include an indicator as to whether the size of the packet exceeds a first threshold (e.g., whether the packet is greater than 1522 bytes) or falls under a second threshold (e.g., whether the packet is 256 bytes or less). This information may be useful in re-assembling packet data. Illustratively, these indicators comprise single-bit flags.

In one alternative embodiment of the invention, status information **912** includes a packet's flow key and/or flow number (e.g., the index of the packet's flow in flow database **110**). The flow key or flow number may, for example, be used for debugging or other diagnostic purposes. In one embodiment of the invention, the packet's flow number may be stored in status information **912** so that multiple packets in a single flow may be identified. Such related packet may then be collectively transferred to and/or processed by a host computer.

FIG. **9** depicts a read pointer and a write pointer for indexing control queue **118**. Read pointer **914** indicates an entry to be read by DMA engine **120**. Write pointer **916** indicates the entry in which to store information concerning the next packet stored in packet queue **116**.

In an alternative embodiment of the invention, a second read pointer (not shown in FIG. **9**) may be used for indexing control queue **118**. As described in a later section, when a packet is to be transferred to the host computer, information drawn from entries in the control queue is searched to determine whether a related packet (e.g., a packet in the same flow as the packet to be transferred) is also going to be transferred. If so, the host computer is alerted so that protocol headers from the related packets may be processed collectively. In this alternative embodiment of the invention, related packets are identified by matching their flow numbers (or flow keys) in status information **912**. The second read pointer may be used to look ahead in the control queue for packets with matching flow numbers.

In one embodiment of the invention CPU number **902** may be stored in the control queue by load distributor **112** and No_Assist signal **904** may be stored by header parser **106**. Operation code **906** may be stored by flow database manager **108**, and payload offset **908** and payload size **910** may be stored by header parser **106**. Portions of other status information may be written by the preceding modules and/or others, such as IPP module **104** and checksum generator **114**. In one particular embodiment of the invention, however, many of these items of information are stored by IPP module **104** or some other module acting in somewhat of a coordinator role.

One Embodiment of a DMA Engine

FIG. **10** is a block diagram of DMA (Direct Memory Access) engine **120** in one embodiment of the invention. One purpose of DMA engine **120** in this embodiment is to transfer packets from packet queue **116** into buffers in host computer memory. Because related packets (e.g., packets that are part of one flow) can be identified by their flow numbers or flow keys, data from the related packets may be transferred together (e.g., in the same buffer). By using one buffer for data from one flow, the data can be provided to an application program or other destination in a highly efficient manner. For example, after the host computer receives the data, a page-flip operation may be performed to transfer the data to an application's memory space rather than performing numerous copy operations.

With reference back to FIGS. **1A–B**, a packet that is to be transferred into host memory by DMA engine **120** is stored in packet queue **116** after being received from network **102**. Header parser **106** parses a header portion of the packet and generates a flow key, and flow database manager **108** assigns an operation code to the packet. In addition, the communication flow that includes the packet is registered in flow database **110**. The packet's flow may be identified by its flow key or flow number (e.g., the index of the flow in flow database **110**). Finally, information concerning the packet (e.g., operation code, a packet size indicator, flow number) is stored in control queue **118** and, possibly, other portions or modules of NIC **100**, and the packet is transferred to the host computer by DMA engine **120**. During the transfer process, the DMA engine may draw upon information stored in the control queue to copy the packet into an appropriate buffer, as described below. Dynamic packet batching module **122** may also use information stored in the control queue, as discussed in detail in a following section.

With reference now to FIG. **10**, one embodiment of a DMA engine is presented. In this embodiment, DMA man-

ager **1002** manages the transfer of a packet, from packet queue **116**, into one or more buffers in host computer memory. Free ring manager **1012** identifies or receives empty buffers from host memory and completion ring manager **1014** releases the buffers to the host computer, as described below. The free ring manager and completion ring managers may be controlled with logic contained in DMA manager **1002**. In the illustrated embodiment, flow re-assembly table **1004**, header table **1006**, MTU table **1008** and jumbo table **1010** store information concerning buffers used to store different types of packets (as described below). Information stored in one of these tables may include a reference to, or some other means of identifying, a buffer. In FIG. **10**, DMA engine **120** is partially or fully implemented in hardware.

Empty buffers into which packets may be stored are identified via a free descriptor ring that is maintained in host memory. As one skilled in the art is aware, a descriptor ring is a data structure that is logically arranged as a circular queue. A descriptor ring contains descriptors for storing information (e.g., data, flag, pointer, address). In one embodiment of the invention, each descriptor stores its index within the free descriptor ring and an identifier (e.g., memory address, pointer) of a free buffer that may be used to store packets. In this embodiment a buffer is identified in a descriptor by its address in memory, although other means of identifying a memory buffer are also suitable. In one embodiment of the invention a descriptor index is thirteen bits large, allowing for a maximum of 8,192 descriptors in the ring, and a buffer address is sixty-four bits in size.

In the embodiment of FIG. **10**, software that executes on a host computer, such as a device driver for NIC **100**, maintains a free buffer array or other data structure (e.g., list, table) for storing references to (e.g., addresses of) the buffers identified in free descriptors. As descriptors are retrieved from the ring their buffer identifiers are placed in the array. Thus, when a buffer is needed for the storage of a packet, it may be identified by its index (e.g., cell, element) in the free buffer array. Then, when the buffer is no longer needed, it may be released to the host computer by placing its array index or reference in a completion descriptor. A packet stored in the buffer can then be retrieved by accessing the buffer identified in the specified element of the array. Thus, in this embodiment of the invention the size of a descriptor index (e.g., thirteen bits) may not limit the number of buffers that may be assigned by free ring manager **1012**. In particular, virtually any number of buffers or descriptors could be managed by the software. For example, in one alternative embodiment of the invention buffer identifiers may be stored in one or more linked lists after being retrieved from descriptors in a free descriptor ring. When the buffer is released to the host computer, a reference to the head of the buffer's linked list may be provided. The list could then be navigated to locate the particular buffer (e.g., by its address).

As one skilled in the art will appreciate, the inclusion of a limited number of descriptors in the free descriptor ring (e.g., 8,192 in this embodiment) means that they may be re-used in a round-robin fashion. In the presently described embodiment, a descriptor is just needed long enough to retrieve its buffer identifier (e.g., address) and place it in the free buffer array, after which it may be re-used relatively quickly. In other embodiments of the invention free descriptor rings having different numbers of free descriptors may be used, thus allowing some control over the rate at which free descriptors must be re-used.

In one alternative embodiment of the invention, instead of using a separate data structure to identify a buffer for storing a packet, a buffer may be identified within DMA engine **120** by the index of the free descriptor within the free descriptor ring that referenced the buffer. One drawback to this scheme when the ring contains a limited number of descriptors, however, is that a particular buffer's descriptor may need to be re-used before its buffer has been released to the host computer. Thus, either a method of avoiding or skipping the re-use of such a descriptor must be implemented or the buffer referenced by the descriptor must be released before the descriptor is needed again. Or, in another alternative, a free descriptor ring may be of such a large size that a lengthy or even virtually infinite period of time may pass from the time a free descriptor is first used until it needs to be re-used.

Thus, in the illustrated embodiment of the invention free ring manager **1012** retrieves a descriptor from the free descriptor ring, stores its buffer identifier (e.g., memory address) in a free buffer array, and provides the array index and/or buffer identifier to flow re-assembly table **1004**, header table **1006**, MTU table **1008** or jumbo table **1010**.

Free ring manager **1012** attempts to ensure that a buffer is always available for a packet. Thus, in one embodiment of the invention free ring manager **1012** includes descriptor cache **1012***a* configured to store a number of descriptors (e.g., up to eight) at a time. Whenever there are less than a threshold number of entries in the cache (e.g., five), additional descriptors may be retrieved from the free descriptor ring. Advantageously, the descriptors are of such a size (e.g., sixteen bytes) that some multiple (e.g., four) of them can be efficiently retrieved in a sixty-four byte cache line transfer from the host computer.

Returning now to the illustrated embodiment of the invention, each buffer in host memory is one memory page in size. However, buffers and the packets stored in the buffers may be divided into multiple categories based on packet size and whether a packet's data is being re-assembled. Re-assembly refers to the accumulation of data from multiple packets of a single flow into one buffer for efficient transfer from kernel space to user or application space within host memory. In particular, re-assembleable packets may be defined as packets that conform to a pre-selected protocol (e.g., a protocol that is parseable by header parser **106**). By filling a memory page with data for one destination, page-flipping may be performed to provide a page in kernel space to the application or user space. A packet's category (e.g., whether re-assembleable or non-re-assembleable) may be determined from information retrieved from the control queue or flow database manager. In particular, and as described previously, an operation code may be used to determine whether a packet contains a re-assembleable portion of data.

In the illustrated embodiment of the invention, data portions of related, re-assembleable, packets are placed into a first category of buffers—which may be termed re-assembly buffers. A second category of buffers, which may be called header buffers, stores the headers of those packets whose data portions are being re-assembled and may also store small packets (e.g., those less than or equal to 256 bytes in size). A third category of buffers, MTU buffers, stores non-re-assembleable packets that are larger than 256 bytes, but no larger than MTU size (e.g., 1522 bytes). Finally, a fourth category of buffers, jumbo buffers, stores jumbo packets (e.g., large packets that are greater than 1522 bytes in size) that are not being re-assembled. Illustratively, a jumbo packet may be stored intact (e.g., its headers and data portions kept together in one buffer) or its headers may be stored in a header buffer while its data portion is stored in an appropriate (e.g., jumbo) non-re-assembly buffer.

In one alternative embodiment of the invention, no distinction is made between MTU and jumbo packets. Thus, in this alternative embodiment, just three types of buffers are used: re-assembly and header buffers, as described above, plus non-re-assembly buffers. Illustratively, all non-small packets (e.g., larger than 256 bytes) that are not re-assembled are placed in a non-re-assembly buffer.

In another alternative embodiment, jumbo packets may be re-assembled in jumbo buffers. In particular, in this embodiment data portions of packets smaller than a predetermined size (e.g., MTU) are re-assembled in normal re-assembly buffers while data portions of jumbo packets (e.g., packets greater in size than MTU) are re-assembled in jumbo buffers. Re-assembly of jumbo packets may be particularly effective for a communication flow that comprises jumbo frames of a size such that multiple frames can fit in one buffer. Header portions of both types of packets may be stored in one type of header buffer or, alternatively, different header buffers may be used for the headers of the different types of re-assembleable packets.

In yet another alternative embodiment of the invention buffers may be of varying sizes and may be identified in different descriptor rings or other data structures. For example, a first descriptor ring or other mechanism may be used to identify buffers of a first size for storing large or jumbo packets. A second ring may store descriptors referencing buffers for MTU-sized packets, and another ring may contain descriptors for identifying page-sized buffers (e.g., for data re-assembly).

A buffer used to store portions of more than one type of packet—such as a header buffer used to store headers and small packets, or a non-re-assembly buffer used to store MTU and jumbo packets—may be termed a "hybrid" buffer.

Illustratively, each time a packet or a portion of a packet is stored in a buffer, completion ring manager **1014** populates a descriptor in a completion descriptor ring with information concerning the packet. Included in the information stored in a completion descriptor in this embodiment is a number or reference identifying the free buffer array cell or element in which an identifier (e.g., memory address) of a buffer in which a portion of the packet is stored. The information may also include an offset into the buffer (e.g., to the beginning of the packet portion), the identity of another free buffer array entry that stores a buffer identifier for a buffer containing another portion of the packet, a size of the packet, etc. A packet may be stored in multiple buffers, for example, if the packet data and header are stored separately (e.g., the packet's data is being re-assembled in a re-assembly buffer while the packet's header is placed in a header buffer). In addition, data portions of a jumbo packet or a re-assembly packet may span two or more buffers, depending on the size of the data portion.

A distinction should be kept in mind between a buffer identifier (e.g., the memory address of a buffer) and the entry in the free buffer array in which the buffer identifier is stored. In particular, it has been described above that when a memory buffer is released to a host computer it is identified to the host computer by its position within a free buffer array (or other suitable data structure) rather than by its buffer identifier. The host computer retrieves the buffer identifier from the specified array element and accesses the specified buffer to locate a packet stored in the buffer. As one skilled in the art will appreciate, identifying memory buffers in completion descriptors by the buffers' positions in a free buffer array can be more efficient than identifying them by their memory addresses. In particular, in FIG. **10** buffer identifiers are sixty-four bits in size while an index in a free buffer array or similar data structure will likely be far smaller. Using array positions thus saves space compared to using buffer identifiers. Nonetheless, buffer identifiers may be used to directly identify buffers in an alternative embodiment of the invention, rather than filtering access to them through a free buffer array. However, completion descriptors would have to be correspondingly larger in order to accommodate them.

A completion descriptor may also include one or more flags indicating the type or size of a packet, whether the packet data should be re-assembled, whether the packet is the last of a datagram, whether the host computer should delay processing the packet to await a related packet, etc. As described in a following section, in one embodiment of the invention dynamic packet batching module **122** determines, at the time a packet is transferred to the host computer, whether a related packet will be sent shortly. If so, the host computer may be advised to delay processing the transferred packet and await the related packet in order to allow more efficient processing.

A packet's completion descriptor may be marked appropriately when the buffer identified by its buffer identifier is to be released to the host computer. For example, a flag may be set in the descriptor to indicate that the packet's buffer is being released from DMA engine **120** to the host computer or software operating on the host computer (e.g., a driver associated with NIC **100**). In one embodiment of the invention, completion ring manager **1014** includes completion descriptor cache **1014***a*. Completion descriptor cache **1014***a* may store one or more completion descriptors for collective transfer from DMA engine **120** to the host computer.

Thus, empty buffers are retrieved from a free ring and used buffers are released to the host computer through a completion ring. One reason that a separate ring is employed to release used buffers to the host computer is that buffers may not be released in the order in which they were taken. In one embodiment of the invention, a buffer (especially a flow re-assembly buffer) may not be released until it is full. Alternatively, a buffer may be released at virtually any time, such as when the end of a communication flow is detected. Free descriptors and completion descriptors are further described below in conjunction with FIG. **12**.

Another reason that separate rings are used for free and completion descriptors is that the number of completion descriptors that are required in an embodiment of the invention may exceed the number of free descriptors provided in a free descriptor ring. For example, a buffer provided by a free descriptor may be used to store multiple headers and/or small packets. Each time a header or small packet is stored in the header buffer, however, a separate completion descriptor is generated. In an embodiment of the invention in which a header buffer is eight kilobytes in size, a header buffer may store up to thirty-two small packets. For each packet stored in the header buffer, another completion descriptor is generated.

FIG. **11** includes diagrams of illustrative embodiments of flow re-assembly table **1004**, header table **1006**, MTU table **1008** and jumbo table **1010**. One alternative embodiment of the invention includes a non-re-assembly table in place of MTU table **1008** and jumbo table **1010**, corresponding to a single type of non-re-assembly buffer for both MTU and jumbo packets. Jumbo table **1010** may also be omitted in another alternative embodiment of the invention in which jumbo buffers are retrieved or identified only when needed. Because a jumbo buffer is used only once in this alternative embodiment, there is no need to maintain a table to track its use.

Flow re-assembly table **1004** in the illustrated embodiment stores information concerning the re-assembly of packets in one or more communication flows. For each flow that is active through DMA engine **120**, separate flow re-assembly buffers may be used to store the flow's data. More than one buffer may be used for a particular flow, but each flow has one entry in flow re-assembly table **1004** with which to track the use of a buffer. As described in a previous section, one embodiment of the invention supports the interleaving of up to sixty-four flows. Thus, flow re-assembly buffer table **1004** in this embodiment maintains up to sixty-four entries. A flow's entry in the flow re-assembly table may match its flow number (e.g., the index of the flow's flow key in flow database **110**) or, in an alternative embodiment, an entry may be used for any flow.

In FIG. **11**, an entry in flow re-assembly table **1004** includes flow re-assembly buffer index **1102**, next address **1104** and validity indicator **1106**. Flow re-assembly buffer index **1102** comprises the index, or position, within a free buffer array or other data structure for storing buffer identifiers identified in free descriptors, of a buffer for storing data from the associated flow. Illustratively, this value is written into each completion descriptor associated with a packet whose data portion is stored in the buffer. This value may be used by software operating on the host computer to access the buffer and process the data. Next address **1104** identifies the location within the buffer (e.g., a memory address) at which to store the next portion of data. Illustratively, this field is updated each time data is added to the buffer. Validity indicator **1106** indicates whether the entry is valid. Illustratively, each entry is set to a valid state (e.g., stores a first value) when a first portion of data is stored in the flow's re-assembly buffer and is invalidated (e.g., stores a second value) when the buffer is full. When an entry is invalidated, the buffer may be released or returned to the host computer (e.g., because it is full).

Header table **1006** in the illustrated embodiment stores information concerning one or more header buffers in which packet headers and small packets are stored. In the illustrated embodiment of the invention, only one header buffer is active at a time. That is, headers and small packets are stored in one buffer until it is released, at which time a new buffer is used. In this embodiment, header table **1006** includes header buffer index **1112**, next address **1114** and validity indicator **1116**. Similar to flow re-assembly table **1004**, header buffer index **1112** identifies the cell or element in the free buffer array that contains a buffer identifier for a header buffer. Next address **1114** identifies the location within the header buffer at which to store the next header or small packet. This identifier, which may be a counter, may be updated each time a header or small packet is stored in the header buffer. Validity indicator **1116** indicates whether the header buffer table and/or the header buffer is valid. This indicator may be set to valid when a first packet or header is stored in a header buffer and may be invalidated when it is released to the host computer.

MTU table **1008** stores information concerning one or more MTU buffers for storing MTU packets (e.g., packets larger than 256 bytes but less than 1523 bytes) that are not being re-assembled. MTU buffer index **1122** identifies the free buffer array element that contains a buffer identifier (e.g., address) of a buffer for storing MTU packets. Next address **1124** identifies the location in the current MTU buffer at which to store the next packet. Validity indicator **1126** indicates the validity of the table entry. The validity indicator may be set to a valid state when a first packet is stored in the MTU buffer and an invalid state when the buffer is to be released to the host computer.

Jumbo table **1010** stores information concerning one or more jumbo buffers for storing jumbo packets (e.g., packets larger than 1522 bytes) that are not being re-assembled. Jumbo buffer index **1132** identifies the element within the free buffer array that stores a buffer identifier corresponding to a jumbo buffer. Next address **1134** identifies the location in the jumbo buffer at which to store the next packet. Validity indicator **1136** indicates the validity of the table entry. Illustratively, the validity indicator is set to a valid state when a first packet is stored in the jumbo buffer and is set to an invalid state when the buffer is to be released to the host computer.

In the embodiment of the invention depicted in FIG. **11**, a packet larger than a specified size (e.g., 256 bytes) is not re-assembled if it is incompatible with the pre-selected protocols for NIC **100** (e.g., TCP, IP, Ethernet) or if the packet is too large (e.g., greater than 1522 bytes). Although two types of buffers (e.g., MTU and jumbo) are used for non-re-assembleable packets in this embodiment, in an alternative embodiment of the invention any number may be used, including one. Packets less than the specified size are generally not re-assembled. Instead, as described above, they are stored intact in a header buffer.

In the embodiment of the invention depicted in FIG. **11**, next address fields may store a memory address, offset, pointer, counter or other means of identifying a position within a buffer. Advantageously, the next address field of a table or table entry is initially set to the address of the buffer assigned to store packets of the type associated with the table (and, for re-assembly table **1004**, the particular flow). As the buffer is populated, the address is updated to identify the location in the buffer at which to store the next packet or portion of a packet.

Illustratively, each validity indicator stores a first value (e.g., one) to indicate validity, and a second value (e.g., zero) to indicate invalidity. In the illustrated embodiment of the invention, each index field is thirteen bits, each address field is sixty-four bits and the validity indicators are each one bit in size.

Tables **1004**, **1006**, **1008** and **1010** may take other forms and remain within the scope of the invention as contemplated. For example, these data structures may take the form of arrays, lists, databases, etc., and may be implemented in hardware or software. In the illustrated embodiment of the invention, header table **1006**, MTU table **1008** and jumbo table **1010** each contain only one entry at a time. Thus, only one header buffer, MTU buffer and jumbo buffer are active (e.g., valid) at a time in this embodiment. In an alternative embodiment of the invention, multiple header buffers, MTU buffers and/or jumbo buffers may be used (e.g., valid) at once.

In one embodiment of the invention, certain categories of buffers (e.g., header, non-re-assembly) may store a predetermined number of packets or packet portions. For example, where the memory page size of a host computer processor is eight kilobytes, a header buffer may store a maximum of thirty-two entries, each of which is 256 bytes. Illustratively, even when one packet or header is less than 256 bytes, the next entry in the buffer is stored at the next 256-byte boundary. A counter may be associated with the buffer and decremented (or incremented) each time a new entry is stored in the buffer. After thirty-two entries have been made, the buffer may be released.

In one embodiment of the invention, buffers other than header buffers may be divided into fixed-size regions. For example, in an eight-kilobyte MTU buffer, each MTU packet may be allocated two kilobytes. Any space remaining

in a packet's area after the packet is stored may be left unused or may be padded.

In one alternative embodiment of the invention, entries in a header buffer and/or non-re-assembly buffer (e.g., MTU, jumbo) are aligned for more efficient transfer. In particular, two bytes of padding (e.g., random bytes) are stored at the beginning of each entry in such a buffer. Because a packet's layer two Ethernet header is fourteen bytes long, by adding two pad bytes each packet's layer three protocol header (e.g., IP) will be aligned with a sixteen-byte boundary. Sixteen-byte alignment, as one skilled in the art will appreciate, allows efficient copying of packet contents (such as the layer three header). The addition of two bytes may, however, decrease the size of the maximum packet that may be stored in a header buffer (e.g., to 254 bytes).

As explained above, counters and/or padding may also be used with non-re-assembly buffers. Some non-re-assembleable packets (e.g., jumbo packets) may, however, be split into separate header and data portions, with each portion being stored in a separate buffer—similar to the re-assembly of flow packets. In one embodiment of the invention padding is only used with header portions of split packets. Thus, when a non-re-assembled (e.g., jumbo) packet is split, padding may be applied to the header/small buffer in which the packet's header portion is stored but not to the non-re-assembly buffer in which the packet's data portion is stored. When, however, a non-re-assembly packet is stored with its header and data together in a non-re-assembly buffer, then padding may be applied to that buffer.

In another alternative embodiment of the invention, a second level of padding may be added to each entry in a buffer that stores non-re-assembled packets that are larger than 256 bytes (e.g., MTU packets and jumbo packets that are not split). In this alternative embodiment, a cache line of storage (e.g., sixty-four bytes for a Solaris™ workstation) is skipped in the buffer before storing each packet. The extra padding area may be used by software that processes the packets and/or their completion descriptors. The software may use the extra padding area for routing or as temporary storage for information needed in a secondary or later phase of processing.

For example, before actually processing the packet, the software may store some data that promotes efficient multi-tasking in the padding area. The information is then available when the packet is finally extracted from the buffer. In particular, in one embodiment of the invention a network interface may generate one or more data values to identify multicast or alternate addresses that correspond to a layer two address of a packet received from a network. The multicast or alternate addresses may be stored in a network interface memory by software operating on a host computer (e.g., a device driver). By storing the data value(s) in the padding, enhanced routing functions can be performed when the host computer processes the packet.

Reserving sixty-four bytes at the beginning of a buffer also allows header information to be modified or prepended if necessary. For example, a regular Ethernet header of a packet may, because of routing requirements, need to be replaced with a much larger FDDI (Fiber Distributed Data Interface) header. One skilled in the art will recognize the size disparity between these headers. Advantageously, the reserved padding area may be used for the FDDI header rather than allocating another block of memory.

In a present embodiment of the invention DMA engine **120** may determine which category a packet belongs in, and which type of buffer to store the packet in, by examining the packet's operation code. As described in a previous section, an operation code may be stored in control queue **118** for each packet stored in packet queue **116**. Thus, when DMA engine **120** detects a packet in packet queue **116**, it may fetch the corresponding information in the control queue and act appropriately.

An operation code may indicate whether a packet is compatible with the protocols pre-selected for NIC **100**. In an illustrative embodiment of the invention, only compatible packets are eligible for data re-assembly and/or other enhanced operations offered by NIC **100** (e.g., packet batching or load distribution). An operation code may also reflect the size of a packet (e.g., less than or greater than a predetermined size), whether a packet contains data or is a control packet, and whether a packet initiates, continues or ends a flow. In this embodiment of the invention, eight different operation codes are used. In alternative embodiments of the invention more or less than eight codes may be used. TABLE 1 lists operation codes that may be used in one embodiment of the invention.

FIGS. **12A–12B** illustrate descriptors from a free descriptor ring and a completion descriptor ring in one embodiment of the invention. FIG. **12A** also depicts a free buffer array for storing buffer identifiers retrieved from free descriptors.

Free descriptor ring **1200** is maintained in host memory and is populated with descriptors such as free descriptor **1202**. Illustratively, free descriptor **1202** comprises ring index **1204**, the index of descriptor **1202** in free ring **1200**, and buffer identifier **1206**. A buffer identifier in this embodiment is a memory address, but may, alternatively, comprise a pointer or any other suitable means of identifying a buffer in host memory.

In the illustrated embodiment, free buffer array **1210** is constructed by software operating on a host computer (e.g., a device driver). An entry in free buffer array **1210** in this embodiment includes array index field **1212**, which may be used to identify the entry, and buffer identifier field **1214**. Each entry's buffer identifier field thus stores a buffer identifier retrieved from a free descriptor in free descriptor ring **1200**.

In one embodiment of the invention, free ring manager **1012** of DMA engine **120** retrieves descriptor **1202** from the ring and stores buffer identifier **1206** in free buffer array **1210**. The free ring manager also passes the buffer identifier to flow re-assembly table **1004**, header table **1006**, MTU table **1008** or jumbo table **1010** as needed. In another embodiment the free ring manager extracts descriptors from the free descriptor ring and stores them in a descriptor cache until a buffer is needed, at which time the buffer's buffer identifier is stored in the free buffer array. In yet another embodiment, a descriptor may be used (e.g., the buffer that it references may be used to store a packet) while still in the cache.

In one embodiment of the invention descriptor **1202** is sixteen bytes in length. In this embodiment, ring index **1204** is thirteen bits in size, buffer identifier **1206** (and buffer identifier field **1214** in free buffer array **1210**) is sixty-four bits, and the remaining space may store other information or may not be used. The size of array index field **1212** depends upon the dimensions of array **1210**; in one embodiment the field is thirteen bits in size.

Completion descriptor ring **1220** is also maintained in host memory. Descriptors in completion ring **1220** are written or configured when a packet is transferred to the host computer by DMA engine **120**. The information written to a descriptor, such as descriptor **1222**, is used by software operating on the host computer (e.g., a driver associated with NIC **100**) to process the packet. Illustratively, an ownership

indicator (described below) in the descriptor indicates whether DMA engine **120** has finished using the descriptor. For example, this field may be set to a particular value (e.g., zero) when the DMA engine finishes using the descriptor and a different value (e.g., one) when it is available for use by the DMA engine. However, in another embodiment of the invention, DMA engine **120** issues an interrupt to the host computer when it releases a completion descriptor. Yet another means of alerting the host computer may be employed in an alternative embodiment. Descriptor **1222**, in one embodiment of the invention, is thirty-two bytes in length.

In the illustrated embodiment of the invention, information stored in descriptor **1222** concerns a transferred packet and/or the buffer it was stored in, and includes the following fields. Data size **1230** reports the amount of data in the packet (e.g., in bytes). The data size field may contain a zero if there is no data portion in the packet or no data buffer (e.g., flow re-assembly buffer, non-re-assembly buffer, jumbo buffer, MTU buffer) was used. Data buffer index **1232** is the index, within free buffer array **1210**, of the buffer identifier for the flow re-assembly buffer, non-re-assembly buffer, jumbo buffer or MTU buffer in which the packet's data was stored. When the descriptor corresponds to a small packet fully stored in a header buffer, this field may store a zero or remain unused. Data offset **1234** is the offset of the packet's data within the flow re-assembly buffer, non-re-assembly buffer, jumbo buffer or MTU buffer (e.g., the location of the first byte of data within the data buffer).

In FIG. 12B, flags field **1236** includes one or more flags concerning a buffer or packet. For example, if a header buffer or data is being released (e.g., because it is full), a release header or release data flag, respectively, is set. A release flow flag may be used to indicate whether a flow has, at least temporarily, ended. In other words, if a release flow flag is set (e.g., stores a value of one), this indicates that there are no other packets waiting in the packet queue that are in the same flow as the packet associated with descriptor **1222**. Otherwise, if this flag is not set (e.g., stores a value of zero), software operating on the host computer may queue this packet to await one or more additional flow packets so that they may be processed collectively. A split flag may be included in flags field **1236** to identify whether a packet's contents (e.g., data) spans multiple buffers. Illustratively, if the split flag is set, there will be an entry in next data buffer index **1240**, described below.

Descriptor type **1238**, in the presently described embodiment of the invention, may take any of three values. A first value (e.g., one) indicates that DMA engine **120** is releasing a flow buffer for a flow that is stale (e.g., no packet has been received in the flow for some period of time). A second value (e.g., two) may indicate that a non-re-assembleable packet was stored in a buffer. A third value (e.g., three) may be used to indicate that a flow packet (e.g., a packet that is part of a flow through NIC **100**) was stored in a buffer.

Next buffer index **1240** stores an index, in free buffer array **1210**, of an entry containing a buffer identifier corresponding to a buffer storing a subsequent portion of a packet if the entire packet, or its data, could not fit into the first assigned buffer. The offset in the next buffer may be assumed to be zero. Header size **1242** reports the length of the header (e.g., in bytes). The header size may be set to zero if the header buffer was not used for this packet (e.g., the packet is not being re-assembled and is not a small packet). Header buffer index **1244** is the index, in free buffer array **1210**, of the buffer identifier for the header buffer used to store this packet's header. Header offset **1246** is the offset of the packet's header within the buffer (e.g., header buffer) in which the header was stored. The header offset may take the form of a number of bytes into the buffer at which the header can be found. Alternatively, the offset may be an index value, reporting the index position of the header. For example, in one embodiment of the invention mentioned above, entries in a header buffer are stored in 256-byte units. Thus, each entry begins at a 256-byte boundary regardless of the actual size of the entries. The 256-byte entries may be numbered or indexed within the buffer.

In the illustrated embodiment, flow number **1250** is the packet's flow number (e.g., the index in flow database **110** of the packet's flow key). Flow number **1250** may be used to identify packets in the same flow. Operation code **1252** is a code generated by flow database manager **108**, as described in a previous section, and used by DMA engine **120** to process the packet and transfer it into an appropriate buffer. Methods of transferring a packet depending upon its operation code are described in detail in the following section. No_Assist signal **1254**, also described in a previous section, may be set or raised when the packet is not compatible with the protocols pre-selected for NIC **100**. One result of incompatibility is that header parser **106** may not extensively parse the packet, in which case the packet will not receive the subsequent benefits. Processor identifier **1256**, which may be generated by load distributor **112**, identifies a host computer system processor for processing the packet. As described in a previous section, load distributor **112** attempts to share or distribute the load of processing network packets among multiple processors by having all packets within one flow processed by the same processor. Layer three header offset **1258** reports an offset within the packet of the first byte of the packet's layer three protocol (e.g., IP) header. With this value, software operating on the host computer may easily strip off one or more headers or header portions.

Checksum value **1260** is a checksum computed for this packet by checksum generator **114**. Packet length **1262** is the length (e.g., in bytes) of the entire packet.

Ownership indicator **1264** is used in the presently described embodiment of the invention to indicate whether NIC **100** or software operating on the host computer "owns" completion descriptor **1222**. In particular, a first value (e.g., zero) is placed in the ownership indicator field when NIC **100** (e.g., DMA engine **120**) has completed configuring the descriptor. Illustratively, this first value is understood to indicate that the software may now process the descriptor. When finished processing the descriptor, the software may store a second value (e.g., one) in the ownership indicator to indicate that NIC **100** may now use the descriptor for another packet.

One skilled in the art will recognize that there are numerous methods that may be used to inform host software that a descriptor has been used by, or returned to, DMA engine **120**. In one embodiment of the invention, for example, one or more registers, pointers or other data structures are maintained to indicate which completion descriptors in a completion descriptor ring have or have not been used. In particular, a head register may be used to identify a first of a series of descriptors that are owned by host software, while a tail register identifies the last descriptor in the series. DMA engine **120** may update these registers as it configures and releases descriptors. Thus, by examining these registers the host software and the DMA engine can determine how many descriptors have or have not been used.

Finally, other information, flags and indicators may be stored in other field **1266**. Other information that may be

stored in one embodiment of the invention includes the length and/or offset of a TCP payload, flags indicating a small packet (e.g., less than 257 bytes) or a jumbo packet (e.g., more than 1522 bytes), a flag indicating a bad packet (e.g., CRC error), a checksum starting position, etc.

In alternative embodiments of the invention only information and flags needed by the host computer (e.g., driver software) are included in descriptor **1222**. Thus, in one alternative embodiment one or more fields other than the following may be omitted: data size **1230**, data buffer index **1232**, data offset **1234**, a split flag, next data buffer index **1240**, header size **1242**, header buffer index **1244**, header offset **1246** and ownership indicator **1264**.

In addition, a completion descriptor may be organized in virtually any form; the order of the fields of descriptor **1222** in FIG. **12** is merely one possible configuration. It is advantageous, however, to locate ownership indicator **1264** towards the end of a completion descriptor since this indicator may be used to inform host software when the DMA engine has finished populating the descriptor. If the ownership indicator were placed in the beginning of the descriptor, the software may read it and attempt to use the descriptor before the DMA engine has finished writing to it.

One skilled in the art will recognize that other systems and methods than those described in this section may be implemented to identify storage areas in which to place packets being transferred from a network to a host computer without exceeding the scope of the invention.

Methods of Transferring a Packet into a Memory Buffer by a DMA Engine

FIGS. **13–20** are flow charts describing procedures for transferring a packet into a host memory buffer. In these procedures, a packet's operation code helps determine which buffer or buffers the packet is stored in. An illustrative selection of operation codes that may be used in this procedure are listed and explained in TABLE 1.

The illustrated embodiments of the invention employ four categories of host memory buffers, the sizes of which are programmable. The buffer sizes are programmable in order to accommodate various host platforms, but are programmed to be one memory page in size in present embodiments in order to enhance the efficiency of handling and processing network traffic. For example, the embodiments discussed in this section are directed to the use of a host computer system employing a SPARC™ processor, and so each buffer is eight kilobytes in size. These embodiments are easily adjusted, however, for host computer systems employing memory pages having other dimensions.

One type of buffer is for re-assembling data from a flow, another type is for headers of packets being re-assembled and for small packets (e.g., those less than or equal to 256 bytes in size) that are not re-assembled. A third type of buffer stores packets up to MTU size (e.g., 1522 bytes) that are not re-assembled, and a fourth type stores jumbo packets that are greater than MTU size and which are not re-assembled. These buffers are called flow re-assembly, header, MTU and jumbo buffers, respectively.

The procedures described in this section make use of free descriptors and completion descriptors as depicted in FIG. **12**. In particular, in these procedures free descriptors retrieved from a free descriptor ring store buffer identifiers (e.g., memory addresses, pointers) for identifying buffers in which to store a portion of a packet. A used buffer may be returned to a host computer by identifying the location within a free buffer array or other data structure used to store the buffer's buffer identifier. One skilled in the art will recognize that these procedures may be readily adapted to work with alternative methods of obtaining and returning buffers for storing packets.

FIG. **13** is a top-level view of the logic controlling DMA engine **120** in this embodiment of the invention. State **1300** is a start state.

In state **1302**, a packet is stored in packet queue **116** and associated information is stored in control queue **118**. One embodiment of a packet queue is depicted in FIG. **8** and one embodiment of a control queue is depicted in FIG. **9**. DMA engine **120** may detect the existence of a packet in packet queue **116** by comparing the queue's read and write pointers. As long as they do not reference the same entry, then it is understood that a packet is stored in the queue. Alternatively, DMA engine **120** may examine control queue **118** to determine whether an entry exists there, which would indicate that a packet is stored in packet queue **116**. As long as the control queue's read and write pointers do not reference the same entry, then an entry is stored in the control queue and a packet must be stored in the packet queue.

In state **1304**, the packet's associated entry in the control queue is read. Illustratively, the control queue entry includes the packet's operation code, the status of the packet's No_Assist signal (e.g., indicating whether or not the packet is compatible with a pre-selected protocol), one or more indicators concerning the size of the packet (and/or its data portion), etc.

In state **1306**, DMA engine **120** retrieves the packet's flow number. As described previously, a packet's flow number is the index of the packet's flow in flow database **110**. A packet's flow number may, as described in a following section, be provided to and used by dynamic packet batching module **122** to enable the collective processing of headers from related packets. In one embodiment of the invention, a packet's flow number may be provided to any of a number of NIC modules (e.g., IPP module **104**, packet batching module **122**, DMA engine **120**, control queue **118**) after being generated by flow database manager **108**. The flow number may also be stored in a separate data structure (e.g., a register) until needed by dynamic packet batching module **122** and/or DMA engine **120**. In one embodiment of the invention DMA engine **120** retrieves a packet's flow number from dynamic packet batching module **122**. In an alternative embodiment of the invention, the flow number may be retrieved from a different location or module.

Then, in states **1308–1318**, DMA engine **120** determines the appropriate manner of processing the packet by examining the packet's operation code. The operation code may, for example, indicate which buffer the engine should transfer the packet into and whether a flow is to be set up or torn down in flow re-assembly buffer table **1004**.

The illustrated procedure continues at state **1400** (FIG. **14**) if the operation code is 0, state **1500** (FIG. **15**) for operation code 1, state **1600** (FIG. **16**) for operation code 2, state **1700** (FIG. **17**) for operation code 3, state **1800** (FIG. **18**) for operation code 4, state **1900** (FIG. **19**) for operation code 5 and state **2000** (FIG. **20**) for operation codes 6 and 7.

A Method of Transferring a Packet with Operation Code 0

FIG. **14** depicts an illustrative procedure in which DMA engine **120** transfers a packet associated with operation code 0 to a host memory buffer. As reflected in TABLE 1, operation code 0 indicates in this embodiment that the packet is compatible with the protocols that may be parsed by NIC **100**. As explained above, compatible packets are eligible for re-assembly, such that data from multiple packets of one flow may be stored in one buffer that can then be efficiently provided (e.g., via a page-flip) to a user or

program's memory space. Packets having operation code 0, however, are small and contain no flow data for re-assembly. They are thus likely to be control packets. Therefore, no new flow is set up, no existing flow is torn down and the entire packet may be placed in a header buffer.

In state **1400**, DMA engine **120** (e.g., DMA manager **1002**) determines whether there is a valid (e.g., active) header buffer. Illustratively, this determination is made by examining validity indicator **1116** of header buffer table **1006**, which manages the active header buffer. If the validity indicator is set (e.g., equal to one), then there is a header buffer ready to receive this packet and the procedure continues at state **1404**.

Otherwise, in state **1402** a header buffer is prepared or initialized for storing small packets (e.g., packets less than 257 bytes in size) and headers of re-assembled packets (and, possibly, headers of other packets—such as jumbo packets). In the illustrated embodiment, this initialization process involves obtaining a free ring descriptor and retrieving its buffer identifier (e.g., its reference to an available host memory buffer). The buffer identifier may then be stored in a data structure such as free buffer array **1210** (shown in FIG. 12A). As described above, in one embodiment of the invention free ring manager **1012** maintains a cache of descriptors referencing empty buffers. Thus, a descriptor may be retrieved from this cache and its buffer allocated to header buffer table **1006**. If the cache is empty, new descriptors may be retrieved from a free descriptor ring in host memory to replenish the cache.

When a new buffer identifier is retrieved from the cache or from the free descriptor ring, the buffer identifier's position in the free buffer array is placed in header buffer index **1112** of header buffer table **1006**. Further, an initial storage location in the buffer identifier (e.g., its starting address) is stored in next address field **1114** and validity indicator **1116** is set to a valid state.

In state **1404**, the packet is copied or transferred (e.g., via a DMA operation) into the header buffer at the address or location specified in the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes are inserted before the packet in order to align the beginning of the packet's layer three (e.g., IP) header with a sixteen-byte boundary. In addition, a header buffer may be logically partitioned into cells of predetermined size (e.g., 256 bytes), in which case the packet or padding may begin at a cell boundary.

In state **1406**, a completion descriptor is written or configured to provide information to the host computer (e.g., a software driver) for processing the packet. In particular, the header buffer index (e.g. the index within the free buffer array of the buffer identifier that references the header buffer) and the packet's offset in the header buffer are placed in the descriptor. Illustratively, the offset may identify the location of the cell in which the header is stored, or it may identify the first byte of the packet. The size of the packet is also stored in the descriptor, illustratively within a header size field. A data size field within the descriptor is set to zero to indicate that the entire packet was placed in the header buffer (e.g., there was no data portion to store in a separate data buffer). A release header flag is set in the descriptor if the header buffer is full. However, the header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release header flag may be set (or cleared) at that time.

As described in a later section, in one embodiment of the invention a release flow flag may also be set, depending upon dynamic packet batching module **122**. For example, if the packet batching module determines that another packet in the same flow will soon be transferred to the host computer, the release flow flag will be cleared (e.g., a zero will be stored). This indicates that the host computer should await the next flow packet before processing this one. Then, by collectively processing multiple packets from a single flow, the packets can be processed more efficiently while requiring less processor time.

In the descriptor type field, a value is stored to indicate that a flow packet was transferred to host memory. Also, a predetermined value (e.g., zero) is stored in the ownership indicator field to indicate that DMA engine **120** is done using the descriptor and/or is releasing a packet to the host computer. Illustratively, the host computer will detect the change in the ownership indicator (e.g., from one to zero) and use the stored information to process the packet. In one alternative embodiment of the invention, DMA engine **120** issues an interrupt or other signal to alert the host computer that a descriptor is being released. In another alternative embodiment, the host computer polls the NIC to determine when a packet has been received and/or transferred. In yet another alternative embodiment, the descriptor type field is used to inform the host computer that the DMA engine is releasing a descriptor. In this alternative embodiment, when a non-zero value is placed in the descriptor type field the host computer may understand that the DMA engine is releasing the descriptor.

In a present embodiment of the invention, the ownership indicator field is not changed until DMA engine **120** is finished with any other processing involving this packet or is finished making all entries in the descriptor. For example, as described below a header buffer or other buffer may be found to be full at some time after state **1406**. By delaying the setting of the ownership indicator, a release header flag can be set before the descriptor is reclaimed by the host computer, thus avoiding the use of another descriptor.

In state **1408**, it is determined whether the header buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, up to thirty-two entries may be stored in a header buffer. Thus, a counter may be used to keep track of entries placed in each new header buffer and the buffer can be considered full when thirty-two entries are stored. Other methods of determining whether a buffer is full are also suitable. For example, after a packet is stored in the header buffer a new next address field may be calculated and the difference between the new next address field and the initial address of the buffer may be compared to the size of the buffer (e.g., eight kilobytes). If less than a predetermined number of bytes (e.g., 256) are unused, the buffer may be considered full.

If the buffer is full, in state **1410** the header buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer via a descriptor. In this embodiment of the invention a release header flag in the descriptor is set. If the descriptor that was written in state **1406** was already released (e.g., its ownership indicator field changed), another descriptor may be used in this state. If another descriptor is used simply to report a full header buffer, the descriptor's header size and data size fields may be set to zero to indicate that no new packet was transferred with this descriptor.

If the header buffer is not full, then in state **1412** the next address field of header buffer table **1006** is updated to indicate the address at which to store the next header or

small packet. The processing associated with a packet having operation code 0 then ends with end state **1499**. In one embodiment of the invention, the ownership indicator field of a descriptor that is written in state **1406** is not changed, or an interrupt is not issued, until end state **1499**. Delaying the notification of the host computer allows the descriptor to be updated or modified for as long as possible before turning it over to the host.

A Method of Transferring a Packet with Operation Code 1

FIG. **15** depicts an illustrative procedure in which DMA engine **120** transfers a packet associated with operation code 1 to a host memory buffer. As reflected in TABLE 1, in this embodiment operation code 1 indicates that the packet is compatible with the protocols that may be parsed by NIC **100**. A packet having operation code 1, however, may be a control packet having a particular flag set. No new flow is set up, but a flow should already exist and is to be torn down; there is no data to re-assemble and the entire packet may be stored in a header buffer.

In state **1500**, DMA engine **120** (e.g., DMA manager **1002**) determines whether there is a valid (e.g., active) header buffer. Illustratively, this determination is made by examining validity indicator **1116** of header buffer table **1006**, which manages the active header buffer. If the validity indicator is set, then there is a header buffer ready to receive this packet and the procedure continues at state **1504**.

Otherwise, in state **1502** a new header buffer is prepared or initialized for storing small packets and headers of re-assembled packets. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, its buffer identifier (e.g., pointer, address, index) is stored in free buffer array **1210** and its initial storage location (e.g., address or cell location) is stored in next address field **1114** of header buffer table **1006**. The index or position of the buffer identifier within the free buffer array is stored in header buffer index **1112**. Finally, validity indicator **1116** is set to a valid state.

In state **1504** the packet is copied into the header buffer at the address or location specified in the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes are inserted before the packet in order to align the beginning of the packet's layer three (e.g., IP) header with a sixteen-byte boundary. And, the packet (with or without padding) may be placed into a pre-defined area or cell of the buffer.

In the illustrated embodiment, operation code 1 indicates that the packet's existing flow is to be torn down. Thus, in state **1506** it is determined whether a flow re-assembly buffer is valid (e.g., active) for this flow by examining the flow's validity indicator in flow re-assembly buffer table **1004**. If, for example, the indicator is valid, then there is an active buffer storing data from one or more packets in this flow. Illustratively, the flow is torn down by invalidating the flow re-assembly buffer and releasing it to the host computer. If there is no valid flow re-assembly buffer for this flow, the illustrated procedure continues at state **1512**. Otherwise, the procedure proceeds to state **1508**.

In state **1508**, a completion descriptor is configured to release the flow's re-assembly buffer and to provide information to the host computer for processing the current packet. In particular, the header buffer index and the offset of the first byte of the packet (or location of the packet's cell) within the header buffer are placed in the descriptor. The index within the free buffer array of the entry containing the re-assembly buffer's buffer identifier is stored in a data index field of the descriptor. The size of the packet is stored in a header size field and a data size field is set to zero to indicate that no separate buffer was used for storing this packet's data. A release header flag is set in the descriptor if the header buffer is full and a release data flag is set to indicate that no more data will be placed in this flow's present re-assembly buffer (e.g., it is being released). In addition, a release flow flag is set to indicate that DMA engine **120** is tearing down the packet's flow. The header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release header flag may be set at that time.

In state **1510**, the flow's entry in flow re-assembly buffer table **1004** is invalidated. After state **1510**, the procedure continues at state **1514**.

In state **1512**, a completion descriptor is configured with information somewhat different than that of state **1508**. In particular, the header buffer index, the offset to this packet within the header buffer and the packet size are placed within the same descriptor fields as above. The data size field is set to zero, as above, but no data index needs to be stored and no release data flag is set (e.g., because there is no flow re-assembly buffer to release). A release header flag is still set in the descriptor if the header buffer is full and a release flow flag is again set to indicate that DMA engine **120** is tearing down the packet's flow. Also, the descriptor type field is changed to a value indicating that DMA engine **120** transferred a flow packet into host memory.

In state **1514**, it is determined whether the header buffer is now full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, a counter is used to keep track of entries placed into each new header buffer. The buffer is considered full when thirty-two entries are stored.

If the buffer is full, in state **1516** the header buffer is invalidated. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer via the descriptor configured in state **1508** or state **1512**. In this embodiment of the invention a release header flag in the descriptor is set to indicate that the header buffer is full.

If the header buffer is not full, then in state **1518** the next address field of header buffer table **1006** is updated to indicate the address at which to store the next header or small packet.

The processing associated with a packet having operation code 1 then ends with end state **1599**. In this end state, the descriptor used for this packet is turned over to the host computer by changing its ownership indicator field (e.g., from one to zero), issuing an interrupt, or some other mechanism.

One skilled in the art will appreciate that in an alternative embodiment of the invention a change in the descriptor type field to any value other than the value (e.g., zero) it had when DMA engine **120** was using it, may constitute a surrender of "ownership" of the descriptor to the host computer or software operating on the host computer. The host computer will detect the change in the descriptor type field and subsequently use the stored information to process the packet.

A Method of Transferring Packet with Operation Code 2

FIGS. **16A–16F** illustrate a procedure in which DMA engine **120** transfers a packet associated with operation code 2 to a host memory buffer. As reflected in TABLE 1,

operation code 2 may indicate that the packet is compatible with the protocols that may be parsed by NIC **100**, but that it is out of sequence with another packet in the same flow. It may also indicate an attempt to re-establish a flow, but that no more data is likely to be received after this packet. For operation code 2, no new flow is set up and any existing flow with the packet's flow number is to be torn down. The packet's data is not to be re-assembled with data from other packets in the same flow.

Because an existing flow is to be torn down (e.g., the flow's re-assembly buffer is to be invalidated and released to the host computer), in state **1600** it is determined whether a flow re-assembly buffer is valid (e.g., active) for the flow having the flow number that was read in state **1306**. This determination may be made by examining the validity indicator in the flow's entry in flow re-assembly buffer table **1004**. Illustratively, if the indicator is valid then there is an active buffer storing data from one or more packets in the flow. If there is a valid flow re-assembly buffer for this flow, the illustrated procedure continues at state **1602**. Otherwise, the procedure proceeds to state **1606**.

In state **1602**, a completion descriptor is written or configured to release the existing flow re-assembly buffer. In particular, the flow re-assembly buffer's index (e.g., the location within the free buffer array that contains the buffer identifier corresponding to the flow re-assembly buffer) is written to the descriptor. In this embodiment of the invention, no offset needs to be stored in the descriptor's data offset field and the data size field may be set to zero because no new data was stored in the re-assembly buffer. Similarly, the header buffer is not yet being released, therefore the header index and header offset fields of the descriptor need not be used and a zero may be stored in the header size field.

Illustratively, the descriptor's release header flag is cleared (e.g., a zero is stored in the flag) because the header buffer is not to be released. The release data flag is set (e.g., a one is stored in the flag), however, because no more data will be placed in the released flow re-assembly buffer. Further, a release flow flag in the descriptor is also set, to indicate that the flow associated with the released flow re-assembly buffer is being torn down.

The descriptor type field may be changed to a value indicating that DMA engine **120** is releasing a stale flow buffer (e.g., a flow re-assembly buffer that has not been used for some time). Finally, the descriptor is turned over to the host computer by changing its ownership indicator field or by issuing an interrupt or using some other mechanism. In one embodiment of the invention, however, the descriptor is not released to the host computer until end state **1699**.

Then, in state **1604**, the flow re-assembly buffer is invalidated by modifying validity indicator **1106** in the flow's entry in flow re-assembly buffer table **1004** appropriately.

In state **1606**, it is determined whether the present packet is a small packet (e.g., less than or equal to 256 bytes in size), suitable for storage in a header buffer. If so, the illustrated procedure proceeds to state **1610**. Information stored in packet queue **116** and/or control queue **118** may be used to make this determination.

In state **1608**, it is determined whether the present packet is a jumbo packet (e.g., greater than 1522 bytes in size), such that it should be stored in a jumbo buffer. If so, the illustrated procedure proceeds to state **1650**. If not, the procedure continues at state **1630**.

In state **1610** (reached from state **1606**), it has been determined that the present packet is a small packet suitable for storage in a header buffer. Therefore, DMA engine **120** (e.g., DMA manager **1002**) determines whether there is a valid (e.g., active) header buffer. Illustratively, this determination is made by examining validity indicator **1116** of header buffer table **1006**, which manages the active header buffer. If the validity indicator is set, then there should be a header buffer ready to receive this packet and the procedure continues at state **1614**.

Otherwise, in state **1612** a new header buffer is prepared or initialized for storing small packets and headers of re-assembled packets. This initialization process may involve obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or some other indicator of the first storage location in the buffer is placed in next address field **1114** of header buffer table **1006**. The buffer identifier's position or index within the free buffer array is stored in header buffer index **1112**, and validity indicator **1116** is set to a valid state.

In state **1614** the packet is copied or transferred (e.g., via a DMA operation) into the header buffer at the address or location specified in the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes are inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In addition, the packet may be positioned within a cell of predetermined size (e.g., 256 bytes) within the header buffer.

In state **1616**, a completion descriptor is written or configured to provide necessary information to the host computer (e.g., a software driver) for processing the packet. In particular, the header buffer index (e.g. the position within the free buffer array of the header buffer's buffer identifier) and the packet's offset within the header buffer are placed in the descriptor. Illustratively, this offset may serve to identify the first byte of the packet, the first pad byte before the packet or the beginning of the packet's cell within the buffer. The size of the packet is also stored in the descriptor in a header size field. A data size field within the descriptor may be set to zero to indicate that the entire packet was placed in the header buffer (e.g., no separate data portion was stored). A release header flag is set in the descriptor if the header buffer is full. However, the header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release header flag may be set (or cleared) at that time. A release data flag is cleared (e.g., set to a value of zero), because there is no separate data portion being conveyed to the host computer.

Also, the descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-reassembleable packet into host memory. And, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention the ownership indicator field is not changed until end state **1699** below. In one alternative embodiment of the invention, DMA engine **120** issues an interrupt or other signal to alert the host computer that a descriptor is being released.

In state **1618**, it is determined whether the header buffer is full. In this embodiment of the invention, where each

buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, a counter may be used to keep track of entries placed into each new header buffer. The buffer is considered full when thirty-two entries are stored.

If the buffer is full, in state **1620** the header buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to an invalid state and communicating this status to the host computer. In this embodiment of the invention, a release header flag in the descriptor is set. The illustrated procedure then ends with end state **1699**.

If the header buffer is not full, then in state **1622** the next address field of header buffer table **1006** is updated to indicate the address or cell boundary at which to store the next header or small packet. The illustrated procedure then ends with end state **1699**.

In state **1630** (reached from state **1608**), it has been determined that the packet is not a small packet or a jumbo packet. The packet may, therefore, be stored in a non-re-assembly buffer (e.g., an MTU buffer) used to store packets that are up to MTU in size (e.g., 1522 bytes). Thus, in state **1630** DMA engine **120** determines whether a valid (e.g., active) MTU buffer exists. Illustratively, this determination is made by examining validity indicator **1126** of MTU buffer table **1008**, which manages an active MTU buffer. If the validity indicator is set, then there is an MTU buffer ready to receive this packet and the procedure continues at state **1634**.

Otherwise, in state **1632** a new MTU buffer is prepared or initialized for storing non-re-assembleable packets up to 1522 bytes in size. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer (e.g., a buffer identifier). If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in the free buffer array. The buffer's initial address or some other indication of the first storage location in the buffer is placed in next address field **1124** of MTU buffer table **1008**. Further, the position of the buffer identifier within the free buffer array is stored in MTU buffer index **1122** and validity indicator **1126** is set to a valid state.

In state **1634** the packet is copied or transferred (e.g., via a DMA operation) into the MTU buffer at the address or location specified in the next address field. As described above, in one embodiment of the invention pad bytes may be inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In another embodiment of the invention packets may be aligned in an MTU buffer in cells of predefined size (e.g., two kilobytes), similar to entries in a header buffer.

In state **1636**, a completion descriptor is written or configured to provide necessary information to the host computer (e.g., a software driver) for processing the packet. In particular, the MTU buffer index (e.g. the free buffer array element that contains the buffer identifier for the MTU buffer) and offset (e.g., the offset of the first byte of this packet within the MTU buffer) are placed in the descriptor in data index and data offset fields, respectively. The size of the packet is also stored in the descriptor, illustratively within a data size field. A header size field within the descriptor is set to zero to indicate that the entire packet was placed in the MTU buffer (e.g., no separate header portion was stored in a header buffer). A release data flag is set in the descriptor if the MTU buffer is full. However, the MTU buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release data flag may be set (or cleared) at that time. A release header flag is cleared (e.g., set to zero), because there is no separate header portion being conveyed to the host computer.

Further, the descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. Also, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In a present embodiment of the invention the ownership field is not set until end state **1699** below. In one alternative embodiment of the invention, DMA engine **120** issues an interrupt or other signal to alert the host computer that a descriptor is being released, or communicates this event to the host computer through the descriptor type field.

In state **1638**, it is determined whether the MTU buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the MTU buffer are allotted two kilobytes, a counter may be used to keep track of entries placed into each new header buffer. The buffer may be considered full when a predetermined number of entries (e.g., four) are stored. In an alternative embodiment of the invention DMA engine **120** determines how much storage space within the buffer has yet to be used. If no space remains, or if less than a predetermined amount of space is still available, the buffer may be considered full.

If the MTU buffer is full, in state **1640** it is invalidated to ensure that it is not used again. Illustratively, this involves setting the MTU buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention, a release data flag in the descriptor is set. The illustrated procedure then ends with end state **1699**.

If the MTU buffer is not full, then in state **1642** the next address field of MTU buffer table **1008** is updated to indicate the address or location (e.g., cell boundary) at which to store the next packet. The illustrated procedure then ends with end state **1699**.

In state **1650** (reached from state **1608**), it has been determined that the packet is a jumbo packet (e.g., that it is greater than 1522 bytes in size). In this embodiment of the invention jumbo packets are stored in jumbo buffers and, if splitting of jumbo packets is enabled (e.g., as determined in state **1654** below), headers of jumbo packets are stored in a header buffer. DMA engine **120** determines whether a valid (e.g., active) jumbo buffer exists. Illustratively, this determination is made by examining validity indicator **1136** of jumbo buffer table **1010**, which manages the active jumbo buffer. If the validity indicator is set, then there is a jumbo buffer ready to receive this packet and the procedure continues at state **1654**. As explained above, a jumbo buffer table may not be used in an embodiment of the invention in which a jumbo buffer is used only once (e.g., to store just one, or just part of one, jumbo packet).

Otherwise, in state **1652** a new jumbo buffer is prepared or initialized for storing a non-re-assembleable packet that is larger than 1522 bytes. This initialization process may involve obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer (e.g., a buffer identifier). If the cache is

empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, its buffer identifier (e.g., pointer, address, index) is stored in a free buffer array (or other data structure). The buffer's initial address or other indication of the first storage location in the buffer is placed in next address field **1134** of jumbo buffer table **1010**. Also, the location of the buffer identifier within the free buffer array is stored in jumbo buffer index **1132** and validity indicator **1136** is set to a valid state.

Then, in state **1654** DMA engine **120** determines whether splitting of jumbo buffers is enabled. If enabled, the header of a jumbo packet is stored in a header buffer while the packet's data is stored in one or more jumbo buffers. If not enabled, the entire packet will be stored in one or more jumbo buffers. Illustratively, splitting of jumbo packets is enabled or disabled according to the configuration of a programmable indicator (e.g., flag, bit, register) that may be set by software operating on the host computer (e.g., a device driver). If splitting is enabled, the illustrated procedure continues at state **1670**. Otherwise, the procedure continues with state **1656**.

In state **1656**, DMA engine **120** determines whether the packet will fit into one jumbo buffer. For example, in an embodiment of the invention using eight kilobyte pages, if the packet is larger than eight kilobytes a second jumbo buffer will be needed to store the additional contents. If the packet is too large, the illustrated procedure continues at state **1662**.

In state **1658**, the packet is copied or transferred (e.g., via a DMA operation) into the current jumbo buffer, at the location specified in the next address field **1134** of jumbo buffer table **1010**. When the packet is transferred intact like this, padding may be added to align a header portion of the packet with a sixteen-byte boundary. One skilled in the art will appreciate that the next address field may not need to be updated to account for this new packet because the jumbo buffer will be released. In other words, in one embodiment of the invention a jumbo buffer may be used just once (e.g., to store one packet or a portion of one packet).

In state **1660**, a completion descriptor is written or configured to release the jumbo buffer and to provide information to the host computer for processing the packet. The jumbo buffer index (e.g., the position within the free buffer array of the buffer identifier for the jumbo buffer) and the offset of the packet within the jumbo buffer are placed in the descriptor. Illustratively, these values are stored in data index and data offset fields, respectively. The size of the packet (e.g., the packet length) may be stored in a data size field.

A header size field is cleared (e.g., a zero is stored) to indicate that the header buffer was not used (e.g., the header was not stored separately from the packet's data). Because there is no separate packet header, header index and header offset fields are not used or are set to zero (e.g., the values stored in their fields do not matter). A release header flag is cleared and a release data flag is set to indicate that no more data will be placed in this jumbo buffer (e.g., because it is being released).

Also, the descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-reassembleable packet into host memory. And, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In an alternative embodiment, the descriptor may be released by issuing an interrupt or other alert. In yet another embodiment, changing the descriptor type field (e.g., to a non-zero value) may signal the release of the descriptor. In one embodiment of the invention the ownership indicator is not set until end state **1699** below. After state **1660**, the illustrated procedure resumes at state **1668**.

In state **1662**, a first portion of the packet is stored in the present (e.g., valid) jumbo buffer, at the location identified in the buffer's next address field **1134**. Then, because the full packet will not fit into this buffer, in state **1664** a new jumbo buffer is prepared and the remainder of the packet is stored in that buffer.

In state **1666**, a completion descriptor is written or configured. The contents are similar to those described in state **1660** but this descriptor must reflect that two jumbo buffers were used to store the packet.

Thus, the jumbo buffer index (e.g., the index, within the free buffer array, of the buffer identifier that identifies the header buffer) and the offset of the packet within the first jumbo buffer are placed in the descriptor, as above. The size of the packet (e.g., the packet length) is stored in a data size field.

A header size field is cleared (e.g., a zero is stored) to indicate that the header buffer was not used (e.g., the header was not stored separately from the packet's data). Because there is no separate packet header, header index and header offset fields are not used (e.g., the values stored in their fields do not matter).

A release header flag is cleared and a release data flag is set to indicate that no more data will be placed in these jumbo buffers (e.g., because they are being released). Further, a split packet flag is set to reflect the use of a second jumbo buffer, and the index (within the free buffer array) of the buffer identifier for the second buffer is stored in a next index field.

Further, the descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-reassembleable packet into host memory. Finally, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field, or some other mechanism is employed, to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention, the descriptor is not released to the host computer until end state **1699** below.

In state **1668**, the jumbo buffer entry or entries in jumbo buffer table **1010** are invalidated (e.g., validity indicator **1136** is set to invalid) to ensure that they are not used again. In the procedure described above a jumbo packet was stored in, at most, two jumbo buffers. In an alternative embodiment of the invention a jumbo buffer may be stored across any number of buffers. The descriptor(s) configured to report the transfer of such a packet is/are constructed accordingly, as will be obvious to one skilled in the art.

After state **1668**, the illustrated procedure ends with end state **1699**.

In state **1670** (reached from state **1654**), it has been determined that the present jumbo packet will be split to store the packet header in a header buffer and the packet data in one or more jumbo buffers. Therefore, DMA engine **120** (e.g., DMA manager **1002**) determines whether there is a valid (e.g., active) header buffer. Illustratively, this determination is made by examining validity indicator **1116** of header buffer table **1006**, which manages the active header buffer. If the validity indicator is set, then there is a header buffer ready to receive this packet and the procedure continues at state **1674**.

Otherwise, in state **1672** a new header buffer is prepared or initialized for storing small packets and headers of other packets. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or some other indication of the first storage location or cell in the buffer is placed in next address field **1114** of header buffer table **1006**. Also, the index of the buffer identifier within the free buffer array is stored in header buffer index **1112** and validity indicator **1116** is set to a valid state.

In state **1674** the packet's header is copied or transferred (e.g., via a DMA operation) into the header buffer at the address or location specified in the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes are inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In addition, the packet's header may be positioned within a cell of predetermined size (e.g., 256 bytes) within the buffer.

In state **1676**, DMA engine **120** determines whether the packet's data (e.g., the TCP payload) will fit into one jumbo buffer. If the packet is too large, the illustrated procedure continues at state **1682**.

In state **1678**, the packet's data is copied or transferred (e.g., via a DMA operation) into the current jumbo buffer, at the location specified in the next address field **1134** of jumbo buffer table **1010**. One skilled in the art will appreciate that the next address field may not need to be updated to account for this new packet because the jumbo buffer will be released. In other words, in one embodiment of the invention a jumbo buffer may be used just once (e.g., to store one packet or a portion of one packet).

In state **1680**, a completion descriptor is written or configured to release the jumbo buffer and to provide information to the host computer for processing the packet. The header buffer index (e.g. the index of the header buffer's buffer identifier within the free buffer array) and offset of the packet's header within the buffer are placed in the descriptor in header index and header offset fields, respectively. Illustratively, this offset may serve to identify the first byte of the header, the first pad byte before the header or the location of the cell in which the header is stored. The jumbo buffer index (e.g., the position or index within the free buffer array of the buffer identifier that identifies the jumbo buffer) and the offset of the first byte of the packet's data within the jumbo buffer are placed in data index and data offset fields, respectively. Header size and data size fields are used to store the size of the packet's header (e.g., the offset of the payload within the packet) and data (e.g., payload size), respectively.

A release header flag is set in the descriptor if the header buffer is full. However, the header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release header flag may be set (or cleared) at that time. A release data flag is also set, because no more data will be placed in the jumbo buffer (e.g., it is being released to the host computer).

The descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. Also, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention the ownership indicator is not changed until end state **1699** below. In an alternative embodiment, the descriptor may be released by issuing an interrupt or other alert. In yet another alternative embodiment, changing the descriptor type value may signal the release of the descriptor.

After state **1680**, the illustrated procedure proceeds to state **1688**.

In state **1682**, a first portion of the packet's data is stored in the present (e.g., valid) jumbo buffer, at the location identified in the buffer's next address field **1134**.

Because all of the packet's data will not fit into this buffer, in state **1684** a new jumbo buffer is prepared and the remainder of the packet is stored in that buffer.

In state **1686**, a completion descriptor is written or configured. The contents are similar to those described in states **1680** but this descriptor must reflect that two jumbo buffers were used to store the packet. The header buffer index (e.g. the index of the free buffer array element containing the header buffer's buffer identifier) and offset (e.g., the location of this packet's header within the header buffer) are placed in the descriptor in header index and header offset fields, respectively. The jumbo buffer index (e.g., the index, within the free buffer array, of the buffer identifier that references the jumbo buffer) and the offset of the first byte of the packet's data within the jumbo buffer are placed in data index and data offset fields, respectively. Header size and data size fields are used to store the size of the packet's header (e.g., as measured by the offset of the packet's payload from the start of the packet) and data (e.g., payload size), respectively.

A release header flag is set in the descriptor if the header buffer is full. However, the header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release header flag may be set (or cleared) at that time. A release data flag is also set, because no more data will be placed in the jumbo buffer (e.g., it is being released to the host computer). Further, a split packet flag is set to indicate that a second jumbo buffer was used, and the location (within the free buffer array or other data structure) of the second buffer's buffer identifier is stored in a next index field

The descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. Finally, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention the ownership indicator is not changed until end state **1699** below.

In state **1688**, the jumbo buffer's entry in jumbo buffer table **1010** is invalidated (e.g., validity indicator **1136** is set to invalid) to ensure that it is not used again. In the procedure described above, a jumbo packet was stored in, at most, two jumbo buffers. In an alternative embodiment of the invention a jumbo packet may be stored across any number of buffers. The descriptor that is configured to report the transfer of such a packet is constructed accordingly, as will be obvious to one skilled in the art.

In state **1690**, it is determined whether the header buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, a counter may be used

to keep track of entries placed into each new header buffer. The buffer may be considered full when thirty-two entries are stored.

If the buffer is full, in state **1692** the header buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention, a release header flag in the descriptor is set. The illustrated procedure then ends with end state **1699**.

If the header buffer is not full, then in state **1694** the next address field of header buffer table **1006** is updated to indicate the address at which to store the next header or small packet. The illustrated procedure then ends with end state **1699**.

In end state **1699**, a descriptor may be turned over to the host computer by changing a value in the descriptor's descriptor type field (e.g., from one to zero), as described above. Illustratively, the host computer (or software operating on the host computer) detects the change and understands that DMA engine **120** is returning ownership of the descriptor to the host computer.

A Method of Transferring a Packet with Operation Code 3

FIGS. 17A–17C illustrate one procedure in which DMA engine **120** transfers a packet associated with operation code 3 to a host memory buffer. As reflected in TABLE 1, operation code 3 may indicate that the packet is compatible with a protocol that can be parsed by NIC **100** and that it carries a final portion of data for its flow. No new flow is set up, but a flow should already exist and is to be torn down. The packet's data is to be re-assembled with data from previous flow packets. Because the packet is to be re-assembled, the packet's header should be stored in a header buffer and its data in the flow's re-assembly buffer. The flow's active re-assembly buffer may be identified by the flow's entry in flow re-assembly buffer table **1004**.

In state **1700**, DMA engine **120** (e.g., DMA manager **1002**) determines whether there is a valid (e.g., active) header buffer. Illustratively, this determination is made by examining validity indicator **1116** of header buffer table **1006**, which manages the active header buffer. If the validity indicator is set (e.g., equal to one), then it is assumed that there is a header buffer ready to receive this packet and the procedure continues at state **1704**.

Otherwise, in state **1702** a new header buffer is prepared or initialized for storing small packets and headers of re-assembled packets. This initialization process may involve obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its buffer identifier (e.g., a reference to an available memory buffer). If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

Illustratively, when a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or some other indication of the first storage location or cell in the buffer is placed in next address field **1114** of header buffer table **1006**. Further, the index of the buffer identifier within the free buffer array is stored in header buffer index **1112** and validity indicator **1116** is set to a valid state.

In state **1704** the packet's header is copied or transferred into the header buffer at the address or location specified in the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes may be inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In addition, the header may be positioned within a cell of predetermined size (e.g., 256 bytes) within the header buffer.

In the illustrated embodiment, operation code 3 indicates that an existing flow is to be torn down (e.g., the flow re-assembly buffer is to be invalidated and released to the host computer). Thus, in state **1706** it is determined whether a flow re-assembly buffer is valid (e.g., active) for this flow by examining the validity indicator in the flow's entry in flow re-assembly buffer table **1004**. Illustratively, if the indicator is valid then there should be an active buffer storing data from one or more packets in this flow. If there is a valid flow re-assembly buffer for this flow, the illustrated procedure continues at state **1712**. Otherwise, the procedure proceeds to state **1708**.

In state **1708**, a new flow re-assembly buffer is prepared to store this packet's data. Illustratively, a free ring descriptor is obtained from a cache maintained by free ring manager **1012** and its reference to an empty buffer is retrieved. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or other indication of its first storage location is placed in next address field **1104** of the flow's entry in flow re-assembly buffer table **1004**. The flow's entry in the re-assembly buffer table may be recognized by its flow number. The location within the free buffer array of the buffer identifier is stored in re-assembly buffer index **1102**, and validity indicator **1106** is set to a valid state.

In state **1710**, the packet's data is copied or transferred (e.g., via a DMA operation) into the address or location specified in the next address field of the flow's entry in flow re-assembly buffer table **1004**.

In state **1712**, a completion descriptor is written or configured to release the flow's re-assembly buffer and to provide information to the host computer for processing the packet. In particular, the header buffer index (e.g., the index, within the free buffer array, of the header buffer's identifier) and the offset of the packet's header within the header buffer are placed in the descriptor. Illustratively, this offset serves to identify the first byte of the header, the first pad byte preceding the header or the cell in which the header is stored. The flow re-assembly buffer index (e.g., the index, within the free buffer array, of the flow re-assembly buffer's identifier) and the offset of the packet's data within that buffer are also stored in the descriptor.

The size of the packet's data (e.g., the size of the packet's TCP payload) and header (e.g., the offset of the TCP payload within the packet) portions are stored in data size and header size fields, respectively. The descriptor type field is given a value that indicates that a flow packet has been transferred to host memory. A release header flag may be set if the header buffer is full and a release data flag may be set to indicate that no more data will be placed in this flow re-assembly buffer (e.g., because it is being released). In addition, a release flow flag is set to indicate that DMA engine **120** is tearing down the packet's flow. The header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment, the release header flag may be set (or cleared) at that time.

Then, in state **1714**, the flow re-assembly buffer is invalidated by modifying validity indicator **1106** in the flow's entry in flow re-assembly buffer table **1004** appropriately. After state **1714**, the procedure continues at state **1730**.

In state **1716**, DMA engine **120** determines whether the packet's TCP payload (e.g., the packet's data portion) will fit into the valid flow re-assembly buffer. If not, the illustrated procedure continues at state **1722**.

In state **1718**, the packet data is copied or transferred (e.g., via a DMA operation) into the flow's re-assembly buffer, at the location specified in the next address field **1104** of the flow's entry in flow re-assembly table **1004**. One skilled in the art will appreciate that the next address field may or may not be updated to account for this new packet because the re-assembly buffer is being released.

In state **1720**, a completion descriptor is written or configured to release the flow's re-assembly buffer and to provide information to the host computer for processing the packet. The header buffer index (e.g., the location or index, within the free buffer array, of the header buffer's identifier) and the offset of the packet's header within the header buffer are placed in the descriptor. The flow re-assembly buffer index (e.g., the location or index within the free buffer array of the flow re-assembly buffer's identifier) and the offset of the packet's data within that buffer are also stored in the descriptor.

The size of the packet's data (e.g., the size of the packet's TCP payload) and header (e.g., the offset of the TCP payload within the packet) are stored in data size and header size fields, respectively. The descriptor type field is given a value that indicates that a flow packet has been transferred to host memory. A release header flag is set if the header buffer is full and a release data flag is set to indicate that no more data will be placed in this flow re-assembly buffer (e.g., because it is being released). As explained above, the header buffer may not be tested to see if it is full until a later state of this procedure, at which time the release header flag may be set. Finally, a release flow flag is set to indicate that DMA engine **120** is tearing down the packet's flow. After state **1720**, the illustrated procedure resumes at state **1728**.

In state **1722**, a first portion of the packet's payload (e.g., data) is stored in the flow's present (e.g., valid) re-assembly buffer, at the location identified in the buffer's next address field **1104**.

Because the full payload will not fit into this buffer, in state **1724** a new flow re-assembly buffer is prepared and the remainder of the payload is stored in that buffer. In one embodiment of the invention information concerning the first buffer is stored in a completion descriptor. This information may include the position within the free buffer array of the first buffer's buffer identifier and the offset of the first portion of data within the buffer. The flow's entry in flow re-assembly buffer table **1004** may then be updated for the second buffer (e.g., store a first address in next address field **1104** and the location of buffer's identifier in the free buffer array in re-assembly buffer index **1102**).

In state **1726**, a completion descriptor is written or configured. The contents are similar to those described for states **1712** and **1720** but this descriptor must reflect that two re-assembly buffers were used.

Thus, the header buffer index (e.g., the position within the free buffer array of the buffer identifier corresponding to the header buffer) and the offset of the packet's header within the header buffer are placed in the descriptor, as above. The first flow re-assembly buffer index (e.g., the position, within the free buffer array, of the buffer identifier corresponding to the first flow re-assembly buffer used to store this packet's payload) and the offset of the packet's first portion of data within that buffer are also stored in the descriptor.

The size of the packet's data (e.g., the size of the packet's TCP payload) and header (e.g., the offset of the TCP payload within the packet) are stored in data size and header size fields, respectively. The descriptor type field is given a value that indicates that a flow packet has been transferred to host memory. A release header flag is set if the header buffer is full and a release data flag is set to indicate that no more data will be placed in this flow re-assembly buffer. A release flow flag is set to indicate that DMA engine **120** is tearing down the packet's flow.

Because two re-assembly buffers were used, a split packet flag is set and the index, within the free buffer array, of the re-assembly buffer's buffer identifier is stored in a next index field. Additionally, because the packet contains the final portion of data for the flow, a release next data buffer flag may also be set to indicate that the second flow re-assembly buffer is being released.

In state **1728**, the flow's entry in flow re-assembly buffer table **1004** is invalidated to ensure that it is not used again.

In state **1730**, it is determined whether the header buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, a counter is used to keep track of entries placed into each new header buffer. The buffer is considered full when thirty-two entries are stored.

If the buffer is full, in state **1732** the header buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention a release header flag in the descriptor is set.

If the header buffer is not full, then in state **1734** the next address field of header buffer table **1006** is updated to indicate the address at which to store the next header or small packet.

The processing associated with a packet having operation code 3 then ends with end state **1799**. In this end state, the descriptor used for this packet is turned over to the host computer by changing its ownership indicator field (e.g., from one to zero). Alternatively, some other mechanism may be used, such as issuing an interrupt or changing the descriptor' descriptor type field. Illustratively, the descriptor type field would be changed to a value indicating that DMA engine **120** transferred a flow packet into host memory.

In one alternative embodiment of the invention an optimization may be performed when processing a packet with operation code 3. This optimization takes advantage of the knowledge that the packet contains the last portion of data for its flow. In particular, instead of loading a descriptor into flow re-assembly buffer table **1004** the descriptor may be used where it is—in a descriptor cache maintained by free ring manager **1012**.

For example, instead of retrieving a buffer identifier from a descriptor and storing it in an array in state **1708** above, only to store one packet's data in the identified buffer before releasing it, it may be more efficient to use the descriptor without removing it from the cache. In this embodiment, when a completion descriptor is written the values stored in its data index and data offset fields are retrieved from a descriptor in the descriptor cache. Similarly, when the first portion of a code 3 packet's data fits into the flow's active buffer but a new one is needed just for the remaining data, a descriptor in the descriptor cache may again be used without first loading it into a free buffer array and the flow re-assembly buffer table. In this situation, the completion descriptor's next index field is retrieved from the descriptor in the descriptor cache.

A Method of Transferring a Packet with Operation Code 4

FIGS. **18A–18D** depict an illustrative procedure in which DMA engine **120** transfers a packet associated with opera-

tion code 4 to a host memory buffer. As reflected in TABLE 1, operation code 4 in this embodiment indicates that the packet is compatible with the protocols that may be parsed by NIC **100** and continues a flow that is already established. No new flow is set up, the existing flow is not to be torn down, and the packet's data is to be re-assembled with data from other flow packets. Because the packet is to be re-assembled, the packet's header should be stored in a header buffer and its data in the flow's re-assembly buffer.

In state **1800**, DMA engine **120** determines whether there is a valid (e.g., active) header buffer. Illustratively, this determination is made by examining validity indicator **1116** of header buffer table **1006**, which manages the active header buffer. If the validity indicator is set, then there should be a header buffer ready to receive this packet and the procedure continues at state **1804**.

Otherwise, in state **1802** a new header buffer is prepared or initialized for storing small packets and headers of re-assembled packets. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or some other indication of the first storage location in the buffer is place in next address field **1114** of header buffer table **1006**. Also, the position or index of the buffer identifier within the free buffer array is stored in header buffer index **1112** and validity indicator **1116** is set to a valid state.

In state **1804** the packet's header is copied or transferred into the header buffer at the address or location specified in the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes are inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In addition, the packet's header may be positioned within a cell of predetermined size (e.g., 256 bytes) within the buffer.

In the illustrated embodiment, operation code 4 indicates that an existing flow is to be continued. Thus, in state **1806** it is determined whether a flow re-assembly buffer is valid (e.g., active) for this flow by examining the validity indicator in the flow's entry in flow re-assembly buffer table **1004**. Illustratively, if the indicator is valid then there is an active buffer storing data from one or more packets in this flow. If there is a valid flow re-assembly buffer for this flow, the illustrated procedure continues at state **1808**. Otherwise, the procedure proceeds to state **1810**.

In state **1808**, it is determined whether the packet's data (e.g., its TCP payload) portion is too large for the current flow re-assembly buffer. If the data portion is too large, two flow re-assembly buffers will be used and the illustrated procedure proceeds to state **1830**. Otherwise, the procedure continues at state **1820**.

In state **1810**, because it was found (in state **1806**) that there was no valid flow re-assembly buffer for this packet, a new flow re-assembly buffer is prepared. Illustratively, a free ring descriptor is obtained from a cache maintained by free ring manager **1012** and its reference to an empty buffer is retrieved. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or other indicator of its first storage location is placed in next address field **1104** of the flow's entry in flow re-assembly buffer table **1004**. The flow's entry in the table may be recognized by its flow number. The location of the buffer identifier in the free buffer array is stored in re-assembly buffer index **1102**, and validity indicator **1106** is set to a valid state.

In state **1812**, the packet's data is copied or transferred (e.g., via a DMA operation) into the address or location specified in the next address field of the flow's entry in flow re-assembly buffer table **1004**.

In state **1814**, a completion descriptor is written or configured to provide information to the host computer for processing the packet. In particular, the header buffer index (e.g., the index within the free buffer array of the buffer identifier that identifies the header buffer) and the offset of the packet's header within the header buffer are placed in the descriptor. Illustratively, this offset may serve to identify the first byte of the header, the first pad byte preceding the header or the header's cell within the header buffer. The flow re-assembly buffer index (e.g., the index within the free buffer array of the buffer identifier that identifies the flow re-assembly buffer) and the offset of the packet's data within that buffer are also stored in the descriptor.

The size of the packet's data (e.g., the size of the packet's TCP payload) and header (e.g., the offset of the TCP payload within the packet) are stored in data size and header size fields, respectively. The descriptor type field is given a value indicating that a flow packet has been transferred to host memory. A release header flag is set if the header buffer is fill but a release data flag is not set, because more data will be placed in this flow re-assembly buffer. The header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment, the release header flag may be set (or cleared) at that time.

In one embodiment of the invention a release flow flag may also be set, depending upon dynamic packet batching module **122**. For example, if the packet batching module determines that another packet in the same flow will soon be transferred to the host computer, the release flow flag will be cleared (e.g., a zero will be stored). This indicates that the host computer should await the next flow packet before processing this one. By collectively processing multiple packets from a single flow, the packets can be processed more efficiently and less processor time is required. If, however, no other packets in the same flow are identified, the release flow flag may be set (e.g., a one is stored) to indicate that the host computer should process the flow packets it has received so far, without waiting for more.

In state **1816**, the flow's entry in flow re-assembly buffer table **1004** is updated. In particular, next address field **1104** is updated to identify the location in the re-assembly buffer at which the next flow packet's data should be stored. After state **1816**, the illustrated procedure continues at state **1838**.

In state **1820** (reached from state **1808**), it is known that the packet's data, or TCP payload, will fit within the flow's current re-assembly buffer. Thus, the packet data is copied or transferred into the buffer at the location identified in next address field **1104** of the flow's entry in flow re-assembly buffer table **1004**.

In state **1822**, a completion descriptor is written or configured to provide information to the host computer for processing the packet. In particular, the header buffer index (e.g., the index within the free buffer array of the buffer identifier that identifies the header buffer) and the offset of the packet's header within the header buffer are placed in the

descriptor. The flow re-assembly buffer index (e.g., the index within the free buffer array of the buffer identifier that identifies the flow re-assembly buffer) and the offset of the packet's data within that buffer are also stored in the descriptor.

The size of the packet's data (e.g., the size of the packet's TCP payload) and header (e.g., the offset of the TCP payload within the packet) are stored in data size and header size fields, respectively. The descriptor type field is given a value indicating that a flow packet has been transferred to host memory. A release header flag is set if the header buffer is full but a release data flag is set only if the flow re-assembly buffer is now full. The header and flow re-assembly buffers may not be tested to see if they are full until a later state of this procedure. In such an embodiment, the flags may be set (or cleared) at that time.

In one embodiment of the invention a release flow flag may also be set, depending upon dynamic packet batching module **122**. For example, if the packet batching module determines that another packet in the same flow will soon be transferred to the host computer, the release flow flag will be cleared. This indicates that the host computer should await the next flow packet before processing this one. By collectively processing multiple packets from a single flow, the packets can be processed more efficiently and less processor time is required. If, however, no other packets in the same flow are identified, the release flow flag may be set to indicate that the host computer should process the flow packets received so far, without waiting for more.

In state **1824**, the flow re-assembly buffer is examined to determine if it is full. In the presently described embodiment of the invention this test is conducted by first determining how much data (e.g., how many bytes) has been stored in the buffer. Illustratively, the flow's next address field and the amount of data stored from this packet are summed. Then, the initial buffer address (e.g., before any data was stored in it) is subtracted from this sum. This value, representing how much data is now stored in the buffer, is then compared to the size of the buffer (e.g., eight kilobytes).

If the amount of data currently stored in the buffer equals the size of the buffer, then it is full. In the presently described embodiment of the invention it is desirable to completely fill flow re-assembly buffers. Thus, a flow re-assembly buffer is not considered full until its storage space is completely populated with flow data. This scheme enables the efficient processing of network packets.

If the flow re-assembly buffer is full, in state **1826** the buffer is invalidated to ensure it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention, a release data flag in the descriptor is set. After state **1826**, the procedure continues at state **1838**.

If the flow re-assembly buffer is not full, then in state **1828** next address field **1104** in the flow's entry in flow re-assembly buffer table **1004** is updated to indicate the address at which to store the next portion of flow data. After state **1828**, the procedure continues at state **1838**.

In state **1830** (reached from state **1808**), it is known that the packet's data will not fit into the flow's current re-assembly buffer. Therefore, some of the data is stored in the current buffer and the remainder in a new buffer. In particular, in state **1830** a first portion of data (e.g., an amount sufficient to fill the buffer) is copied or transferred into the current flow re-assembly buffer.

In state **1832**, a new descriptor is loaded from a descriptor cache maintained by free ring manager **1012**. Its identifier of a new buffer is retrieved and the remaining data from the packet is stored in the new buffer. In one embodiment of the invention, after the first portion of data is stored information from the flow's entry in flow re-assembly table **1004** is stored in a completion descriptor. Illustratively, this information includes re-assembly buffer index **1102** and the offset of the first portion of data within the full buffer. Then the new descriptor can be loaded—its index is stored in re-assembly buffer index **1102** and an initial address is stored in next address **1104**.

In state **1834**, a completion descriptor is written or configured to provide information to the host computer for processing the packet. In particular, the header buffer index (e.g., the location of the header buffer's buffer identifier within the free buffer array) and the offset of the packet's header within the header buffer are placed in the descriptor. The flow re-assembly buffer index (e.g., the location of the flow re-assembly buffer's buffer identifier within the free buffer array) and the offset of the packet's data within that buffer are also stored in the descriptor.

The size of the packet's data (e.g., the size of the packet's TCP payload) and header (e.g., the offset of the TCP payload within the packet) are stored in data size and header size fields, respectively. The descriptor type field is given a value indicating that a flow packet has been transferred to host memory. A release header flag is set if the header buffer is full and a release data flag is set because the first flow re-assembly buffer is being released. The header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment, the release header flag may be set (or cleared) at that time.

Because two re-assembly buffers were used, a split packet flag in the descriptor is set and the index, within the free descriptor ring, of the descriptor that references the second re-assembly buffer is stored in a next index field.

In one embodiment of the invention a release flow flag may also be set, depending upon dynamic packet batching module **122**. For example, if the packet batching module determines that another packet in the same flow will soon be transferred to the host computer, the release flow flag will be cleared. This indicates that the host computer should await the next flow packet before processing this one. By collectively processing multiple packets from a single flow, the packets can be processed more efficiently and less processor time is required. If, however, no other packets in the same flow are identified, the release flow flag may be set to indicate that the host computer should process the flow packets received so far, without waiting for more.

In state **1836**, next address field **1104** in the flow's entry in flow re-assembly buffer table **1004** is updated to indicate the address in the new buffer at which to store the next portion of flow data.

In state **1838**, it is determined whether the header buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, a counter may be used to keep track of entries placed into each new header buffer. The buffer is considered full when thirty-two entries are stored.

If the buffer is full, in state **1840** the header buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention, a release header flag in the descriptor is set.

If the header buffer is not full, then in state **1842** the next address field of header buffer table **1006** is updated to indicate the address at which to store the next header or small packet.

The processing associated with a packet having operation code 4 then ends with end state **1899**. In this end state, the descriptor used for this packet is turned over to the host computer by changing its ownership indicator field (e.g., from one to zero). In one alternative embodiment of the invention, DMA engine **120** issues an interrupt or uses other means to alert the host computer that a descriptor is being released.

In one alternative embodiment of the invention the optimization described above for packets associated with operation code 3 may be performed when processing a packet with operation code 4. This optimization is useful, for example, when a code 4 packet's data is too large to fit in the current flow re-assembly buffer. Instead of loading a new descriptor for the second portion of data, the descriptor may be used where it is—in a descriptor cache maintained by free ring manager **1012**. This allows DMA engine **120** to finish transferring the packet and turn over the completion descriptor before adjusting flow re-assembly buffer table **1004** to reflect a new buffer.

In particular, instead of loading information from a new descriptor in state **1832** above, it may be more efficient to use the descriptor without removing it from the cache. In this embodiment a new buffer for storing a remainder of the packet's data is accessed by retrieving its buffer identifier from a descriptor in the free ring manager's descriptor cache. The data is stored in the buffer and, after the packet's completion descriptor is configured and released, the necessary information is loaded into the flow re-assembly table as described above. Illustratively, re-assembly buffer index **1102** stores the buffer identifier's index within the free buffer array, and an initial memory address of the buffer, taking into account the newly stored data, is placed in next address **1104**.

A Method of Transferring a Packet with Operation Code 5

FIGS. **19A–19E** depict a procedure in which DMA engine **120** transfers a packet associated with operation code 5 to a host memory buffer. As reflected in TABLE 1, operation code 5 in one embodiment of the invention may indicate that a packet is incompatible with the protocols that may be parsed by NIC **100**. It may also indicate that a packet contains all of the data for a new flow (e.g., no more data will be received for the packet's flow). Therefore, for operation code 5, no new flow is set up and there should not be any flow to tear down. The packet's data, if there is any, is not to be re-assembled.

In state **1900**, it is determined whether the present packet is a small packet (e.g., less than or equal to 256 bytes in size) suitable for storage in a header buffer. If so, the illustrated procedure proceeds to state **1920**.

Otherwise, in state **1902** it is determined whether the present packet is a jumbo packet (e.g., greater than 1522 bytes in size), such that it should be stored in a jumbo buffer. If so, the illustrated procedure proceeds to state **1940**. If not, the procedure continues at state **1904**.

In state **1904**, it has been determined that the packet is not a small packet or a jumbo packet. The packet may, therefore, be stored in a non-re-assembly buffer used to store packets that are no greater in size than MTU (Maximum Transfer Unit) in size, which is 1522 bytes in a present embodiment. This buffer may be called an MTU buffer. Therefore, DMA engine **120** determines whether a valid (e.g., active) MTU buffer exists. Illustratively, this determination is made by examining validity indicator **1126** of MTU buffer table **1008**, which manages the active MTU buffer. If the validity indicator is set, then there should be a MTU buffer ready to receive this packet and the procedure continues at state **1908**.

Otherwise, in state **1906** a new MTU buffer is prepared or initialized for storing non-re-assembleable packets up to 1522 bytes in size. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its buffer identifier (e.g., a reference to an empty host memory buffer). If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or some other indication of the first storage location in the buffer is placed in next address field **1124** of MTU buffer table **1008**. The buffer identifier's index or position within the free buffer array is stored in MTU buffer index **1122**, and validity indicator **1126** is set to a valid state.

In state **1908** the packet is copied or transferred (e.g., via a DMA operation) into the MTU buffer at the address or location specified in the next address field of MTU buffer table **1008**. As described above, in one embodiment of the invention pad bytes may be inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In addition, the packet may be positioned within a cell of predetermined size (e.g., two kilobytes) within the MTU buffer.

In state **1910**, a completion descriptor is written or configured to provide necessary information to the host computer for processing the packet. In particular, the MTU buffer index (e.g. the location within the free buffer array of the buffer identifier for the MTU buffer) and offset (e.g., the offset to the packet or the packet's cell within the buffer) are placed in the descriptor in data index and data offset fields, respectively. The size of the packet is stored in a data size field. A header size field within the descriptor may be set to zero to indicate that the entire packet was placed in the MTU buffer (e.g., no separate header portion was stored in a header buffer). A release data flag is set in the descriptor if the MTU buffer is full. The MTU buffer may not, however, be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release data flag may be set (or cleared) at that time. A release header flag may be cleared (e.g., not set), because there is no separate header portion being conveyed to the host computer.

Further, the descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. Also, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention, the ownership indicator is not set until end state **1999** below. In an alternative embodiment of the invention, the descriptor may be released by issuing an interrupt or other alert. In yet another alternative embodiment, changing the descriptor's descriptor type field may signal the descriptor's release.

In state **1912**, DMA engine **120** determines whether the MTU buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size, each entry in the MTU buffer may be allotted two kilobytes of space and a counter may be used to keep track of entries placed into an MTU buffer. The buffer may be considered full when a predetermined number of entries (e.g., four) are stored. In an alternative embodiment of the invention entries in an MTU

buffer may or may not be allocated a certain amount of space, in which case DMA engine **120** may calculate how much storage space within the buffer has yet to be used. If no space remains, or if less than a predetermined amount of space is still available, the buffer may be considered full.

If the MTU buffer is full, in state **1914** the buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the MTU buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention a release data flag in the descriptor is set. The illustrated procedure then ends with end state **1999**.

If the MTU buffer is not full, then in state **1916** the next address field of MTU buffer table **1008** is updated to indicate the address at which to store the next packet. The illustrated procedure then ends with end state **1999**.

In state **1920** (reached from state **1900**), it has been determined that the present packet is a small packet suitable for storage in a header buffer. Therefore, DMA engine **120** (e.g., DMA manager **1002**) determines whether there is a valid (e.g., active) header buffer. Illustratively, this determination is made by examining validity indicator **1116** of header buffer table **1006**, which manages the active header buffer, If the validity indicator is set, then there is a header buffer ready to receive this packet and the procedure continues at state **1924**.

Otherwise, in state **1922** a new header buffer is prepared or initialized for storing small packets and headers of re-assembled packets. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or some other indicator of the first storage location or cell in the buffer is placed in next address field **1114** of header buffer table **1006**. Further, the buffer identifier's position within the free buffer array is stored in header buffer index **1112** and validity indicator **1116** is set to a valid state.

In state **1924** the packet is copied or transferred (e.g., via a DMA operation) into the header buffer at the address or location specified in the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes may be inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In addition, the packet may be positioned within a cell of predetermined size (e.g., 256 bytes) within the buffer.

In state **1926**, a completion descriptor is written or configured to provide necessary information to the host computer (e.g., a software driver) for processing the packet. In particular, the header buffer index (e.g. the index of the free buffer array element that contains the header buffer's identifier) and offset are placed in the descriptor, in header index and header offset fields, respectively. Illustratively, this offset serves to identify the first byte of the packet, the first pad byte preceding the packet or the location of the packet's cell within the buffer. The size of the packet is also stored in the descriptor, illustratively within a header size field. A data size field within the descriptor may be set to zero to indicate that the entire packet was placed in the header buffer (e.g., no separate data portion was stored in another buffer). A release header flag may be set in the descriptor if the header buffer is full. However, the header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release header flag may be set (or cleared) at that time. A release data flag may be cleared (e.g., not set), because there is no separate data portion being conveyed to the host computer.

The descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. Also, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention the ownership indicator is not set until end state **1999** below.

In state **1928** it is determined whether the header buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, a counter is used to keep track of entries placed into each new header buffer. The buffer is considered full when thirty-two entries are stored.

If the buffer is full, in state **1930** the header buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention a release header flag in the descriptor is set. The illustrated procedure then ends with end state **1999**.

If the header buffer is not fill, then in state **1932** the next address field of header buffer table **1006** is updated to indicate the address at which to store the next header or small packet. The illustrated procedure then ends with end state **1999**.

In state **1940** (reached from state **1902**), it has been determined that the packet is a jumbo packet (e.g., that it is greater than 1522 bytes in size). In this embodiment of the invention a jumbo packet's data portion is stored in a jumbo buffer. Its header is also stored in the jumbo buffer unless splitting of jumbo packets is enabled, in which case its header is stored in a header buffer. DMA engine **120** thus determines whether a valid (e.g., active) jumbo buffer exists. Illustratively, this determination is made by examining validity indicator **1136** of jumbo buffer table **1010**, which manages an active jumbo buffer. If the validity indicator is set, then there is a jumbo buffer ready to receive this packet and the procedure continues at state **1944**.

Otherwise, in state **1942** a new jumbo buffer is prepared or initialized for storing a non-re-assembleable packet that is larger than 1522 bytes. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or other indication of the first storage location within the buffer is placed in next address field **1134** of jumbo buffer table **1010**. The position of the buffer identifier within the free buffer array is stored in jumbo buffer index **1132**, and validity indicator **1136** is set to a valid state.

Then, in state **1944**, DMA engine **120** determines whether splitting of jumbo buffers is enabled. If enabled, the header of a jumbo packet is stored in a header buffer while the packet's data is stored in one or more jumbo buffers. If not

enabled, the entire packet will be stored in one or more jumbo buffers. Illustratively, splitting of jumbo packets is enabled or disabled according to the configuration of a programmable indicator (e.g., flag, bit, register) that is set by software operating on the host computer (e.g., a device driver). If splitting is enabled, the illustrated procedure continues at state **1960**. Otherwise, the procedure proceeds to state **1946**.

In state **1946**, DMA engine **120** determines whether the packet will fit into one jumbo buffer. For example, in an embodiment of the invention using eight kilobyte pages, if the packet is larger than eight kilobytes a second jumbo buffer will be needed to store the additional contents. If the packet is too large, the illustrated procedure continues at state **1952**.

Otherwise, in state **1948** the packet is copied or transferred (e.g., via a DMA operation) into the current jumbo buffer, at the location specified in the next address field **1134** of jumbo buffer table **1010**. When the packet is transferred intact like this, padding may be added to align a header portion of the packet with a sixteen-byte boundary. One skilled in the art will appreciate that the next address field may not need to be updated to account for this new packet because the jumbo buffer will be released. In other words, in one embodiment of the invention a jumbo buffer is only used once (e.g., to store one packet or a portion of one packet). In an alternative embodiment of the invention a jumbo buffer may store portions of two or more packets, in which case next address field **1134** may need to be updated.

In state **1950**, a completion descriptor is written or configured to release the jumbo buffer and to provide information to the host computer for processing the packet. The jumbo buffer index (e.g., the index, within the free buffer array, of the buffer identifier that corresponds to the jumbo buffer) and the offset of the first byte of the packet within the jumbo buffer are placed in the descriptor, in data index and data size fields, respectively. The size of the packet (e.g., the packet length) is stored in a data size field.

A header size field may be cleared (e.g., a zero is stored) to indicate that the header buffer was not used (e.g., the header was not stored separately from the packet's data). Because the packet was stored intact, header index and header offset fields may or may not be used (e.g., the values stored in their fields do not matter). A release header flag is cleared and a release data flag is set to indicate that no more data will be placed in this jumbo buffer (e.g., because it is being released).

The descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. Finally, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention, the ownership indicator is not changed until end state **1999** below. After state **1950**, the illustrated procedure resumes at state **1958**. In one alternative embodiment of the invention, DMA engine **120** issues an interrupt or uses some other means, possibly not until end state **1999**, to alert the host computer that a descriptor is being released.

In state **1952**, a first portion of the packet is stored in the present (e.g., valid) jumbo buffer, at the location identified in the buffer's next address field **1134**. Because the whole packet will not fit into this buffer, in state **1954** a new jumbo buffer is prepared and the remainder of the packet is stored in that buffer.

In state **1956**, a completion descriptor is written or configured. The contents are similar to those described in state **1950** but this descriptor must reflect that two jumbo buffers were used to store the packet. Thus, the jumbo buffer index (e.g., the index, within the free buffer array, of the array element containing the header buffer's buffer identifier) and the offset of the first byte of the packet within the first jumbo buffer are placed in the descriptor, as above. The size of the packet (e.g., the packet length) is stored in a data size field.

A header size field may be cleared (e.g., a zero is stored) to indicate that the header buffer was not used (e.g., the header was not stored separately from the packet's data). Because there is no separate packet header, header index and header offset fields may or may not be used (e.g., the values stored in their fields do not matter).

A release header flag is cleared and a release data flag is set to indicate that no more data will be placed in these jumbo buffers (e.g., because they are being released). Further, a split packet flag is set to indicate that a second jumbo buffer was used, and the index (within the free buffer array) of the buffer identifier for the second buffer is stored in a next index field.

The descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. And, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention the ownership indicator is not changed until end state **1999** below.

In state **1958**, the jumbo buffer's entry in jumbo buffer table **1010** is invalidated (e.g., validity indicator **1136** is set to invalid) to ensure that it is not used again. In the procedure described above, a jumbo packet was stored in, at most, two jumbo buffers. In an alternative embodiment of the invention, a jumbo buffer may be stored across any number of buffers. The descriptor that is configured to report the transfer of such a packet is constructed accordingly, as will be obvious to one skilled in the art.

After state **1958**, the illustrated procedure ends at end state **1999**.

In state **1960** (reached from state **1944**), it has been determined that the present jumbo packet will be split to store the packet header in a header buffer and the packet data in one or more jumbo buffers. Therefore, DMA engine **120** (e.g., DMA manager **1002**) first determines whether there is a valid (e.g., active) header buffer. Illustratively, this determination is made by examining validity indicator **1116** of header buffer table **1006**, which manages the active header buffer. If the validity indicator is set, then there is a header buffer ready to receive this packet and the procedure continues at state **1964**.

Otherwise, in state **1962** a new header buffer is prepared or initialized for storing small packets and headers of other packets. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or some other indication of the first storage location or cell in the buffer is placed in next address field **1114** of header buffer table **1006**. The index or position of the buffer identifier within the free buffer array is stored in header buffer index **1112**, and validity indicator **1116** is set to a valid state.

In state **1964** the packet's header is copied or transferred (e.g., via a DMA operation) into the header buffer at the address or location specified in the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes may be inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In addition, the header may be positioned within a cell of predetermined size (e.g., 256 bytes) in the buffer.

In state **1966**, DMA engine **120** determines whether the packet's data (e.g., the TCP payload) will fit into one jumbo buffer. If the packet is too large to fit into one (e.g., the current jumbo buffer), the illustrated procedure continues at state **1972**.

In state **1968**, the packet's data is copied or transferred (e.g., via a DMA operation) into the current jumbo buffer, at the location specified in the next address field **1134** of jumbo buffer table **1010**. One skilled in the art will appreciate that the next address field may not need to be updated to account for this new packet because the jumbo buffer will be released. In other words, in one embodiment of the invention a jumbo buffer is only used once (e.g., to store one packet or a portion of one packet).

In state **1970**, a completion descriptor is written or configured to release the jumbo buffer and to provide information to the host computer for processing the packet. The header buffer index (e.g. the free buffer array position of the buffer identifier corresponding to the header buffer) and offset of the packet's header are placed in the descriptor in header index and header offset fields, respectively. Illustratively, this offset serves to identify the first byte of the header, the first pad byte preceding the header or the cell in which the header is stored. The jumbo buffer index (e.g., the index within the free buffer array of the buffer identifier that references the jumbo buffer) and the offset of the first byte of the packet's data within the jumbo buffer are placed in data index and data offset fields, respectively. Header size and data size fields are used to store the size of the packet's header (e.g., the offset of the payload within the packet) and data (e.g., payload size), respectively.

A release header flag may be set in the descriptor if the header buffer is full. However, the header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release header flag may be set (or cleared) at that time. A release data flag is also set, because no more data will be placed in the jumbo buffer (e.g., it is being released to the host computer).

The descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. Also, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention the ownership indicator is not set until end state **1999** below.

After state **1970**, the illustrated procedure proceeds to state **1978**.

In state **1972**, a first portion of the packet's data is stored in the present (e.g., valid) jumbo buffer, at the location identified in the buffer's next address field **1134**. Because all of the packet's data will not fit into this buffer, in state **1974** a new jumbo buffer is prepared and the remainder of the packet is stored in that buffer.

In state **1976**, a completion descriptor is written or configured. The contents are similar to those described in states **1970** but this descriptor must reflect that two jumbo buffers were used to store the packet. The header buffer index (e.g. the free buffer array element that contains the header buffer's identifier) and offset of the header are placed in the descriptor in header index and header offset fields, respectively. The jumbo buffer index (e.g., the free buffer array element containing the jumbo buffer's buffer identifier) and the offset of the first byte of the packet's data within the jumbo buffer are placed in data index and data offset fields, respectively. Header size and data size fields are used to store the size of the packet's header (e.g., the offset of the payload within the packet) and data (e.g., payload size), respectively.

A release header flag is set in the descriptor if the header buffer is full. However, the header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release header flag may be set (or cleared) at that time. A release data flag is also set, because no more data will be placed in the jumbo buffer (e.g., it is being released to the host computer). Further, a split packet flag is set to indicate that a second jumbo buffer was used, and the position or index within the free buffer array of the second buffer's buffer identifier is stored in a next index field.

The descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. Finally, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention the ownership indicator is not set until end state **1999** below. In an alternative embodiment of the invention DMA engine **120** issues an interrupt or uses some other signal to alert the host computer that a descriptor is being released.

In state **1978**, the jumbo buffer's entry in jumbo buffer table **1010** is invalidated (e.g., validity indicator **1136** is set to invalid) to ensure that it is not used again. In the procedure described above, a jumbo packet was stored in, at most, two jumbo buffers. In an alternative embodiment of the invention a jumbo buffer may be stored across any number of buffers. The descriptor that is configured to report the transfer of such a packet is constructed accordingly, as will be obvious to one skilled in the art.

In state **1980**, it is determined whether the header buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, a counter may be used to keep track of entries placed into each new header buffer. The buffer is considered full when thirty-two entries are stored.

If the buffer is full, in state **1982** the header buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention a release header flag in the descriptor is set. The illustrated procedure then ends with end state **1999**.

If the header buffer is not full, then in state **1984** the next address field of header buffer table **1006** is updated to indicate the address at which to store the next header or small packet. The illustrated procedure then ends with end state **1999**.

In end state **1999**, a descriptor may be turned over to the host computer by storing a particular value (e.g., zero) in the descriptor's ownership indicator field as described above. Illustratively, the host computer (or software operating on the host computer) detects the change and understands that DMA engine **120** is returning ownership of the descriptor to the host computer.

A Method of Transferring a Packet with Operation Code 6 or Operation Code 7

FIGS. **20A–20B** depict an illustrative procedure in which DMA engine **120** transfers a packet associated with operation code 6 or 7 to a host memory buffer. As reflected in TABLE 1, operation codes 6 and 7 may indicate that a packet is compatible with the protocols pre-selected for NIC **100** and is the first packet of a new flow. The difference between these operation codes in this embodiment of the invention is that operation code 7 is used when an existing flow is to be replaced (e.g., in flow database **110** and/or flow re-assembly buffer table **1004**) by the new flow. With operation code 6, in contrast, no flow needs to be torn down. For both codes, however, a new flow is set up and the associated packet's data may be re-assembled with data from other packets in the newly established flow. Because the packet data is to be re-assembled, the packet's header should be stored in a header buffer and its data in a new flow re-assembly buffer.

As described in a previous section, the flow that is torn down to make room for a new flow (in the case of operation code 7) may be the least recently used flow. Because flow database **110** and flow re-assembly buffer table **1004** contain only a limited number of entries in the presently described embodiment of the invention, when they are full and a new flow arrives an old one must be torn down. Choosing the least recently active flow for replacement is likely to have the least impact on network traffic through NIC **100**. In one embodiment of the invention DMA engine **120** tears down the flow in flow re-assembly buffer table **1004** that has the same flow number as the flow that has been replaced in flow database **110**.

In state **2000**, DMA engine **120** determines whether there is a valid (e.g., active) header buffer. Illustratively, this determination is made by examining validity indicator **1116** of header buffer table **1006**, which manages the active header buffer. If the validity indicator is set, then there is a header buffer ready to receive this packet and the procedure continues at state **2004**.

Otherwise, in state **2002** a new header buffer is prepared or initialized for storing small packets and headers of re-assembled packets. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or some other indication of the first storage location or cell in the buffer is placed in next address field **1114** of header buffer table **1006**. The position or index of the buffer identifier within the free buffer array is stored in header buffer index **1112**, and validity indicator **1116** is set to a valid state.

In state **2004** the packet's header is copied or transferred into the header buffer at the address or location specified in the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes may be inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In addition, the packet's header may be positioned in a cell of predetermined size (e.g., 256 bytes) within the buffer.

As discussed above, operation code 7 indicates that an old flow is to be torn down in flow re-assembly buffer table **1004** to make room for a new flow. This requires the release of any flow re-assembly buffer that may be associated with the flow being torn down.

Thus, in state **2006** it is determined whether a flow re-assembly buffer is valid (e.g., active) for a flow having the flow number that was read from control queue **118** for this packet. As explained in a previous section, for operation code 7 the flow number represents the entry in flow database **110** (and flow re-assembly buffer table **1004**) that is being replaced with the new flow. DMA engine **120** thus examines the validity indicator in the flow's entry in flow re-assembly buffer table **1004**. Illustratively, if the indicator is valid then there is an active buffer storing data from one or more packets in the flow that is being replaced. If there is a valid flow re-assembly buffer for this flow, the illustrated procedure continues at state **2008**. Otherwise, the procedure proceeds to state **2010**. It will be understood that the illustrated procedure will normally proceed to state **2008** for operation code 7 and state **2010** for operation code 6.

In state **2008**, a completion descriptor is written or configured to release the replaced flow's re-assembly buffer. In particular, the flow re-assembly buffer index (e.g., the index within the free buffer array of the flow re-assembly buffer's buffer identifier) is written to the descriptor. In this embodiment of the invention, no offset needs to be stored in the descriptor's data offset field and the data size field is set to zero because no new data was stored in the buffer that is being released. Similarly, the header buffer is not yet being released, and therefore the header index and header offset fields of the descriptor need not be used and a zero may be stored in the header size field.

The descriptor's release header flag is cleared (e.g., a zero is stored in the flag) because the header buffer is not being released. The release data flag is set (e.g., a one is stored in the flag), however, because no more data will be placed in the released flow re-assembly buffer. Further, a release flow flag in the descriptor is set to indicate that the flow associated with the released flow re-assembly buffer is being torn down.

The descriptor type field is changed to a value indicating that DMA engine **120** is releasing a stale flow buffer (e.g., a flow re-assembly buffer that has not been used for some time). Finally, the descriptor used to release the replaced flow's re-assembly buffer and terminate the associated flow is turned over to the host computer by changing its ownership indicator field (e.g., from one to zero). In one alternative embodiment of the invention, DMA engine **120** issues an interrupt or employs some other means of alerting the host computer that a descriptor is being released.

In state **2010**, a new flow re-assembly buffer is prepared for the flow that is being set up. Illustratively, a free ring descriptor is obtained from a cache maintained by free ring manager **1012** and its buffer identifier (e.g., a reference to an empty memory buffer) is retrieved. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or other indication of the first storage location in the buffer is placed in next address field **1104** of the flow's entry in flow re-assembly buffer table **1004**. The flow's entry in the table may be recognized by its flow number. The position or index of the buffer identifier within the free buffer array is stored in re-assembly buffer index **1102**, and validity indicator **1106** is set to a valid state.

In state **2012**, the packet's data is copied or transferred (e.g., via a DMA operation) into the address or location specified in the next address field of the flow's entry in flow re-assembly buffer table **1004**.

In state **2014**, a completion descriptor is written or configured to provide information to the host computer for processing the packet. In particular, the header buffer index (e.g., the location or position within the free buffer array of the buffer identifier that references the header buffer) and the offset of the packet's header within the header buffer are placed in the descriptor. Illustratively, the offset identifies the first byte of the header, the first pad byte preceding the header or the location of the header's cell in the header buffer.

The flow re-assembly buffer index (e.g., the location or position, within the free buffer array, of the buffer identifier that references the flow re-assembly buffer) and the offset of the packet's data within that buffer are also stored in the descriptor. It will be recognized, however, that the offset reported for this packet's data may be zero, because the packet data is stored at the very beginning of the new flow re-assembly buffer.

The size of the packet's data (e.g., the size of the packet's TCP payload) and header (e.g., the offset of the TCP payload within the packet) are stored in data size and header size fields, respectively. The descriptor type field is changed to a value indicating that DMA engine **120** transferred a flow packet into host memory. A release header flag is set if the header buffer is full but a release data flag is not set, because more data will be placed in this flow re-assembly buffer. The header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment, the release header flag may be set (or cleared) at that time.

In one embodiment of the invention a release flow flag may also be set, depending upon dynamic packet batching module **122**. For example, if the packet batching module determines that another packet in the same flow will soon be transferred to the host computer, the release flow flag will be cleared (e.g., a zero will be stored). This indicates that the host computer should await the next flow packet before processing this one. By collectively processing multiple packets from a single flow, the packets can be processed more efficiently and less processor time will be required for network traffic. If, however, no other packets in the same flow are identified, the release flow flag may be set to indicate that the host computer should process the flow packets received so far, without waiting for more.

In state **2016**, the flow's entry in flow re-assembly buffer table **1004** is updated. In particular, next address field **1104** is updated to identify the location in the re-assembly buffer at which the next flow packet's data should be stored.

In state **2018**, it is determined whether the header buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, a counter may be used to keep track of entries placed into each new header buffer. The buffer is considered full when thirty-two entries are stored.

If the buffer is full, in state **2020** the header buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention, a release header flag in the descriptor is set.

If the header buffer is not full, then in state **2022** the next address field of header buffer table **1006** is updated to indicate the address at which to store the next header or small packet.

The processing associated with a packet having operation codes 6 and 7 then ends with end state **2099**. In this end state, the descriptor used for this packet (e.g., the descriptor that was configured in state **2014**) is turned over to the host computer by changing its ownership indicator field (e.g., from one to zero). In one alternative embodiment of the invention, DMA engine **120** issues an interrupt or employs other means (e.g., such as the descriptor's descriptor type field) to alert the host computer that a descriptor is being released.

One Embodiment of a Packet Batching Module

FIG. **21** is a diagram of dynamic packet batching module **122** in one embodiment of the invention. In this embodiment, packet batching module **122** alerts a host computer to the transfer, or impending transfer, of multiple packets from one communication flow. The related packets may then be processed through an appropriate protocol stack collectively, rather than processing one at a time. As one skilled in the art will recognize, this increases the efficiency with which network traffic may be handled by the host computer.

In the illustrated embodiment, a packet is transferred from NIC **100** to the host computer by DMA engine **120** (e.g., by copying its payload into an appropriate buffer). When a packet is transferred, packet batching module **122** determines whether a related packet (e.g., a packet in the same flow) will soon be transferred as well. In particular, packet batching module **122** examines packets that are to be transferred after the present packet. One skilled in the art will appreciate that the higher the rate of packet arrival at NIC **100**, the more packets that are likely to await transfer to a host computer at a given time. The more packets that await transfer, the more packets that may be examined by the dynamic packet batching module and the greater the benefit it may provide. In particular, as the number of packets awaiting transfer increases, packet batching module **122** may identify a greater number of related packets for collective processing. As the number of packets processed together increases, the amount of host processor time required to process each packet decreases.

Thus, if a related packet is found the packet batching module alerts the host computer so that the packets may be processed as a group. As described in a previous section, in one embodiment of the invention dynamic packet batching module **122** alerts the host computer to the availability of a related packet by clearing a release flow flag in a completion descriptor associated with a transferred packet. The flag may, for example, be cleared by DMA engine **120** in response to a signal or alert from dynamic packet batching module **122**.

In contrast, in an alternative embodiment of the invention dynamic packet batching module **122** or DMA engine **120** may alert the host computer when no related packets are found or when, for some other reason, the host processor should not delay processing a transferred packet. In particular, a release flow flag may be set when the host computer is not expected to receive a packet related to a transferred packet in the near future (e.g., thus indicating that the associated flow is being released or torn down). For example, it may be determined that the transferred packet is the last packet in its flow or that a particular packet doesn't even belong to a flow (e.g., this may be reflected in the packet's associated operation code).

With reference now to FIG. **21**, packet batching module **122** in one embodiment of the invention includes memory **2102** and controller **2104**. Illustratively, each entry in memory **2102**, such as entry **2106**, comprises two fields:

flow number **2108** and validity indicator **2110**. In alternative embodiments of the invention, other information may be stored in memory **2102**. Read pointer **2112** and write pointer **2114** serve as indices into memory **2102**.

In the illustrated embodiment, memory **2102** is an associative memory (e.g., a CAM) configured to store up to 256 entries. Each entry corresponds to and represents a packet stored in packet queue **116**. As described in a previous section, packet queue **116** may also contain up to **256** packets in one embodiment of the invention. When a packet is, or is about to be transferred, by DMA engine **120** from packet queue **116** to the host computer, memory **2102** may be searched for an entry having a flow number that matches the flow number of the transferred packet. Because memory **2102** is a CAM in this embodiment, all entries in the memory may be searched simultaneously or nearly simultaneously. In this embodiment, memory **2102** is implemented in hardware, with the entries logically arranged as a ring. In alternative embodiments, memory **2102** may be virtually any type of data structure (e.g., array, table, list, queue) implemented in hardware or software. In one particular alternative embodiment, memory **2102** is implemented as a RAM, in which case the entries may be examined in a serial manner.

The maximum of 256 entries in the illustrated embodiment matches the maximum number of packets that may be stored in a packet queue. Because the depth of memory **2102** matches the depth of the packet queue, when a packet is stored in the packet queue its flow number may be automatically stored in memory **2102**. Although the same number of entries are provided for in this embodiment, in an alternative embodiment of the invention memory **2102** may be configured to hold a smaller or greater number of entries than the packet queue. And, as discussed in a previous section, for each packet stored in the packet queue, related information may also be stored in the control queue.

In the illustrated embodiment of the invention, flow number **2108** is the index into flow database **110** of the flow comprising the corresponding packet. As described above, in one embodiment of the invention a flow includes packets carrying data from one datagram sent from a source entity to a destination entity. Illustratively, each related packet has the same flow key and the same flow number. Flow number **2108** may comprise the index of the packet's flow key in flow database **110**.

Validity indicator **2110** indicates whether the information stored in the entry is valid or current. In this embodiment, validity indicator **2110** may store a first value (e.g., one) when the entry contains valid data, and a second value (e.g., zero) when the data is invalid. For example, validity indicator **2110** in entry **2106** may be set to a valid state when the corresponding entry in packet queue **116** contains a packet awaiting transfer to the host computer and belongs to a flow (e.g., which may be indicated by the packet's operation code). Similarly, validity indicator **2110** may be set to an invalid state when the entry is no longer needed (e.g., when the corresponding packet is transferred to the host computer).

Flow validity indicator **2110** may also be set to an invalid state when a corresponding packet's operation code indicates that the packet does not belong to a flow. It may also be set to an invalid state when the corresponding packet is a control packet (e.g., contains no data) or is otherwise non-re-assembleable (e.g., because it is out of sequence, incompatible with a pre-selected protocol, has an unexpected control flag set). Validity indicator **2110** may be managed by controller **2104** during operation of the packet batching module.

In the illustrated embodiment of the invention, an entry's flow number is received from a register in which it was placed for temporary storage. A packet's flow number may be temporarily stored in a register, or other data structure, in order to facilitate its timely delivery to packet batching module **122**. Temporary storage of the flow number also allows the flow database manager to turn its attention to a later packet. A flow number may, for example, be provided to dynamic packet batching module **122** at nearly the same time that the associated packet is stored in packet queue **116**. Illustratively, the flow number may be stored in the register by flow database manager **108** or by IPP module **104**. In an alternative embodiment, the flow number is received from control queue **118** or some other module of NIC **100**.

In the illustrated embodiment of the invention, memory **2102** contains an entry corresponding to each packet in packet queue **116**. When a packet in the packet queue is transferred to a host computer (e.g., when it is written to a re-assembly buffer), controller **2104** invalidates the memory entry that corresponds to that packet. Memory **2102** is then searched for another entry having the same flow number as the transferred packet. Afterwards, when a new packet is stored in packet queue **116**, perhaps in place of the transferred packet, a new entry is stored in memory **2102**.

In an alternative embodiment of the invention, memory **2102** may be configured to hold entries for only a subset of the maximum number of packets stored in packet queue **116** (e.g., just re-assembleable packets). Entries in memory **2102** may still be populated when a packet is stored in the packet queue. However, if memory **2102** is full when a new packet is received, then creation of an entry for the new packet must wait until a packet is transferred and its entry in memory **2102** invalidated. Therefore, in this alternative embodiment entries in memory **2102** may be created by extracting information from entries in control queue **118** rather than packet queue **116**. Controller **2104** would therefore continually attempt to copy information from entries in control queue **118** into memory **2102**. The function of populating memory **2102** may be performed independently or semi-independently of the function of actually comparing the flow numbers of memory entries to the flow number of a packet being transferred to the host computer.

In this alternative embodiment a second read pointer may be used to index control queue **118** to assist in the population of memory **2102**. In particular, the second read pointer may be used by packet batching module **122** to find and fetch entries for memory **2102**. Illustratively, if the second, or "lookahead" read pointer references the same entry as the control queue's write pointer, then it could be determined that no new entries were added to control queue **118** since the last check by controller **2104**. Otherwise, as long as there is an empty (e.g., invalid) entry in memory **2102**, the necessary information (e.g., flow number) may be copied into memory **2102** for the packet corresponding to the entry referenced by the lookahead read pointer. The lookahead read pointer would then be incremented.

Returning now to FIG. **21**, read pointer **2112** of dynamic packet batching module **122** identifies the current entry in memory **2102** (e.g., the entry corresponding to the packet at the front of the packet queue or the next packet to be transferred). Illustratively, this pointer is incremented each time a packet is transferred to the host computer. Write pointer **2114** identifies the position at which the next entry in memory **2102** is to be stored. Illustratively, the write pointer is incremented each time an entry is added to memory **2102**. One manner of collectively processing headers from related packets is to form them into one "super-

"header. In this method, the packets' data portions are stored separately (e.g., in a separate memory page or buffer) from the super-header.

Illustratively, a super-header comprises one combined header for each layer of the packets' associated protocol stack (e.g., one TCP header and one IP header). To form each layer's portion of a super-header, the packet's individual headers may be merged to make a regular-sized header whose fields accurately reflect the assembled data and combined headers. For example, merged header fields relating to payload or header length would indicate the size of the aggregated data or aggregated headers, the sequence number of a merged TCP header would be set appropriately, etc. The super-header portion may then be processed through its protocol stack similar to the manner in which a single packet's header is processed.

This method of collectively processing related packets' headers (e.g., with "super-"headers) may require modification of the instructions for processing packets (e.g., a device driver). For example, because multiple headers are merged for each layer of the protocol stack, the software may require modification to recognize and handle the super-headers. In one embodiment of the invention the number of headers folded or merged into a super-header may be limited. In an alternative embodiment of the invention the headers of all the aggregated packets, regardless of number, may be combined.

In another method of collectively processing related packets' header portions, packet data and headers may again be stored separately (e.g., in separate memory pages). But, instead of combining the packets' headers for each layer of the appropriate protocol stack to form a super-header, they may be submitted for individual processing in quick succession. For example, all of the packets' layer two headers may be processed in a rapid sequence—one after the other—then all of the layer three headers, etc. In this manner, packet processing instructions need not be modified, but headers are still processed more efficiently. In particular, a set of instructions (e.g., for each protocol layer) may be loaded once for all related packets rather than being separately loaded and executed for each packet.

As discussed in a previous section, data portions of related packets may be transferred into storage areas of predetermined size (e.g., memory pages) for efficient transfer from the host computer's kernel space into application or user space. Where the transferred data is of memory page size, the data may be transferred using highly efficient "page-flipping," wherein a full page of data is provided to application or user memory space.

FIGS. **22A–22B** present one method of dynamic packet batching with packet batching module **122**. In the illustrated method, memory **2102** is populated with flow numbers of packets stored in packet queue **116**. In particular, a packet's flow number and operation code are retrieved from control queue **118**, IPP module **104**, flow database manager **108** or other module(s) of NIC **100**. The packet's flow number is stored in the flow number portion of an entry in memory **2102**, and validity indicator **2110** is set in accordance with the operation code. For example, if the packet is not re-assembleable (e.g., codes 2 and 5 in TABLE 1), the validity indicator may be set to zero; otherwise it may be set to one.

The illustrated method may operate in parallel to the operation of DMA engine **120**. In other words, dynamic packet batching module **122** may search for packets related to a packet in the process of being transferred to a host memory buffer. Alternatively, a search may be conducted shortly after or before the packet is transferred. Because memory **2102** may be associative in nature, the search operation may be conducted quickly, thus introducing little, if any, delay into the transfer process.

FIG. **22A** may be considered a method of searching for a related packet, while FIG. **22B** may be considered a method of populating the dynamic packet batching module's memory.

FIGS. **22A–22B** each reflect one "cycle" of a dynamic packet batching operation (e.g., one search and creation of one new memory entry). Illustratively, however, the operation of packet batching module **122** runs continuously. That is, at the end of one cycle of operation another cycle immediately begins. In this manner, controller **2104** strives to ensure memory **2102** is populated with entries for packets as they are stored in packet queue **116**. If memory **2102** is not large enough to store an entry for each packet in packet queue **116**, then controller **2104** attempts to keep the memory as full as possible and to quickly replace an invalidated entry with a new one.

State **2200** is a start state for a memory search cycle. In state **2202**, it is determined whether a packet (e.g., the packet at the front of the packet queue) is being transferred to the host computer. This determination may, for example, be based on the operation of DMA engine **120** or the status of a pointer in packet queue **116** or control queue **118**. Illustratively, state **2202** is initiated by DMA engine **120** as a packet is copied into a buffer in the host computer. One purpose of state **2202** is simply to determine whether memory **2102** should be searched for a packet related to one that was, will be, or is being transferred. Until a packet is transferred, or about to be transferred, the illustrated procedure continues in state **2202**.

When, however, it is time for a search to be conducted (e.g., a packet is being transferred), the method continues at state **2204**. In state **2204**, the entry in memory **2102** corresponding to the packet being transferred is invalidated. Illustratively, this consists of storing a predetermined value (e.g., zero) in validity indicator **2110** for the packet's entry. In a present embodiment of the invention read pointer **2112** identifies the entry corresponding to the packet to be transferred. As one skilled in the art will recognize, one reason for invalidating a transferred packet's entry is so that when memory **2102** is searched for an entry associated with a packet related to the transferred packet, the transferred packet's own entry will not be identified.

In one embodiment of the invention the transferred packet's flow number is copied into a register (e.g., a hardware register) when dynamic packet batching module **122** is to search for a related packet. This may be particularly helpful (e.g., to assist in comparing the flow number to flow numbers of other packets) if memory **2102** is implemented as a RAM instead of a CAM.

In state **2206**, read pointer **2112** is incremented to point to the next entry in memory **2102**. If read pointer is incremented to the same entry that is referenced by write pointer **2114**, and that entry is also invalid (as indicated by validity indicator **2110**), it may be determined that memory **2102** is now empty.

Then, in state **2208**, memory **2102** is searched for a packet related to the packet being transferred (e.g., the memory is searched for an entry having the same flow number). As described above, entries in memory **2102** are searched associatively in one embodiment of the invention. Thus, the result of the search operation may be a single signal indicating whether or not a match was found.

In the illustrated embodiment of the invention, only valid entries (e.g., those having a value of one in their validity

indicators) are searched. As explained above, an entry may be marked invalid (e.g., its validity indicator stores a value of zero) if the associated packet is considered incompatible. Entries for incompatible packets may be disregarded because their data is not ordinarily re-assembled and their headers are not normally batched. In an alternative embodiment of the invention, all entries may be searched but a match is reported only if a matching entry is valid.

In state **2210**, the host computer is alerted to the availability or non-availability of a related packet. In this embodiment of the invention, the host computer is alerted by storing a predetermined value in a specific field of the transferred packet's completion descriptor (described in a previous section). As discussed in the previous section, when a packet is transferred a descriptor in a descriptor ring in host memory is populated with information concerning the packet (e.g., an identifier of its location in host memory, its size, an identifier of a processor to process the packet's headers). In particular, a release flow flag or indicator is set to a first value (e.g., zero) if a related packet is found, and a second value if no related packet is found. Illustratively, DMA engine **120** issues the alert or stores the necessary information to indicate the existence of a related packet in response to notification from dynamic packet batching module **122**. Other methods of notifying the host computer of the presence of a related packet are also suitable (e.g., an indicator, flag, key), as will be appreciated by one skilled in the art.

In FIG. 22B, state **2220** is a start state for a memory population cycle.

In state **2222**, it is determined whether a new packet has been received at the network interface. Illustratively, a new entry is made in the packet batching module's memory for each packet received from the network. The receipt of a new packet may be signaled by IPP module **104**. For example, the receipt of a new packet may be indicated by the storage of the packet's flow number, by IPP module **104**, in a temporary location (e.g., a register). Until a new packet is received, the illustrated procedure waits. When a packet is received, the procedure continues at state **2224**.

In state **2224**, if memory **2102** is configured to store fewer entries than packet queue **116** (and, possibly, control queue **118**), memory **2102** is examined to determine if it is full.

In one embodiment of the invention memory **2102** may be considered full if the validity indicator is set (e.g., equal to one) for each entry or for the entry referenced by write pointer **2114**. If the memory is full, the illustrated procedure waits until the memory is not full. As one skilled in the art will recognize, memory **2102** and other data structures in NIC **100** may be tested for saturation (e.g., whether they are filled) by comparing their read and write pointers.

In state **2226**, a new packet is represented in memory **2102** by storing its flow number in the entry identified by write pointer **2114** and storing an appropriate value in the entry's validity indicator field. If, for example, the packet is not re-assembleable (e.g., as indicated by its operation code), the entry's validity indicator may be set to an invalid state. For purposes of the operation of dynamic packet batching module **122**, a TCP control packet may or may not be considered re-assembleable. Thus, depending upon the implementation of a particular embodiment the validity indicator for a packet that is a TCP control packet may be set to a valid or invalid state.

In an alternative embodiment of the invention an entry in memory **2102** is populated with information from the control queue entry identified by the second read pointer described above. This pointer may then be incremented to the next entry in control queue **118**.

In state **2228**, write pointer **2114** is incremented to the next entry of memory **2102**, after which the illustrated method ends at end state **2230**. If write pointer **2114** references the same entry as read pointer **2112**, it may be determined that memory **2102** is full. One skilled in the art will recognize that many other suitable methods of managing pointers for memory **2102** may be employed.

As mentioned above, in one embodiment of the invention one or both of the memory search and memory population operations run continuously. Thus, end state **2230** may be removed from the procedure illustrated in FIG. 22B, in which case the procedure would return to state **2222** after state **2228**.

Advantageously, in the illustrated embodiment of the invention the benefits provided to the host computer by dynamic packet batching module **122** increase as the host computer becomes increasingly busy. In particular, the greater the load placed on a host processor, the more delay that will be incurred until a packet received from NIC **100** may be processed. As a result, packets may queue up in packet queue **116** and, the more packets in the packet queue, the more entries that can be maintained in memory **2102**.

The more entries that are stored in memory **2102**, the further ahead dynamic packet batching module can look for a related packet. The further ahead it scans, the more likely it is that a related packet will be found. As more related packets are found and identified to the host computer for collective processing, the amount of processor time spent on network traffic decreases and overall processor utilization increases.

One skilled in the art will appreciate that other systems and methods may be employed to identify multiple packets from a single communication flow or connection without exceeding the scope of the present invention.

Early Random Packet Discard in One Embodiment of the Invention

Packets may arrive at a network interface from a network at a rate faster than they can be transferred to a host computer. When such a situation exists, the network interface must often drop, or discard, one or more packets. Therefore, in one embodiment of the present invention a system and method for randomly discarding a packet are provided. Systems and methods discussed in this section may be applicable to other communication devices as well, such as gateways, routers, bridges, modems, etc.

As one skilled in the art will recognize, one reason that a packet may be dropped is that a network interface is already storing the maximum number of packets that it can store for transfer to a host computer. In particular, a queue that holds packets to be transferred to a host computer, such as packet queue **116** (shown in FIG. 1A), may be fully populated when another packet is received from a network. Either the new packet or a packet already stored in the queue may be dropped.

Partly because of the bursty nature of much network traffic, multiple packets may often be dropped when a network interface is congested. And, in some network interfaces, if successive packets are dropped one particular network connection or flow (e.g., a connection or flow that includes all of the dropped packets) may be penalized even if it is not responsible for the high rate of packet arrival. If a network connection or flow is penalized too heavily, the network entity generating the traffic in that connection or flow may tear it down in the belief that a "broken pipe" has been encountered. As one skilled in the art will recognize, a broken pipe occurs when a network entity interprets a communication problem as indicating that a connection has been severed.

For certain network traffic (e.g., TCP traffic), the dropping of a packet may initiate a method of flow control in which a network entity's window (e.g., number of packets it transmits before waiting for an acknowledgement) shrinks or is reset to a very low number. Thus, every time a packet from a TCP communicant is dropped by a network interface at a receiving entity, the communicant must re-synchronize its connection with the receiving entity. If one or a subset of communicants are responsible for a large percentage of network traffic received at the entity, then it seems fair that those communicants should be penalized in proportion to the amount of traffic that it is responsible for.

In addition, it may be wise to prevent certain packets or types of packets from being discarded. For example, discarding a small control packet may do very little to alleviate congestion in a network interface and yet have a drastic and negative effect upon a network connection or flow. Further, if a network interface is optimized for packets adhering to a particular protocol, it may be more efficient to avoid dropping such packets. Even further, particular connections, flows or applications may be prioritized, in which case higher priority traffic should not be dropped.

Thus, in one embodiment of a network interface according to the present invention, a method is provided for randomly discarding a packet when a communication device's packet queue is full or is filled to some threshold level. Intelligence may be added to such a method by selecting certain types of packets for discard (e.g., packets from a particular flow, connection or application) or excepting certain types of packets from being discarded (e.g., control packets, packets conforming to a particular protocol or set of protocols).

A provided method is random in that discarded packets are selected randomly from those packets that are considered discardable. Applying a random discard policy may be sufficient to avoid broken pipes by distributing the impact of dropped packets among multiple connections or flows. In addition, if a small number of transmitting entities are responsible for a majority of the traffic received at a network interface, dropping packets randomly may ensure that the offending entities are penalized proportionately. Different embodiments of the invention that are discussed below provide various combinations of randomness and intelligence, and one of these attributes may be omitted in one or more embodiments.

FIG. **24** depicts a system and method for randomly discarding packets in a present embodiment of the invention. In this embodiment, packet queue **2400** is a hardware FIFO (e.g., first-in first-out) queue that is 16 KB in size. In other embodiments of the invention the packet queue may be smaller or larger or may comprise another type of data structure (e.g., list, array, table, heap) implemented in hardware or software.

Similar to packet queue **116** discussed in a previous section, packet queue **2400** receives packets from a network and holds them for transfer to a host computer. Packets arriving from a network may arrive from the network at a high rate and may be processed or examined by one or more modules (e.g., header parser **106**, flow database manager **108**) prior to being stored in packet queue **2400**. For example, where the network is capable of transmitting one gigabit of traffic per second, packets conforming to one set of protocols (e.g., Ethernet, IP and TCP) may be received at a rate of approximately 1.48 million packets per second. After being stored in packet queue **2400**, packets are transferred to a host computer at a rate partially dependent upon events and conditions internal to the host computer. Thus, the network interface may not be able to control the rate of packet transmittal to the host computer.

In the illustrated embodiment, packet queue **2400** is divided into a plurality of zones or regions, any of which may overlap or share a common boundary. Packet queue **2400** may be divided into any number of regions, and the invention is not limited to the three regions depicted in FIG. **24**. Illustratively, region zero (represented by the numeral **2402**) encompasses the portion of packet queue **2400** from 0 KB (e.g., no packets are stored in the queue) to 8 KB (e.g., half full). Region one (represented by the numeral **2404**) encompasses the portion of the packet queue from 8 KB to 12 KB. Region two (represented by the numeral **2406**) encompasses the remaining portion of the packet queue, from 12 KB to 16 KB. In an alternative embodiment, regions may only be defined for a portion of packet queue **2400**. For example, only the upper half (e.g., above 8 KB) may be divided into one or more regions.

The number and size of the different regions and the location of boundaries between the regions may vary according to several factors. Among the factors are the type of packets received at the network interface (e.g., the protocols according to which the packets are configured), the size of the packets, the rate of packet arrival (e.g., expected rate, average rate, peak rate), the rate of packet transfer to the host computer, the size of the packet queue, etc. For example, in another embodiment of the invention, packet queue **2400** is divided into five regions. A first region extends from 0 KB to 8 KB; a second region ranges from 8 KB to 10 KB; a third from 10 KB to 12 KB; a fourth from 12 KB to 14 KB; and a final region extends from 14 KB to 16 KB.

During operation of a network interface according to a present embodiment, traffic indicator **2408** indicates how full packet queue **2400** is. Traffic indicator **2408**, in one embodiment of the invention, comprises read pointer **810** and/or write pointer **812** (shown in FIG. **8**). In the presently discussed embodiment in which packet queue **2400** is fully partitioned, traffic indicator **2408** will generally be located in one of the regions into which the packet queue was divided or at a dividing boundary. Thus, during operation of a network interface appropriate action may be taken, as described below, depending upon how full the packet queue is (e.g., depending upon which region is identified by traffic indicator **2408**).

In FIG. **24**, counter **2410** is incremented as packets arrive at packet queue **2400**. In the illustrated embodiment, counter **2410** continuously cycles through a limited range of values, such as zero through seven. In one embodiment of the invention, each time a new packet is received the counter is incremented by one. In an alternative embodiment, counter **2410** may not be incremented when certain "non-discardable" packets are received. Various illustrative criteria for identifying non-discardable packets are presented below.

For one or more regions of packet queue **2400**, an associated programmable probability indicator indicates the probability that a packet will be dropped when traffic indicator **2408** indicates that the level of traffic in the packet queue has reached the associated region. Therefore, in the illustrated embodiment probability indicator **2412** indicates the probability that a packet will be dropped while the packet queue is less than half full (e.g., when traffic indicator **2408** is located in region zero). Similarly, probability indicators **2414** and **2416** specify the probability that a new packet will be dropped when traffic indicator **2408** identifies regions one and two, respectively.

For certain network traffic (e.g., TCP traffic), the dropping of a packet may initiate a method of flow control in which a network entity's window (e.g., number of packets it transmits before waiting for an acknowledgement) shrinks or is reset to a very low number. Thus, every time a packet from a TCP communicant is dropped by a network interface at a receiving entity, the communicant must re-synchronize its connection with the receiving entity. If one or a subset of communicants are responsible for a large percentage of network traffic received at the entity, then it seems fair that those communicants should be penalized in proportion to the amount of traffic that it is responsible for.

In addition, it may be wise to prevent certain packets or types of packets from being discarded. For example, discarding a small control packet may do very little to alleviate congestion in a network interface and yet have a drastic and negative effect upon a network connection or flow. Further, if a network interface is optimized for packets adhering to a particular protocol, it may be more efficient to avoid dropping such packets. Even further, particular connections, flows or applications may be prioritized, in which case higher priority traffic should not be dropped.

Thus, in one embodiment of a network interface according to the present invention, a method is provided for randomly discarding a packet when a communication device's packet queue is full or is filled to some threshold level. Intelligence may be added to such a method by selecting certain types of packets for discard (e.g., packets from a particular flow, connection or application) or excepting certain types of packets from being discarded (e.g., control packets, packets conforming to a particular protocol or set of protocols).

A provided method is random in that discarded packets are selected randomly from those packets that are considered discardable. Applying a random discard policy may be sufficient to avoid broken pipes by distributing the impact of dropped packets among multiple connections or flows. In addition, if a small number of transmitting entities are responsible for a majority of the traffic received at a network interface, dropping packets randomly may ensure that the offending entities are penalized proportionately. Different embodiments of the invention that are discussed below provide various combinations of randomness and intelligence, and one of these attributes may be omitted in one or more embodiments.

FIG. **24** depicts a system and method for randomly discarding packets in a present embodiment of the invention. In this embodiment, packet queue **2400** is a hardware FIFO (e.g., first-in first-out) queue that is 16 KB in size. In other embodiments of the invention the packet queue may be smaller or larger or may comprise another type of data structure (e.g., list, array, table, heap) implemented in hardware or software.

Similar to packet queue **116** discussed in a previous section, packet queue **2400** receives packets from a network and holds them for transfer to a host computer. Packets arriving from a network may arrive from the network at a high rate and may be processed or examined by one or more modules (e.g., header parser **106**, flow database manager **108**) prior to being stored in packet queue **2400**. For example, where the network is capable of transmitting one gigabit of traffic per second, packets conforming to one set of protocols (e.g., Ethernet, IP and TCP) may be received at a rate of approximately 1.48 million packets per second. After being stored in packet queue **2400**, packets are transferred to a host computer at a rate partially dependent upon events and conditions internal to the host computer. Thus, the network interface may not be able to control the rate of packet transmittal to the host computer.

In the illustrated embodiment, packet queue **2400** is divided into a plurality of zones or regions, any of which may overlap or share a common boundary. Packet queue **2400** may be divided into any number of regions, and the invention is not limited to the three regions depicted in FIG. **24**. Illustratively, region zero (represented by the numeral **2402**) encompasses the portion of packet queue **2400** from 0 KB (e.g., no packets are stored in the queue) to 8 KB (e.g., half full). Region one (represented by the numeral **2404**) encompasses the portion of the packet queue from 8 KB to 12 KB. Region two (represented by the numeral **2406**) encompasses the remaining portion of the packet queue, from 12 KB to 16 KB. In an alternative embodiment, regions may only be defined for a portion of packet queue **2400**. For example, only the upper half (e.g., above 8 KB) may be divided into one or more regions.

The number and size of the different regions and the location of boundaries between the regions may vary according to several factors. Among the factors are the type of packets received at the network interface (e.g., the protocols according to which the packets are configured), the size of the packets, the rate of packet arrival (e.g., expected rate, average rate, peak rate), the rate of packet transfer to the host computer, the size of the packet queue, etc. For example, in another embodiment of the invention, packet queue **2400** is divided into five regions. A first region extends from 0 KB to 8 KB; a second region ranges from 8 KB to 10 KB; a third from 10 KB to 12 KB; a fourth from 12 KB to 14 KB; and a final region extends from 14 KB to 16 KB.

During operation of a network interface according to a present embodiment, traffic indicator **2408** indicates how full packet queue **2400** is. Traffic indicator **2408**, in one embodiment of the invention, comprises read pointer **810** and/or write pointer **812** (shown in FIG. **8**). In the presently discussed embodiment in which packet queue **2400** is fully partitioned, traffic indicator **2408** will generally be located in one of the regions into which the packet queue was divided or at a dividing boundary. Thus, during operation of a network interface appropriate action may be taken, as described below, depending upon how full the packet queue is (e.g., depending upon which region is identified by traffic indicator **2408**).

In FIG. **24**, counter **2410** is incremented as packets arrive at packet queue **2400**. In the illustrated embodiment, counter **2410** continuously cycles through a limited range of values, such as zero through seven. In one embodiment of the invention, each time a new packet is received the counter is incremented by one. In an alternative embodiment, counter **2410** may not be incremented when certain "non-discardable" packets are received. Various illustrative criteria for identifying non-discardable packets are presented below.

For one or more regions of packet queue **2400**, an associated programmable probability indicator indicates the probability that a packet will be dropped when traffic indicator **2408** indicates that the level of traffic in the packet queue has reached the associated region. Therefore, in the illustrated embodiment probability indicator **2412** indicates the probability that a packet will be dropped while the packet queue is less than half full (e.g., when traffic indicator **2408** is located in region zero). Similarly, probability indicators **2414** and **2416** specify the probability that a new packet will be dropped when traffic indicator **2408** identifies regions one and two, respectively.

In the illustrated embodiment, probability indicators **2412**, **2414** and **2416** each comprise a set, or mask, of sub-indicators such as bits or flags. Illustratively, the number of sub-indicators in a probability indicator matches the range of counter values—in this case, eight. In one embodiment of the invention, each sub-indicator may have one of two values (e.g., zero or one) indicating whether a packet is dropped. Thus, the sub-elements of a probability indicator may be numbered from zero to seven (illustratively, from right to left) to correspond to the eight possible values of counter **2410**. For each position in a probability indicator that stores a first value (e.g., one), when the value of counter **2410** matches the number of that bit, the next discardable packet received for packet queue **2400** will be dropped. As discussed above, certain types of packets (e.g., control packets) may not be dropped. Illustratively, counter **2410** is only incremented for discardable packets.

In FIG. **24**, probability indicator **2412** (e.g., 00000000) indicates that no packets are to be dropped as long as the packet queue is less than half full (e.g., as long as traffic indicator **2408** is in region zero). Probability indicator **2414** (e.g., 00000001) indicates that every eighth packet is to be dropped when there is at least 8 KB stored in the packet queue. In other words, when traffic indicator **2408** is located in region one, there is a 12.5% probability that a discardable packet will be dropped. In particular, when counter **2410** equals zero the next discardable packet, or a packet already stored in the packet queue, is discarded. Probability indicator **2416** (e.g., 01010101) specifies that every other discardable packet is to be dropped. There is thus a 50% probability that a discardable packet will be dropped when the queue is more than three-quarters full. Illustratively, when a packet is dropped, counter **2410** is still incremented.

As another example, in the alternative embodiment described above in which the packet queue is divided into five regions, suitable probability indicators may include the following. For regions zero and one, 00000000; for region two, 00000001; for region three, 00000101; and for region four, 01111111. Thus, in this alternative embodiment, region one is treated as an extension to region zero. Further, the probability of dropping a packet has a wider range, from 0% to 87.5%.

In one alternative embodiment described above, only a portion of a packet queue is partitioned into regions. In this alternative embodiment, a default probability or null probability (e.g., 00000000) of dropping a packet may be associated with the un-partitioned portion. Illustratively, this ensures that no packets are dropped before the level of traffic stored in the queue reaches a first threshold. Even in an embodiment where the entire queue is partitioned, a default or null probability may be associated with a region that encompasses or borders a 0 KB threshold.

Just as a packet queue may be divided into any number of regions for purposes of the present invention, probability indicators may comprise bit masks of any size or magnitude, and need not be of equal size or magnitude. Further, probability indicators are programmable in a present embodiment, thus allowing them to be altered even during the operation of a network interface.

One skilled in the art will recognize that discarding packets on the basis of a probability indicator injects randomness into the discard process. A random early discard policy may be sufficient to avoid the problem of broken pipes discussed above. In particular, in one embodiment of the invention, all packets are considered discardable, such that all packets are counted by counter **2410** and all are candidates for being dropped. As already discussed, however, in another embodiment of the invention intelligence is added in the process of excluding certain types of packets from being discarded.

It will be understood that probability indicators and a counter simply constitute one system for enabling the random discard of packets in a network interface. Other mechanisms are also suitable. In one alternative embodiment, a random number generator may be employed in place of a counter and/or probability indicators to enable a random discard policy. For example, when a random number is generated, such as M, the Mth packet (or every Mth packet) after the number is generated may be dropped. Or, the random number may specify a probability of dropping a packet. The random number may thus be limited to (e.g., hashed into) a certain range of values or probabilities. As another alternative, a random number generator may be used in tandem with multiple regions or thresholds within a packet queue. In this alternative embodiment a programmable value, represented here as N, may be associated with a region or queue threshold. Then, when a traffic indicator reaches that threshold or region, the Nth packet (or every Nth packet) may be dropped until another threshold or boundary is reached.

In yet another alternative embodiment of the invention, the probability of dropping a packet is expressed as a binary fraction. As one skilled in the art will recognize, a binary fraction consists of a series of bits in which each bit represents one half of the magnitude of its more significant neighbor. For example, a binary fraction may use four digits in one embodiment of the invention. From left to right, the bits may represent 0.5, 0.25, 0.125 and 0.0625, respectively. Thus, a binary fraction of 1010 would be interpreted as indicating a 62.5% probability of dropping a packet (e.g., 50% plus 12.5%). The more positions (e.g., bits) used in a binary fraction, the greater precision that may be attained.

In one implementation of this alternative embodiment a separate packet counter is associated with each digit. The counter for the leftmost bit increments at twice the rate of the next counter, which increments twice as fast as the next counter, etc. In other words, when the counter for the most significant (e.g., left) bit increments from 0 to 1 the other counters do not change. When the most significant counter increments again, from 1 back to 0, then the next counter increments from 0 to 1. Likewise, the counter for the third bit does not increment from 0 to 1 until the second counter returns to 0. In summary, the counter for the most significant bit changes (i.e., increments) each time a packet is received. The counter for the next most significant bit maintains each value (i.e., 0 or 1) for two packets before incrementing. Similarly, the counter for the third most significant bit maintains each counter value for four packets before incrementing and the counter for the least significant bit maintains its values for eight packets before incrementing.

Each time a packet is received or a counter is incremented the counters are compared to the probability indicator (e.g., the specified binary fraction). In one embodiment the determination of whether a packet is dropped depends upon which of the fraction's bits are equal to one. Illustratively, for each fraction bit equal to one a random packet is dropped if the corresponding counter is equal to one and the counters for any bits of higher significance are equal to zero. Thus for the example fraction **1010**, whenever the most significant bit's counter is equal to one a random packet is dropped. In addition, a random packet is also dropped whenever the counter for the third bit is equal to one and the counters for the first two bits are equal to zero.

A person skilled in the art may also derive other suitable mechanisms for specifying and enforcing a probability of

dropping a packet received at a network interface without exceeding the scope of the present invention.

As already mentioned, intelligence may be imparted to a random discard policy in order to avoid discarding certain types of packets. In a previous section, methods of parsing a packet received from a network were described. In particular, in a present embodiment of the invention a packet received from a network is parsed before it is placed into a packet queue such as packet queue **2400**. During the parsing procedure various information concerning the packet may be gleaned. This information may be used to inject intelligence into a random discard policy. In particular, one or more fields of a packet header may be copied, an originating or destination entity of the packet may be identified, a protocol may be identified, etc.

Thus, in various embodiments of the invention, certain packets or types of packets may be immune from being discarded. In the embodiment illustrated in FIG. **24**, for example, control packets are immune. As one skilled in the art will appreciate, control packets often contain information essential to the establishment, re-establishment or maintenance of a communication connection. Dropping a control packet may thus have a more serious and damaging effect than dropping a packet that is not a control packet. In addition, because control packets generally do not contain data, dropping a control packet may save very little space in the packet queue.

Many other criteria for immunizing packets are possible. For example, when a packet is parsed according to a procedure described in a previous section, a No_Assist flag or signal may be associated with the packet to indicate whether the packet is compatible with a set of pre-selected communication protocols. Illustratively, if the flag is set to a first value (e.g., one) or the signal is raised, the packet is considered incompatible and is therefore ineligible for certain processing enhancements (e.g., re-assembly of packet data, batch processing of packet headers, load-balancing). Because a packet for which a No_Assist flag is set to the first value may be a packet conforming to an unexpected protocol or unique format, it may be better not to drop such packets. For example, a network manager may want to ensure receipt of all such packets in order to determine whether a parsing procedure should be augmented with the ability to parse additional protocols.

Another reason for immunizing a No_Assist packet (e.g., packets that are incompatible with a set of selected protocols) from being discarded concerns the reaction to dropping the packet. Because the packet's protocols were not identified, it may not be known how the packet's protocols respond to the loss of a packet. In particular, if the sender of the packet does not lower its transmission rate in response to the dropped packet (e.g., as a form of congestion control), then there is no benefit to dropping it.

A packet's flow number may be used to immunize certain packets in another alternative embodiment of the invention. As discussed in a previous section, a network interface may include a flow database and flow database manager to maintain a record of multiple communication flows received by the network interface. It may be efficacious to prevent packets from one or more certain flows from being discarded. Immunized flows may include a flow involving a high-priority network entity, a flow involving a particular application, etc. For example, it may be considered relatively less damaging to discard packets from an animated or streaming graphics application in which a packet, or a few packets, may be lost without seriously affecting the destination entity and the packets may not even need to be retransmitted. In contrast, the consequences may be more severe if a few packets are dropped from a file transfer connection. The packets will likely need to be retransmitted, and the transmitting entity's window may be shrunk as a result—thus decreasing the rate of file transfer.

In yet another alternative embodiment of the invention, a probability indicator may comprise a bit mask in which each bit corresponds to a separate, specific flow through the network interface. In particular, the bits may correspond to the flows maintained in the flow database described in a previous section.

Although embodiments of the invention discussed thus far in this section involve discarding packets as they arrive at a packet queue, in an alternative embodiment packets may be discarded from within the packet queue. In particular, as the packet queue is filled (e.g., as a traffic indicator reaches pre-defined regions or thresholds), packets already stored in the queue may be discarded at random according to one or more probability indicators. In the embodiment illustrated in FIG. **24**, for example, when traffic indicator **2408** reaches a certain threshold, such as the boundary between regions one and two or the end of the queue, packets may be deleted in one or more regions according to related probability indicators. Such probability indicators would likely have different values than those indicated in FIG. **24**.

In a present embodiment of the invention, probability indicators and/or the specifications (e.g., boundaries) into which a packet queue is partitioned are programmable and may be adjusted by software operating on a host computer (e.g., a device driver). Criteria for immunizing packets may also be programmable. Methods of discarding packets in a network interface or other communication device may thus be altered in accordance with the embodiments described in this section, even during continued operation of such a device. Various other embodiments and criteria for randomly discarding packets and/or applying criteria for the intelligent discard of packets will be apparent to those skilled in the art.

FIGS. **25A–25B** comprise a flow chart demonstrating one method of implementing a policy for randomly discarding packets in a network interface according to the embodiment of the invention substantially similar to the embodiment illustrated in FIG. **24**. In this embodiment, a packet is received while packet queue **2400** is not yet full. As one skilled in the will appreciate, this embodiment provides a method of determining whether to discard the packet. Once packet queue **2400** is full, when another packet is received the network interface generally must drop a packet—either the one just received or one already stored in the queue—in which case the only decision is which packet to drop.

In FIG. **25A**, state **2500** is a start state. State **2500** may reflect the initialization of the network interface (and packet queue **2400**) or may reflect a point in the operation of the network interface at which one or more parameters or aspects concerning the packet queue and the random discard policy are to be modified.

In state **2502**, one or more regions are identified in packet queue **2400**, perhaps by specifying boundaries such as the 8 KB and 12 KB boundaries depicted in FIG. **24**. Although the regions depicted in FIG. **24** fully encompass packet queue **2400** when viewed in unison, regions in an alternative embodiment of the invention may encompass less than the entire queue.

In state **2504**, one or more probability indicators are assigned and configured. In the illustrated embodiment, one probability indicator is associated with each region. Alternatively, multiple regions may be associated with one

probability indicator. Even further, one or more regions may not be explicitly associated with a probability indicator, in which case a default or null probability indicator may be assumed. As described above, a probability indicator may take the form of a multi-bit mask, whereby the number of bits in the mask reflect the range of possible values maintained by a packet counter. In another embodiment of the invention, a probability indicator may take the form of a random number or a thresbold value against which a randomly generated number is compared when a decision must be whether to discard a packet.

In state **2506**, if certain types of packets are to be prevented from being discarded, criteria are expressed to identify the exempt packets. Some packets that may be exempted are control packets, packets conforming to unknown or certain known protocols, packets belonging to a particular network connection or flow, etc. In one embodiment of the invention, no packets are exempt from being discarded.

In state **2508**, a packet or traffic counter is initialized. As described above, the counter may be incremented, possibly through a limited range of values, when a discardable packet is received for storage in packet queue **2400**. The limited range of counter values may correspond to the number of bits in a mask form of a probability indicator. Alternatively, the counter may be configured to increment through a greater range, in which case a counter value may be filtered through a modulus or hash function prior to being compared to a probability indicator as described below.

In state **2510**, a packet is received from a network and may be processed through one or more modules (e.g., a header parser, an IPP module) prior to its arrival at packet queue **2400**. Thus, in state **2510** the packet is ready to be stored in the packet queue. One or more packets may already be stored in the packet queue and a traffic indicator (e.g., a pointer or index) identifies the level of traffic stored in the queue (e.g., by a storage location and/or region in the queue).

In state **2512**, it may be determined whether the received packet is discardable. For example, if the random discard policy that is in effect allows for the exemption of some packets from being discarded, in state **2512** it is determined whether the received packet meets any of the exemption criteria. If so, the illustrated procedure continues at state **2522**. Otherwise, the procedure continues at state **2514**.

In state **2514**, an active region of packet queue **2400** is identified. In particular, the region of the packet queue to which the queue is presently populated with traffic is determined. The level of traffic stored in the queue depends upon the number and size of packets that have been stored in the queue to await transfer to a host computer. The slower the transfer process, the higher the level of traffic may reach in the queue. Although the level of traffic stored in the queue rises and falls as packets are stored and transferred, the level may be identified at a given time by examining the traffic indicator. The traffic indicator may comprise a pointer identifying the position of the last or next packet to be stored in the queue. Such a pointer may be compared to another pointer that identifies the next packet to be transferred to the host computer in order to reveal how much traffic is stored in the queue.

In state **2516**, the counter value (e.g., a value between zero and seven in the embodiment of FIG. **24**) is compared to the probability indicator associated with the active region. As previously described, the counter is incremented as discardable packets are received at the queue. This comparison is conducted so as to determine whether the received packet should be discarded. As explained above, in the embodiment of FIG. **24** the setting of the probability indicator bit corresponding to the counter value is examined. For example, if the counter has a value of N, then bit number N of the probability indicator mask is examined. If the bit is set to a first state (e.g., one) the packet is to be discarded; otherwise it is not to be discarded.

In state **2518**, the counter is incremented to reflect the receipt of a discardable packet, whether or not the packet is to be discarded. In the presently discussed embodiment of the invention, if the counter contains its maximum value (e.g., seven) prior to being incremented, incrementing it entails resetting it to its minimum value (e.g., zero).

In state **2520**, if the packet is to be discarded the illustrated procedure continues at state **2524**. Otherwise, the procedure continues at state **2522**. In state **2522**, the packet is stored in packet queue **2400** and the illustrated procedure ends with end state **2526**. In state **2524**, the packet is discarded and the illustrated procedure ends with end state **2526**.

Sun, Sun Microsystems, SPARC and Solaris are trademarks or registered trademarks of Sun Microsystems, Incorporated in the United States and other countries.

The foregoing descriptions of embodiments of the invention have been presented for purposes of illustration and description only. They are not intended to be exhaustive or to limit the invention to the forms disclosed. Many modifications and variations will be apparent to practitioners skilled in the art. Accordingly, the above disclosure is not intended to limit the invention; the scope of the invention is defined by the appended claims.

What is claimed is:

1. A method of identifying multiple packets in a communication flow between a source entity and a destination entity, comprising:

storing a first flow identifier of a first packet received from a source entity for a destination entity, wherein said first flow identifier comprises an identifier of the source entity and an identifier of the destination entity;

storing said first packet in a packet memory for transfer toward the destination entity;

storing a second flow identifier of a second packet;

storing said second packet in said packet memory;

determining whether said first flow identifier matches said second flow identifier;

storing a first indicator in the destination entity if a first communication flow identified by said first flow identifier comprises said second packet; and

storing a second indicator in the destination entity if said first packet is the only packet stored in the packet memory that is part of said first communication flow.

2. The method of claim **1**, further comprising, prior to said storing a first flow identifier, parsing said first packet to retrieve said identifier of the source entity and said identifier of the destination entity.

3. A method of identifying one or more packets in a communication flow between a source entity and a destination entity, comprising:

receiving a first packet at a communication device;

identifying a first communication flow comprising said first packet with a first flow identifier configured to identify both the source entity and the destination entity;

determining whether said first communication flow also comprises a second packet received at said communication device after said first packet was received at said communication device; and

transferring said first packet to a host computer for processing in accordance with a communication protocol associated with said first packet.

4. The method of claim 3, further comprising:

transferring said second packet to said host computer;

wherein said host computer is configured to collectively process a header portion of said first packet and a header portion of said second packet in accordance with said communication protocol.

5. The method of claim 3, wherein said identifying comprises:

receiving a flow key generated by concatenating an identifier of the source entity and an identifier of the destination entity;

wherein said first flow identifier comprises said flow key.

6. The method of claim 3, wherein said identifying comprises:

receiving an index of said first communication flow in a flow database;

wherein said first flow identifier comprises said index.

7. The method of claim 3, wherein said determining comprises comparing said first flow identifier with a second flow identifier associated with a second packet received at said communication device.

8. The method of claim 7, wherein said determining further comprises:

storing said first flow identifier in a flow memory; and

storing said second flow identifier in said flow memory; and

comparing said stored first flow identifier and said stored second flow identifier.

9. The method of claim 8, wherein said flow memory is an associative memory in said communication device.

10. The method of claim 3, further comprising storing said first packet in a packet memory.

11. The method of claim 10, wherein said determining comprises comparing said first flow identifier configured to identify said first communication flow with a second flow identifier configured to identify a second communication flow comprising a packet stored in said packet memory.

12. The method of claim 3, further comprising informing said host computer of said transfer of said first packet.

13. The method of claim 12, wherein said informing comprises configuring an indicator in a host memory.

14. The method of claim 13, wherein said indicator is configured to indicate that said host computer should delay processing said first packet until said second packet is transferred to said host computer.

15. The method of claim 13, wherein said indicator indicates that said host computer should not delay processing said first packet.

16. A method of transferring a packet from a network interface to a host computer, comprising:

receiving a first packet at a network interface;

storing said first packet in a packet memory;

receiving a first flow identifier configured to identify a communication flow comprising said first packet;

storing said first flow identifier in a flow memory;

searching said flow memory for a second packet in said communication flow received at the network interface after said first packet;

transferring said first packet to said host computer; and

configuring an indicator in a host memory to indicate whether processing of said first packet by said host computer should be delayed to await transfer of said second packet to said host memory.

17. The method of claim 16, wherein said generating comprises:

receiving an index of said communication flow in a flow database;

wherein said flow identifier comprises said index.

18. The method of claim 16, wherein said receiving comprises:

receiving a flow key comprising an identifier of a source of said first packet and an identifier of a destination of said first packet;

wherein said flow identifier comprises said flow key.

19. The method of claim 16, wherein said packet memory comprises said flow memory.

20. The method of claim 16, wherein said configuring comprises:

storing a first indicator in a host memory if said communication flow comprises said second packet; and

storing a second indicator in said host memory if said first packet is the only packet in said packet memory that is part of said communication flow.

21. A computer system for processing a packet received from a network interface, comprising:

a network interface configured to receive a first packet from a network and transfer said first packet to a host computer memory, said network interface comprising:

a packet memory configured to store said first packet;

a flow memory for storing a first flow number associated with said first packet, wherein said first flow number is configured to identify a communication flow comprising said first packet;

a packet batcher configured to determine whether the communication flow includes a second packet stored in said packet memory after said first packet; and

a notifier configured to:

store a first code in a host indicator if said packet memory includes the second packet; and

store a second code in said host indicator if said packet memory does not include the second packet; and

a processor for processing a header portion of said first packet.

22. A computer readable storage medium storing instructions that, when executed by a computer, cause the computer to perform a method of transferring a packet from a network interface to a host computer, the method comprising:

receiving a first packet at a communication device;

identifying a first communication flow comprising said first packet with a first flow identifier configured to identify both the source entity and the destination entity;

determining whether said first communication flow also comprises a second packet received at said communication device after said first packet was received at said communication device; and

transferring said first packet to a host computer for processing in accordance with a communication protocol associated with said first packet.

23. A processor readable storage medium containing a data structure configured to store information concerning a packet to be transferred from a network interface to a host computer, the data structure including one or more entries, each entry comprising:

a flow number configured to identify a communication flow comprising a first packet received at the network

interface from a source entity for a destination entity associated with the host computer; and

a validity indicator configured to provide:

a first indication if said first packet is ready for transfer to the host computer; and

a second indication if said first packet is a control packet;

wherein said data structure is searched for a second entry containing said flow number when said first packet is transferred to the host computer to determine if said communication flow also comprises a second packet received at the network interface after said first packet.

24. The method of claim **3**, wherein said identifying comprises:

parsing said first packet to retrieve an identifier of the source entity and an identifier of the destination entity; and

combining said source entity identifier and said destination entity identifier to form said first flow identifier.

25. A communication interface, comprising:

a header parser configured to parse a header of a first packet received at the communication interface, wherein the first packet was issued from a source entity for a destination entity;

a flow database configured to facilitate management of a communication flow comprising the first packet, the flow database comprising:

a flow key configured to identify the communication flow using identifiers of the source entity and the destination entity;

an activity indicator configured to indicate a recency with which a packet in the communication flow has been received; and

a validity indicator for indicating whether the communication flow is valid;

a code generator configured to generate an operation code for the first packet, to facilitate forwarding of the first packet toward the destination entity; and

a packet batching module configured to determine whether a second packet received at the communication interface is part of the communication flow.

26. A method of processing a packet through a communication interface, the method comprising:

receiving a first packet from a network, wherein the first packet is part of a communication flow between a source entity and a destination entity;

determining whether a header portion of the first packet conforms to one of a set of communication protocols;

assembling a flow identifier to identify the communication flow, wherein said flow identifier comprises a source entity identifier and a destination entity identifier;

updating a flow database configured to facilitate management of communication flows through the communication interface, wherein said updating comprises:

configuring a flow activity indicator associated with the communication flow to reflect receipt of the first packet; and

configuring a flow validity indicator associated with the communication flow to indicate that the communication flow is valid;

assigning an operation code to the first packet, said operation code indicating whether a portion of data in the first packet is reassembleable with another portion of data in another packet in the communication flow; and

determining whether a second packet received at the communication interface is part of the communication flow.

27. The method of claim **3**, further comprising:

storing a first indicator in the host computer if said first communication flow comprises said second packet; and

storing a second indicator in the host computer if said first packet is the only packet stored in the communication device that is part of said communication flow.

* * * * *

IW 7696177

THE UNITED STATES OF AMERICA

TO ALL TO WHOM THESE PRESENTS SHALL COME:

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office

October 17, 2018

THIS IS TO CERTIFY THAT ANNEXED IS A TRUE COPY FROM THE RECORDS OF THIS OFFICE OF THE FILE WRAPPER AND CONTENTS OF:

APPLICATION NUMBER: *09/608,237*
FILING DATE: *June 30, 2000*
PATENT NUMBER: *6,651,099*
ISSUE DATE: *November 18, 2003*

By Authority of the
Under Secretary of Commerce for Intellectual Property
and Director of the United States Patent and Trademark Office

P. R. GRANT
Certifying Officer

PART 2 OF (2) PART(S)

US006570875B1

(12) **United States Patent**
Hegde

(10) **Patent No.: US 6,570,875 B1**
(45) **Date of Patent: May 27, 2003**

(54) **AUTOMATIC FILTERING AND CREATION OF VIRTUAL LANS AMONG A PLURALITY OF SWITCH PORTS**

(75) Inventor: **Gopal D. Hegde**, San Jose, CA (US)

(73) Assignee: **Intel Corporation**, Santa Clara, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/172,723**

(22) Filed: **Oct. 13, 1998**

(51) **Int. Cl.**[7] **H04L 12/28**; H04L 12/56

(52) **U.S. Cl.** **370/389**; 370/392; 370/395.53; 370/395.32

(58) **Field of Search** 370/389, 352, 370/353, 354, 356, 360, 390, 392, 396, 398, 395.3, 395.31, 395.42, 395.5, 395.53, 401, 413, 415, 417, 422, 428, 395.32, 432

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,715,250 A	2/1998	Watanabe	370/395
5,920,699 A	7/1999	Bare	395/200.55

(List continued on next page.)

OTHER PUBLICATIONS

Douglas E. Comer and David L. Stevens, *Adress Discovery and Binding (ARP)*, Internetworking with TCP/IP, vol. II: Design, Implementation, and Internals, Chapter 4, 1994, pp.39–59.

(List continued on next page.)

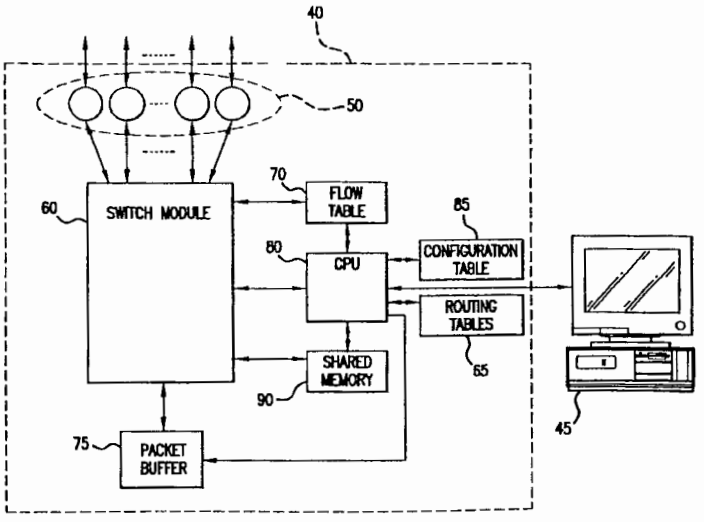
Primary Examiner—Douglas Olms
Assistant Examiner—Phirin Sam

(74) *Attorney, Agent, or Firm*—Pillsbury Winthrop LLP

(57) **ABSTRACT**

In a method and apparatus for performing multiprotocol switching and routing, incoming data packets are examined and the flow (i.e., source and destination) with which they are associated is determined. A flow table contains forwarding information that can be applied to all the packets belonging to the flow. If an entry is not present in the table for the particular flow, the packet is forwarded to the CPU to be processed. The CPU can then update the table with new forwarding information to be applied to all future packets of the same flow. When the forwarding information is already present in the table, packets can be forwarded at wire-speed. A dedicated ASIC is preferably employed to contain the table, as well as the engine for examining the packets and forwarding them according to the stored information. Decision-making tasks are thus more efficiently partitioned between the switch and the CPU so as to minimize processing overhead. Processes executing on the CPU maintain information regarding filters, mirrors, priorities, and VLANs. Such information is further integrated with the flow table forwarding information when flows corresponding to the established filters, mirrors, priorities and VLANs are detected. Accordingly, filters, mirrors, priorities and VLANs can be automatically implemented when forwarding decisions are made, which implementation is done at wire speeds. According to another aspect, VLANs are automatically created and updated based on the automatic detection of multicast groups existing among the hosts connected to the ports of the switch. After such VLANs are established, broadcast packets destined for the detected multicast groups are forwarded only along ports whose hosts are members thereof, thereby preventing needless and burdensome traffic from congesting other network segments and host connection.

14 Claims, 14 Drawing Sheets



U.S. PATENT DOCUMENTS

6,005,863	A *	12/1999	Deng et al.	370/392
6,047,325	A	4/2000	Jain et al.	709/227
6,091,725	A *	7/2000	Cheriton et al.	370/392
6,094,435	A *	7/2000	Hoffman et al.	370/414
6,128,298	A	10/2000	Wootton et al.	370/392
6,216,167	B1 *	4/2001	Momirov	709/238
6,243,758	B1 *	6/2001	Okanoue	709/238
6,246,680	B1 *	6/2001	Muller et al.	370/389
6,256,306	B1 *	7/2001	Bellenger	370/389
6,272,134	B1 *	8/2001	Bass et al.	370/390
6,331,983	B1 *	12/2001	Haggerty et al.	370/400
6,335,935	B2 *	1/2002	Kadambi et al.	370/396

OTHER PUBLICATIONS

Douglas E. Comer and David L. Stevens, *RIP: Active Route Propagation and Passive Acquisition*, Internetworking with TCP/IP, vol. II: Design, Implementation, and Internals, Chapter 18, 1994, pp. 355–379.

Keith Turner, *Is It a Switch or Is It a Router*, PC Magazine, Nov. 18, 1997.

* cited by examiner

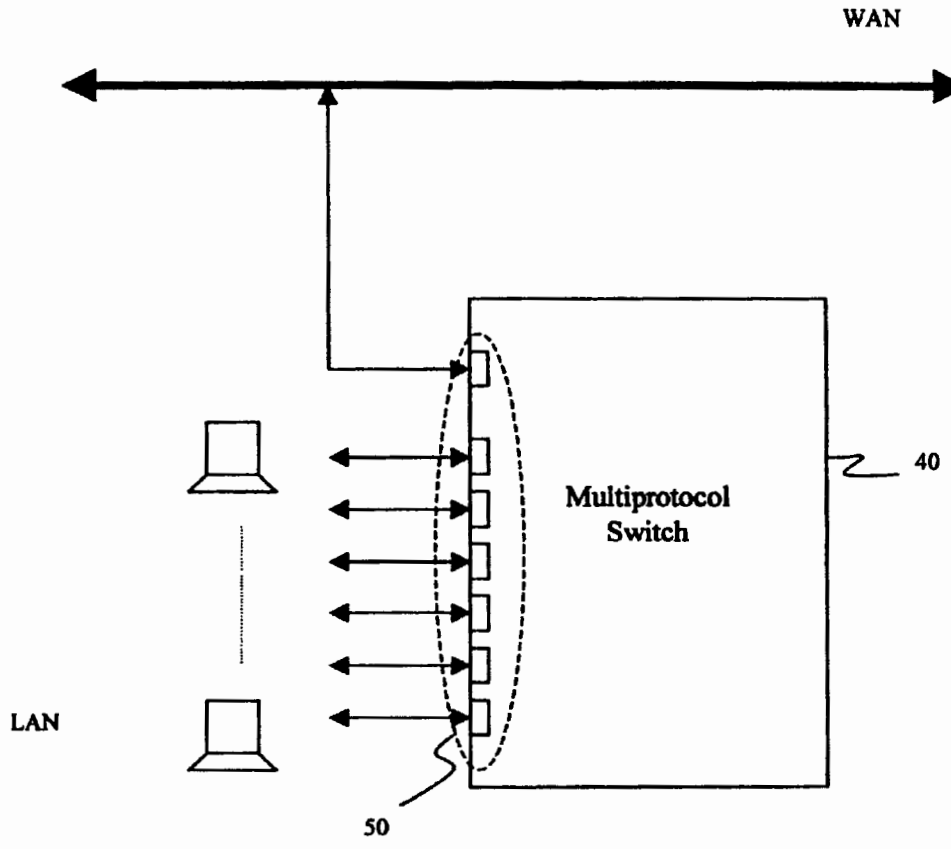


FIG. 1

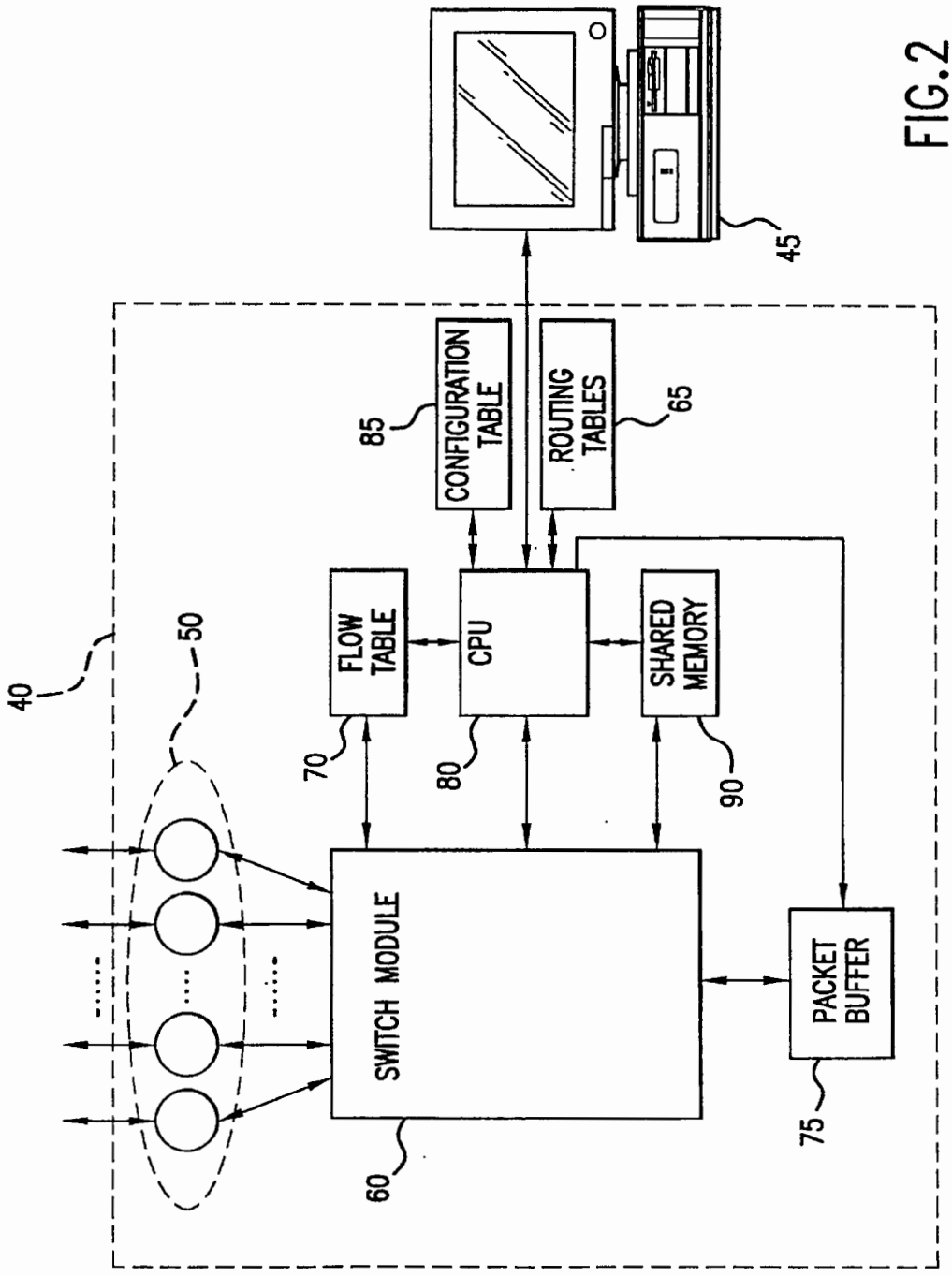

45
CONFIGURATION TABLE
85
ROUTING TABLES
65
FLOW TABLE
70
CPU
80
SHARED MEMORY
90
40
50
SWITCH MODULE
60
PACKET BUFFER
75

FIG.2

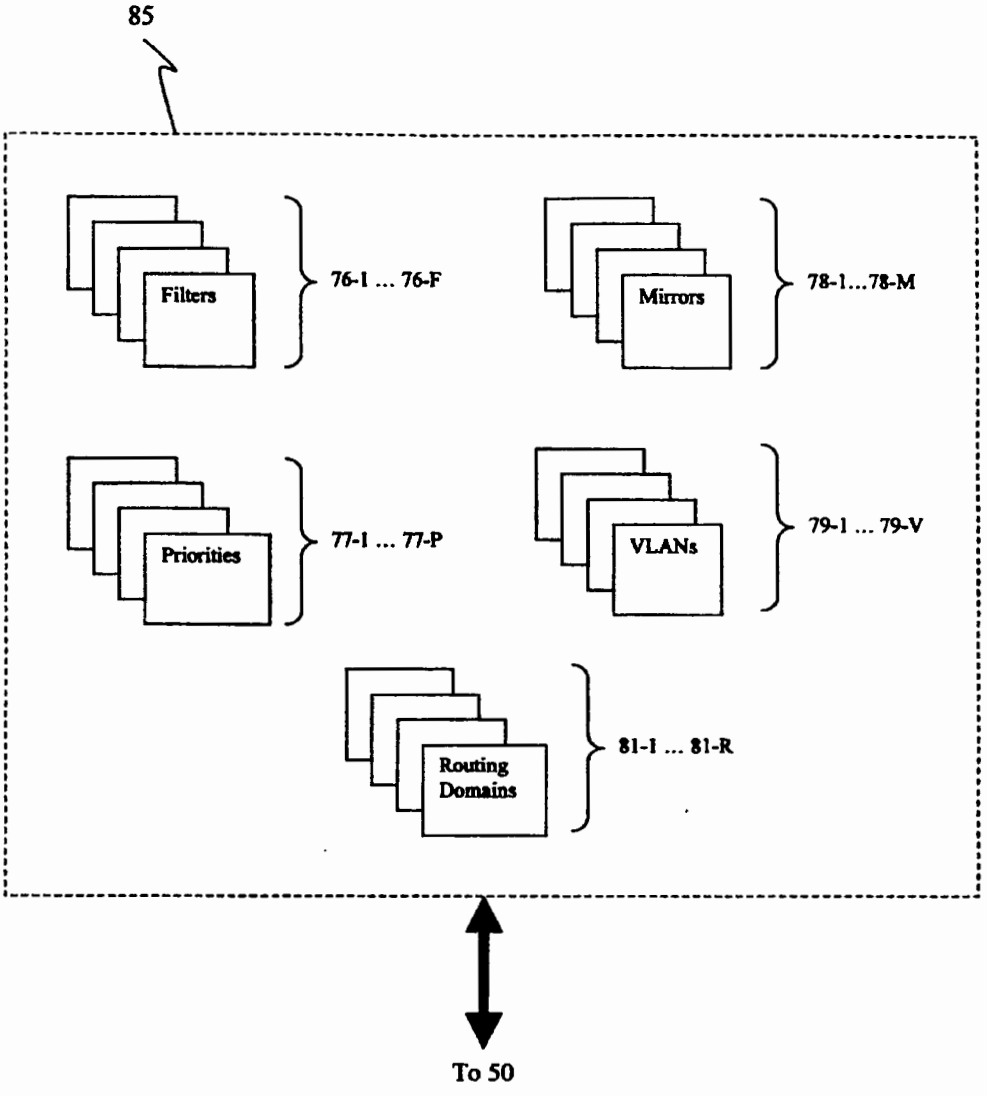

85
Filters
76-1 ... 76-F
Mirrors
78-1...78-M
Priorities
77-1 ... 77-P
VLANs
79-1 ... 79-V
Routing Domains
81-1 ... 81-R
To 50

FIG. 3

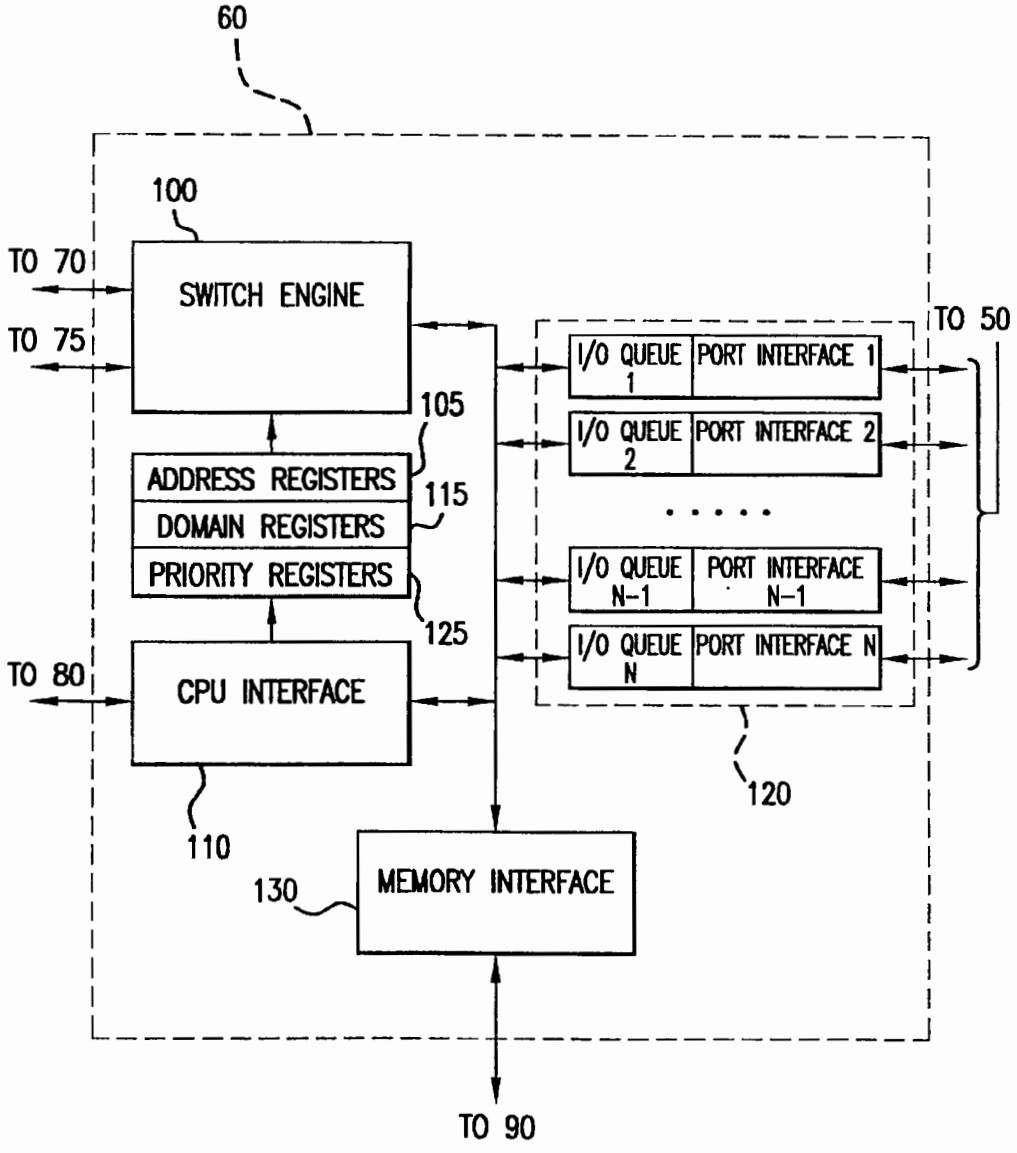

60
100
TO 70
TO 75
SWITCH ENGINE
105
ADDRESS REGISTERS
DOMAIN REGISTERS
PRIORITY REGISTERS
115
125
TO 80
CPU INTERFACE
110
130
MEMORY INTERFACE
TO 90
I/O QUEUE 1
PORT INTERFACE 1
I/O QUEUE 2
PORT INTERFACE 2
I/O QUEUE N-1
PORT INTERFACE N-1
I/O QUEUE N
PORT INTERFACE N
120
TO 50

FIG.4

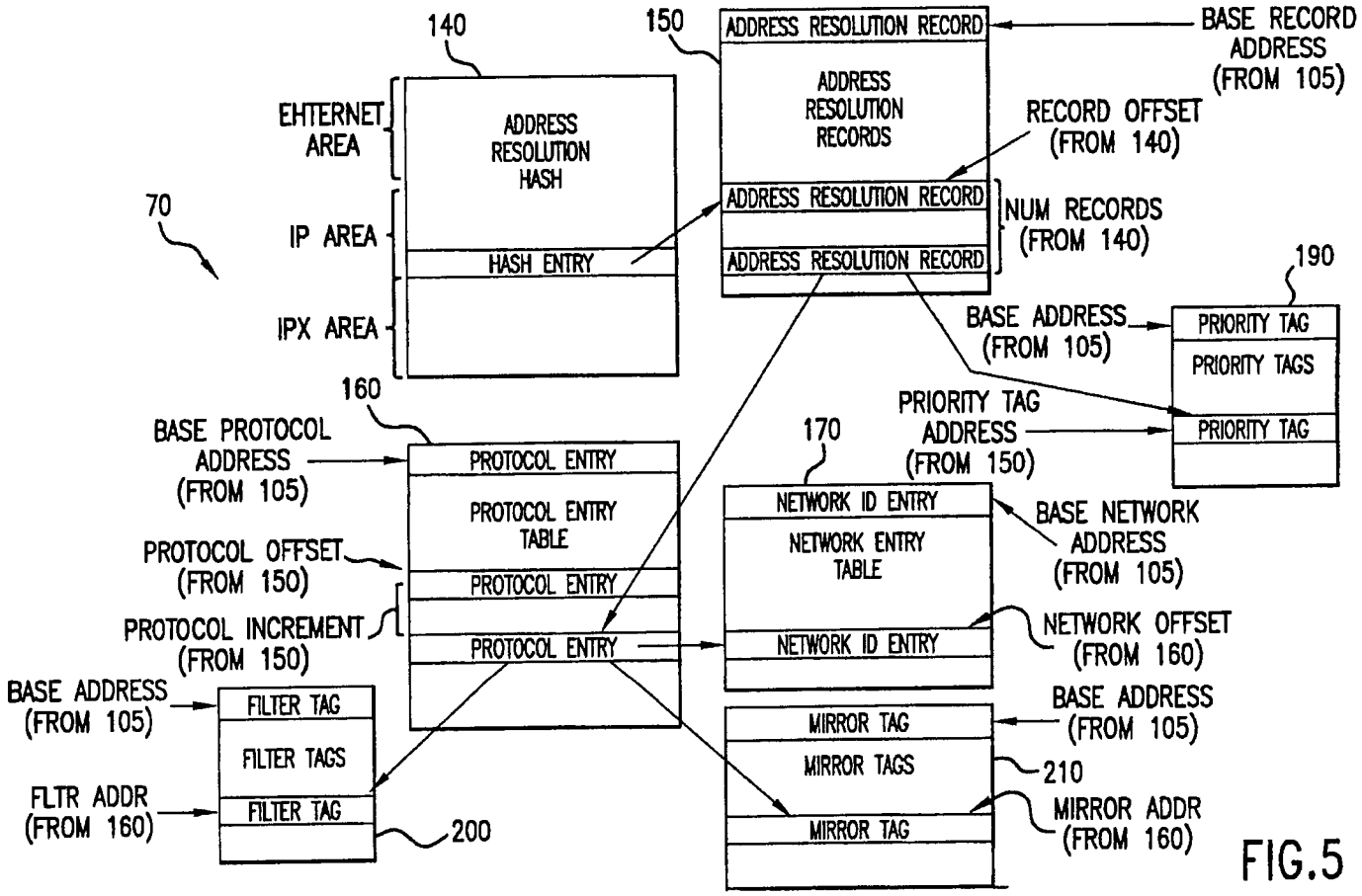

140
150
ADDRESS RESOLUTION RECORD
BASE RECORD ADDRESS (FROM 105)
EHTERNET AREA
ADDRESS RESOLUTION HASH
ADDRESS RESOLUTION RECORDS
RECORD OFFSET (FROM 140)
70
IP AREA
HASH ENTRY
ADDRESS RESOLUTION RECORD
ADDRESS RESOLUTION RECORD
NUM RECORDS (FROM 140)
IPX AREA
190
BASE ADDRESS (FROM 105)
PRIORITY TAG
PRIORITY TAGS
PRIORITY TAG
160
PRIORITY TAG ADDRESS (FROM 150)
170
BASE PROTOCOL ADDRESS (FROM 105)
PROTOCOL ENTRY
PROTOCOL ENTRY TABLE
NETWORK ID ENTRY
NETWORK ENTRY TABLE
BASE NETWORK ADDRESS (FROM 105)
PROTOCOL OFFSET (FROM 150)
PROTOCOL ENTRY
PROTOCOL INCREMENT (FROM 150)
PROTOCOL ENTRY
NETWORK ID ENTRY
NETWORK OFFSET (FROM 160)
BASE ADDRESS (FROM 105)
FILTER TAG
FILTER TAGS
MIRROR TAG
MIRROR TAGS
210
FLTR ADDR (FROM 160)
FILTER TAG
MIRROR TAG
MIRROR ADDR (FROM 160)
200

FIG.5

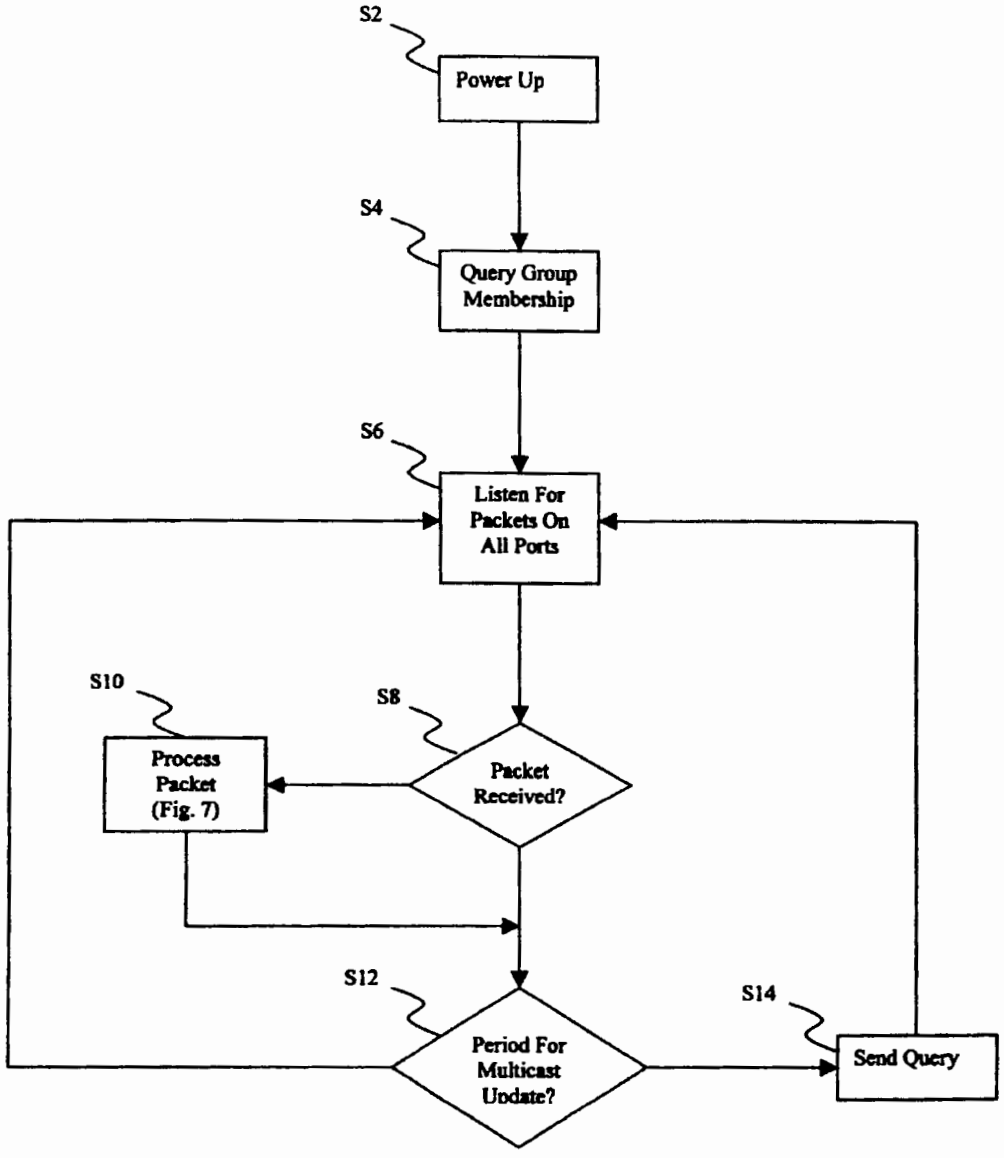

S2
Power Up
S4
Query Group Membership
S6
Listen For Packets On All Ports
S10
Process Packet (Fig. 7)
S8
Packet Received?
S12
Period For Multicast Update?
S14
Send Query

FIG. 6

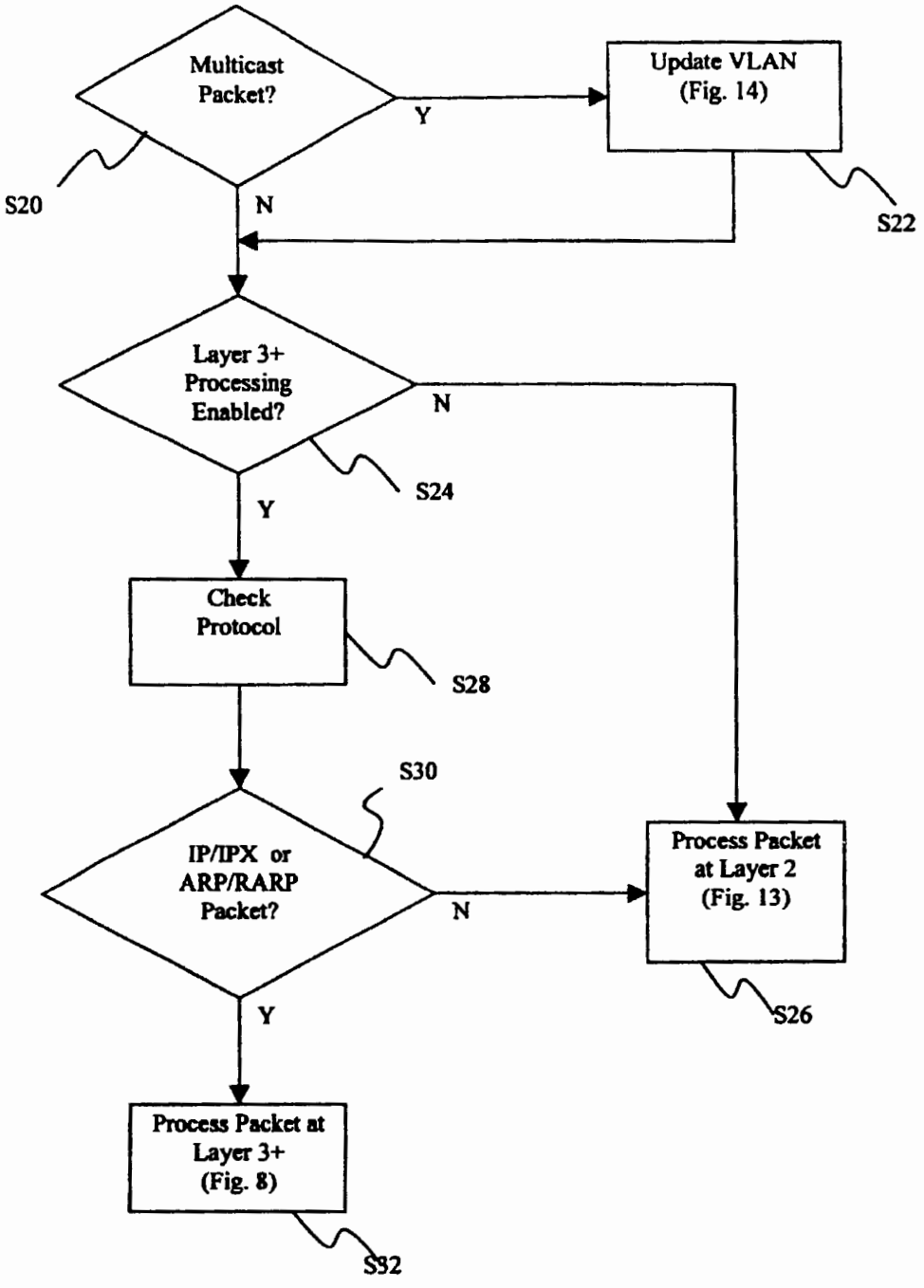

Multicast Packet?
Y
Update VLAN (Fig. 14)
S20
N
S22
Layer 3+ Processing Enabled?
N
S24
Y
Check Protocol
S28
S30
IP/IPX or ARP/RARP Packet?
N
Process Packet at Layer 2 (Fig. 13)
S26
Y
Process Packet at Layer 3+ (Fig. 8)
S32

FIG. 7

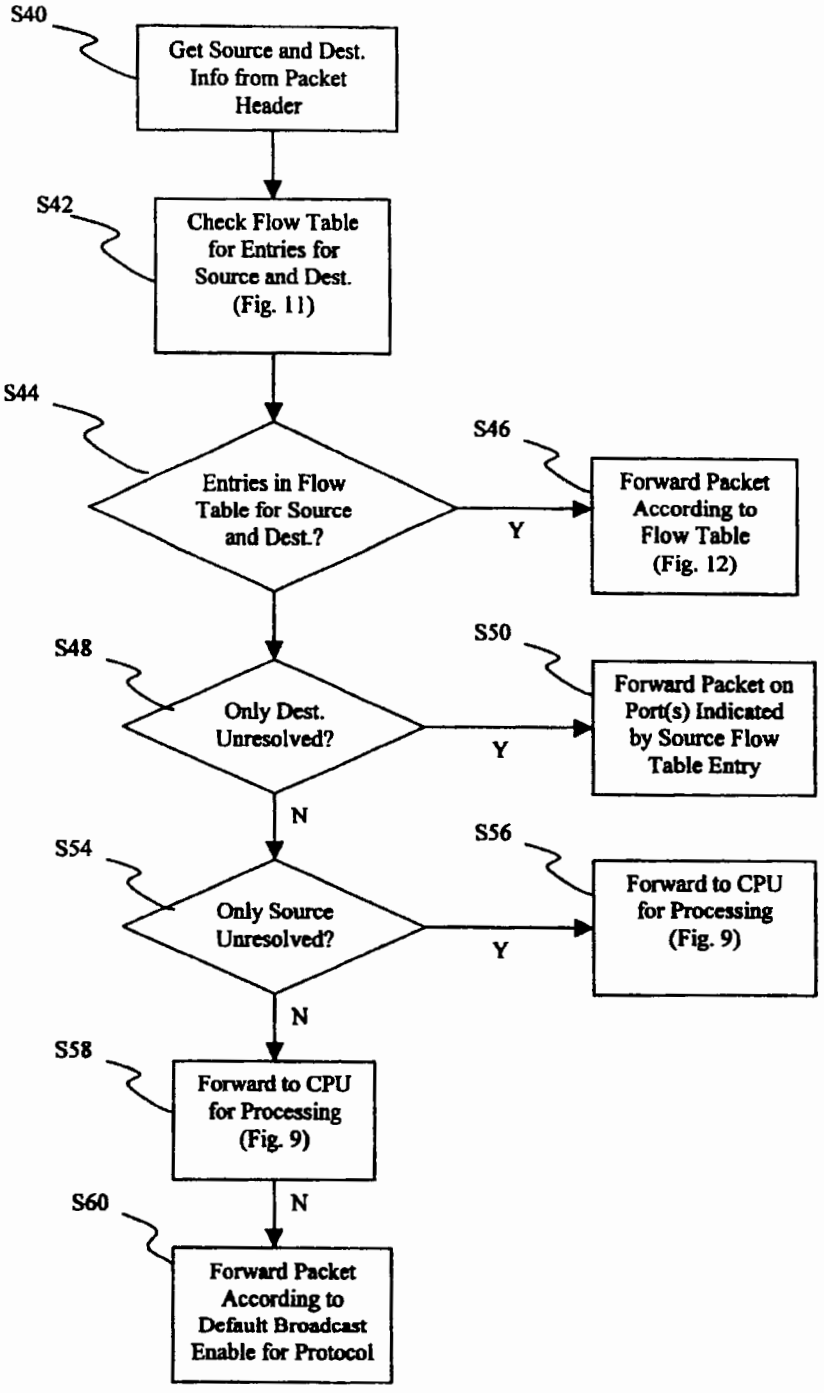

S40
Get Source and Dest. Info from Packet Header
S42
Check Flow Table for Entries for Source and Dest. (Fig. 11)
S44
Entries in Flow Table for Source and Dest.?
S46
Y
Forward Packet According to Flow Table (Fig. 12)
S48
Only Dest. Unresolved?
S50
Y
Forward Packet on Port(s) Indicated by Source Flow Table Entry
N
S54
Only Source Unresolved?
S56
Y
Forward to CPU for Processing (Fig. 9)
N
S58
Forward to CPU for Processing (Fig. 9)
N
S60
Forward Packet According to Default Broadcast Enable for Protocol

FIG. 8

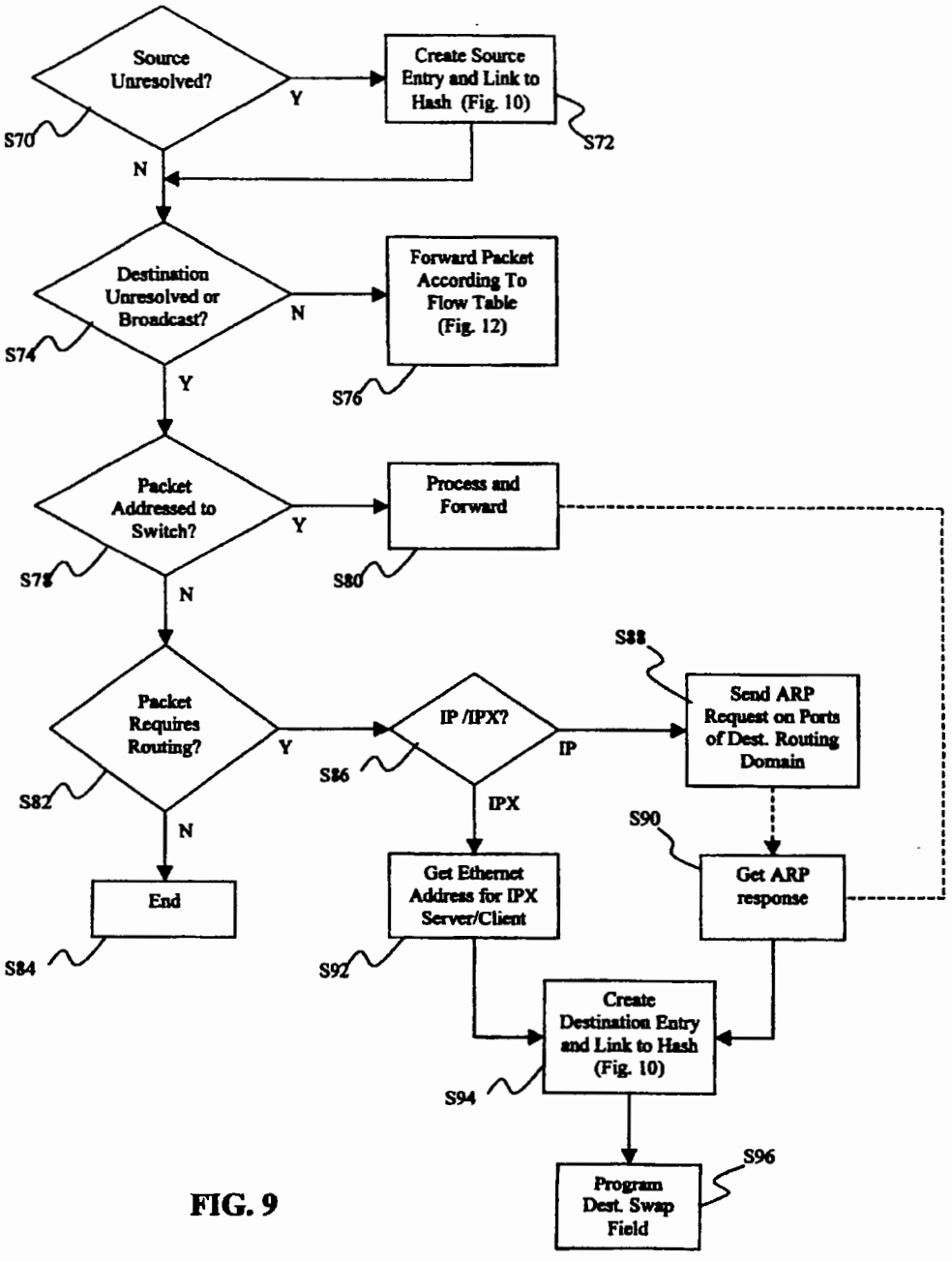

Source Unresolved?
Y
Create Source Entry and Link to Hash (Fig. 10)
S70
S72
N
Destination Unresolved or Broadcast?
N
Forward Packet According To Flow Table (Fig. 12)
S74
S76
Y
Packet Addressed to Switch?
Y
Process and Forward
S78
S80
N
S88
Packet Requires Routing?
Y
IP /IPX?
IP
Send ARP Request on Ports of Dest. Routing Domain
S82
S86
IPX
S90
N
End
Get Ethernet Address for IPX Server/Client
Get ARP response
S84
S92
Create Destination Entry and Link to Hash (Fig. 10)
S94
S96
Program Dest. Swap Field

FIG. 9

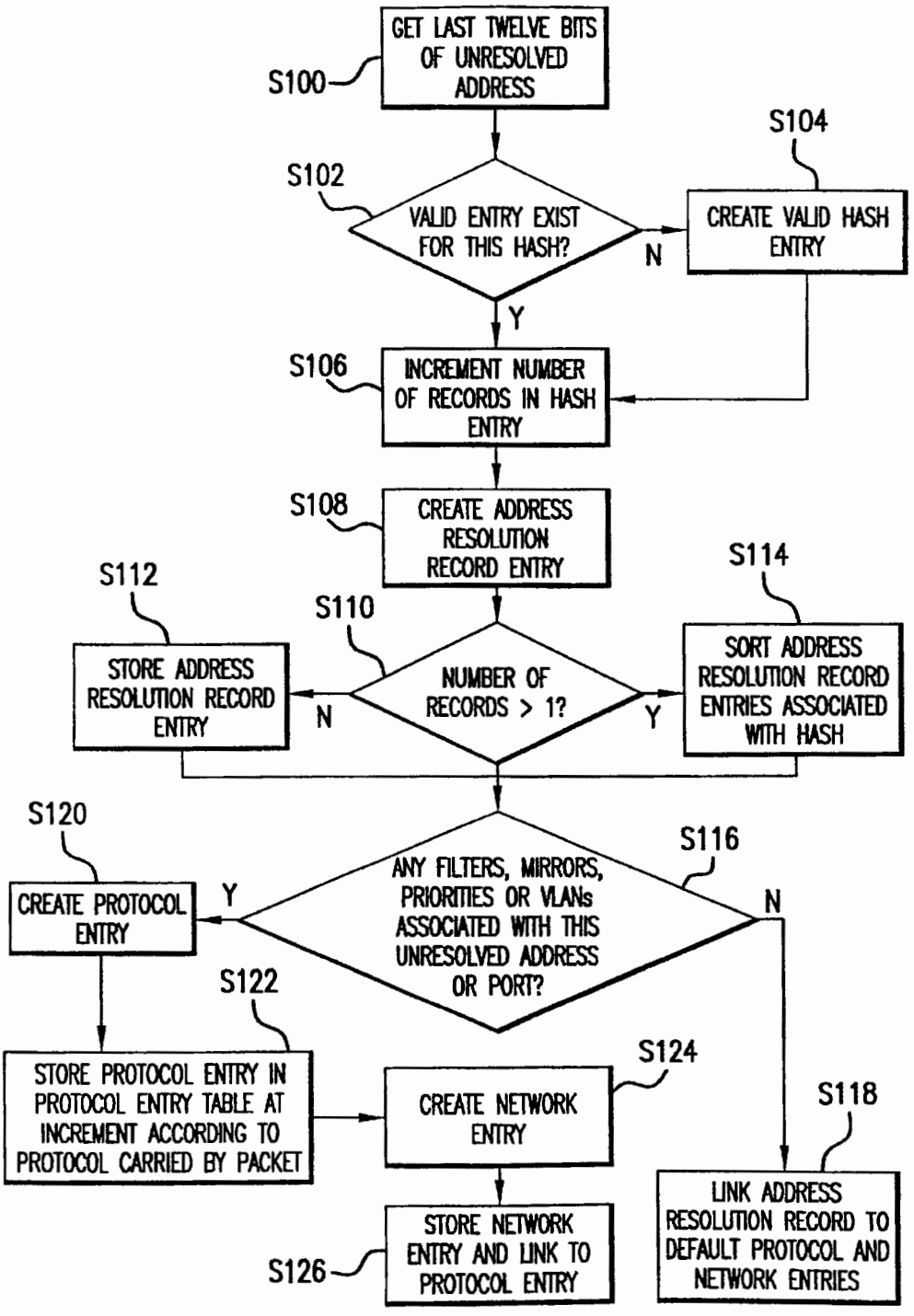

GET LAST TWELVE BITS OF UNRESOLVED ADDRESS
S100
VALID ENTRY EXIST FOR THIS HASH?
S102
N
Y
CREATE VALID HASH ENTRY
S104
INCREMENT NUMBER OF RECORDS IN HASH ENTRY
S106
CREATE ADDRESS RESOLUTION RECORD ENTRY
S108
NUMBER OF RECORDS > 1?
S110
N
Y
STORE ADDRESS RESOLUTION RECORD ENTRY
S112
SORT ADDRESS RESOLUTION RECORD ENTRIES ASSOCIATED WITH HASH
S114
ANY FILTERS, MIRRORS, PRIORITIES OR VLANs ASSOCIATED WITH THIS UNRESOLVED ADDRESS OR PORT?
S116
Y
N
CREATE PROTOCOL ENTRY
S120
STORE PROTOCOL ENTRY IN PROTOCOL ENTRY TABLE AT INCREMENT ACCORDING TO PROTOCOL CARRIED BY PACKET
S122
CREATE NETWORK ENTRY
S124
STORE NETWORK ENTRY AND LINK TO PROTOCOL ENTRY
S126
LINK ADDRESS RESOLUTION RECORD TO DEFAULT PROTOCOL AND NETWORK ENTRIES
S118

FIG.10

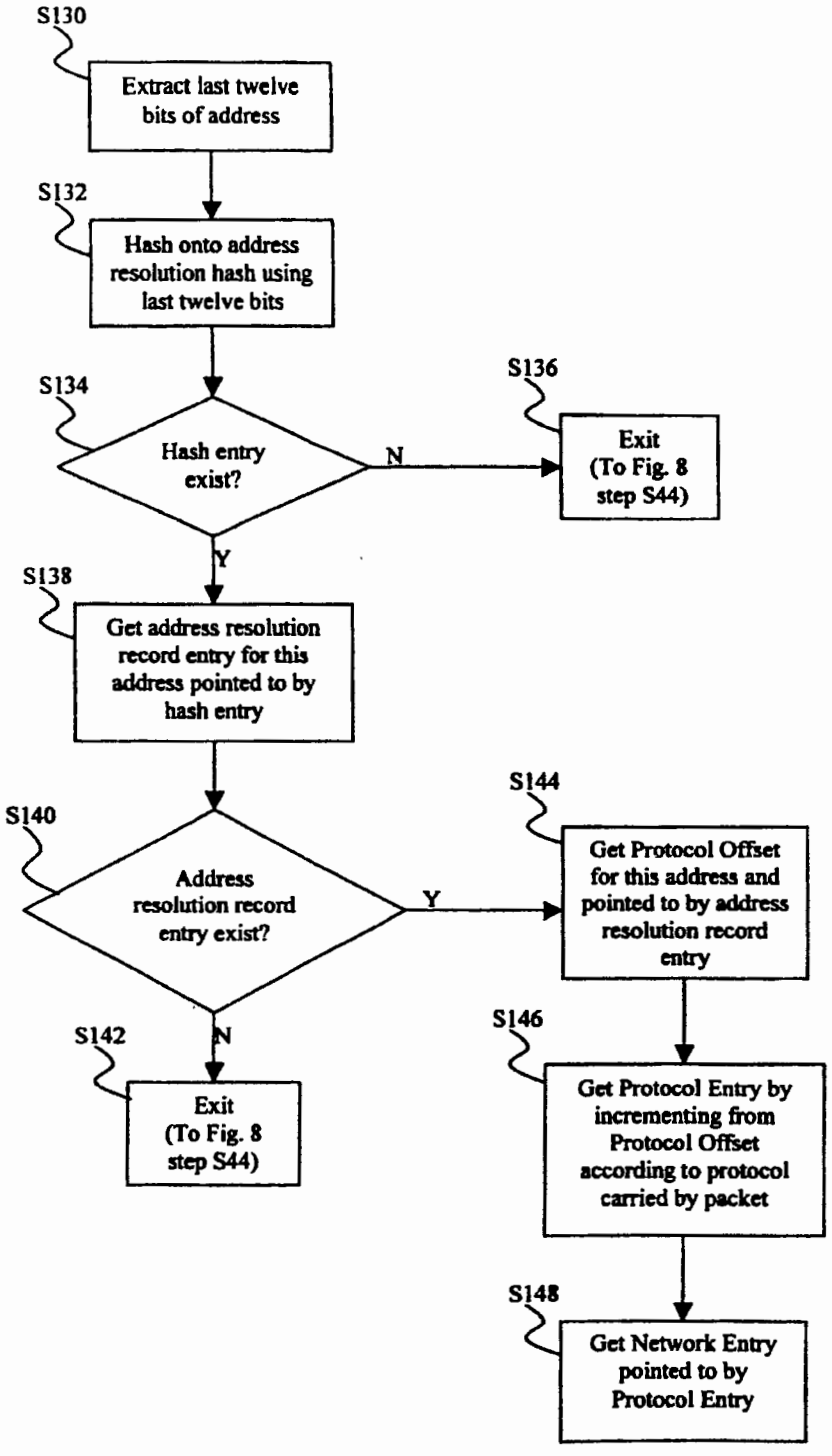
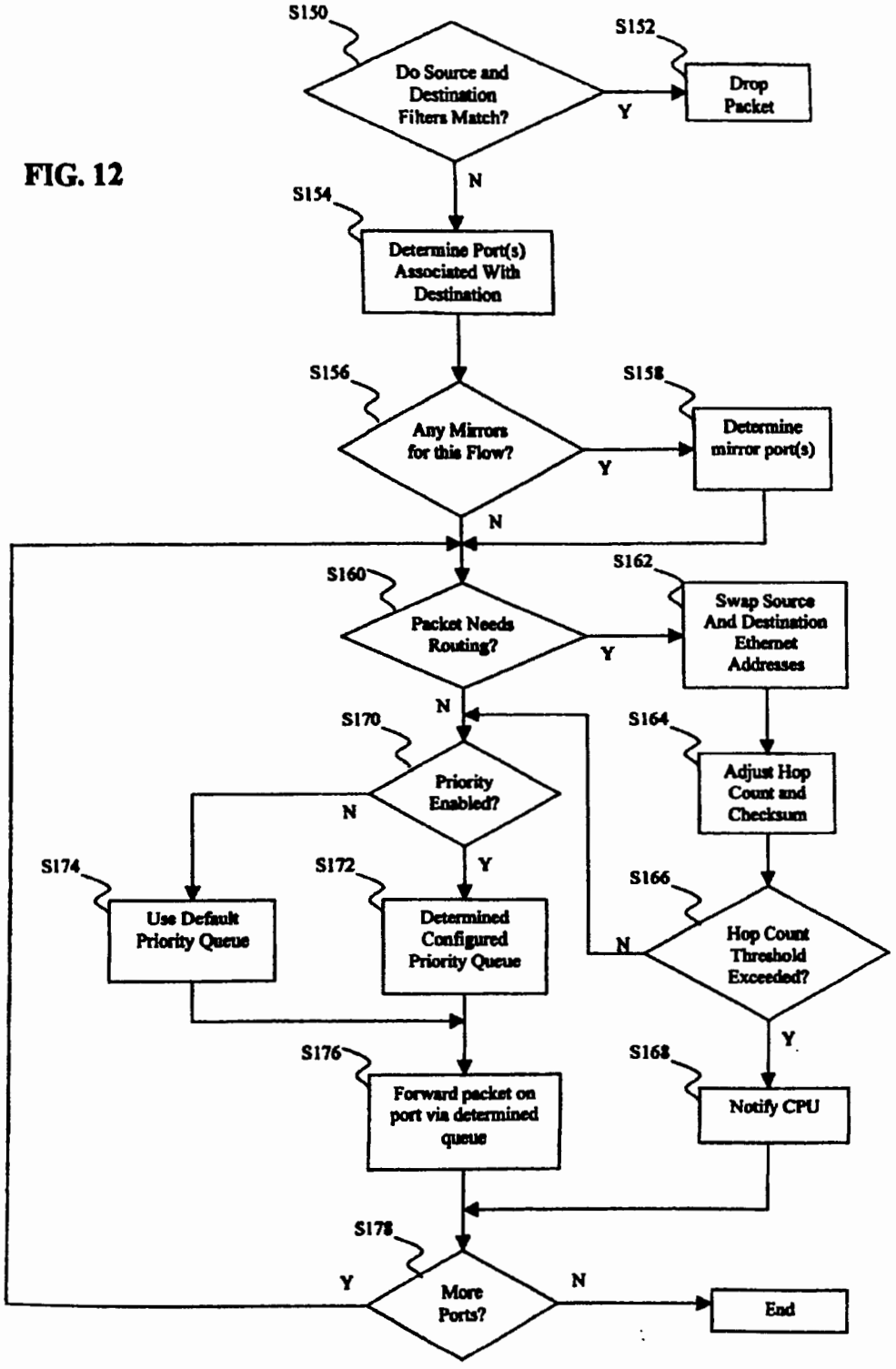

S130
Extract last twelve bits of address
S132
Hash onto address resolution hash using last twelve bits
S134
Hash entry exist?
N
S136
Exit (To Fig. 8 step S44)
Y
S138
Get address resolution record entry for this address pointed to by hash entry
S140
Address resolution record entry exist?
Y
S144
Get Protocol Offset for this address and pointed to by address resolution record entry
N
S142
Exit (To Fig. 8 step S44)
S146
Get Protocol Entry by incrementing from Protocol Offset according to protocol carried by packet
S148
Get Network Entry pointed to by Protocol Entry

FIG. 11

FIG. 12


S150
Do Source and Destination Filters Match?
S152
Drop Packet
Y
N
S154
Determine Port(s) Associated With Destination
S156
Any Mirrors for this Flow?
S158
Determine mirror port(s)
Y
N
S160
Packet Needs Routing?
S162
Swap Source And Destination Ethernet Addresses
Y
N
S170
Priority Enabled?
S164
Adjust Hop Count and Checksum
N
Y
S174
Use Default Priority Queue
S172
Determined Configured Priority Queue
S166
Hop Count Threshold Exceeded?
N
Y
S176
Forward packet on port via determined queue
S168
Notify CPU
S178
More Ports?
Y
N
End

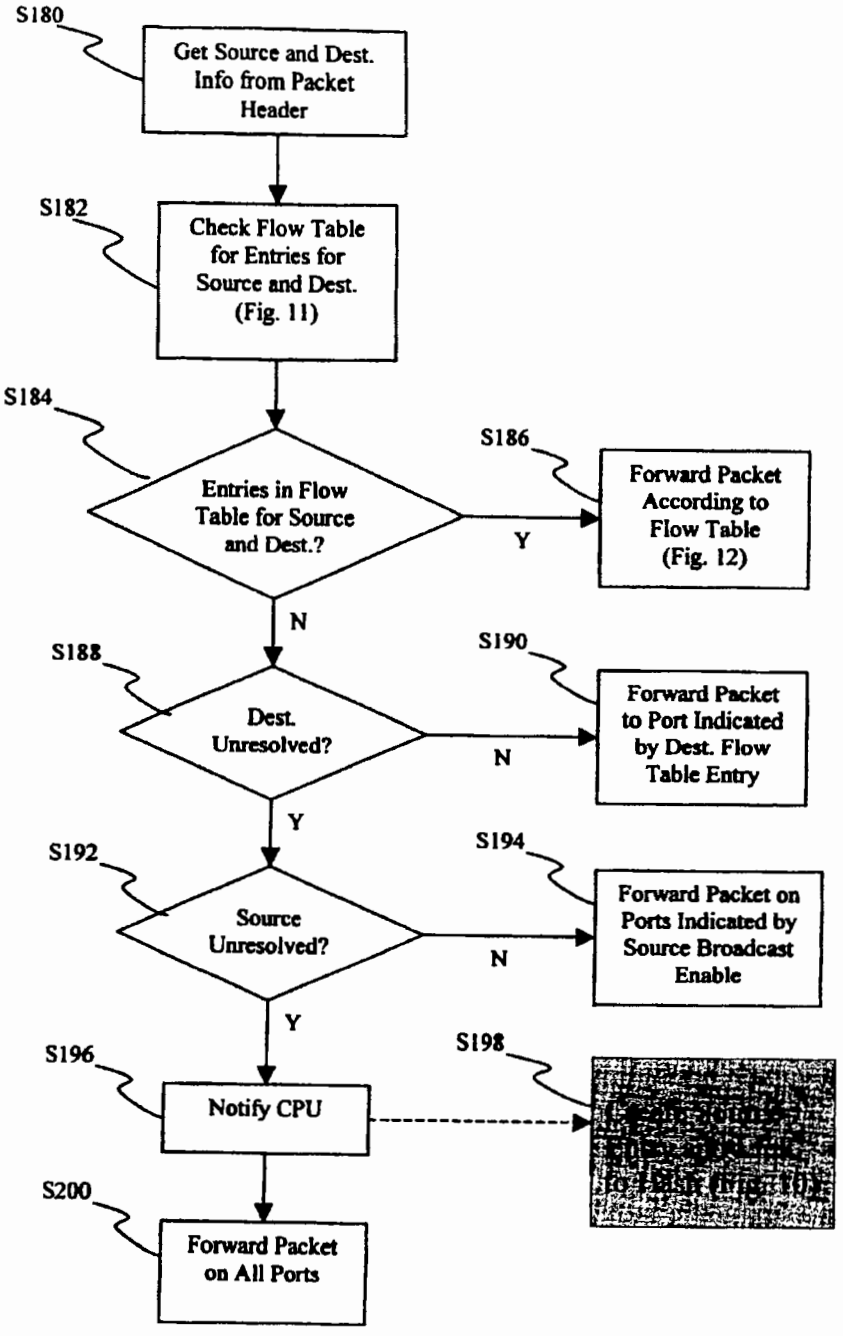

S180
Get Source and Dest. Info from Packet Header
S182
Check Flow Table for Entries for Source and Dest. (Fig. 11)
S184
Entries in Flow Table for Source and Dest.?
Y
S186
Forward Packet According to Flow Table (Fig. 12)
N
S188
Dest. Unresolved?
N
S190
Forward Packet to Port Indicated by Dest. Flow Table Entry
Y
S192
Source Unresolved?
N
S194
Forward Packet on Ports Indicated by Source Broadcast Enable
Y
S196
Notify CPU
S198
S200
Forward Packet on All Ports

FIG. 13

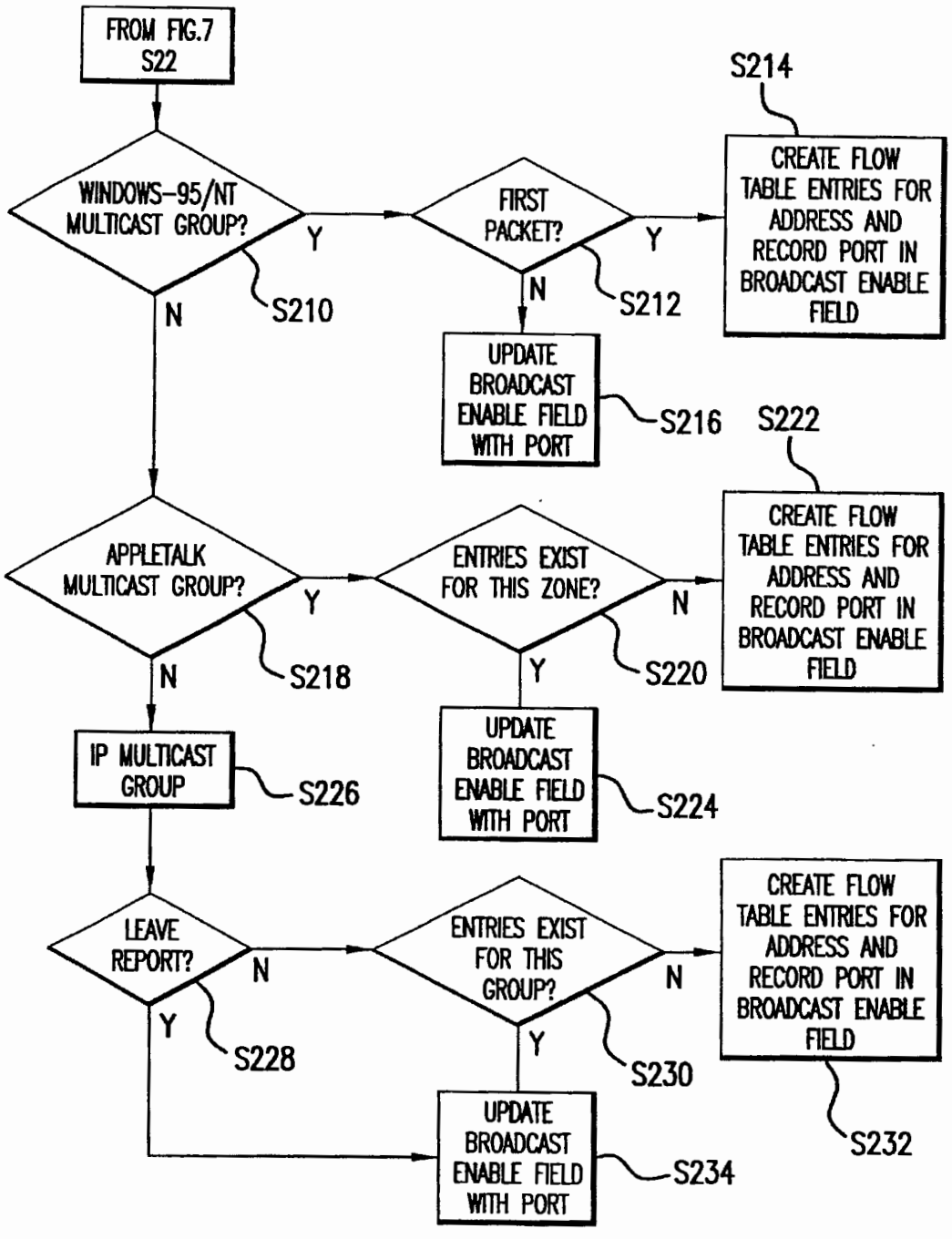

FROM FIG.7 S22
WINDOWS-95/NT MULTICAST GROUP?
Y
N
S210
FIRST PACKET?
Y
N
S212
S214
CREATE FLOW TABLE ENTRIES FOR ADDRESS AND RECORD PORT IN BROADCAST ENABLE FIELD
UPDATE BROADCAST ENABLE FIELD WITH PORT
S216
APPLETALK MULTICAST GROUP?
Y
N
S218
ENTRIES EXIST FOR THIS ZONE?
N
Y
S220
S222
CREATE FLOW TABLE ENTRIES FOR ADDRESS AND RECORD PORT IN BROADCAST ENABLE FIELD
UPDATE BROADCAST ENABLE FIELD WITH PORT
S224
IP MULTICAST GROUP
S226
LEAVE REPORT?
N
Y
S228
ENTRIES EXIST FOR THIS GROUP?
N
Y
S230
CREATE FLOW TABLE ENTRIES FOR ADDRESS AND RECORD PORT IN BROADCAST ENABLE FIELD
S232
UPDATE BROADCAST ENABLE FIELD WITH PORT
S234

FIG. 14

AUTOMATIC FILTERING AND CREATION OF VIRTUAL LANS AMONG A PLURALITY OF SWITCH PORTS

RELATED APPLICATION

This application is related to co-pending U.S. application Ser. No. 09/058,335, filed Apr. 10, 1998, and entitled, "Method And Apparatus For Multiprotocol Switching And Routing," commonly owned by the assignee of the present application, the contents of which are incorporated herein by reference.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to packet switches and routers, and more particularly, to a switching and routing method and apparatus capable of automatically filtering flows of packets between switch ports allowing for creation of a high performance hardware assisted firewall for Intranet applications and automatically creating virtual LANs among switch ports. In addition, the present invention describes a mechanism to reserve bandwidth for end to end applications and provide guaranteed quality of service (QoS) for them.

2. Description of the Related Art

Packet switches and routers forward data packets between nodes in a network. However, securing machines and data from unauthorized access is fast becoming a very important issue for corporate networks. According to industry experts, more than 70% of breaking are internal (i.e. employees stealing sensitive information from their own company). Also HR department in a company would not want engineers to get access to payroll data. This has created a need for a high performance firewall to secure and separate different networks. In conventional routers, this is done by software which inspects every packet that is being routed and determines whether any filters have been configured for that session. This information is typically manually configured by a system administrator. However, the processing required to inspect packets and apply the appropriate filter significantly reduces the packets rate through the router. The rate further reduces if a large number of filters have been configured.

Multimedia networking (voice and video on LAN/WAN) requires Quality of Service guarantees. Protocols such as Resource Reservation Protocol (RSVP), Real Time Protocol (RTP), Real Time Control Protocol (RTCP) have been defined to provide these services on LANs/WANs. Underlying hardware however needs to support prioritization of traffic and bandwidth reservation for these protocols to operate. Network traffic contains normal and high priority data. A good switch should be able to prioritize traffic in such a way that while high priority traffic gets its share of bandwidth, low priority traffic does not starve completely. This is called Weighted Fair Queuing (WFQ). This invention describes mechanisms to provide these services in hardware.

Likewise, virtual LANs (VLANs) are often desired for controlling broadcast and multicast packet flows in computer networks. Broadcast and multicast packets are typically forwarded on all ports of a switch and each node connected to the switch will have to process such packets. Some switches allow system administrators to manually set up VLANs among groups of nodes such that broadcasts and multicasts from nodes belonging to one group are confined to that group. This reduces the number of packets that nodes on the switched network must process. However, much

administrative overhead is required to create and maintain VLAN groups, and to assign and update memberships in the groups.

Accordingly, there remains a need in the art for a switching device that can support prioritization and QoS guarantees of network traffic and/or create VLANs automatically without any administrator intervention. The present invention fulfills this need.

SUMMARY OF THE INVENTION

An object of the invention is to provide a method and apparatus that can forward packets to their destination at high throughput rates without requiring substantial processing overhead.

Another object of the invention is to provide a method and apparatus that can both switch and route packets with the same minimal processing overhead.

Another object of the invention is to provide a method and apparatus that is capable of both switching and routing packets at wire speed.

Another object of the invention is to provide a method and apparatus that is capable of wire-speed switching and routing of packets that are associated with all possible Layer 2 and Layer 3 traffic protocols.

Another object of the invention is to provide a method and apparatus that provides wire-speed switching and routing functionality in a switched internetwork, but does not require reconfiguration of existing end stations or network infrastructure.

Another object of the invention is to provide a method and apparatus that provides wire-speed application of filters of flows between nodes in a switched internetwork.

Another object of the invention is to provide a method and apparatus that provides wire-speed application of mirrors of flows between nodes in a switched internetwork.

Another object of the invention is to provide a method and apparatus that provides wire-speed application of priorities for flows between nodes in a switched internetwork.

Another object of the invention is to provide a method and apparatus that enhances network security.

Another object of the invention is to provide a method an apparatus that reduces unnecessary network traffic.

Another object of the invention is to provide a method and apparatus that provides wire-speed switch and routing functionality while supporting application or network level filters for intranet security applications.

Another object of the invention is to provide a method and apparatus that provides wire-speed switch and routing functionality while supporting VLANs that are created automatically with no administrator intervention.

Another object of the invention is to provide a method and apparatus for wire speed switching and routing functionality while supporting bandwidth reservation.

Another object of the invention is to provide a method and apparatus for wire speed switching and routing functionality while supporting multilevel priority queueing.

Another object of the invention is to provide a method and apparatus for wire speed switching and routing functionality while supporting weighted fair queueing.

The present invention fulfills these objects, among others, by providing a method and apparatus for performing multiprotocol switching and routing. Incoming data packets are examined and the flow (i.e., source and destination) with which they are associated is determined. A flow table

contains forwarding information that can be applied to the flow. If an entry is not present in the table for the particular flow, the packet is forwarded to the CPU to be processed. The CPU can then update the table with new forwarding information to be applied to all future packets of the same flow. When the forwarding information is already present in the table, packets can thus be forwarded at wire-speed. A high speed static memory is preferably used to contain the table. A dedicated ASIC is preferably used to implement the engine for examining individual packets and forwarding them according to the stored information. Decision-making tasks are thus more efficiently partitioned between the switch and the CPU so as to minimize processing overhead.

Information regarding filters, priorities, and VLANs is maintained by processes executing on the CPU and are programmed into the forwarding table for the hardware to apply when it detects a matching flow.

According to another aspect of the invention, Internet Group Management Protocol (IGMP) packets (for IP multicast control), Zone Information Protocol (ZIP) packets (for AppleTalk) and NetBios & DLC/LLC packets with multicast addresses are forwarded to the CPU by the hardware. The CPU can then create and update VLANs automatically for those multicast groups in the forwarding table with no administrator intervention. Once such VLANs are established, packets destined for the detected multicast groups are forwarded only on the ports whose hosts are members thereof, preventing needless and burdensome traffic from congesting other network segments and host connections.

A further aspect of the invention provides mechanisms for administrators to reserve bandwidths and assign priorities to traffic flows. Protocols such as RSVP can then be used to automatically reserve bandwidth for certain flows. This provides Quality of Service guarantees for traffic being switched.

BRIEF DESCRIPTION OF THE DRAWINGS

These and other objects and advantages of the present invention will become apparent to those skilled in the art after considering the following detailed specification, together with the accompanying drawings wherein:

FIG. **1** is a block diagram illustrating a packet switching architecture in accordance with the present invention;

FIG. **2** is a block diagram illustrating a multiprotocol switch of the present invention in an architecture such as that illustrated in FIG. **1**;

FIG. **3** is a block diagram illustrating a configuration table of the present invention in a multiprotocol switch such as that illustrated in FIG. **2**;

FIG. **4** is a block diagram illustrating a switch module of the present invention in a multiprotocol switch such as that illustrated in FIG. **2**;

FIG. **5** is a block diagram illustrating a flow table of the present invention in a multiprotocol switch such as that illustrated in FIG. **2**;

FIG. **6** is a flowchart illustrating a method used during operation of a multiprotocol switch according to the present invention;

FIG. **7** is a flowchart illustrating a method used to process data packets received in a multiprotocol switch according to the present invention;

FIG. **8** is a flowchart illustrating a method used to process data packets according to Layer 3+ protocols in a multiprotocol switch according to the present invention;

FIG. **9** is a flowchart illustrating a method used to process unresolved Layer 3+ data packets received in a multiprotocol switch according to the present invention;

FIG. **10** is a flowchart illustrating a method used to create flow processing entries in a multiprotocol switch according to the present invention;

FIG. **11** is a flowchart illustrating a method used to resolve flow processing information according to flow identification information contained in data packets processed in a multiprotocol switch according to the present invention;

FIG. **12** is a flowchart illustrating a method used to forward data packets according to flow processing information programmed for the particular flow with which the data packets are associated in a multiprotocol switch according to the present invention;

FIG. **13** is a flowchart illustrating a method used to process data packets according to Layer **2** protocols in a multiprotocol switch according to the present invention; and

FIG. **14** is a flowchart illustrating a method used to automatically configure and update VLAN information in a multiprotocol switch built according to the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

A device and method capable of performing wire-speed multiprotocol switching and routing of data packets between nodes in a network is described in the aforementioned related co-pending U.S. application Ser. No. 09/058,335. FIG. **1** is a block diagram illustrating a switch architecture in accordance with the present invention, which switch architecture is more fully described in the co-pending application. It includes a multiprotocol switch **40** having N input/output ports **50**-**1** . . . **50**-N. The input/output ports can be attached to nodes in a local area network (LAN) or they can be attached to different network segments or different networks in a wide area network (WAN) directly or via routers. As explained in more detail in the co-pending application, the multiprotocol switch has the ability to forward packets among and between local nodes and external networks attached to it at wire speeds, and in accordance with a plurality of Layer **2** and Layer **3** protocols.

FIG. **2** further illustrates a multiprotocol switch **40** in accordance with the principles of the invention. In addition to input/output ports **50**, it includes a switch module **60** and a flow table **70**. Switch module **60** further communicates with a packet buffer **75**, a CPU **80** and a shared memory **90**. Flow table **70** and shared memory **90** are mapped memory spaces that are accessible by both switch module **60** and CPU **80**. CPU **80** also communicates with a routing table **65**, a configuration table **85** and a system administrator **45**.

Although shown separately for clarity, switch module **60** and flow table **70** are preferably implemented together as an application specific integrated circuit (ASIC). Such an implementation permits data packets to be switched between ports **50** at wire speed in accordance with flows, filters and priorities specified in flow table **70**. However, other specific implementations of switch module **60** and flow table **70** in accordance with the invention will be apparent to those skilled in the art after being taught by the following disclosures of their logical functions and data structures, for example.

CPU **80** can be implemented by a MIPS microprocessor made by IDT Inc. of Santa Clara, Calif., and shared memory **90** can be implemented by a fast static RAM (SRAM) such

as that manufactured by ISSI. Packet buffer **75** for storing packets can be implemented using Synchronous DRAM (SDRAM) such as that manufactured by Samsung, Inc. CPU **80** partitions packet buffer **75** on a periport basis. The amount of memory allocated to each partition depends on port speed. So, for example, a gigabit port is allocated more memory than a 10/100 Mbps port.

Although not shown for clarity, it should be understood that CPU **80** includes program and data memory for storing programs that are executed by CPU **80** and data needed by those programs. Such data can include routing tables and the like. Programs executed by CPU **80** can include conventional routing update and costing functions implemented with known protocols such as Routing Information Protocol (RIP) for setting and maintaining conventional routing table information in routing tables **65**, as well as processes for setting and maintaining system configuration information for the network in configuration table **85** in accordance with commands by system administrator **45**, which system configuration information can include routing domains for example. Such conventional routing processes are in addition to the novel processes performed by the multiprotocol switch of the present invention that will be described in more detail below. However, a detailed description of such conventional processes will not be given so as not to obscure the invention.

Ports **50** are preferably RJ45 10/100 Mb ports, and can include port modules such as, for example, a 8×10/100 Mb port module (100 Base TX), a 1-Gigabit port module, or a 4×100 Base FX port module.

The term "routing domain" is used in this document to describe multiple ports (**50-1** . . . **50-N**) that belong to the same IP or IPX network. All the ports that belong to a routing domain have the same IP address and subnet mask or same IPX address. Each routing domain represents a virtual router port on the switch.

In the architecture shown in FIG. **2**, data packets arrive at ports **50-1** . . . **50-N**. As will be described in more detail below, switch module **60** continually monitors each of the ports for incoming traffic. When a data packet arrives, it checks the packet header for information that identifies the flow to which the packet belongs. For example, a flow of packets between two hosts in the network can be identified by the Ethernet and/or IP/IPX addresses of the hosts, and perhaps further by IP/IPX sockets and the protocol by which the hosts are communicating. This flow identification information is extracted from the header of each packet that traverses the network through the multiprotocol switch. IP/IPX data packets are buffered in packet buffer **75** while flow identification and forwarding processing is performed.

Software processes executing on CPU **80** handle interfacing with a system administrator **45** to retrieve, store and manage configuration information in configuration table **85**. The software processes and interfaces can be implemented in many ways known to those skilled in the art, and so they will not be described in detail here so as not to obscure the invention. However, some of the contents of configuration table **85** should be noted. In addition to conventional system configuration information such as routing domains, this table includes information relating to filters, priorities, bandwidth reservations for applications and VLANs established between ports and hosts of the network.

As further illustrated in FIG. **3**, in addition to routing domain settings **81-1** . . . **81-R**, sets of filters **76-1** . . . **76-F**, priorities **77-1** . . . **77-P**, and mirrors **78-1** . . . **78-M**, are maintained in configuration table **85**. Also maintained in configuration table **85** is a list of VLANs **79-1** . . . **79-V**, which list includes each established VLAN and the members thereof. Filters, priorities, mirrors and can be port-specific, host-specific, application-specific, or protocol-specific. That is, for example, a filter may be established between two ports of the switch (e.g. forbid any communication between ports A and B), between two hosts connected to ports of the switch (e.g. forbid any communication between host A having Ethernet address X, and host B having Ethernet address Y), between two applications running on hosts connected to ports of the switch (e.g. forbid any telnet sessions between hosts A and B), or between two hosts using a certain protocol (e.g. forbid ICMP communications between IP hosts A and B). When a priority level is assigned to a port, host, application or protocol, packets associated therewith are forwarded via a selected one of multiple priority queues, as will be described in more detail below. A mirror permits packets destined for one port, host or application to be duplicated and forwarded on one or more ports.

In addition to the VLANs automatically created and maintained by the present invention, as will be described in more detail below, the list of VLANs **79-1** . . . **79-V** allows system administrators to manually create and maintain VLANs, or to disable automatic creation of VLANs, by the switch.

Routing domains **81-1** . . . **81-R** contain the lists of routing domains established for the network and the members thereof. For example, a typical routing domain configuration for IP networks involves assigning ports to routing domains and specifying a separate IP address and subnet mask for each routing domain. For IPX networks, administrators need to configure an IPX network address and a frame type for the routing domain in addition to specifying ports that belong to the routing domain. Such configuration information for IP and IPX networks are maintained and updated by processes executing on CPU **80** and stored as routing domains **81-1** . . . **81-R** in configuration table **85**. Each individual port can belong to only one routing domain. In accordance with an aspect of the invention that will be described in more detail below, the routing domain configurations are used to automatically configure rules in flow table **70** such that IP and IPX flows of packets from nodes belonging to the same routing domain are switched at Layer **3**+ at wire speed, while IP and IPX flows of packets from communicating nodes on different routing domains are routed at wire speed at Layer **3**+.

FIG. **4** further illustrates a switch module **60** in accordance with the architecture illustrated in FIG. **3**. As can be seen, it includes switch engine **100**, address registers **105**, domain configuration registers **115**, priority level configuration registers **125**, CPU interface **110**, port interfaces **120-1** . . . **120-N** with associated I/O queues, and memory interface **130**. As is further apparent from the figure, switch engine **100** accesses information contained in flow table **70**, address registers **105**, domain configuration registers **115** and priority level configuration registers **125**, and manages packets buffered in packet buffer **75**. CPU interface **110** communicates with CPU **80**, thereby providing communication means between CPU **80** and switch engine **100**, address registers **105**, domain configuration registers **115**, priority level configuration registers **125**, port interfaces **120-1** . . . **120-N**, and memory interface **130**. Port interfaces **120-1** . . . **120-N** respectively communicate with ports **50-1** . . . **50-N**, and memory interface **130** manages access to shared memory **90**. It should be noted that in this configuration, both switch engine **100** and CPU **80** (via CPU interface **110** and memory interface **130**) can forward pack-

ets on ports **50-1** . . . **50**-N via port interfaces **120-1** . . . **120**-N and their associated I/O queues, although in the preferred embodiment, switch engine **100** can forward packets at wire speeds with no intervention from CPU **80**.

Switch engine **100** performs the flow identification and processing operations for forwarding packets received via port interfaces **120-1** . . . **120**-N. It accesses flow table **70** to look up the forwarding information associated with the flows. Address registers **105** provide address information to assist switch engine **100** in locating appropriate flow processing information in flow table **70**. The contents of these registers can be configured by CPU **80** via CPU interface **110**, and include the base Ethernet address of ports **50-1** . . . **50**-N.

Domain configuration registers **115** provide routing domain configuration information in accordance with routing domains **81-1** . . . **81**-R established in configuration table **85**. For each port, separate IP and IPX routing domain identifiers associated therewith are stored in domain configuration registers **115**, which registers are configured by CPU **80** via interface **110**. Routing domain configuration information for IP and IPX networks is maintained and updated by processes executing on CPU **80** and stored as routing domains **81-1** . . . **81**-R in configuration table **85**. CPU **80** then uses this configuration information to configure domain configuration registers **115** via interface **110**.

Although shown singly for clarity, the I/O queue associated with each of port interfaces **120-1** . . . **120**-N includes several queues, each having a corresponding priority level that can be configured by the software running on CPU **80**. Switch engine **100** forwards packets destined for one of ports **50-1** . . . **50**-N using the appropriate queue, in accordance with priority rules configured in the flow table. The number of times in a service interval packets in each queue will be processed is programmed such that higher priority queues get serviced more frequently than lower priority queues. Each flow can be assigned a particular priority level, and thus have a desired QoS, as will be explained in more detail below. Priority level configuration registers **125** provide service level settings for each respective level of priority corresponding to the I/O queues associated with port interfaces **120-1** . . . **120**-N.

FIG. **5** further illustrates the contents of flow table **70**. In this example, flow table **70** includes address resolution hash **140**, address resolution record table **150**, protocol entry table **160** and network entry table **170**. Flow table **70** further includes priority tags **190**, filter tags **200** and mirror tags **210**. The contents of these tables, and their interrelations, will be described in more detail below in connection with the creation and maintenance of flow table entries according to the invention. Switch engine **100** has read access of the tables in flow table **70**, and CPU **80** has read and write access to the tables.

The operation of the multiprotocol switch of the present invention will now be described in more detail with reference to FIGS. **6** to **14**.

After powerup (step S2), and before any packets are received, flow table **70** is empty. Configuration table **85** contains routing domain, filter, mirror, priority, and VLAN configurations that have been established already for the network, and can be updated at any time by a system administrator via software processes executing on CPU **80**.

In addition to switching and routing packets, the multiprotocol switch of the present invention performs tasks that are performed by conventional switches and routers such as route determination, routing table updates, and the like. Such tasks are well known to those skilled in the art and are not depicted in the overall operation of the switch, as illustrated in FIG. **6**, so as not to obscure the invention. Of note, however, are the unique capabilities of the multiprotocol switch with respect to multicast group management for IP, AppleTalk and NetBios networks.

In accordance with an object of the invention, virtual LANs are automatically created for every IP multicast group associated with nodes and segments attached to the switch. Upon powerup (step S2), software processes executing on CPU **80** cause the multiprotocol switch to send out host membership query messages (IGMP messages addressed to 224.0.0.1) (step S4). Hosts attached to the switch that belong to multicast groups send IGMP host membership report messages. The switch can thus determine all the multicast groups to which nodes attached to it belong and forward IP multicast packets within multicast groups appropriately.

The switch then enters into an operational state wherein switch engine **100** continually monitors for data packets arriving on each of ports **50** via port interfaces **120-1** . . . **120**-N (step S6). When a packet is received (step S8), it is processed in accordance with the algorithm further illustrated in FIG. **7** (step S10). CPU **80** also periodically sends out IGMP queries (step S12 and step S14) to determine all active IP multicast groups to which hosts attached to the ports of the switch belong and keeps its multicast group table updated. Moreover, hosts send IGMP packets to join and leave IP multicast groups. These IGMP packets are intercepted by switch engine **100** and sent to CPU **80**. CPU **80** can therefore have up to date information about IP multicast group membership.

Packet processing and switching in the multiprotocol switch of the invention, that can be included in step S10 of FIG. **6** for example, is further illustrated in FIG. **7**.

When a packet is received, switch engine **100** first examines the packet header and if the packet is a multicast packet (step S22), the packet is forwarded to CPU **80** for special processing to automatically create and/or update VLAN information (step S24). Whether or not it is a multicast packet, switch engine **100** keeps the packet for further processing.

Layer **3**+ switching and routing can be disabled for the system by setting a flag in configuration table **85**, for example. In that event, each packet traversing through the port is processed and switched as a Layer **2** packet, regardless of the protocol carried by the packet. Accordingly, if Layer **3**+ switching and routing is disabled (as determined in step S24), processing branches to step S26.

All data packet headers include the source and destination Ethernet addresses of the packets. The protocol used by the communicating hosts can also be determined from the Ethernet data packet header. For example, Ethernet Type II packets contain a type field that explicitly indicates the protocol (such as IP, IPX, ARP, RARP and LAT) that packet belongs to. Moreover, it is known that different protocols use different Ethernet frame formats. For example, IP uses Ethernet II or SNAP packet formats. IPX can use all four Ethernet formats (depending on configuration), AppleTalk (a trademark of Apple Computer Corp., Cupertino, Calif.) uses the SNAP packet format and NetBios typically uses the Ethernet 802.2 LLC format. The protocol carried by the packet thus can be learned either explicitly or implicitly from the packet header.

Layer **4** protocols such as Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are built on top of the Layer **3** IP protocol. In addition to the information

above, the headers for these protocols further contain source and destination socket numbers, which can identify individual applications such as FTP, Telnet, e-mail and HTTP, running on IP hosts. Specifically, the protocol carried field in the IP header identifies the protocol carried by the IP datagram. For example, if the protocol carried field has a value of 6, this indicates that the IP datagram carries a TCP packet, whereas a value of 17 indicates that a UDP packet follows the IP header.

The IPX protocol is also at Layer 3 of the OSI model. Most Novell NetWare (trademark of Novell, Inc. of Provo, Utah) applications run on top of IPX. IPX headers contain source and destination socket numbers which identify applications running on the hosts, in addition to IPX source and destination network and IPX source and destination node addresses, which identify end-to-end IPX hosts. Different Novell Netware Layer 4 protocols use the IPX datagram to send and receive packets. The protocol field in the IPX header indicates which protocol (SAP or RIP) the IPX datagram is carrying.

As evident from the foregoing, flows of packets associated with IP and IPX protocols can thus be identified by their IP/IPX source and destination node and/or socket information, and those associated with other protocols can be identified by their source and destination Ethernet addresses. In this example of the invention, flows of packets associated with IP and IPX protocols, as well as ARP and RARP packets, are processed by the multiprotocol switch to support special Layer 3+ processing and/or routing across different networks. Accordingly, as shown in FIG. 7, switch engine **100** determines the protocol type from the packet (step S28). As shown above, this may be explicitly defined in the header as in the Ethernet Type II type field or may be implicitly derived from other information in the IP or IPX header. If the packet is not an IP/IPX or ARP/RARP packet (determined in step S30), processing branches to step S26 and it is processed and switched as a Layer 2 packet, regardless of the protocol carried by the packet. Otherwise, processing advances to step S32 and the packet is processed in accordance with Layer 3+ protocols.

FIG. 8 further illustrates switch engine processing of Layer 3+ packets, which processing can be included in step S32 of FIG. 7 for example.

Switch engine **100** first extracts the address information for both the source and destination ends of the flow with which the packet is associated (step S40). As shown above, headers of packets associated with IP and IPX protocols include IP/IPX source and destination node addresses and may further include socket numbers.

Switch engine **100** then checks to see whether entries exist in the flow table for both ends of the flow (step S42). If entries exists for both ends of the flow (determined in step S44), the packet is forwarded at wire speed in accordance with any filters, mirrors, priorities or VLANs established in the flow table entry (step S46).

If flow table entries do not exist for both ends of the flow, the packet is "unresolved." For unresolved IP/IPX and ARP/RARP packets, switch engine **100** forwards or broadcasts the packet as best it can under the circumstances. If only the destination end of the flow was unresolved (determined in step S48), a flow table entry exists for the source end of the flow, containing a broadcast enable entry for the source. Accordingly, switch engine **100** forwards the packet on the port(s) indicated by the broadcast enable entry associated with the source (step S56).

If only the source end of the flow is unresolved, a flow table entry exists for the destination. In this situation, switch engine **100** notifies CPU **80** via CPU interface **110** that the packet is unresolved (step S56). The message to the CPU contains the source and destination addresses (and possibly socket numbers), the port on which the packet arrived, and an indication that the source end of the flow was unresolved. As will be described in more detail below, CPU **80** will process the packet, and if appropriate, will create entries in flow table **70** containing forwarding information and links to information relating to any previously-configured filters, priorities, mirrors or VLANs corresponding to the unresolved ends of the flow.

If both the source and destination ends of the flow are unresolved, switch engine **100** notifies CPU **80** via CPU interface **110** that the both ends of the flow were unresolved (step S58). Moreover, switch engine **100** forwards the packet on all ports indicated by the default broadcast enable entry for this protocol. As will be explained in more detail below, default entries exist for IP, IPX, and non-IP/IPX types of flows. Switch engine **100** can determine which ports on which to forward the packet according to the default broadcast enable entry, and causes the packet to be forwarded to a default I/O queue (typically one with lowest priority) associated with each of the indicated ports (step S60).

After the unresolved packet is broadcast or forwarded in steps S50 or S60, the destination node, if attached to the switch, will respond. The response packet will be processed as described above and a flow table entry for the responding node will be created by the CPU as in step S56. At that point, flow table entries for both ends of the flow will have been created so that any subsequent packets belonging to that flow will be forwarded by switch engine **100** at wire speed.

CPU processing for unresolved IP/IPX or ARP/RARP packets, as initiated in step S48 of FIG. 8 for example, is further illustrated in FIG. 9.

First, CPU **80** determines whether the source of the flow is unresolved, from the message sent by switch engine **100** for example (step S70). If so, processing advances to step S72, where a flow table entry for the source of the flow is created by software executing in CPU **80**, in accordance with any filters, mirrors, and priorities associated with the source node (that may have been configured by the network administrator) in flow table **70**.

Processing then advances to step S74, where CPU **80** determines whether the destination of the flow is unresolved, from the message sent by switch engine **100** for example. If not, then the source was the only unresolved portion of the flow, and so the packet can be forwarded in accordance with the flow table information (step S76).

If the destination of the flow is unresolved, as determined in step S74, processing continues to step S78, where CPU **80** determines whether the packet is addressed to the switch. Such packets can include, for example, ARP packets from hosts that are attempting to get the Ethernet address of their IP gateway. Since the IP address of the gateway is actually associated with a port of the switch, such ARP requests must be processed by the switch and responded to appropriately by, for example, sending an ARP response back to the requesting host containing the Ethernet address of the gateway interface (step S80). Such packets can also include responses to ARP requests sent by CPU **80** to determine the Ethernet address of the host for programming the swap fields of a flow table entry, as will be described in more detail below in connection with the processing of steps S88 and S90. The dashed line connecting steps S80 and S90 in FIG. 9 thus represents the logical processing flow when an ARP request sent by CPU **80** in step S88 is responded to by the

host at the requested IP node, which processing will be further described below.

If the destination of the flow is unresolved and the packet is not addressed to the switch, a flow table entry needs to be created for the destination of the flow. First, it must be determined whether the flow requires switching or routing. This is determined in step **S62**. If the destination Ethernet address of the packet is the Ethernet address of the port of the switch on which the packet arrived (as determined from address registers **105**) and the destination IP or IPX address is not the IP/IPX address of the switch, then the packet needs to be routed. If the destination Ethernet address is not the Ethernet address of the port of the switch, CPU **80** further looks up the routing domains configured in tables **81-1** . . . **81-R** in configuration table **85**. If a packet is going from aport in one routing domain to a port in another routing domain, then the packet will require routing. Otherwise, it is switched. This is required to support IP multicast routing, as will be described even further below.

If it is determined in step **S62** that the unresolved destination is in another network, the Ethernet address of the destination needs to be determined to perform routing. That is, during routing, switch engine **100** needs to replace the source Ethernet address in the packet with the Ethernet address of the switch port on which the packet is being forwarded, and the destination Ethernet address of the packet needs to be replaced by the Ethernet address of the destination node or the router en route to the destination. It is this destination Ethernet address that needs to be determined.

If the packet is an IP packet (as determined in step **S86**), CPU **80** determines the IP network that the destination belongs to and determines the port(s) that connect to or belong to that network. CPU **80** can do this, for example, by cross-referencing the destination IP address with the contents of routing table **65** (created statically by an administrator or dynamically learned using routing protocols such as RIP and OSPF). If no such network exists, CPU **80** sends an ICMP redirect message to the host indicating that the network was unreachable. If it finds an entry in routing table **65** for the network, CPU **80** sends an ARP request packet on all the ports belonging to that network (step **S88**). The destination or next hop sends an ARP response containing its Ethernet address. Switch engine **100** sends this response to CPU **80** (step S48 in FIG. **8**). CPU **80** extracts the Ethernet address contained in the response packet, and records the port on which the packet arrived (step **S80** and step **S90**). For further information regarding binding machine level addresses with network level addresses using ARP, see generally Douglas E. Comer and David L. Stevens, Internetworking with TCP/IP—Vol. II: Design, Implementation, and Internals, 1994, Chapter 4, pp. 39–59.

If the packet is an IPX packet that needs to be routed and destination address is unresolved (as determined in steps 74 and S86), the Ethernet address of the destination is termined using IPX RIP information in routing tables **65** intained by CPU **80** (if the destination is a Netware yer). If the destination is a Netware client, then the tination Ethernet address is already known. In either nt, the Ethernet address associated with the IPX destion address is determined in step **S92**. For further infortion regarding route determination and updating using see generally IPX Routing Guide, published by Novell s.

CPU **80** then creates a flow table entry for the destination he flow (step **S94**) and programs the Ethernet address swap field of the entry with the Ethernet address information determined in the preceding steps (step **S96**).

Processing for creating flow table entries for unresolved packets, as performed in steps **S72** and **S94** of FIG. **9** for example, is further illustrated in FIG. **10**.

First, in step **S100**, CPU **80** extracts the last twelve (least significant) bits (**0–11**) of the Ethernet or IP/IPX address that could not be resolved by switch engine **100**. CPU **80** uses these twelve bits as a hash into flow table **70** to determine whether an address resolution hash entry exists in address resolution hash table **140** for the unresolved address (step **S102**). The address resolution hash entry is used as a starting link for all forwarding and other packet processing information associated with the node corresponding to the unresolved address.

Separate hash areas are maintained for Ethernet, IP and IPX address tables. Each hash entry is 32 bits long and has a format as shown below (bit positions of each field shown in parentheses):

Hash Accessed (31)	Record Offset (27–10)	Number of Records (9–2)	Record Link Valid (1)	No Entries Valid (0)

The Hash Accessed field indicates whether this hash has been accessed by switch engine **100**. This field can thus be used to age out hashes using the Least Recently Used (LRU) algorithm, for example. Aging software executing on CPU **80** initially sets this bit on all the hash entries. When a node associated with this hash entry sends data on the network, switch engine **100** clears this bit. The aging software can later and/or periodically delete hash entries that do not have the Hash Accessed bit cleared.

The Record Offset field contains the address offset from the Base Record Address of address resolution record table **150** at which the first record entry for the group of addresses that map to this hash is stored. The first address resolution record entry associated with this hash will thus reside at location (Base Record Address+Record Offset). The Base Record Address is stored in a register within address registers **105**. The Record Offset field is originally set to zero, but CPU **80** updates it with the offset of the address resolution record entry for this flow, after such entry is stored in address resolution record table **150**, so as to link it to this hash entry.

The Number of Records field indicates the number of addresses (minus one) that the switch has learned map to this hash. This field is originally set to zero, but is updated when CPU **80** creates additional address resolution record entries that are linked to this hash.

The Record Link Valid field, when set, indicates that the data stored at location (Base Record Address+Record Offset+(Number of Records×2)+2) is actually a Link Entry. Since each hash can only point to 128 address resolution record entries (7 bit field), this bit can be used to increase the number of records for this hash value. If this bit is not set, and the No Entries Valid bit is also not set, then the data stored at (Base Record Address+Record Offset+(Number of Records×2)) is the last possible address resolution record for this particular hash entry.

The No Entries Valid bit indicates that there are no valid addresses that map to this hash. This bit is originally set, but is cleared when CPU **80** creates an address resolution record corresponding to this hash entry.

Accordingly, in step **S102**, when CPU **80** next determines whether a valid hash entry exists in address resolution hash

140 at the position corresponding to the twelve bits of the unresolved address, it inspects the No Entries Valid field of the entry. If the bit is set, CPU **80** clears it and all other bits in the entry, thereby creating a valid hash entry at the position in address resolution hash **140** corresponding to the unresolved address (step **S104**).

CPU **80** next increments the Number of Records field in the hash entry to indicate that an additional address resolution record entry for this hash will be created (step **S106**). If the number of records that will exist for this hash exceeds the field size of the Number of Records field, that field is decremented and the Record Link Valid field bit is set.

Processing advances to step **S108**, where CPU **80** creates an entry in address resolution record table **150** for the host corresponding to the unresolved address. The format of an address resolution record entry is shown below.

Protocol Offset (31–18)			Ethernet Address bits 15–12, 23–16, 31–26 (17–0) OR IP/IPX Address bits 15–12, 23–16, 31–26 (17–0)
Port Number (31–23)	Priority Tag Addr (22–19)	Mirror Enable (18)	Ethernet Address bits 25–24, 39–32, 47–40 (17–0) OR IP/IPX Address bits 25–24 and Socket bits 0–15 (17–0)

CPU **80** fills the Address fields with the remaining 36 bits of the unresolved address that were not used as the initial hash. For non-IP/IPX packets, CPU fills these fields with the remaining most significant bits of the Ethernet address contained in the unresolved entry message from switch engine **100**. For IP and IPX packets, CPU **80** fills these fields with the remaining most significant bits of the IP/IPX address and the host application socket number. If no special configurations (filter, mirror or priority) have been configured for the host application, CPU **80** inserts a "don't care" value of hex 0xffff for the socket number to indicate that the socket number is not used to identify the IP/IPX flow with which this entry is associated. Moreover, if the protocol (such as ICMP, IGMP)carrying the packet does not use socket numbers a "don't care" value is used for the socket value.

The Port Number field indicates the port (**50-1** . . . **50-N**) which this Ethernet (or IP or IPX) address resides. CPU fills this field with the port number contained in the resolved entry message from switch engine **100**.

The Protocol Offset field indicates the offset from the Protocol Address at which the entries in the Protocol for each protocol associated with this flow are stored. Base Protocol Address is stored in a register in address ers **105**. The protocol the packet belongs to is used as crement from the Protocol Offset to point to the priate entry. The increments for each protocol are in a register in address registers **105**. The address of y in the Protocol Table for the flow associated with dress resolution record will thus be Base Protocol +Protocol Offset+Protocol Increment. If and when creates a protocol table entry for this flow, explained detail below, CPU **80** fills the Protocol Offset field the protocol entry to this address resolution record.

Priority Tag Addr field provides the address to one of bit priority tags **190**. The Base Address for priority is stored in a register in address registers **105**. ly, the address to the priority tag field for this flow is located at Base Address+Priority Tag Addr. CPU **80** initially sets the Priority Tag Addr field to zero. If priorities are configured for this flow, as will be explained below, a priority tag will be configured, priority processing will be enabled by setting a bit in the Protocol Offset entry, and the address to the configured priority tag will be programmed in this field to link the priority tag to the address resolution record entry.

The Mirror Enable field qualifies the mirror tag (tag address found in the protocol entry). If this bit is set, mirror processing is enabled for this flow. CPU **80** initially sets this bit to zero. If mirrors are configured for the flow, as explained in more detail below, a mirror tag will be configured, and the mirrors will be enabled by setting this bit.

After creating the address resolution record for the unresolved portion of the flow, processing advances to step **S100**, where CPU **80** determines how many address resolution records now exist for this hash by examining the Number of Records field in the hash entry. If this is the first address resolution record entry created for this hash, CPU **80** simply stores the address resolution record entry it created in address resolution record table **150** at the address corresponding to Base Record Address+Record Offset (step **S112**). It then links this entry to the address resolution hash entry by storing the address offset from the Base Record Address where the entry is stored in the Record Offset field of the hash.

If it is determined in step **S110** that there are more than one address resolution record entries for this hash, CPU **80** sorts the existing entries with the newly created entry in order of the remaining address bits of the unresolved flow address, and stores them accordingly (step **S114**). The entries are already linked to the address resolution hash entry by the previous programming of the Record Offset field.

Processing then advances to step **S116**, where CPU **80** determines whether any special processing has been configured or is otherwise required for this flow. This includes determining whether any filters, mirrors, priorities or VLANs are associated with this flow. CPU **80** does this, by cross-referencing the flow identification information in the unresolved packet with the entries in configuration table **85**.

If it is determined in step **S116** that no filters, mirrors, priorities, VLANs or swap addresses need to be programmed for this flow (i.e. routing is not configured for the switch), the address resolution record is linked to a default set of protocol entries which are all linked in turn to a default network entry (step **S118**). If this is a TCP, UDP or IPX packet with a socket number, the socket number portion of the remaining address bits in the address resolution record entry for this flow will be set to the "don't care" value of 0xffff. The default network entry will be described in more detail below.

Otherwise, if it is determined in step **S116** that this flow requires special handling, unique entries in protocol table **160** and network table **170** for the flow must be created, and processing by CPU **80** continues by first creating a protocol entry (step **S120**).

The address pointed to by the Protocol Offset field of the address resolution record indexes a list of entries in protocol table **160**. Different filters, mirrors and priorities can be established between two nodes and/or processes depending on the particular protocol used for communication between them. The protocol increment for the particular protocol (provided by address registers **105**) points to the appropriate

set of filters, etc. for the flow, as indicated by a 32-bit entry in the protocol table, having the following format:

Fltr Addr (31–27)	Mirror Addr (26–23)	Filter Enable (22)	Priority Queue (21–19)	B/W (18–15)	Priority Enable (14)	Network Offset (13–0)

The Fltr Addr field provides the address to one of the 32 32-bit filter tags **200**. The Base Address for filter tags **200** is stored in a register in address registers **105**. Accordingly, the address to the filter tag for this flow is located at Base Address+Fltr Addr. CPU **80** initially sets the Fltr Addr field to zero. If filters are configured for this flow, CPU **80** will create a filter tag in filter tags **200** and program the address of the tag within filter tags **200** in this field to link the configured filter tag to the protocol entry. CPU **80** determines whether any filters are configured for this flow by cross-referencing the filter tables **76-1** . . . **76-F** in configuration table **85** with the flow identification information for this side of the flow.

Each 32-bit filter tag contains 8 bits of veto information and 24 bits of match information. CPU **80** configures filter tags for each side (source and destination) of the flow to be filtered. If packets belonging to the flow are to be blocked (as in firewalling), CPU **80** configures the filter tags for the source and destination sides of the flow so that their veto bits do not match and their match bits do match. For example, if a firewall is established between a first network having nodes A, B and C and a second network having nodes D, E and F, CPU **80** may configure the filter tags for nodes A, B and C such that they have match bits of 0x000001 and veto bits of 0x02, while nodes D, E and F are configured with filter tags having match bits of 0x000001 and veto bits of 0x03. Accordingly, nodes A, B and C will be able to communicate with each other, but not with nodes D, E and F, who are also able to communicate with each other. Now assume that node C in the first network is to be given the privilege of communicating with nodes in the second network. This can be done by changing its match bits to 0x000002. Now node C will be able to communicate with all nodes in both networks.

The Mirror Addr field provides the address to one of the 16 16-bit mirror tags **210**. If the mirror enable bit in the address resolution record linked to this protocol entry is set, mirroring is enabled for flows to which this node belongs. The Base Address for mirror tags **210** is stored in a register in address registers **105**. Accordingly, the address to the mirror tag for this node is located at Base Address+Mirror Addr.

CPU **80** determines whether any mirrors are configured for this flow by cross-referencing the mirror tables **78-1** . . . **78-M** in configuration table **85** with the flow identification information for this side of the flow. If any of the established mirrors correspond to this flow, one of the 16 16-bit mirror tags **210** will be configured for it. Each mirror tag field contains 1 bit of match information. CPU **80** initially sets the Mirror Addr field to zero. If traffic belonging to the flow is to be mirrored on another port(s), CPU **80** sets the mirror enable bit in the address resolution entry for each end of the flow, configures a tag in mirror tags **210**, and links both ends of the flow to the same tag by filling the Mirror Addr field of both respective protocol entries with the address of the configured tag in mirror tags **210** (offset from the Base Address). Switch engine **100** can thereafter locate the tag by combining the Mirror Addr field with the Base Address for mirror tags **210** which is stored in a register in address registers **105**. The port(s) on which the flow is to be mirrored is configured in the Mirror Entry of the network entry linked to this protocol entry (described below).

The Priority Enable field qualifies the priority tag (addressed by the Priority Tag Addr in the address resolution record entry corresponding to this flow). If this bit is set, the flow is processed at a higher priority. CPU **80** initially sets this bit to zero. If priorities are configured for this flow, a priority tag will be configured, priority processing will be enabled by setting this bit, and the address to the configured priority tag within priority tags **190** will be programmed in the Priority Tag Addr field of the address resolution record entry linked to this protocol entry to link the priority tag to the address resolution record entry.

CPU **80** determines whether any priorities are configured for this flow by cross-referencing the priority tables **77-1** . . . **77-P** in configuration table **85** with the flow identification information for this side of the flow. If any of the established priorities correspond to this flow, a priority tag will be configured for it. Each priority tag contains 1 bit of match information. If packets belonging to the flow are to be forwarded with priority, CPU **80** enables priority by setting the Priority Enable field in the protocol entries of both ends of the flow, configures a priority tag in priority tags **190**, and links the address resolution records of both ends of the flow to the same configured priority tag. The level of priority associated with this flow is determined by the Priority Queue field and the service level for that queue programmed in priority configuration registers **125**.

The Priority Queue field is valid if the Priority Enable bit is set for the flow. CPU **80** initially sets this field to zero. If a priority is configured for this flow, CPU fills this field with the priority level with which all packets belonging to this flow will be forwarded by the switch. For example, if eight levels of priority are supported, there will be eight I/O queues associated with each port **50-1** . . . **50-N**, and this field will indicate which one of the queues into which packets belonging to this flow will be placed.

Each queue's corresponding priority level is user configurable. That is, a system administrator can program the number of times per service interval a queue having that priority level should be serviced on a system wide basis. This configuration is stored in priority level configuration registers **125**. One such register exists for each priority level/queue. For example, if a value of 5 is programmed into priority configuration register for priority level **4**, the queue corresponding to that priority level will be serviced 5 times in a service interval. The service interval is determined by adding together all the priority level values programmed into each priority configuration register. Weighted Fair Queueing is implemented by servicing the queues with equal priority after servicing the queues in accordance with established priorities (i.e. after all the queues are serviced per the priorities in priority level configuration registers **125**, service queue **1**, once, queue **2** once and so on until all the queues are serviced). This ensures that even the lowest priority queues are serviced once every service interval.

After creating the protocol entry for this unresolved portion of the flow, CPU **80** stores the entry in protocol entry table **160** at the Protocol Increment associated with the protocol used by the hosts of this flow, which increment is referenced to the Protocol Offset for this flow. Then CPU **80** links the protocol entry with the address resolution record entry for this flow by writing the Protocol Offset from the Base Protocol Address where this record is stored in the Protocol Offset field in the address resolution record entry (step S122).

Processing advances to step S124, where CPU **80** creates an entry in network entry table **170** for this unresolved portion of the flow. The Network Offset field points to an entry in network entry table **170** associated with this flow and protocol. The base address of the entry will be Base Network Address+Network Offset. The Base Network Address is stored in a register in address registers **105**.

Entries in network entry table **170** contain fields that help configure virtual LANs, fields to configure mirroring, and fields to store the destination Ethernet address values to swap out the ones in the packet if the packet is being routed. At least one network entry exists for every address resolution table entry. However, multiple address resolution entries can share the same network entry. For example, a default network entry exists for every IP, IPX and non-IP/IPX flow that has no associated filters, mirrors, priorities or VLANs configured for it, as will be described in more detail below. Accordingly, for example, all such non-configured IP flows will have address resolution entries that point to the same default protocol entry that in turn points to the same default network entry. The format of each network entry is shown below:

	Broadcast Enable Entry (31–0)
	Reserved (31–0)
Unused	Reserved (16–24) Swap Address Bits 15–0 (15–0)
	Swap Address Bits 47–16 (31–0)
	Mirror Entry (31–0)

The Broadcast Enable field is used to control the ports on which broadcast and multicast packets, originating from this node should be forwarded. This field contains 1 bit of match information corresponding to each port of the switch (32 ports in this example). A 1 in any bit position indicates that a broadcast packet from the node associated with that Broadcast Enable field must be forwarded on that port. For example, if bits **13** and **14** in this field are set, then broadcast or multicast packets from a node having a network entry containing this field will be sent out on ports **14** and **15**. CPU **80** initially sets this field to a default value based on the protocol and routing domain associated with the node. Accordingly, broadcasts and multicasts originating from a node will only be forwarded on ports with attached nodes belonging to the same routing domain, if such a routing domain exists. If the node belongs to a VLAN configured on or learned by the switch, CPU **80** sets this field appropriately to make sure that multicasts are forwarded only on ports belonging to that VLAN. For example, if nodes attached to ports **1** and **2** belong to the same VLAN, then the broadcast enable field for those nodes will be 0x00000003 indicating that multicast packets from nodes on those ports should be sent out only on ports **1** and **2**.

Every attached node learned by the switch has a network entry containing an associated 32 bit Broadcast Enable field. Separate default network entries having default Broadcast Enable fields are created for each of the IP, IPX and non-IP/IPX protocols. These default network entries are used for nodes that do not have filters, mirrors, priorities or VLANs configured for them. Respective "dummy" protocol entries are further provided to link between unique address resolution record entries and the default network entries. The default Broadcast Enable field of the default network entry to which non-IP/IPX flows are linked is set to 0xffffffff indicating that broadcasts must be forwarded on every port. The default Broadcast Enable fields of the default network entries to which IP and IPX flows are linked are configured based on routing domains. For example, if there are two IP routing domains comprised of ports **1** through **15** and **16** through **32** respectively, then there will be one default network entry for each routing domain. The first one is associated with all nodes learned on ports **1** through **15** and will have a Broadcast Enable field containing 0x0000ffff. The second is associated with all nodes learned on ports **16** through **32** and will have a Broadcast Enable field containing 0xffff0000. This ensures that broadcasts from nodes in IP routing domain **1** will not be sent to nodes in IP routing domain **2** and vice versa.

When a flow has one or more of a filter, mirror, priority or VLAN configured for it, non-default protocol and network entries will be created for it. When creating the network entry for the node, however, CPU **80** first copies the Broadcast Enable field of the default network entry associated with this flow into the Broadcast Enable field for the newly created network entry.

CPU **80** determines whether any VLANs are configured for this flow by cross-referencing the VLAN tables **79-1** . . . **79-V** in configuration table **85** with the flow identification information for this side of the flow. If any VLANs are established the node associated with this side of the flow belongs to any of the established VLANs, a non-default network entry will be configured for it. CPU **80** configures the Broadcast Enable field in accordance with the established VLAN by setting the bits corresponding to the port(s) on which each node belonging to the VLAN is attached.

The Mirror Entry field indicates the ports on which the packets belonging to the flow associated with this network entry should be mirrored on. CPU **80** initially sets these fields to zero. As described above, if mirrors are configured for the flow to which this packet belongs, CPU **80** sets the Mirror Enable bit for both sides of the flow, configures a mirror tag and points both sides of the flow to the same mirror tag, and configures the Mirror Entry by setting the bits corresponding to the port(s) on which packets are to be mirrored on.

The Swap Address Bits fields contains the Ethernet addresses used for address swapping during packet routing. For IP and IPX routing, the swap field bits are filled in with the Ethernet address associated with the destination node as described above. When routing a packet, the switch needs to replace the source Ethernet address with the address of the port through which the packet is being forwarded and the destination Ethernet address in the packet to be routed with the address of the next hop router or the end station. Every node that communicates across networks has a unique network entry associated therewith. The swap field of the network entry is filled by CPU **80** with the Ethernet address of the next hop router or the end station as described above in connection with the processing depicted in FIG. **9**. The Ethernet address is determined at the time of entry creation using Address Resolution Protocol (ARP) for IP packets.

After creating the network entry as described above, CPU **80** stores it in network entry table **170**, then links it to the protocol entry for this portion of the flow by writing the Network Offset relative to the Base Network Address where it is stored in the protocol entry (step S126).

The flow table entry creation processing described above will now be further described by way of the following illustrative and non-limiting examples of the invention.

First, assume that a system administrator sets up a routing domain for a group of users in a network containing a multiprotocol switch of the present invention, for example, the engineering department of a company. Further assume

that these users are connected to ports 1–8 of the switch and that they have been configured with IP addresses in the range 206.233.77.0 to 206.233.77.255, so that the associated IP network is 206.233.77.0 (subnet mask 255.255.255.0).

Once an administrator configures ports 1–8 to belong to routing domain 1 (network address 206.233.77.0), CPU 80 creates a default network entry for that domain in network entry table 170, having a Broadcast Enable field value of 0x000000ff, indicating that broadcasts from any of the nodes attached to ports 1–8 will be sent out only on ports 1–8. CPU 80 then locates an empty space in protocol table 160 sufficient to contain entries for all supported protocols and creates dummy protocol entries that all point to the default network entry for routing domain 1 created in the previous step.

If a user connected to port 1 of the switch wishes to open an FTP session with a user connected to port 2 of the switch, and if this is the first communication between them, there will be no flow table entries for this flow, and the processing depicted in FIG. 10 will be performed for both, the host connected to port 1 and the host connected to port 2 one at a time.

The first packet of the flow is unresolved and is sent to CPU 80 for further processing. CPU 80 creates hash table entries in the IP area of address resolution hash 140 at positions corresponding to the last twelve bits of the IP address of both respective hosts, and clears the No Entries Valid fields in both entries (steps S102 and S104).

Next, CPU 80 creates address resolution record entries in address resolution record table 150, fills the Address fields with the remaining address bits and a "don't care" value of 0xffff for the socket bits, and fills the Port Number fields with the respective port numbers. CPU 80 links the newly created address resolution record entries to the address resolution hash entries by programming the Record Offset fields in the hash entries with the offsets of the address resolution record entries from the Base Record Address in the address resolution record table (steps S108–S112).

CPU 80 determines that no mirrors, filters, priorities or VLANs are associated with this flow, and that no routing is required (step S116). Accordingly, the default protocol and network entries can be used (step S118). CPU 80 links the dummy protocol entry for TCP to both address resolution record entries by programming the Protocol Offset fields of both address resolution record entries with the base address of the dummy protocol entries located in protocol table 160, which dummy protocol entries all point to the default network entry for routing domain 1.

In a next example, assume that a system administrator establishes a firewall that forbids any communications between hosts having Ethernet address A and Ethernet address B, respectively connected to ports 8 and 9 of the switch. This filtering information is stored in filter tables 76-1 . . . 79-F in configuration table 85 and specifies the Ethernet addresses, but not the ports.

If the host having Ethernet address A wishes to send a non-IP/IPX packet to the host having Ethernet address B, and if this is the first attempted communication between them, there will be no flow table entries for this flow, and the processing depicted in FIG. 10 will be performed for both hosts one at a time.

CPU 80 creates hash table entries in the Ethernet area of address resolution hash 140 at positions corresponding to the last twelve bits of Ethernet address A and B for the hosts connected to ports 8 and 9, respectively, and clears the No Entries Valid fields of both entries (steps S102 and S104).

Next, CPU 80 creates address resolution record entries in address resolution record table 150, fills the Address fields with the remaining bits of Ethernet address A and B for the host connected to port 8 and 9, respectively, and fills the Port Number fields with the respective ports (8 and 9). CPU 80 links the newly created address resolution record entries to the address resolution hash entries by programming the Record Offset fields in the hash entries with the offsets of the respective address resolution record entries from the Base Record Address in the address resolution record table (steps S108–S112).

CPU 80 then determines that a filter is associated with this flow by comparing the Ethernet addresses of both hosts with the filter set up in configuration table 85, so further processing is required (step S116). CPU 80 first locates empty spaces in protocol table 160 sufficient to contain entries for all supported protocols and creates respective protocol entries in the spaces at an offset predetermined for Ethernet protocol (step S120). This offset is also programmed in a register in address registers 105. Initially, all bits in the protocol entries are cleared. However, since a filter is associated with this flow, CPU 80 sets up filter tags in filter tags 200 and links the filter tags to the respective protocol entries by programming the Fltr Addr fields of the tags with the offsets of the respective tags in filter tags 200.

CPU 80 establishes a filter between the two hosts by setting the veto bits in their respective filter tags to different values and by setting the match bits to the same values. After the flow table entries for this flow have been created, all non-IP/IPX packets between the two hosts will be dropped by switch engine 100. If IP or IPX communications between these hosts are attempted in the future, similar processing will be performed to create IP or IPX flow table entries which will also cause switch engine 100 to drop all packets for such communications.

CPU 80 then links the protocol entries to the address resolution record entries by programming the Protocol Offset fields of the address resolution record entries with the base addresses of the respective empty spaces located in protocol table 160 (step S122).

Next, CPU 80 creates network entries and clears all bits in them (step S124). The Broadcast Enable fields are programmed with a value of 0xffffffff since no VLANs or routing domains have been configured. Finally, CPU 80 links the network entries by programming the Network Offset fields in the protocol entries with the offsets of the respective network entries from the base of network entry table 170 (step S126).

In a next example, assume that a system administrator establishes a priority for all HTTP communications with hosts belonging to an IP subnet being respectively connected to ports 4–7 of the switch. This priority information is stored in one of priority tables 77-1 . . . 77-P in configuration table 85 and specifies the IP network addresses, protocol type and priority level, but not the ports to which the hosts are attached.

An IP host connected to the switch that wishes to access an external Web site uses the default gateway to connect to it. The multiprotocol switch implementing the present invention will act as the default gateway. It will have routing tables established (through traditional routing protocols such as RIP and OSPF) to determine which port (16, for example) will provide the best route to the destination Web site and the port through which the flow should be forwarded. If this is the first attempted communication between the host and the Web site, there will be no flow table entries for this flow, and

the processing depicted in FIG. **10** will be performed for both sides of the flow one at a time.

CPU **80** creates hash table entries in the IP area of address resolution hash **140** at positions corresponding to the last twelve bits of the IP addresses of the respective hosts, and clears their No Entries Valid fields (steps S102 and S104).

Next, CPU **80** creates address resolution record entries in address resolution record table **150**, fills the Address fields with the remaining IP address bits and the socket numbers for both nodes, and fills the Port Number fields with the respective ports. CPU **80** links the newly created address resolution record entries to the address resolution hash entries by programming the Record Offset fields in the hash entries with the offsets of the respective address resolution record entries in the address resolution record table (steps S108–S112).

CPU **80** then determines that a priority is associated with this flow by comparing the IP addresses and protocol with the priority set up in configuration table **85**, so further processing is required (step S116). CPU **80** first locates empty spaces in protocol table **160** sufficient to contain entries for all supported protocols and creates protocol entries in the spaces at an offset predetermined for TCP protocol (since HTTP uses TCP as a transport protocol) (step S1120). This offset is also programmed in a register in address registers **105**. Initially, all bits in the protocol entries are cleared. However, since a priority is associated with this flow, CPU **80** creates a tag in Priority Tags **190** having bits corresponding to the ports of the respective hosts set, sets the Priority Enable bits in the protocol entries and programs the Priority Tag Addr fields in the address resolution record entries for both ends of the flow with the address of the newly created priority tag. Since the priority tags match, bidirectional priority is established for the flow. CPU **80** also programs the Priority Queue fields with the I/O queue corresponding to the priority level configured for the flow in priority tables **77-1** . . . **77-P**. All packets belonging to that flow are then queued in the programmed queue number and are serviced a number of times per service interval as specified by the priority configuration register **125** for that queue (which CPU **80** has already programmed in accordance with a priority level to service level mapping).

(CPU **80** then links the protocol entries to the address resolution record entries by programming the Protocol Offset fields of the address resolution record entries with the base addresses of the respective empty spaces located in protocol table **160** (step S122).

Next, CPU **80** creates network entries and clears all bits (step S124). The Broadcast Enable fields are programmed in accordance with any routing domains or VLANs established (i.e. all ports that belong to the same routing domain have the same broadcast enable field and should send broadcast packets to each other). Since packets belonging to this flow will require routing, swap fields need to be programmed with the destination or next hop Ethernet address. CPU **80** uses Address Resolution Protocol (ARP) to determine these addresses and fills in the Swap Address bits fields accordingly. Finally, CPU **80** links the network entries by programming the Network Offset fields in the protocol entries with the offsets of the respective network entries from the base of network entry table **170** (step S126).

In a final example, assume that a system administrator establishes a mirror so that all Telnet sessions between IP host A and IP host B, respectively connected to ports **4** and **6** of the switch, are mirrored on port 1. This mirror information is stored in one of mirror tables **78-1** . . . **78-M** in configuration table **85** and specifies the IP addresses and process (the socket for Telnet is **23**) and mirror port, but not the ports to which the hosts are attached.

If host A wishes to initiate a Telnet session with host B, and if this is the first attempted communication between them, there will be no flow table entries for this flow, and the processing depicted in FIG. **10** will be performed for both hosts one at a time.

The packet will be sent to CPU **80** as an unresolved flow. CPU **80** creates hash table entries in the IP area of address resolution hash **140** at positions corresponding to the last twelve bits of the IP addresses of the respective hosts, and clears the No Entries Valid fields (steps S102 and S104).

Next, CPU **80** creates address resolution record entries in address resolution record table **150**, fills the Address fields with the remaining IP address bits and socket bits, and fills the Port Number, fields with the respective ports. CPU **80** links the newly created address resolution record entries to the address resolution hash entries by programming the Record Offset fields in the hash entries with the respective offsets of the address resolution record entries in the address resolution record table (steps S108–S112).

CPU **80** then determines that a mirror is associated with this flow by comparing the IP addresses with the mirror set up in configuration table **85**, so further processing is required (step S116). CPU **80** first locates empty spaces in protocol table **160** sufficient to contain entries for all supported protocols and creates protocol entries in the spaces at an offset predetermined for IP protocol (steps S120 and S122). This offset is also programmed in a register in address registers **105**. Initially, all bits in the protocol entries are cleared. However, since a mirror is associated with this flow, CPU **80** creates a tag in Mirror Tags **210** having bits corresponding to the ports of the respective hosts set, sets the Mirror Enable bit in the address resolution record entries and programs Mirror Addr fields for both protocol entries with the address of the newly created mirror tag in Mirror Tags **210** to create bidirectional mirroring for the flow.

CPU **80** then links the protocol entries to the address resolution record entries by programming the Protocol Offset fields of the address resolution record entries with the base addresses of the respective empty spaces located in protocol table **160** (step S122).

Next, CPU **80** creates network entries and sets up the bits in the Broadcast Enable fields in accordance with any routing domains and VLANs established. Since a mirror is required for packets in both directions of the flow, the Mirror Entry fields of both network entries should be programmed. CPU **80** sets bit **0** in the fields so switch engine **100** will send traffic from A to B and B to A to port **1** in addition to the ports associated with A and B. Finally, CPU **80** links the network entries by programming the Network Offset fields in the protocol entries with the offsets of the respective network entries from the base of network entry table **170** (step S126).

FIG. **11** further illustrates address resolution processing performed to determine whether packet processing information exists in flow table **70** associated with the flow to which the packet belongs, which processing can be performed in step S42 of FIG. **8** for example.

As shown in FIG. **11**, switch engine **100** first extracts the last twelve bits of the address associated with the unresolved end of the flow. For IP and IPX packets for which Layer 3+switching and routing is enabled, the last twelve bits of the IP/IPX address are extracted in step S130; for other protocols, the last twelve bits of the Ethernet address are extracted.

The extracted bits are then used to hash onto the area of address resolution hash **140** corresponding to the protocol carried by the packet (IP, IPX or other) (step S132). If an entry exists in address resolution hash **140** for the flow (determined in S134), processing advances to step S138. Otherwise, address resolution processing ends with a negative result and packet processing control branches to step S44 in FIG. **7** (step S136).

In step S138, switch engine **100** locates in address resolution record table **150** the address resolution record that is linked to the hash entry found in the previous step. This is done by combining the Base Record Address contained in address registers **105** with the Record Offset field contained in the hash entry.

Processing advances to step S140, where it is determined whether an address resolution record entry exists for this end of the flow. If the number of records linked to the hash entry is only one, switch engine **100** can readily locate the address resolution record because the record entry is pointed to by combining the Base Record Address from address registers **105** with the Record Offset indicated in the hash entry as is done in the previous step. If, however, the number of linked address resolution records is more than one (denoted by a value in the Num Records field of the hash entry being greater than zero), switch engine **100** performs a binary search on the linked entries based on the remaining bits of the address from which the twelve bits were extracted. For IP/IPX packets, the entry may contain socket numbers along with addresses (a "don't care" socket value of 0xffffh in an entry matches all socket numbers).

If an address resolution record does not exist for the particular flow (determined in step S140), address resolution processing ends with a negative result and packet processing control branches to step S44 in FIG. **8** (step S142).

As described above, the address resolution record identifies one of the ports (**50-1** . . . **50-N**) to which the host associated with this side of the flow is connected. It also may contain links to entries in protocol entry table **160**, and thence to network entry table **170** that can alert switch engine **100** to any filters, priorities, mirrors and VLANs that are configured for the flow.

In step S144, the Protocol Offset field of the address resolution record for this flow is extracted. This field, combined with the Base Protocol Address provided by address registers **105**, points to a list of protocol entries for this flow in protocol entry table **160**. Different protocol entries may exist for the flow depending on the protocol carried by the packet. To get the protocol entry for this flow and protocol, switch engine **100** extracts the protocol carried by the packet and looks up the Protocol Increment associated with the protocol in address registers **105**. This increment is used to point to a particular entry in the list of entries pointed to by the Protocol Offset field in the preceding step, and the protocol entry is thus obtained in step S146.

A pointer to a linked entry in network entry table **170** can be contained within the protocol entry. To get the network entry for this flow and protocol, switch engine **100** obtains the Base Network Address from address registers **105** and combines it with the Network Offset. The result points to the linked entry in network entry table **170** and the network entry for this flow and protocol is thus obtained in step S148.

Switch engine processing for forwarding packets according to flow table information in the multiprotocol switch of the present invention, as performed in step S46 of FIG. **8** and step S76 of FIG. **9** for example, is further illustrated in FIG. **10**. At this point, flow processing information for both the source and destination sides of the flow have been obtained, as described above in connection with the processing illustrated in FIG. **11**.

The filter tag bits for the source and destination, if they are configured (as denoted by the Filter Enable fields of the respective protocol entries), are compared by switch engine **100** (step S150). If they match (i.e., the match bits of the respective filter tags match and the veto bits do not match), then the packet is dropped (S152). Otherwise, processing advances to step S154, where the port associated with the destination is determined from the address resolution record entry for the destination side of the flow. If this is a multicast packet, the port(s) on which the packet should be forwarded is determined from the Broadcast Enable field of the network entry for the destination side of the flow.

In step S156 switch engine **100** determines whether any mirrors have been configured for the flow (as indicated by Mirror Enable bits being set and matching mirror tags for source and destination). If so, these mirror port(s) are determined in step S158.

Packet forwarding processing is performed for each destination port determined in steps S154 and S158. In step S160, switch engine **100** first determines whether the packet needs to be routed to the current destination port. This is done by comparing the contents of the routing domain configuration registers for the source and destination ports. If the source and destination ports are in different routing domains, the packet needs to be routed. In addition, for unicast packets, switch engine **100** compares the destination Ethernet address in the packet with the address of the port on which the packet arrived. If the address matches, then the packet needs to be routed.

For packets that do not need to be routed, processing advances to step S170. For packets to be routed, in step S162, the source address is swapped with the routing domain on which the packet is being forwarded. For unicast packets, the destination Ethernet addresses of the packet is swapped with the address specified in the swap field of the network entry associated with the destination of this flow. The switch engine extracts the hop count from the packet and if the hop count is one or zero for IP or **16** for IPX (as determined in step S166), notifies CPU **80** for further processing (step S168). In addition, switch engine **100** decrements (for IP) or increments (for IPX) the hop count and recomputes the checksum of the packet (for IP packets only) (step S164).

Processing advances to step S170 where, if priority is enabled for the flow (as indicated by the Priority Enable bit in the protocol entry), the priority tags are compared to determine if they match. If so, the Priority Queue field of the destination is determined from the protocol entry associated with the destination of the flow (step S172). Otherwise, a default queue is used (step S174). The packet is then forwarded to the queue determined for this packet and associated with the current destination port (S176). In a service time interval, the queue is serviced the appropriate number of times as specified in priority configuration registers. If this is the last destination port (determined in step S178), packet forwarding processing ends and control branches back to step S12 in FIG. **6**.

Switch engine processing of Layer **2** packets in the multiprotocol switch of the present invention, as performed in step S26 of FIG. **7** for example, is further illustrated in FIG. **13**. Processing steps that are shaded differentiate those which are preferably performed by CPU **80** rather than switch engine **100** in this example of the invention.

Switch engine **100** first extracts the address information for both the source and destination ends of the flow with

which the packet is associated (step S180). As described above, this includes extracting the source and destination MAC addresses of the packet.

Switch engine **100** then checks to see whether entries exist in the flow table for both ends of the flow (step S182). If entries exists for both ends of the flow (determined instep S184), the packet is forwarded at wire speed in accordance with any filters, mirrors, priorities or VLANs established in the flow table entry (step S186).

If only the source end of the flow was unresolved (determined in step S188), a flow table entry exists for the destination end of the flow, containing the port associated with the destination, as well as any broadcast enable entries that are useful if the packet is a multicast packet. Accordingly, switch engine **100** forwards the packet on the port(s) indicated by the flow table entry associated with the destination (step S190).

If only the destination end of the flow was unresolved (determined in step S192), a flow table entry exists for the source end of the flow, containing a broadcast enable entry for the source. Accordingly, switch engine **100** forwards the packet on the port(s) indicated by the broadcast enable entry associated with the source (step S194).

If both the source and destination ends of the flow are unresolved (determined in steps S184, S188 and S192), switch engine **100** notifies CPU **80** via CPU interface **110** that the source address of the packet is unresolved (step S196). The message to the CPU contains the source and destination addresses, the port on which the packet arrived, and an indication that the source was unresolved. CPU **80** will then create an entry in flow table **70** containing forwarding information and links to information relating to any previously-configured filters, priorities, mirrors or VLANs corresponding to the source end of the flow (step S198), in a manner described above in connection with FIG. **10**.

If it is determined in steps S184, S188 and S192 that both ends of the flow are unresolved, the packet is broadcast on all ports. After the unresolved packet is broadcast or forwarded in steps S194 or, S200, the destination node, if attached to the switch, will respond. The response packet will be processed as described above and a flow table entry for the responding node will be created by the CPU as in step S198. At that point, flow table entries for both ends of the flow will have been created so that any subsequent packets belonging to that flow will be forwarded by switch engine at wire speed.

Automatic VLAN creation and management in accordance with the present invention, as performed in step S24 the process illustrated in FIG. **7** for example, is further illustrated in FIG. **14**.

Switch engine **100** uses the Broadcast Enable field in the work entry to determine the ports on which broadcast or cast packets from a node associated with that network needs to be sent. Every node that belongs to a VLAN network entry associated with it, and network entries shared by multiple nodes. As described above, the cast Enable field is one of the 32-bit fields in the k entry. A 1 in a bit position of the Broadcast Enable means that broadcast or multicast packets from that will be forwarded on that port. A 0 in a bit position of roadcast Enable field indicates that broadcast and t packets from the node associated with that net- ntry will not be forwarded on that port. For example, Broadcast Enable field bit position **16** is 1 and **17** a broadcast or multicast packet from that node will out on port **17**, but not on port **18** of the switch.

If a node is assigned to a VLAN, when a packet first arrives from that node, CPU **80** allocate's a network entry for it (if one does not already exist), determines the ports of the switch on which other members of the VLAN are connected and will put a '1' in bit positions corresponding to those ports (e.g. bit **0** for port **1**, bit **1** for port **2**, etc.). A node can belong to more than one VLAN and in that case, the Broadcast Enable field will be updated appropriately (e.g. it will be a logical "OR" of the two entries). If the destination address of a packet arriving from a node is a broadcast address, then it will be unresolved and the switch engine will use the source address to perform address resolution and network entry lookup, and then broadcast the packet according to the Broadcast Enable of the source if such an entry exists. If the destination address is a multicast address, then the switch engine uses the destination multicast address to perform address resolution and network entry lookup, and will multicast the packet on ports indicated by the Broadcast Enable field of the source if such an entry exists.

As described above, a VLAN can be manually created by a system administrator by configuring one via software processes executing on CPU **80**, information regarding which is consequently stored in VLAN tables **79-1** . . . **79-V** in configuration table **85**. As a further aspect of the invention, however, VLANs can be automatically created, updated and maintained by the switch too correspond to multicast groups instantaneously existing among the hosts connected to ports of the switch. The following automatic VLANs are described below automatic VLANs for nodes using Windows DLC/LLC and NetBios (Windows 95 and Windows NT machines), automatic VLANs for AppleTalk nodes (Apple computers), and automatic VLANs for IP nodes using IP multicasting. It should be noted here that automatic VLAN creation can be disabled by the system administrator by means of setting a flag, for example, in VLAN tables **79-1** . . . **79-V**. It should be further noted that automatic VLAN creation can be performed for other types of multicast groups in addition to those described below in accordance with the invention.

When switch engine **100** determines in step S22 that the packet is a multicast packet, it sends the packet to CPU **80** for processing. This includes Windows-95/Windows-NT multicast packets, IP multicasts, IGMP reports and queries, and AppleTalk Zone Information Protocol (ZIP) packets.

Windows-95/Windows-NT (trademarks of Microsoft Corp of Redmond Wash) machines using NetBios protocol on the top of DLC/LLC use a multicast address of 03:00:00:00:00:01 to other Windows-95/NT machines. When a packet is sent by Windows-95 clients and NT servers using this address, switch engine **100** forwards will forward this packet to CPU **80** via CPU interface **110** with a message containing the address and the port on which the packet arrived. When CPU **80** receives this message, processing will advance to step S210. CPU **80** checks to see whether an address resolution record entry exists for this address in flow table **70** (step S212). If not, it creates address resolution hash and address resolution record entries for this multicast address and links them together with a network table entry it further creates for it. Since CPU **80** knows one of the ports on which a node using this multicast protocol resides, it sets the bit corresponding to that port in the Broadcast Enable field of the newly created network table entry. For example, if the packet arrived on port **5**, then CPU **80** sets bit **4** of the Broadcast Enable field to 1. If an entry already exists, then CPU **80** simply updates the Broadcast Enable field appropriately (for example, if the Broadcast

Enable field had a 1 in bit position **5** and if the packet arrived on port **2**, then bit **1** of the Broadcast Enable field is also set to 1). The VLAN is automatically built in this manner, one step at a time with no administrator intervention.

If the packet is an AppleTalk packet (determined in step **S218**), processing advances to step **S220**. In AppleTalk networks, a rough equivalent of an IP subnetwork is a zone. AppleTalk networks use dynamic node address assignment using a protocol called Apple Address Resolution Protocol (AARP). Addresses of nodes can thus change from time to time. However, names of nodes do not change frequently. AppleTalk Name Binding Protocol (NBP) provides a mechanism for translating names into addresses. One such name to address mapping function involves maintaining a mapping between networks and zone names. This is done in AppleTalk routers through Zone Information Protocol (ZIP). Refer to G. S. Sidhu et al., "Inside AppleTalk," pp. 8-1 through 8-24, for more details regarding this protocol.

Zone Information Protocol (ZIP) provides for the following services: maintenance of network-to-zone-name mapping of the internet; support for selection of a zone name by a node at startup; and support for various commands that may be needed by non-router nodes to obtain this mapping. During startup, an AppleTalk node acquires the name of the zone it belongs to. It can either ask the router to put it in a specific zone or it can ask the router to place it in any zone. This is done through ZIP GetNetinfo requests and responses. A datalink multicast address called zone multicast address is associated with a given zone name on a given data link and is determined by the ZIP process in AppleTalk routers. All the nodes that belong to a particular zone use that multicast address instead of a broadcast address.

Switch engine **100** snoops AppleTalk ZIP packets by detecting GetNetInfo responses and ZIP Notify packets. These packets contain the multicast address associated with the zone. When such packets are detected in step S22, they are forwarded to CPU **80** and processing advances to step **S220**, where CPU **80** checks to see whether flow table entries exist for the multicast address for the zone. If not, it creates an entry for it and automatically adds the sender (which in this case would be the AppleTalk router) and the receiver (which is the end node) to a VLAN based on this multicast address provided that automatic VLAN creation is not disabled. This is done by (step S222) creating an address resolution hash and address resolution record entries for the multicast address if such do not exist already, creating and linking a network table entry thereto, and updating the Broadcast Enable field in the network entry so that subsequent packets addressed to this multicast address are forwarded on the ports which contain these nodes (sender and receiver) by switch engine **100** at wire speed. If it is determined in step S220 that flow table entries already exist for the zone associated with the multicast address, CPU **80** updates the Broadcast Enable field appropriately if necessary.

If the packet is an IP multicast packet, processing advances to step **S226**. VLANs based on IP multicast groups are established using IGMP and IP multicast protocols. The multiprotocol switch of the present invention performs IGMP spoofing in hardware and automatically determines the IP multicast groups to which the nodes attached to the switch belong. The switch creates a VLAN for every IP multicast group. The switch also automatically determines the membership of each multicast group associated with its attached to the switch and maintains the VLAN accordingly. Thus, IP multicast frames will be forwarded only on segments with users registered to receive them so that segments without group members are spared of spurious traffic.

Class D IP addresses (in the range of 224.0.0.0 through 239.255.255.255) are used for IP multicast networks (refer to RFC **1112** for more details). IP multicast addresses are resolved in a manner identical to Ethernet multicast addresses (i.e. the destination address is used to perform address lookup). Internet Group Management Protocol (IGMP) as described in RFC **1112** is used for IP multicast group administration. IGMP uses IP datagrams as shown in figure below. If the protocol carried field in the IP header is **2**, then the data following the IP header contains IGMP packet. Switch engine **100** detects IGMP packets in step S22 and passes them along to CPU **80** as described above.

IGMP type (4 bits)	IGMP version (4 bits)	Unused (8 bits)	Checksum (16 bits)
Class D IP Multicast Group Address (32 bits)			

There are only two types of IGMP messages: reports and queries. End stations send reports to join or maintain membership in a multicast group. Routers send queries to locate active group members. As described above in connection with FIG. **6**, the multiprotocol switch of the present invention, on powerup, joins the "all hosts" multicast group (224.0.0.1) and periodically sends out IGMP queries to the "all hosts" multicast address to determine all the multicast groups that are active on its segments. The group address in this query is set to 0.0.0.0 to indicate that the switch is interested in all active IP multicast groups. When a node receives the query, it will respond with a report indicating the multicast group it belongs to. Nodes also send IGMP reports when they leave the multicast group. During the time a node belongs to a multicast group, all messages it sends to other members of the group are addressed with the IP multicast group address.

Switch engine **100** forwards all IGMP reports and IP messages sent to IP multicast addresses to CPU **80** (step S22). In step **S228**, CPU **80** first determines whether the message is an IGMP report indicating that a node is leaving an existing group. If not, CPU **80** determines whether if low table entries exist for the multicast group associated with the IP multicast address (either indicated within an IGMP report or contained as the destination address of the packet) (step **S230**). If not, CPU **80** creates (step S232) or updates (step S234) the VLAN for this IP multicast address, provided that automatic VLAN creation is not disabled. This is done by creating address resolution hash and address resolution record entries for that IP multicast address (if one has not already been created), creating a network table entry and linking it to the address resolution record entry (if not done already), and updating the Broadcast Enable field. Thereafter, multicast packets from nodes in the IP multicast group using that multicast address as a destination are forwarded on all the other ports which have nodes that belong to that IP multicast group at wire speed by switch engine **100**. If the packet is an IGMP report indicating that a node is leaving the multicast group, CPU **80** retrieves the network entry associated with that multicast group and clears the bit corresponding to the port on which that node is connected.

Although the present invention has been described in detail hereinabove with reference to the preferred embodiments thereof, those skilled in the art will appreciate that various substitutions and modifications can be made to the examples provided while remaining within the spirit and

scope of the invention as defined in the appended claims. For example, many processing steps have been described that could be ordered or grouped together differently for efficiency, or could be performed in parallel depending on implementation. Those skilled in the art will realize that such different implementations are possible after being taught by the present invention. Accordingly, the examples of the invention provided herein should be seen as illustrative rather than limiting.

I claim:

1. A method for forwarding data packets in a switch having a plurality of ports, said ports being adapted to transmit and receive data packets from hosts coupled thereto, said method comprising:

receiving a first data packet at a first port;

firstly extracting flow identification information from said first data packet, said flow identification information indicating that said first data packet belongs to a first flow between certain of said hosts;

corresponding said flow identification information with stored configuration information;

creating flow processing information based on said corresponding stored configuration information, said flow processing information comprising at least one of a filter, a mirror, and a priority, said flow processing information further indicating at least a second port, wherein all data packets belonging to said first flow are forwarded between said first and second ports, said first and second ports being associated with said certain hosts;

storing a first record in a flow table containing said flow processing information;

receiving a second data packet at one of said first and second ports;

secondly extracting said flow identification information from said second data packet;

locating said stored first record in said flow table based on said extracted flow identification information; and

preparing to forward said second data packet to the other of said first and second ports based on said flow processing information contained in said stored first record,

wherein said steps of firstly and secondly extracting said flow identification information each includes:

determining a protocol associated with said first flow;

if said protocol is IP or IPX, extracting Layer 3 header information as said flow identification information; and

if said protocol is not IP or IPX, extracting Layer 2 header information as said flow identification information.

2. A method as defined in claim 1, further comprising:

preventing said second data packet from being forwarded to the other of said first and second ports in accordance with said filter.

3. A method as defined in claim 1, further comprising:

forwarding said second data packet to the other of said first and second ports; and

forwarding said second data packet on certain other of said ports in accordance with said mirror.

4. A method as defined in claim 1, further comprising:

forwarding said second data packet to the other of said first and second ports; and

prioritizing transmission of said second data packet to hosts associated with the other of said first and second ports in accordance with said priority.

5. A method of forwarding data packets between a plurality of switch ports, said method comprising:

receiving a data packet at one of said switch ports;

corresponding flow identification information in said data packet with stored flow processing information, said flow identification information including source and destination addresses, source and destination sockets, and a protocol, said stored flow processing information including a destination port, a filter tag and a broadcast enable, said corresponding including:

locating an address resolution hash record in an address resolution hash table according to a portion of said flow identification information, said address resolution hash record storing a link to an address resolution record;

locating said address resolution record in an address resolution record table in accordance with said address resolution hash record, said address resolution record storing said destination port and a link to a protocol entry;

locating said protocol entry in a protocol entry table in accordance with said address resolution record and said protocol, said protocol entry storing a link to said filter tag and a link to a network entry;

locating said network entry in a network entry table according to said protocol entry, said network entry storing said broadcast enable; and

locating said filter tag in a list of filter tags in accordance with said protocol entry; and

forwarding said data packet in accordance with said stored flow processing information.

6. A method according to claim 5, wherein said stored flow processing information further includes a priority, said protocol entry further storing said priority.

7. A method according to claim 5, further comprising alerting a CPU if no stored flow processing information corresponds to said flow identification information of said data packet.

8. A method of forwarding data packets between a plurality of switch ports, said method comprising:

receiving a data packet at one of said switch ports;

corresponding flow identification information in said data packet with stored flow processing information, said flow identification information including source and destination addresses, source and destination sockets, and a protocol, said stored flow processing information including a destination port, a filter tag and a broadcast enable, said corresponding including:

locating an address resolution hash record in an address resolution hash table according to a portion of said flow identification information, said address resolution hash record storing a link to an address resolution record;

locating said address resolution record in an address resolution record table in accordance with said address resolution hash record, said address resolution record storing said destination port and a link to a protocol entry;

locating said protocol entry in a protocol entry table in accordance with said address resolution record and said protocol, said protocol entry storing a link to said filter tag and a link to a network entry;

locating said network entry in a network entry table according to said protocol entry, said network entry storing said broadcast enable; and

locating said filter tag in a list of filter tags in accordance with said protocol entry; and

forwarding said data packet in accordance with said stored flow processing information,

wherein said stored flow processing information further includes a mirror tag, said protocol entry further storing a link to said mirror tag, said method further comprising locating said mirror tag in a list of mirror tags in accordance with said protocol entry.

9. A method of forwarding data packets between a plurality of switch ports, said method comprising:

receiving a data packet at one of said switch ports;

corresponding flow identification information in said data packet with stored flow processing information, said flow identification information including source and destination addresses, source and destination sockets, and a protocol, said stored flow processing information including a destination port, a filter tag and a broadcast enable, said corresponding including:

locating an address resolution hash record in an address resolution hash table according to a portion of said flow identification information, said address resolution hash record storing a link to an address resolution record;

locating said address resolution record in an address resolution record table in accordance with said address resolution hash record, said address resolution record storing said destination port and a link to a protocol entry;

locating said protocol entry in a protocol entry table in accordance with said address resolution record and said protocol, said protocol entry storing a link to said filter tag and a link to a network entry;

locating said network entry in a network entry table according to said protocol entry, said network entry storing said broadcast enable; and

locating said filter tag in a list of filter tags in accordance with said protocol entry; and

forwarding said data packet in accordance with said stored flow processing information,

wherein said stored flow processing information further includes address swapping bits, said network entry further storing said address swapping bits.

10. A method of forwarding data packets between a plurality of switch ports, said method comprising:

receiving a data packet at one of said switch ports;

corresponding flow identification information in said data packet with stored flow processing information, said flow identification information including source and destination addresses, source and destination sockets, and a protocol, said stored flow processing information including a destination port, a filter tag and a broadcast enable, said corresponding including:

locating an address resolution hash record in an address resolution hash table according to a portion of said flow identification information, said address resolution hash record storing a link to an address resolution record;

locating said address resolution record in an address resolution record table in accordance with said address resolution hash record, said address resolution record storing said destination port and a link to a protocol entry;

locating said protocol entry in a protocol entry table in accordance with said address resolution record and said protocol, said protocol entry storing a link to said filter tag and a link to a network entry;

locating said network entry in a network entry table according to said protocol entry, said network entry storing said broadcast enable; and

locating said filter tag in a list of filter tags in accordance with said protocol entry; and

forwarding said data packet in accordance with said stored flow processing information,

wherein said forwarding step includes routing said data packet to another network by swapping said source and destination addresses of said data packet in accordance with said address swapping bits.

11. An apparatus for forwarding data packets in a switch having a plurality of ports, said ports being adapted to transmit and receive data packets from hosts coupled thereto, said apparatus comprising:

means for receiving a first data packet at a first port;

means for firstly extracting flow identification information from said first data packet, said flow identification information indicating that said first data packet belongs to a first flow between certain of said hosts;

means for corresponding said flow identification information with stored configuration information;

means for creating flow processing information based on said corresponding stored configuration information, said flow processing information comprising at least one of a filter, a mirror, and a priority, said flow processing information further indicating at least a second port, wherein all data packets belonging to said first flow are forwarded between said first and second ports, said first and second ports being associated with said certain hosts;

means for storing a first record in a flow table containing said flow processing information;

means for receiving a second data packet at one of said first and second ports;

means for secondly extracting said flow identification information from said second data packet;

means for locating said stored first record in said flow table based on said extracted flow identification information; and

means for preparing to forward said second data packet to the other of said first and second ports based on said flow processing information contained in said stored first record,

wherein said means for firstly and secondly extracting said flow identification information each includes:

means for determining a protocol associated with said first flow;

means, operative if said protocol is IP or IPX, for extracting Layer 3 header information as said flow identification information; and

means, operative if said protocol is not IP or IPX, for extracting Layer 2 header information as said flow identification information.

12. An apparatus as defined in claim 11, further comprising:

means for preventing said second data packet from being forwarded to the other of said first and second ports in accordance with said filter.

13. An apparatus as defined in claim 11, further comprising:

means for forwarding said second data packet to the other of said first and second ports; and

means for forwarding said second data packet on certain other of said ports in accordance with said mirror.

14. An apparatus as defined in claim 11, further comprising:

means for forwarding said second data packet to the other of said first and second ports; and

means for prioritizing transmission of said second data packet to hosts associated with the other of said first and second ports in accordance with said priority.

* * * * *

US006452915B1

(12) **United States Patent**
Jorgensen

(10) **Patent No.: US 6,452,915 B1**
(45) **Date of Patent: Sep. 17, 2002**

(54) **IP-FLOW CLASSIFICATION IN A WIRELESS POINT TO MULTI-POINT (PTMP) TRANSMISSION SYSTEM**

(75) Inventor: **Jacob W. Jorgensen**, Folsom, CA (US)

(73) Assignee: **Malibu Networks, Inc.**, El Dorado Hills, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/350,156**

(22) Filed: **Jul. 9, 1999**

Related U.S. Application Data

(60) Provisional application No. 60/092,452, filed on Jul. 10, 1998.

(51) **Int. Cl.**[7] **H04Q 7/24**
(52) **U.S. Cl.** **370/338**; 370/328
(58) **Field of Search** 370/338, 328, 370/329, 230, 229, 231, 465, 466, 468, 232, 235; 455/455, 575, 525, 422, 426, 430, 437, 440, 11.1, 456, 404

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,742,512 A	5/1988	Akashi et al.	
4,907,224 A	3/1990	Scoles et al.	370/85.2
5,282,222 A	1/1994	Fattouche et al.	

(List continued on next page.)

FOREIGN PATENT DOCUMENTS

CA	2064975	7/1999	
EP	702 462 A1	3/1996	H04B/7/08
EP	841 763 A1	5/1998	H04B/7/26
EP	848 563 A2	6/1998	H04Q/7/20

(List continued on next page.)

OTHER PUBLICATIONS

Bianchi, et al. "C–PRMA: A Centralized Packet Reservation Multiple Access for Local Wiresless Communications" in IEEE Transactions on Vehicular Technology, vol. 46, No. 2 pp. 422–436, May 1997.

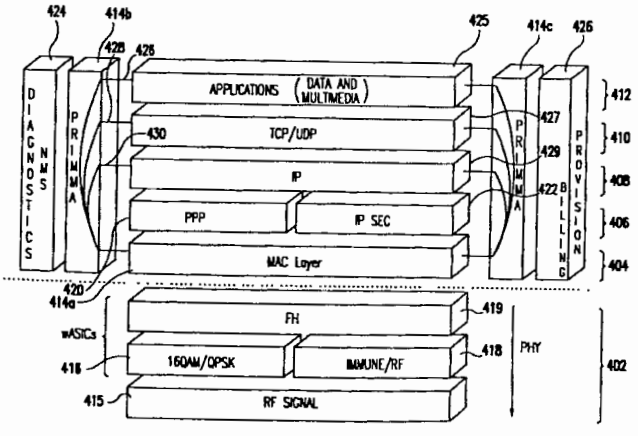
(List continued on next page.)

Primary Examiner—Douglas Olms
Assistant Examiner—Ricardo M. Pizarro
(74) *Attorney, Agent, or Firm*—Venable; Ralph P. Albrecht

(57) **ABSTRACT**

An IP flow classification system is used in a wireless telecommunications system. The IP flow classification system groups IP flows in a packet-centric wireless point to multi-point telecommunications system. The classification system includes: a wireless base station coupled to a first data network; one or more host workstations coupled to the first data network; one or more subscriber customer premise equipment (CPE) stations in wireless communication with the wireless base station over a shared bandwidth using a packet-centric protocol; and one or more subscriber workstations coupled to each of the subscriber CPE stations over a second network; a resource allocation device optimizes end-user quality of service (QoS) and allocates shared bandwidth among the subscriber CPE stations; an analyzing and scheduling device analyzes and schedules internet protocol (IP) flow over the shared wireless bandwidth. The analyzing device includes the above IP flow classifier that classifies the IP flow. The classifier can include a device for associating a packet of an existing IP flow with the IP flow. The classifier can include a QoS grouping device that groups a packet of a new IP flow into a QoS class grouping. The QoS grouping device can include a determining device that determines and takes into account QoS class groupings for the IP flow. The QoS grouping device can include an optional differentiated services (Diff Serv) device that takes into account an optional Diff Servs field priority marking for the IP flow.

163 Claims, 41 Drawing Sheets



U.S. PATENT DOCUMENTS

5,420,851	A		5/1995	Seshadri et al. 370/29
5,493,569	A		2/1996	Buchholz et al. 370/85.7
5,497,504	A		3/1996	Acampora et al.
5,499,243	A		3/1996	Hall
5,515,363	A		5/1996	Ben-Nun et al.
5,581,544	A		12/1996	Hamad et al.
5,602,836	A		2/1997	Papadopoulos et al. 370/280
5,613,198	A		3/1997	Ahmadi et al.
5,648,969	A		7/1997	Pasternak et al.
5,717,689	A		2/1998	Ayanoglu
5,724,513	A		3/1998	Ben-Nun et al.
5,729,542	A		3/1998	Dupont
5,732,077	A		3/1998	Whitehead
5,742,847	A		4/1998	Knoll et al.
5,752,193	A		5/1998	Scholefield et al.
5,757,708	A		5/1998	Eng et al.
5,787,077	A		7/1998	Kuehnel et al.
5,787,080	A		7/1998	Hulyalkar et al.
5,793,416	A		8/1998	Rostoker et al.
5,828,677	A		10/1998	Sayeed et al.
5,831,971	A		11/1998	Bonomi et al.
5,838,670	A		11/1998	Billström
5,841,777	A		11/1998	Cohen
5,864,540	A		1/1999	Bonomi et al.
5,907,822	A		5/1999	Prieto, Jr.
5,930,472	A	*	7/1999	Smith 713/200
5,936,949	A		8/1999	Pasternak et al.
5,956,330	A		9/1999	Kerns
5,970,059	A		10/1999	Ahopelto et al.
5,970,062	A		10/1999	Bauchot
5,974,028	A		10/1999	Ramakrishnan
5,974,085	A	*	10/1999	Smith 375/222
6,002,935	A		12/1999	Wang
6,005,868	A		12/1999	Ito
6,016,311	A		1/2000	Gibert et al.
6,021,158	A		2/2000	Schurr et al.
6,031,832	A		2/2000	Turina
6,031,845	A		2/2000	Walding
6,038,230	A		3/2000	Ofek
6,038,452	A		3/2000	Strawczynski
6,041,051	A		3/2000	Doshi et al.
6,046,980	A	*	4/2000	Packer 370/230
6,052,594	A		4/2000	Chuang et al.
6,058,114	A		5/2000	Sethuram et al.
6,064,649	A		5/2000	Johnston
6,075,787	A		6/2000	Bobeck et al.
6,075,792	A		6/2000	Ozluturk
6,081,536	A		6/2000	Gorsuch et al.
6,084,867	A		7/2000	Meier
6,091,959	A	*	7/2000	Soussi 455/456
6,092,113	A		7/2000	Maeshima
6,097,722	A		8/2000	Graham et al.
6,097,733	A		8/2000	Basu et al.
6,104,721	A		8/2000	Hsu
6,111,863	A		8/2000	Rostoker et al.
6,115,357	A		9/2000	Packer et al.
6,115,370	A		9/2000	Struhsaker et al.
6,115,390	A		9/2000	Chuah
6,151,300	A		11/2000	Hunt et al.
6,151,628	A		11/2000	Xu et al.
6,154,643	A		11/2000	Cox
6,160,793	A		12/2000	Ghani et al.
6,163,532	A		12/2000	Taguchi et al.
6,195,565	B1		2/2001	Dempsey et al.
6,208,620	B1		3/2001	Sen et al.
6,215,769	B1		4/2001	Ghani et al.
6,219,713	B1		4/2001	Ruutu et al.
6,363,209	B2	*	7/2001	Reed 455/456
6,272,333	B1	*	8/2001	Smith 455/456
6,304,564	B1		10/2001	Monin et al.
6,320,846	B1		11/2001	Jamp et al.
6,330,451	B1		12/2001	Sen et al.

FOREIGN PATENT DOCUMENTS

EP	917 317 A1	5/1999	..... H04L/12/28
WO	WO 96/10320	4/1996	..... H04Q/7/22
WO	WO 98/37670	8/1998	..... H04L/12/56
WO	WO 99/26430	5/1999	..... H04Q/7/20
WO	WO 00722626	11/2000	
WO	WO 00/79722	12/2000	

OTHER PUBLICATIONS

Kim et al. "The AT&T Labs Broadband Fixed Wireless Field Experiment", IEEE Communications Magazine, Oct. 1999, pp. 56–62.

Iera et al. "Wireless Broadband Applications: The Teleservice Model and Adaptive QoS Provisioning", IEEE Communications Magazine, Oct. 1999, pp. 71–75.

Celidonio et al. "A Wedeband Two–Layer Radio Access Network Using DECT Technology in the Uplink", IEEE Communications Magazine, Oct. 1999, pp. 76–81.

Yoon et al. "A Wireless Local Loop System Based on Wideband CDMA Technology", IEEE Communications Magazine, Oct. 1999, pp. 128–135.

Balakrishman et al. "Improving Reliable Transprot and Handoff Performance in Cellular Wireless Networks", http://www.cs.berkely.edu/~ss/papers/wunet/html/winet.hml., Computer Science Div., Dept. of Electrical Engineering and Computer Science, Univ. of California at Berkeley, Berkeley, CA 94720–1776, Nov. 1995, pp. 1–18.

"A Cellular Wireless Local Area Network with QoS Guarantees for Heterogeneous Traffic", Authur(s): Sunghyun Choi and Kang G. Shin, *Technical Report CSE–TR–300–96*, Aug. 1996, pp. 1–24.

"The GSM System", Authors: Michel Mouly, Marie–Bernadette Pautet, pp. 272–277, XP–002154762.

"A Comparison of Mechanisms for Improving TCP Performance over Wireless Links" Author(s): Hari Balakrishnan, Venkata N. Padmanabhan, Srinvasan Seshan, and Randy H. Katz; XF000734405 *IEEE/ACM Transactions on Networking*, vol. 5, No. 6, Dec. 1997, pp. 756–769.

"Improving TCP/IP Performance Over Wireless Networks"; Author(s): Hari Balakrishnan, Srinivasan Seshan, Elan Amire and Randy H. Katz; *In Proc. 1st ACM Int'l Conf. On Mobile Computing and Networking (Mobicom)*, Nov. 1995, XP–002920962.

International Search Report; Date: Dec. 14, 2000; International Appln. No. PCT/US 00/18531 for (36792–164878).

International Search Report; Date: Feb. 14, 2000; International Appln. No. PCT/US 00/18584 for (36792–164879).

International Search Report; Date: Dec. 14, 2000; International Appln. No. PCT/US 00/18585 for (36792–164880).

International Search Report; Date: Dec. 22, 2000; International Appln. No. PCT/US 00/18666 for (36792–164881).

* cited by examiner

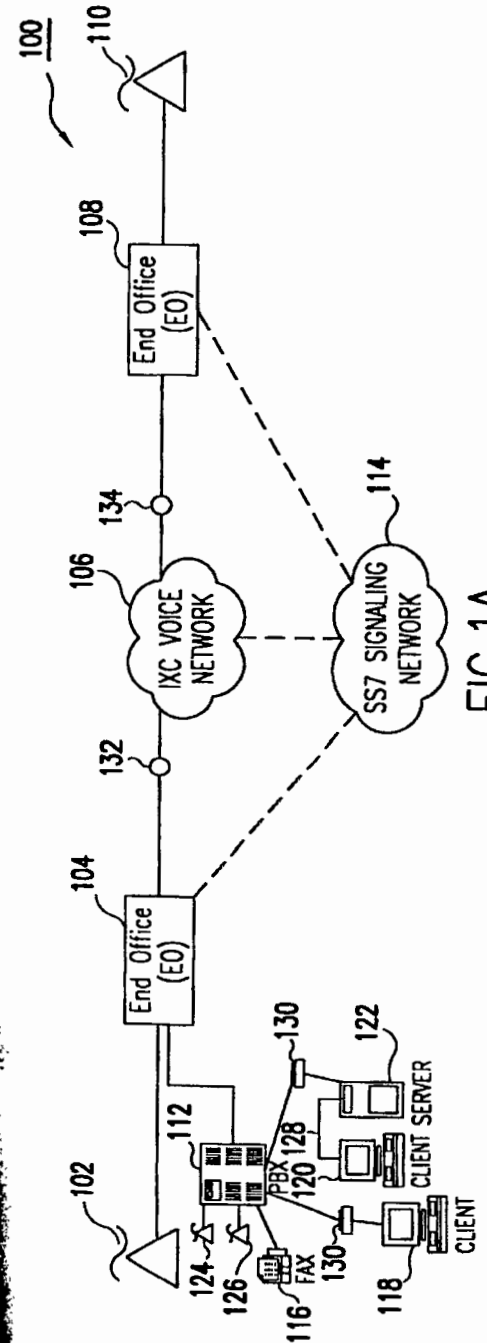


FIG. 1A

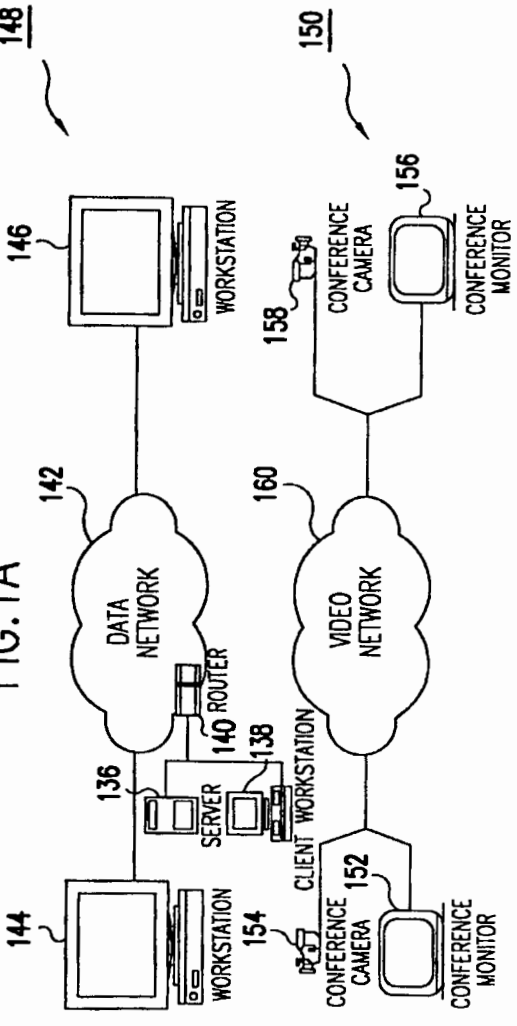


FIG. 1B

FIG. 1C

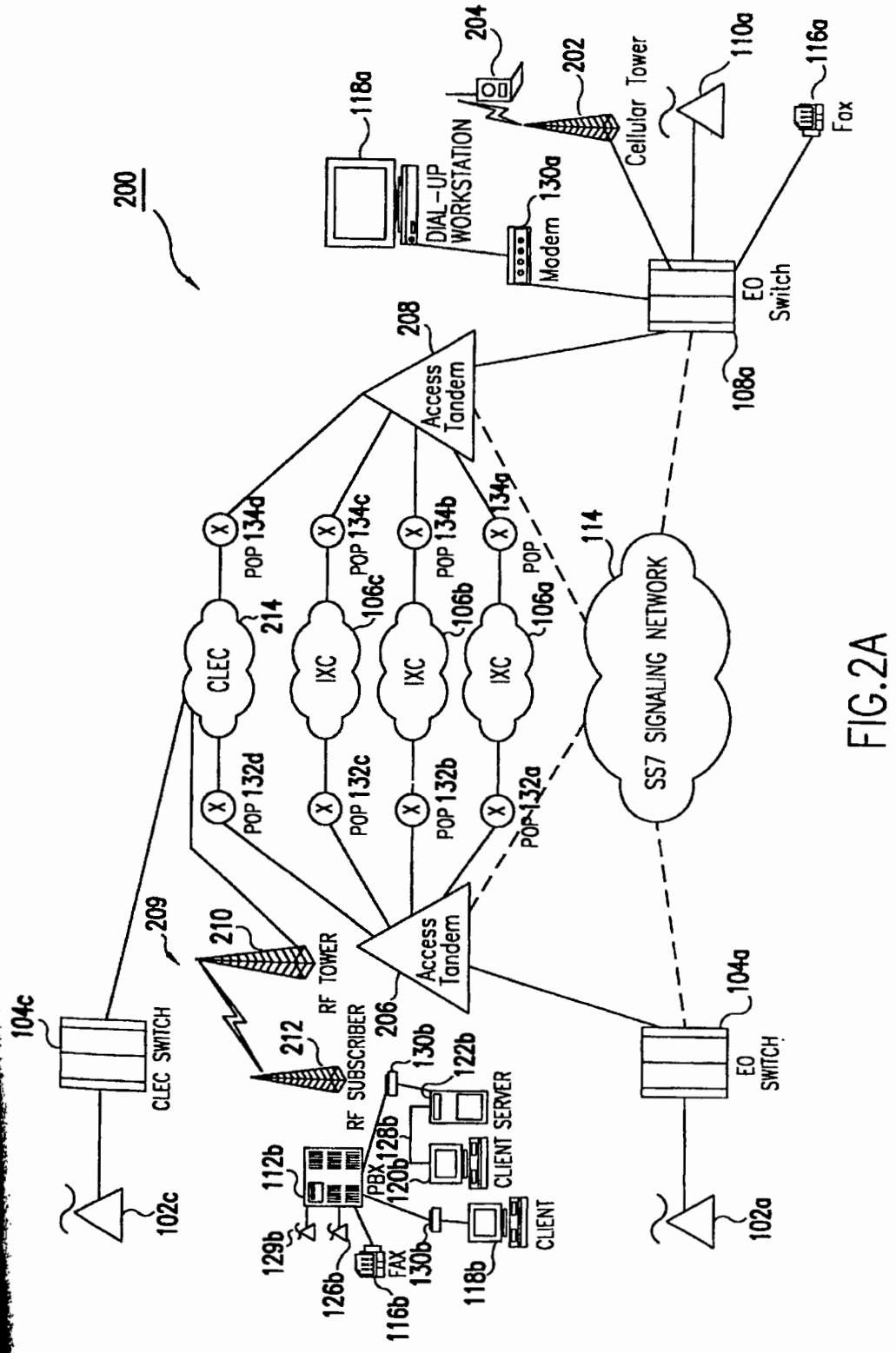

200
118a
DIAL-UP WORKSTATION
204
202
Cellular Tower
110a
116a
Fax
Modem 130a
EO Switch
108a
208
Access Tandem
POP 134d
POP 134c
POP 134b
POP 134a
114
SS7 SIGNALING NETWORK
214
CLEC
IXC
IXC
IXC
106c
106b
106a
POP 132d
POP 132c
POP 132b
POP 132a
209
210
RF TOWER
212
RF SUBSCRIBER
206
Access Tandem
104c
CLEC SWITCH
104a
EO SWITCH
130b
122b
CLIENT SERVER
112b
PBX
120b
128b
CLIENT
102c
102a
129b
126b
FAX
130b
116b
118b

FIG.2A

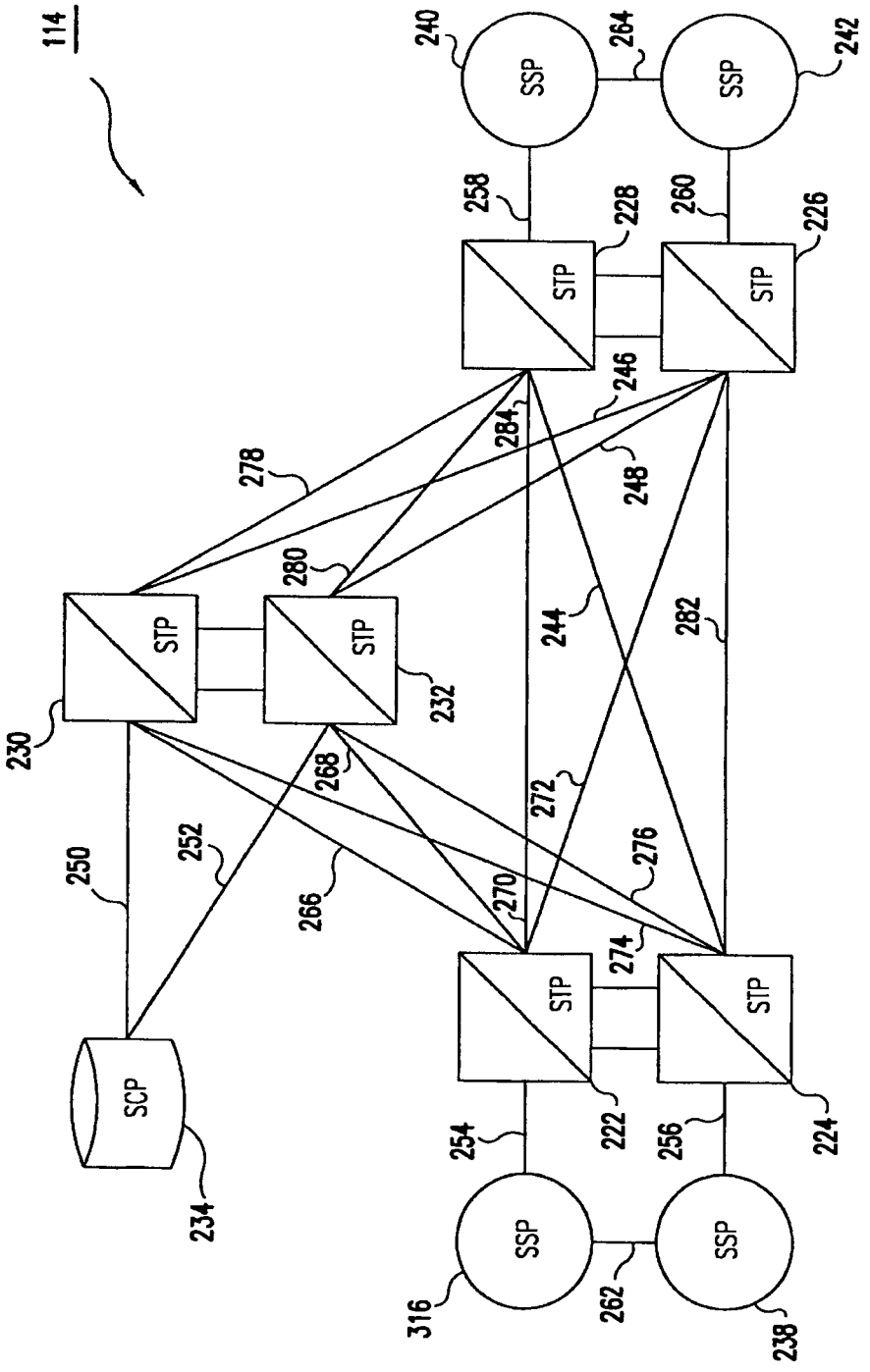

114
240
264
242
SSP
SSP
258
228
260
226
STP
STP
246
284
278
248
280
STP
STP
232
230
244
282
268
272
250
252
276
266
270
274
STP
STP
SCP
234
254
222
256
224
SSP
SSP
316
262
238

FIG.2B

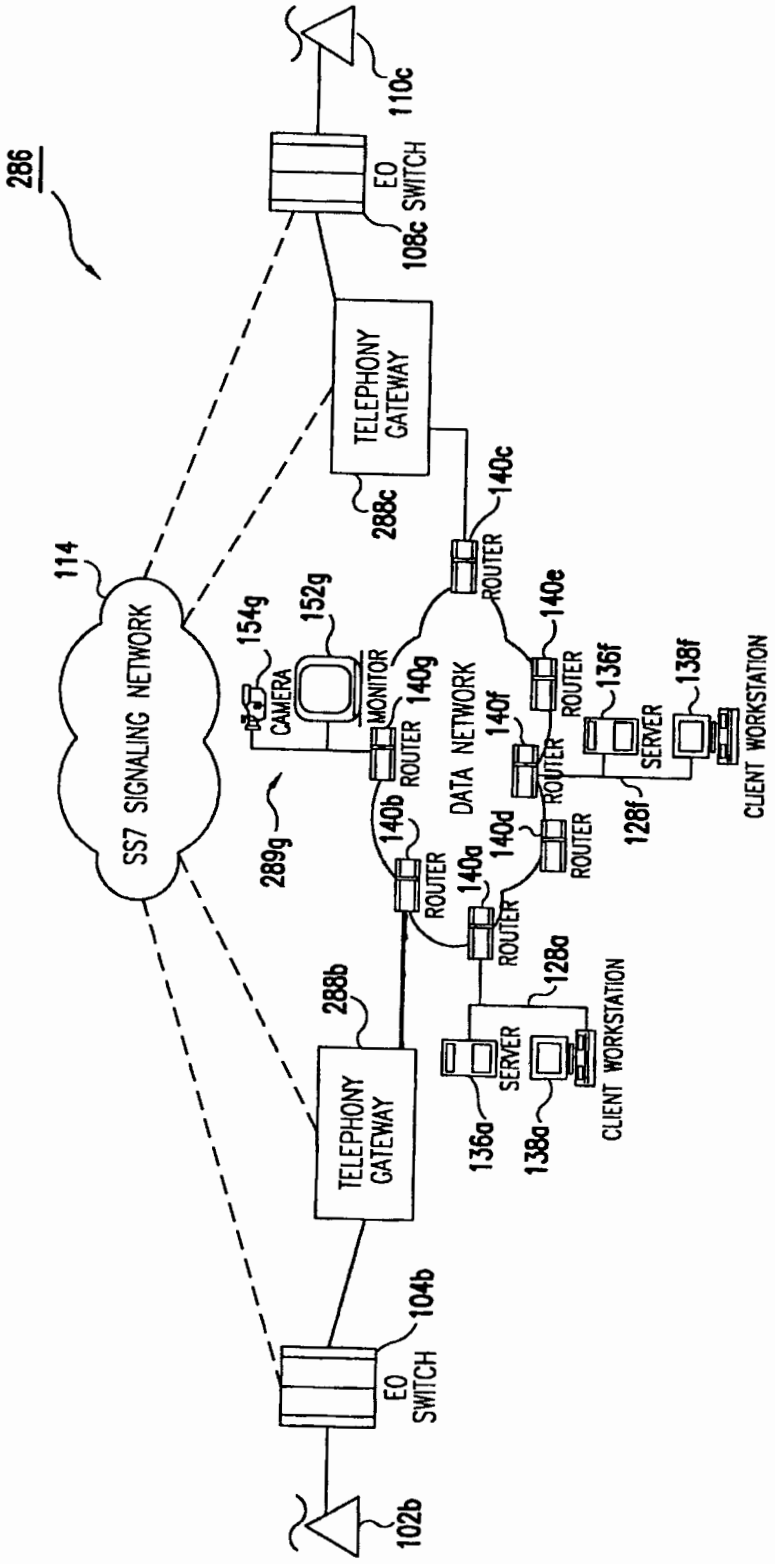

286
110c
EO SWITCH
108c
TELEPHONY GATEWAY
288c
140c
ROUTER
114
SS7 SIGNALING NETWORK
154g
CAMERA
152g
MONITOR
140g
ROUTER
140e
ROUTER
136f
SERVER
138f
CLIENT WORKSTATION
DATA NETWORK
140f
ROUTER
289g
140b
ROUTER
140d
ROUTER
128f
140a
ROUTER
128a
288b
SERVER
CLIENT WORKSTATION
136a
138a
TELEPHONY GATEWAY
104b
EO SWITCH
102b

FIG.2C

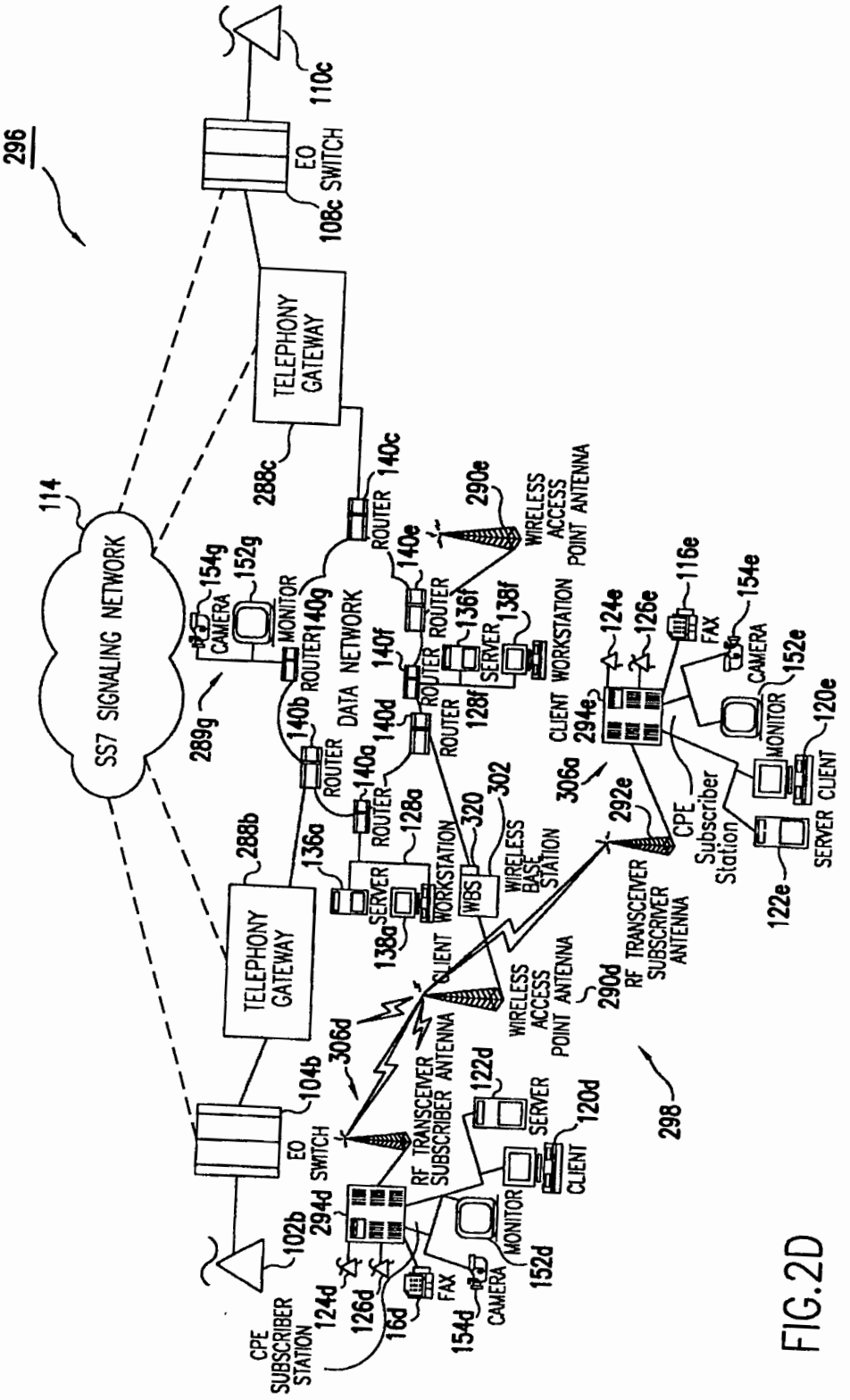

296
114
SS7 SIGNALING NETWORK
110c
EO SWITCH
108c
TELEPHONY GATEWAY
288c
289g
CAMERA
154g
152g
MONITOR
ROUTER
140g
DATA NETWORK
140c
ROUTER
140e
ROUTER
140f
ROUTER
140d
ROUTER
140b
ROUTER
140a
ROUTER
290e
WIRELESS ACCESS POINT ANTENNA
136f
SERVER
138f
WORKSTATION
128f
128a
136a
SERVER
138a
WORKSTATION
WBS
320
302
WIRELESS BASE STATION
288b
TELEPHONY GATEWAY
306d
CLIENT
WIRELESS ACCESS POINT ANTENNA
290d
RF TRANSCEIVER SUBSCRIVER ANTENNA
292e
CPE Subscriber Station
306a
CLIENT
294e
124e
126e
116e
FAX
154e
CAMERA
152e
MONITOR
120e
CLIENT
SERVER
122e
104b
EO SWITCH
RF TRANSCEIVER SUBSCRIBER ANTENNA
122d
SERVER
120d
CLIENT
298
102b
294d
MONITOR
152d
CAMERA
154d
FAX
124d
126d
16d
CPE SUBSCRIBER STATION

FIG.2D

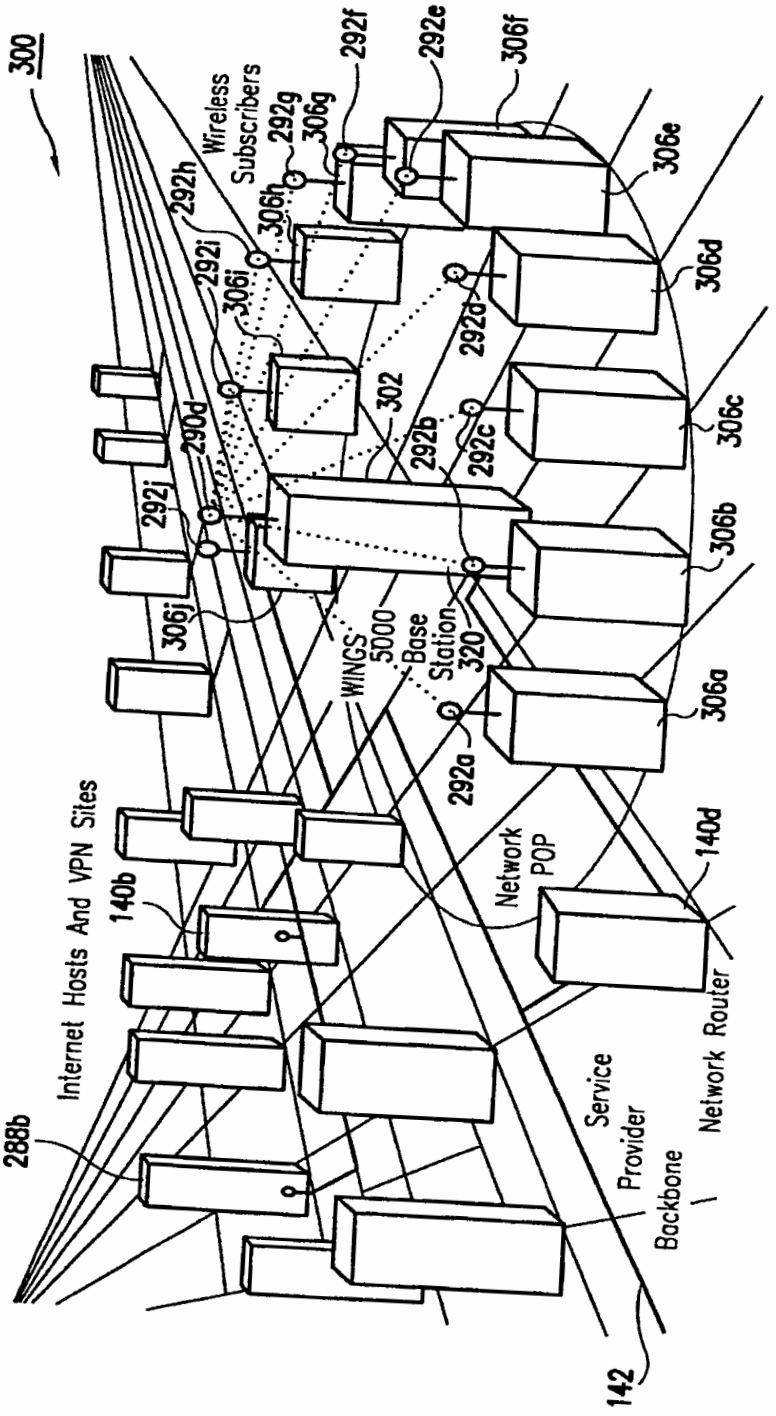

300
Wireless Subscribers
292f
292e
306f
292g
306g
306e
292h
306h
306d
292i
306i
292d
290d
302
306c
292b
292c
292j
306b
306j
WINGS 5000 Base Station 320
306a
292a
Internet Hosts And VPN Sites
140b
Network POP
140d
288b
Service Provider Backbone
Network Router
142

FIG. 3A

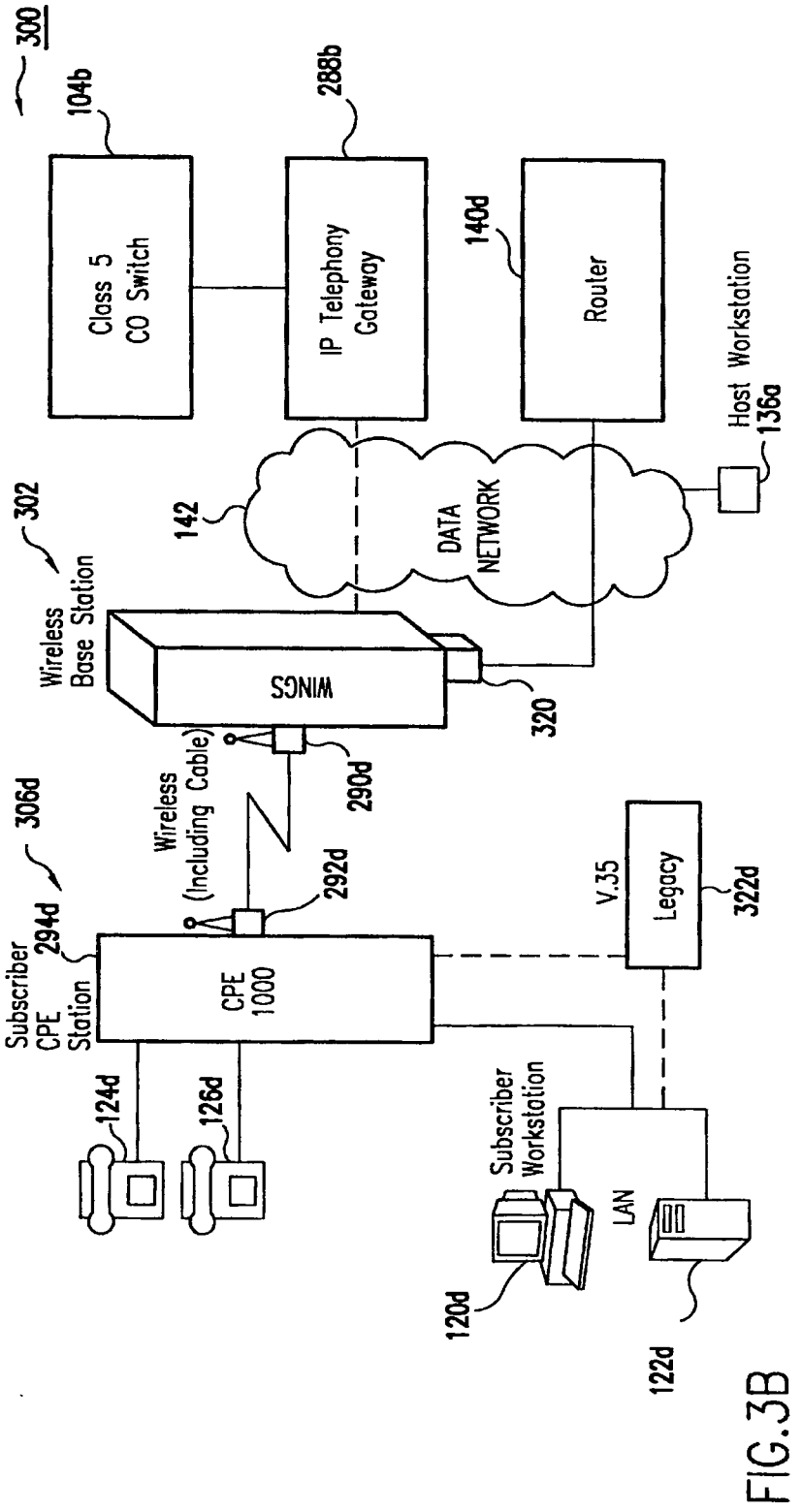

300
104b
Class 5
CO Switch
288b
IP Telephony
Gateway
140d
Router
Host Workstation
136a
142
DATA
NETWORK
302
Wireless
Base Station
WINCS
320
290d
Wireless
(Including Cable)
292d
306d
294d
Subscriber
CPE
Station
CPE
1000
V.35
Legacy
322d
124d
126d
Subscriber
Workstation
LAN
120d
122d

FIG.3B

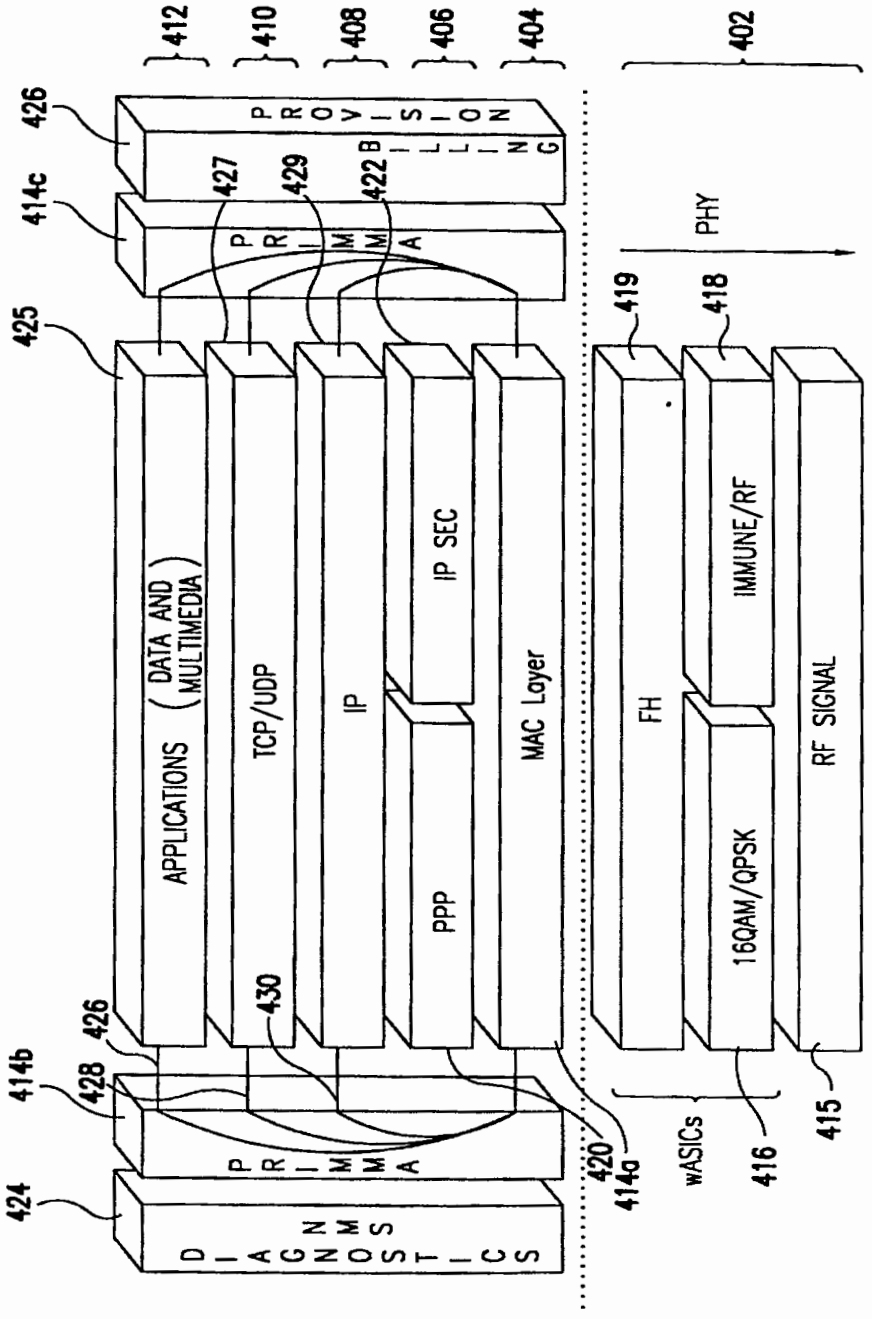

412
410
408
406
404
402
426
414c
425
427
429
422
PROVISIONING
PRIMMA
PHY
419
418
APPLICATIONS (DATA AND MULTIMEDIA)
TCP/UDP
IP
IP SEC
MAC Layer
FH
IMMUNE/RF
RF SIGNAL
430
426
414b
428
PPP
16QAM/QPSK
420
414a
wASICs
416
415
424
DIAGNOSTICS
PRIMMA

FIG. 4

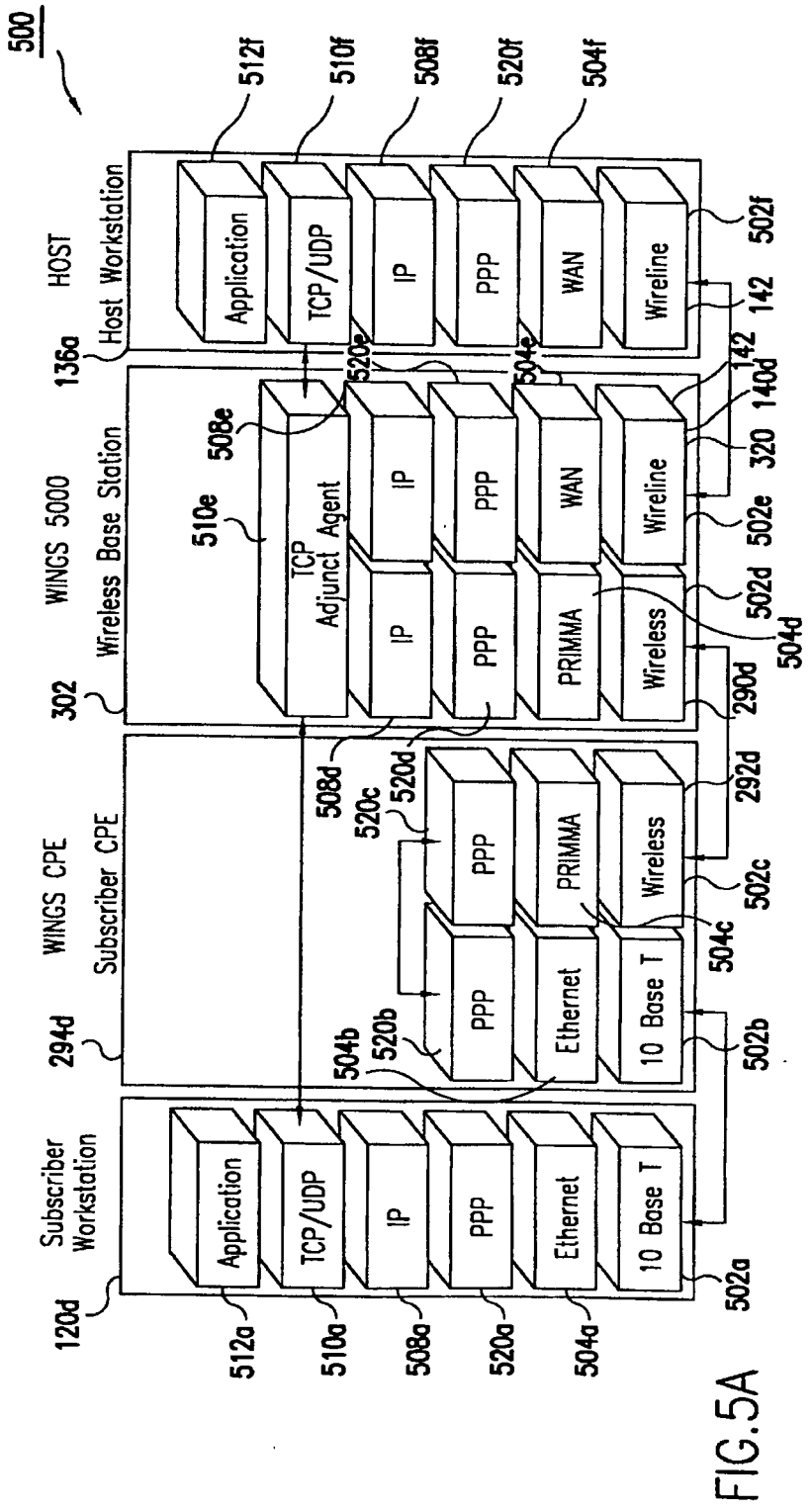

500
512f
510f
508f
520f
504f
Application
TCP/UDP
IP
PPP
WAN
Wireline
HOST
Host Workstation
136a
502f
142
140d
320
502e
WINGS 5000
Wireless Base Station
302
TCP Adjunct Agent
510e
508e
520e
504e
PRIMMA
Wireless
502d
504d
290d
WINGS CPE
Subscriber CPE
294d
508d
520c
520d
292d
502c
504c
Ethernet
10 Base T
504b
520b
502b
Subscriber Workstation
120d
502a
512a
510a
508a
520a
504a

FIG.5A

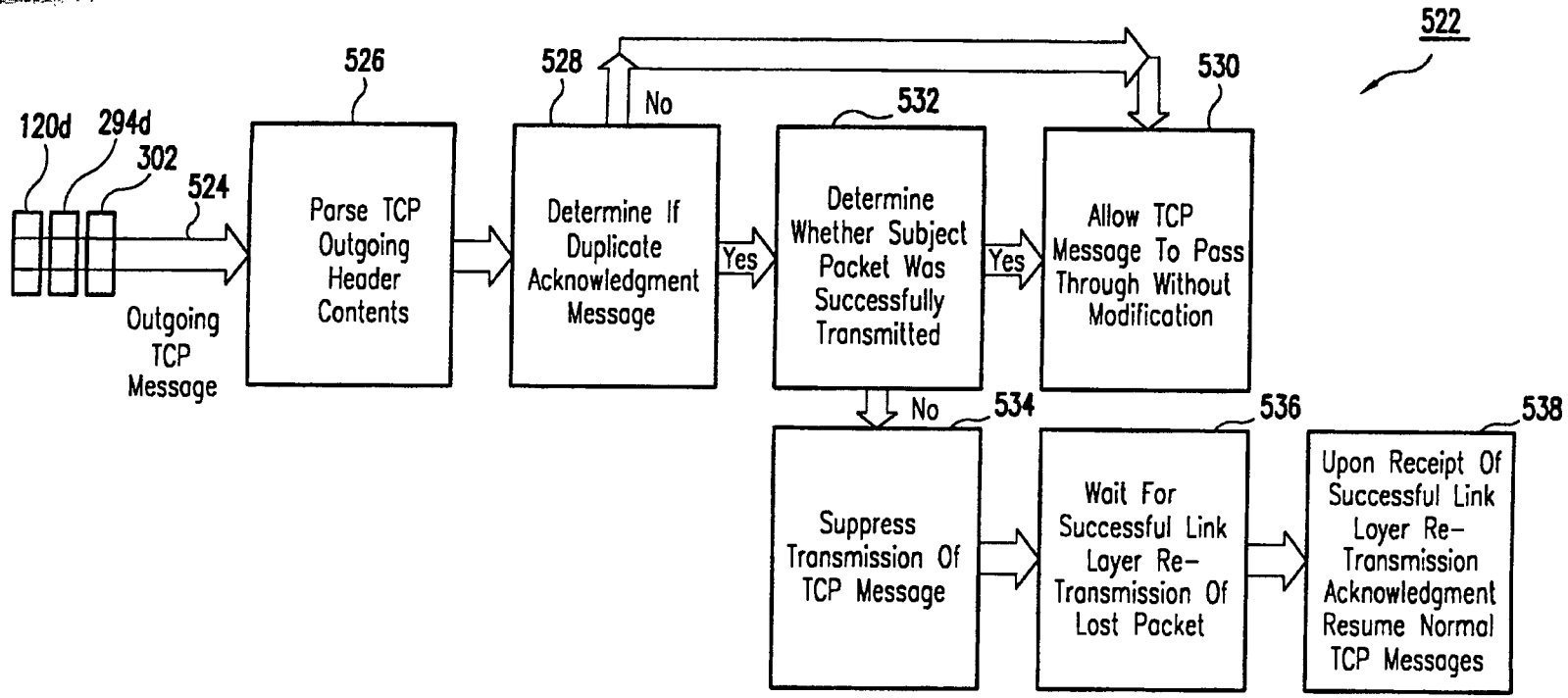

522
526
528
530
532
534
536
538
120d
294d
302
524
No
Yes
Yes
No
Outgoing TCP Message
Parse TCP Outgoing Header Contents
Determine If Duplicate Acknowledgment Message
Determine Whether Subject Packet Was Successfully Transmitted
Allow TCP Message To Pass Through Without Modification
Suppress Transmission Of TCP Message
Wait For Successful Link Layer Re-Transmission Of Lost Packet
Upon Receipt Of Successful Link Layer Re-Transmission Acknowledgment Resume Normal TCP Messages

FIG.5B

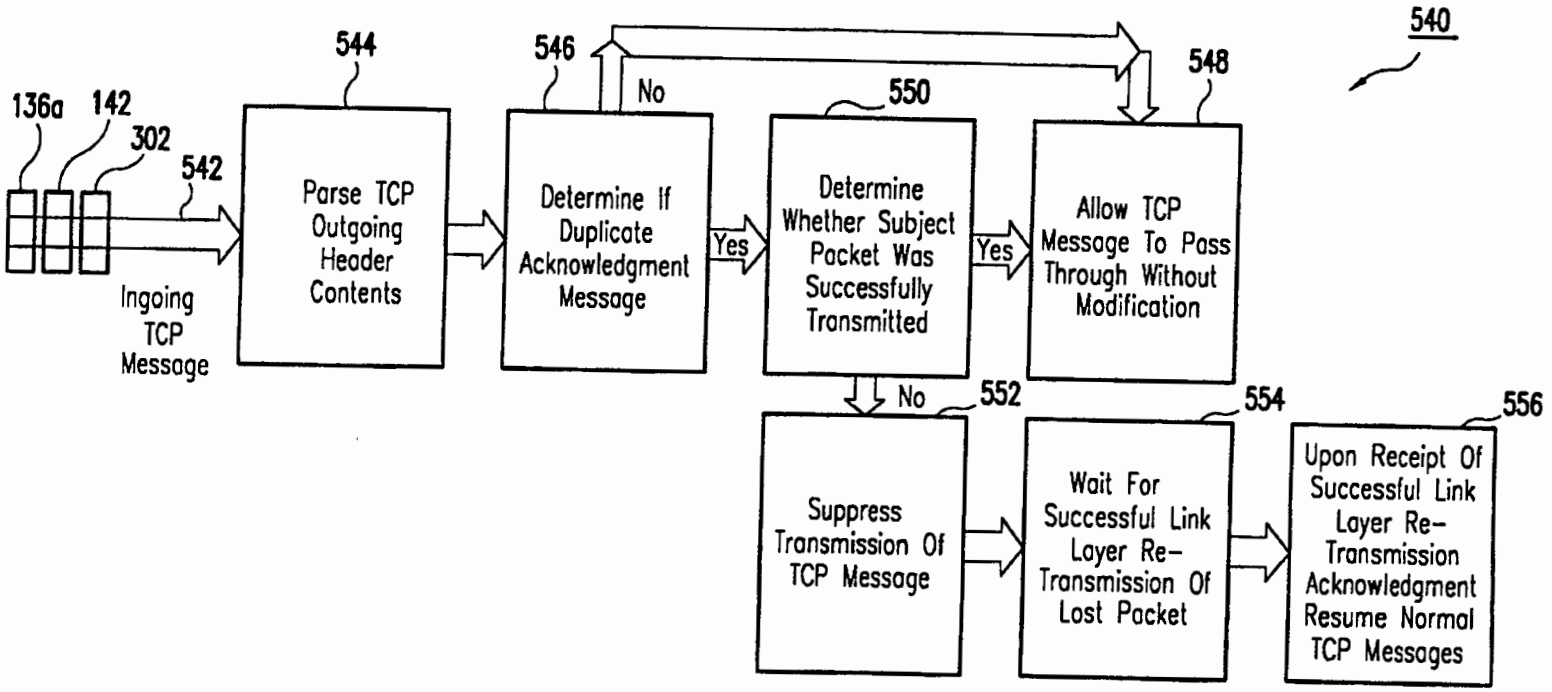

540
136a
142
302
542
Ingoing TCP Message
544
Parse TCP Outgoing Header Contents
546
Determine If Duplicate Acknowledgment Message
No
Yes
550
Determine Whether Subject Packet Was Successfully Transmitted
Yes
548
Allow TCP Message To Pass Through Without Modification
No
552
Suppress Transmission Of TCP Message
554
Wait For Successful Link Layer Re-Transmission Of Lost Packet
556
Upon Receipt Of Successful Link Layer Re-Transmission Acknowledgment Resume Normal TCP Messages

FIG.5C

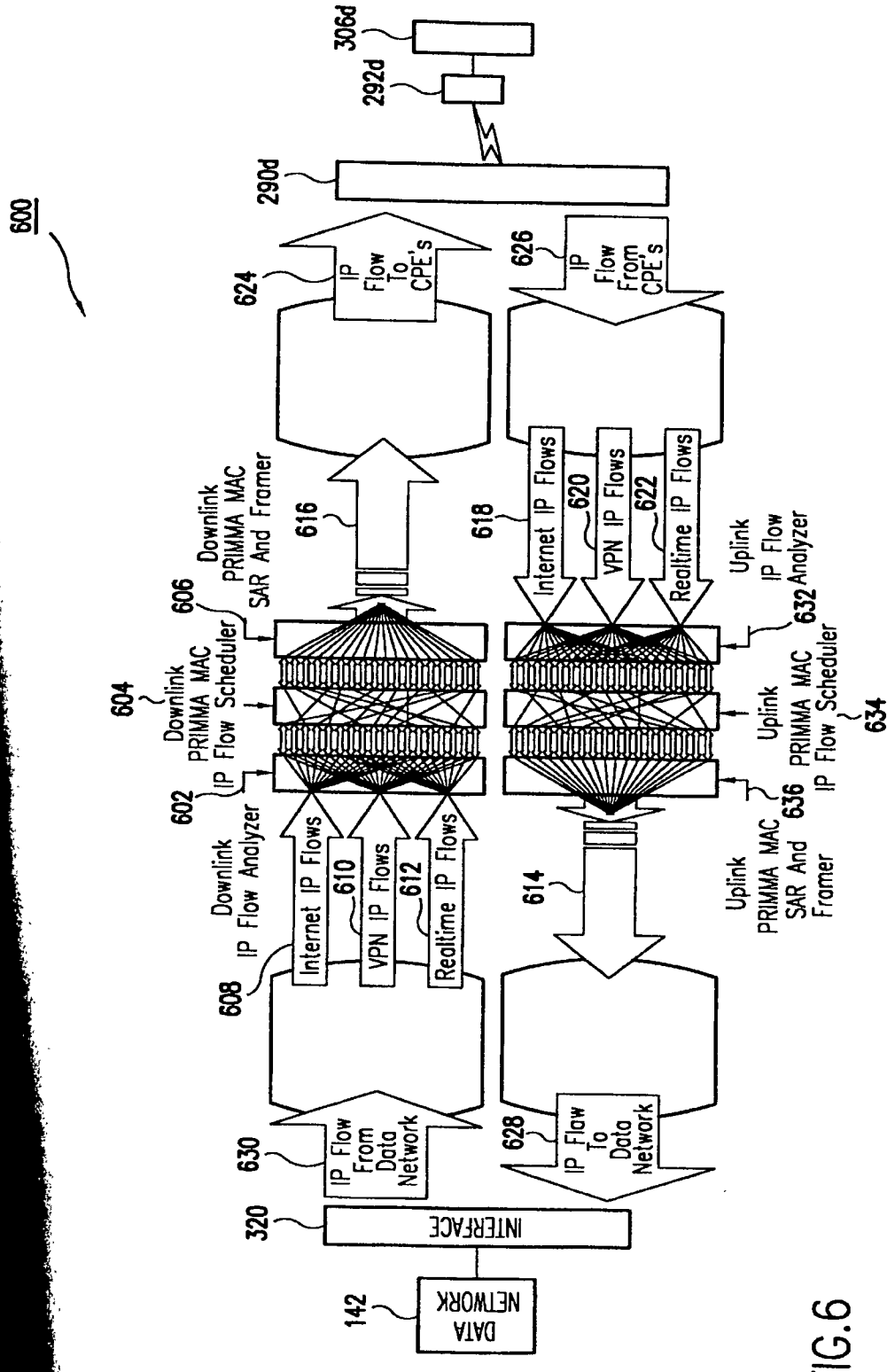

600
306d
292d
290d
624
IP Flow To CPE's
626
IP Flow From CPE's
616
618
Downlink PRIMMA MAC SAR And Framer
Internet IP Flows
620
VPN IP Flows
622
Realtime IP Flows
Uplink IP Flow Analyzer
632
606
Downlink PRIMMA MAC IP Flow Scheduler
604
Uplink PRIMMA MAC IP Flow Scheduler
634
602
636
Downlink IP Flow Analyzer
Internet IP Flows
610
VPN IP Flows
612
Realtime IP Flows
614
Uplink PRIMMA MAC SAR And Framer
608
630
IP Flow From Data Network
628
IP Flow To Data Network
320
INTERFACE
142
DATA NETWORK

FIG.6

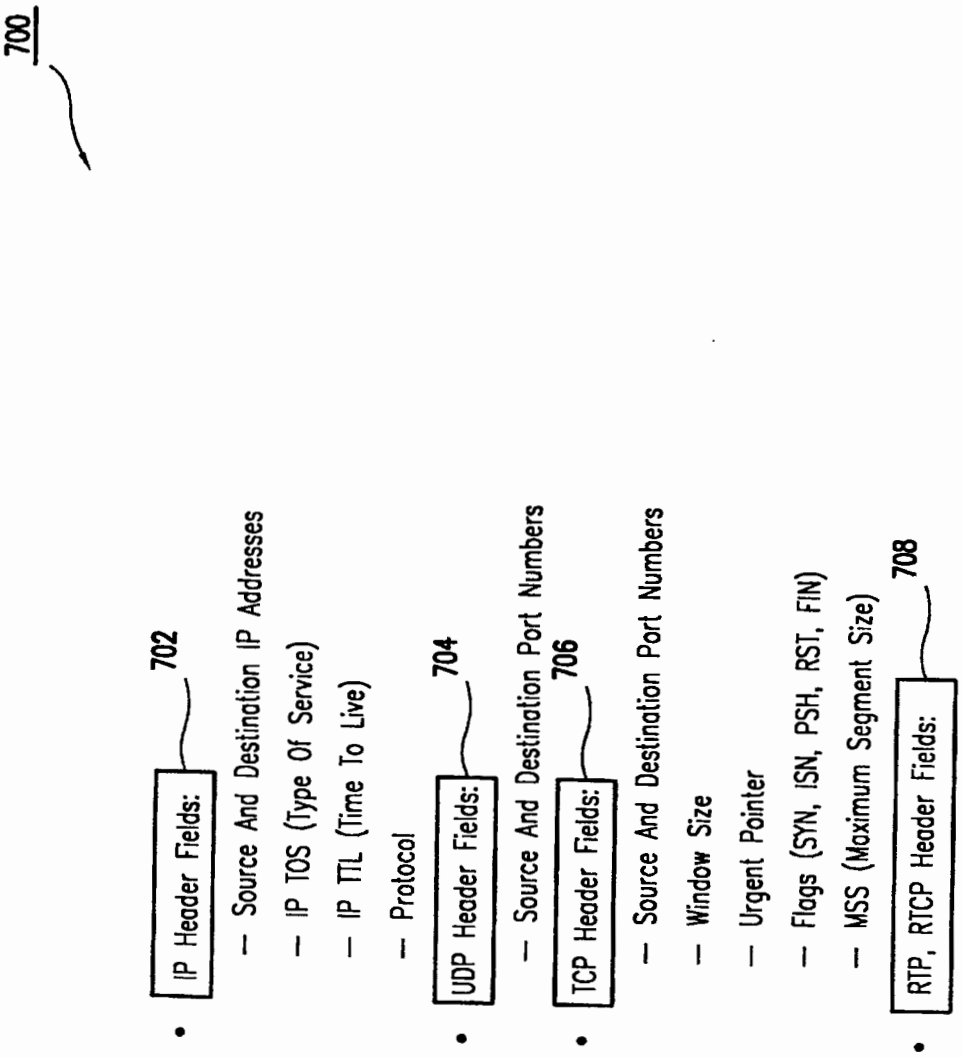

700
IP Header Fields: 702
– Source And Destination IP Addresses
– IP TOS (Type Of Service)
– IP TTL (Time To Live)
– Protocol
UDP Header Fields: 704
– Source And Destination Port Numbers
TCP Header Fields: 706
– Source And Destination Port Numbers
– Window Size
– Urgent Pointer
– Flags (SYN, ISN, PSH, RST, FIN)
– MSS (Maximum Segment Size)
RTP, RTCP Header Fields: 708

FIG. 7

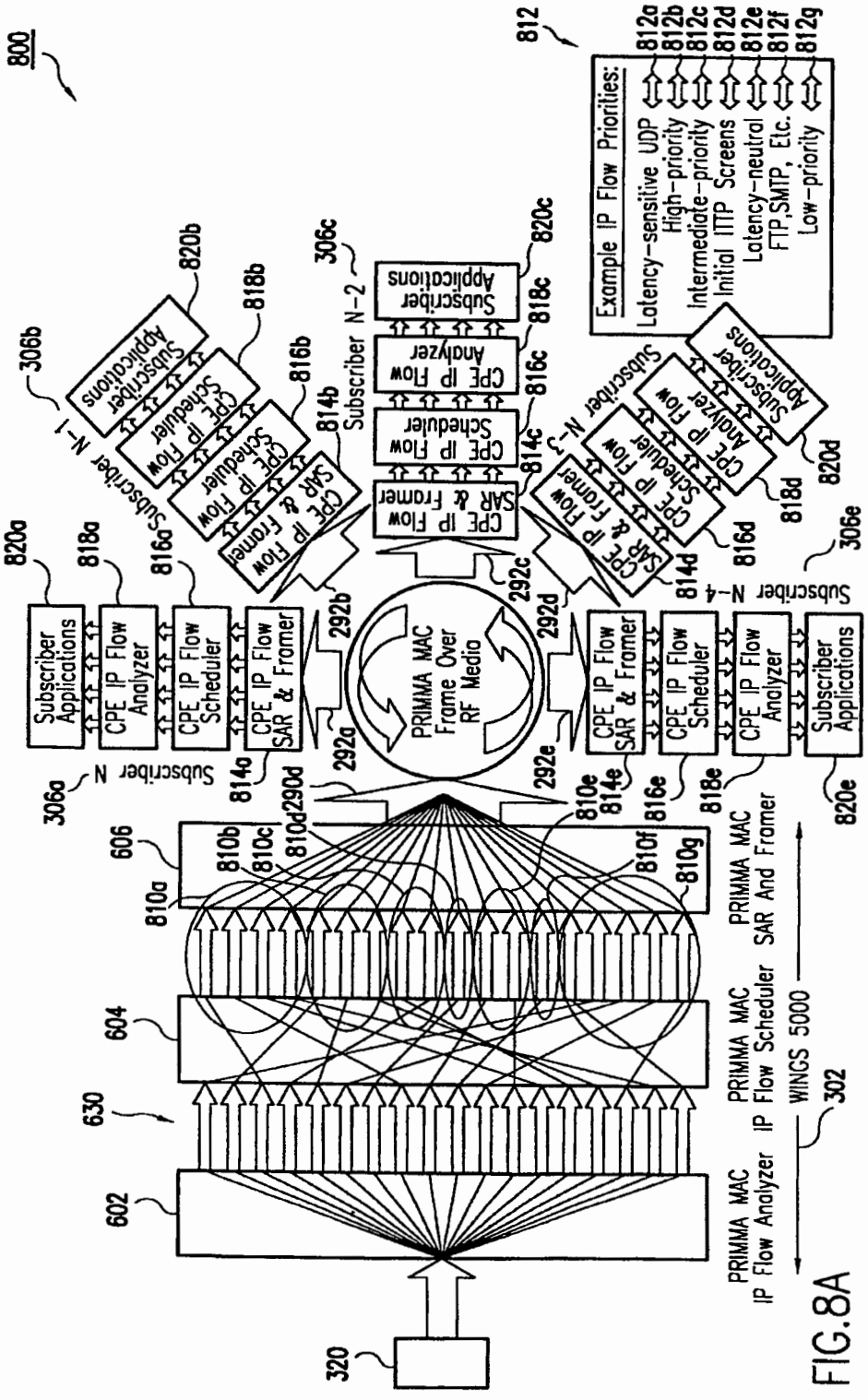

800
812
812a
812b
812c
812d
812e
812f
812g
Example IP Flow Priorities:
Latency-sensitive UDP
High-priority
Intermediate-priority
Initial IITP Screens
Latency-neutral
FTP,SMTP, Etc.
Low-priority
820b
818b
816b
814b
306c
Subscriber N-2
820c
818c
816c
814c
306b
Subscriber N-1
Subscriber Applications
CPE IP Flow Scheduler
CPE IP Flow Analyzer
CPE IP Flow SAR & Framer
Subscriber N-3
820d
818d
816d
814d
306e
Subscriber N-4
820a
818a
816a
814a
306a
Subscriber N
Subscriber Applications
CPE IP Flow Analyzer
CPE IP Flow Scheduler
CPE IP Flow SAR & Framer
292a
292b
292c
292d
292e
PRIMMA MAC Frame Over RF Media
290d
810a
810b
810c
810d
810e
810f
810g
606
604
630
602
320
PRIMMA MAC IP Flow Analyzer
PRIMMA MAC IP Flow Scheduler
PRIMMA MAC SAR And Framer
WINGS 5000
302

FIG.8A

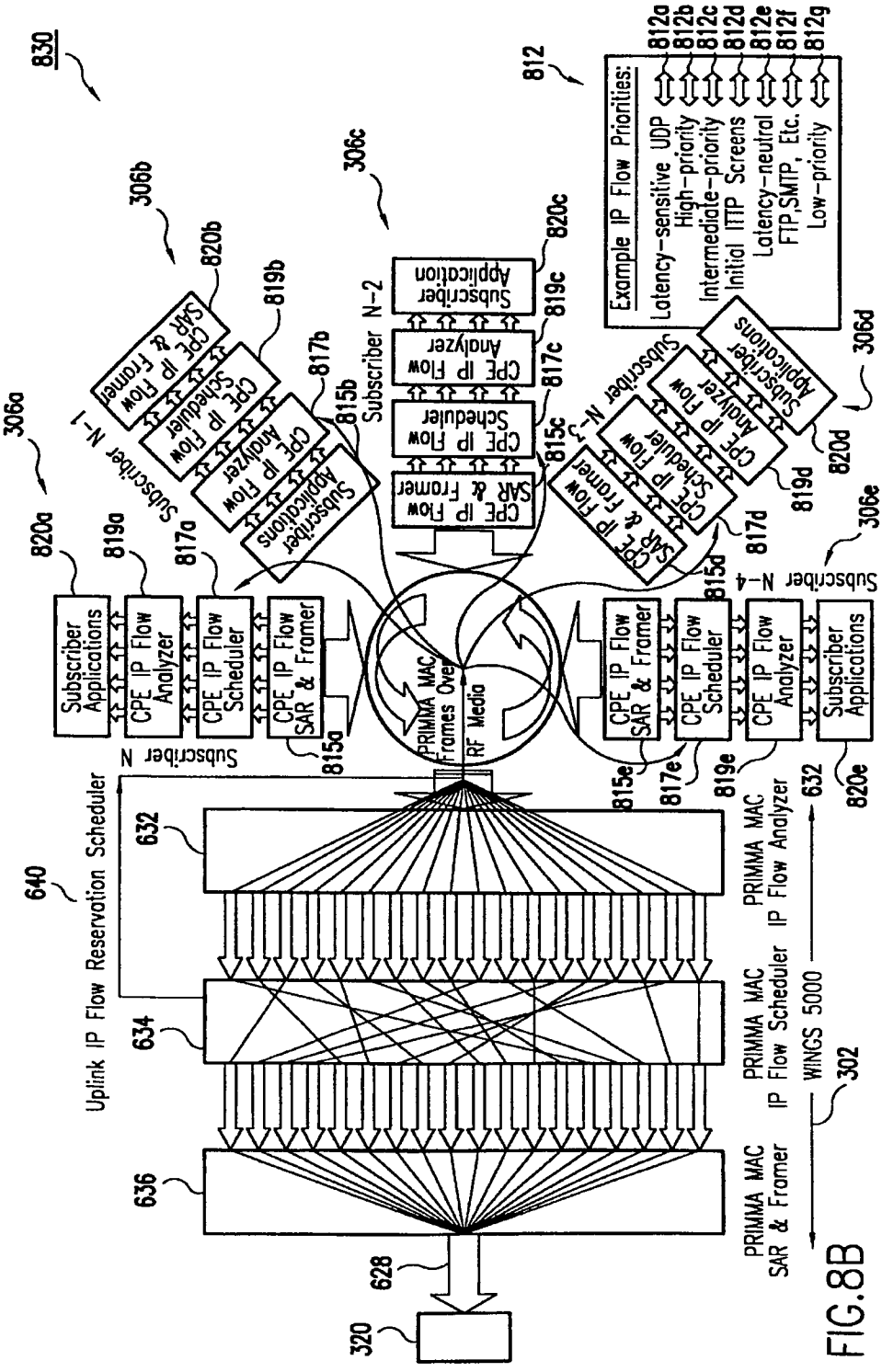

830
812
Example IP Flow Priorities:
812a Latency-sensitive UDP
812b High-priority
812c Intermediate-priority
812d Initial ITTP Screens
812e Latency-neutral
812f FTP,SMTP, Etc.
812g Low-priority
306a
306b
306c
306d
306e
Subscriber N-1
Subscriber N-2
Subscriber N-3
Subscriber N-4
Subscriber N
820a
820b
820c
820d
820e
819a
819b
819c
819d
817a
817b
817c
817d
815a
815b
815c
815d
815e
817e
819e
632
634
636
640
628
320
302
Subscriber Applications
CPE IP Flow Analyzer
CPE IP Flow Scheduler
CPE IP Flow SAR & Framer
PRIMMA MAC Frames Over RF Media
Uplink IP Flow Reservation Scheduler
PRIMMA MAC SAR & Framer
PRIMMA MAC IP Flow Scheduler
PRIMMA MAC IP Flow Analyzer
WINGS 5000

FIG.8B

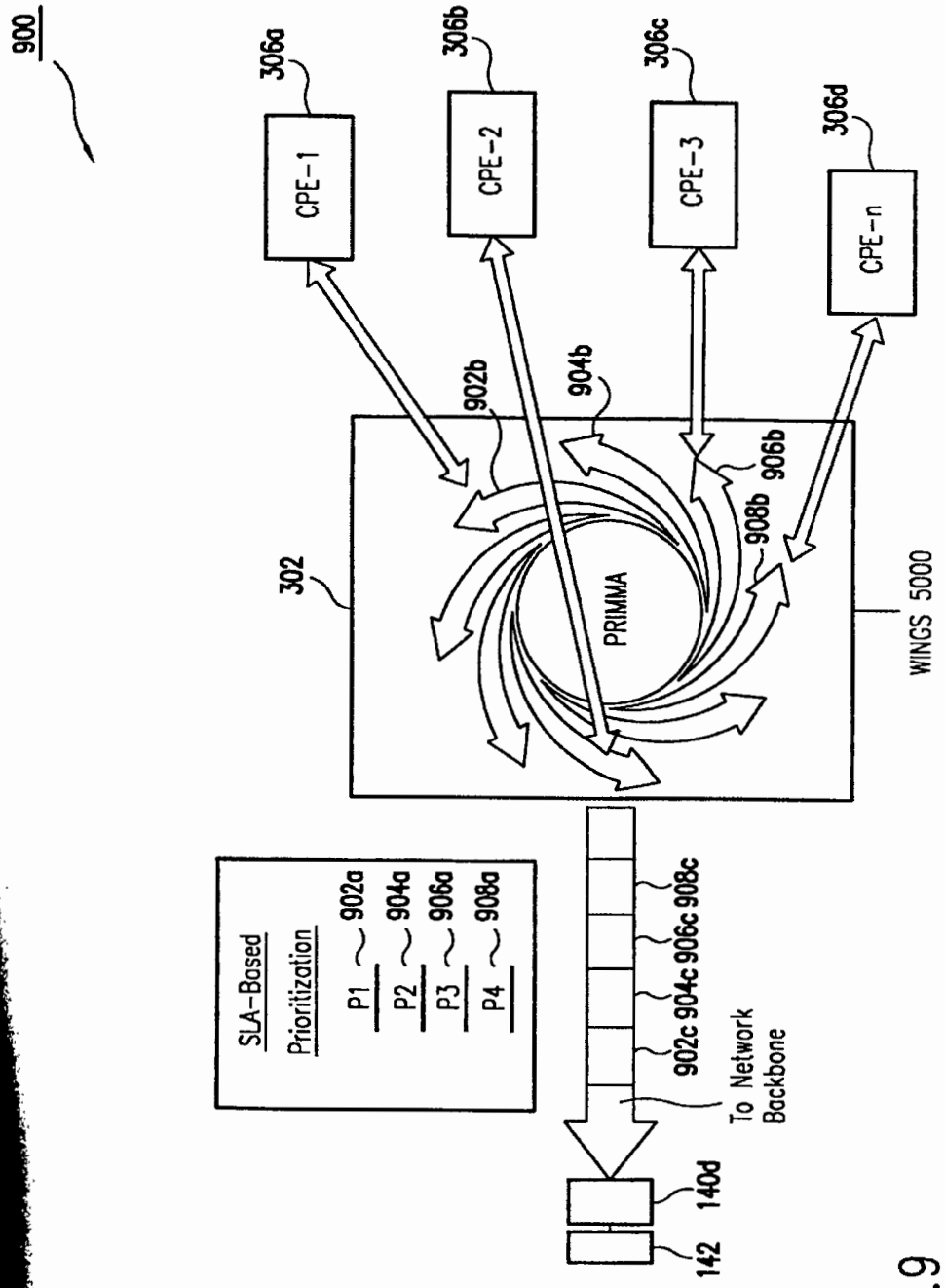

900
306a
CPE-1
306b
CPE-2
306c
CPE-3
306d
CPE-n
902b
904b
906b
908b
302
PRIMMA
WINGS 5000
SLA-Based
Prioritization
P1 902a
P2 904a
P3 906a
P4 908a
902c 904c 906c 908c
To Network
Backbone
140d
142

FIG.9

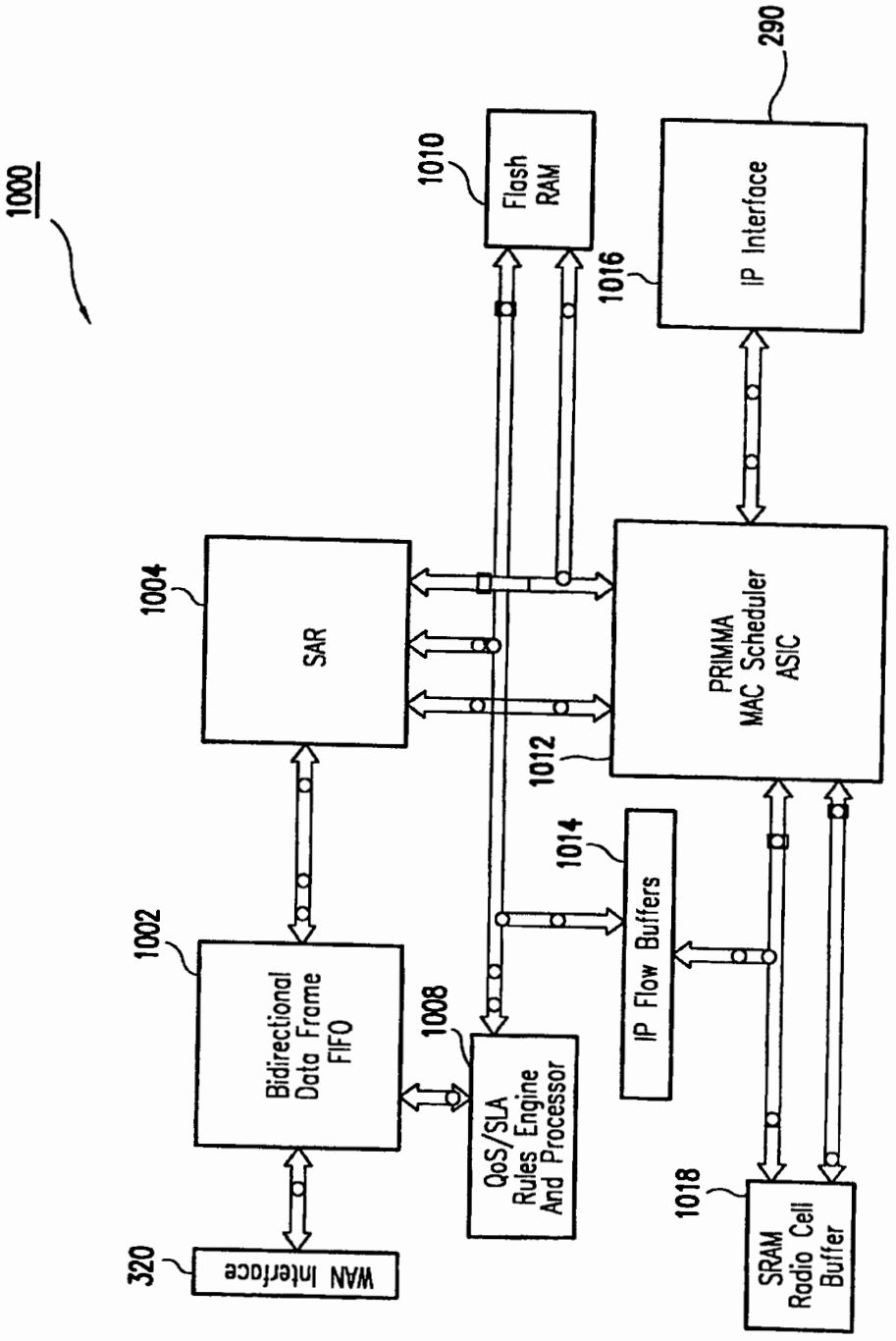

1000
1010
Flash RAM
290
IP Interface
1016
1004
SAR
PRIMA MAC Scheduler ASIC
1012
1014
IP Flow Buffers
1002
Bidirectional Data Frame FIFO
1008
QoS/SLA Rules Engine And Processor
1018
SRAM Radio Cell Buffer
320
WAN Interface

FIG. 10

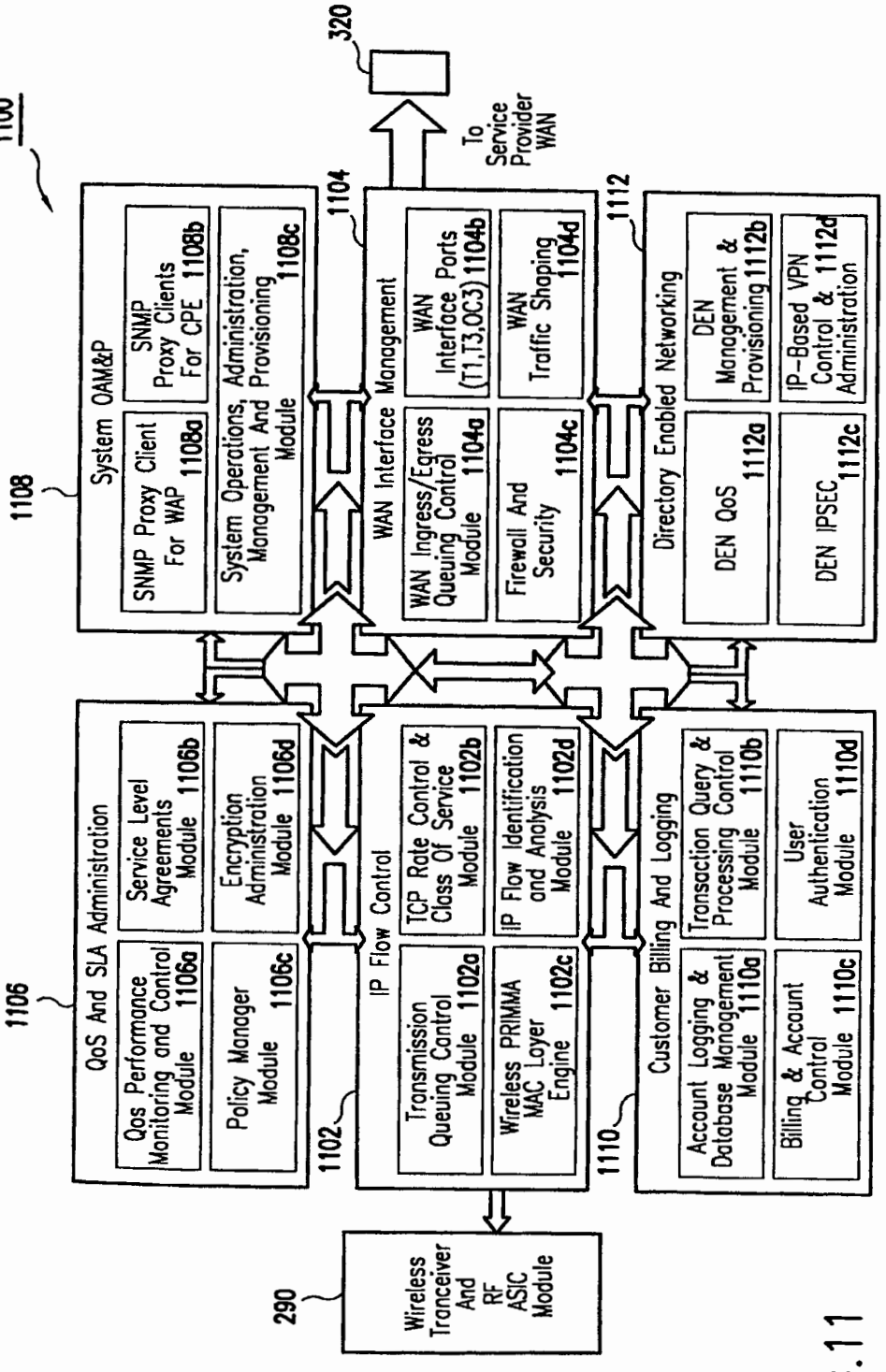

1100
320
To Service Provider WAN
1104
1112
1108
System OAM&P
SNMP Proxy Client For WAP 1108a
SNMP Proxy Clients For CPE 1108b
System Operations, Administration, Management And Provisioning Module 1108c
WAN Interface Management
WAN Interface Ports (T1,T3,OC3) 1104b
WAN Traffic Shaping 1104d
WAN Ingress/Egress Queuing Control Module 1104a
Firewall And Security 1104c
Directory Enabled Networking
DEN Management & Provisioning 1112b
IP-Based VPN Control & Administration 1112d
DEN QoS 1112a
DEN IPSEC 1112c
1106
QoS And SLA Administration
Service Level Agreements Module 1106b
Encryption Administration Module 1106d
Qos Performance Monitoring and Control Module 1106a
Policy Manager Module 1106c
1102
IP Flow Control
TCP Rate Control & Class Of Service Module 1102b
IP Flow Identification and Analysis Module 1102d
Transmission Queuing Control Module 1102a
Wireless PRIMMA MAC Layer Engine 1102c
1110
Customer Billing And Logging
Transaction Query & Processing Control Module 1110b
User Authentication Module 1110d
Account Logging & Database Management Module 1110a
Billing & Account Control Module 1110c
290
Wireless Tranceiver And RF ASIC Module

FIG. 11

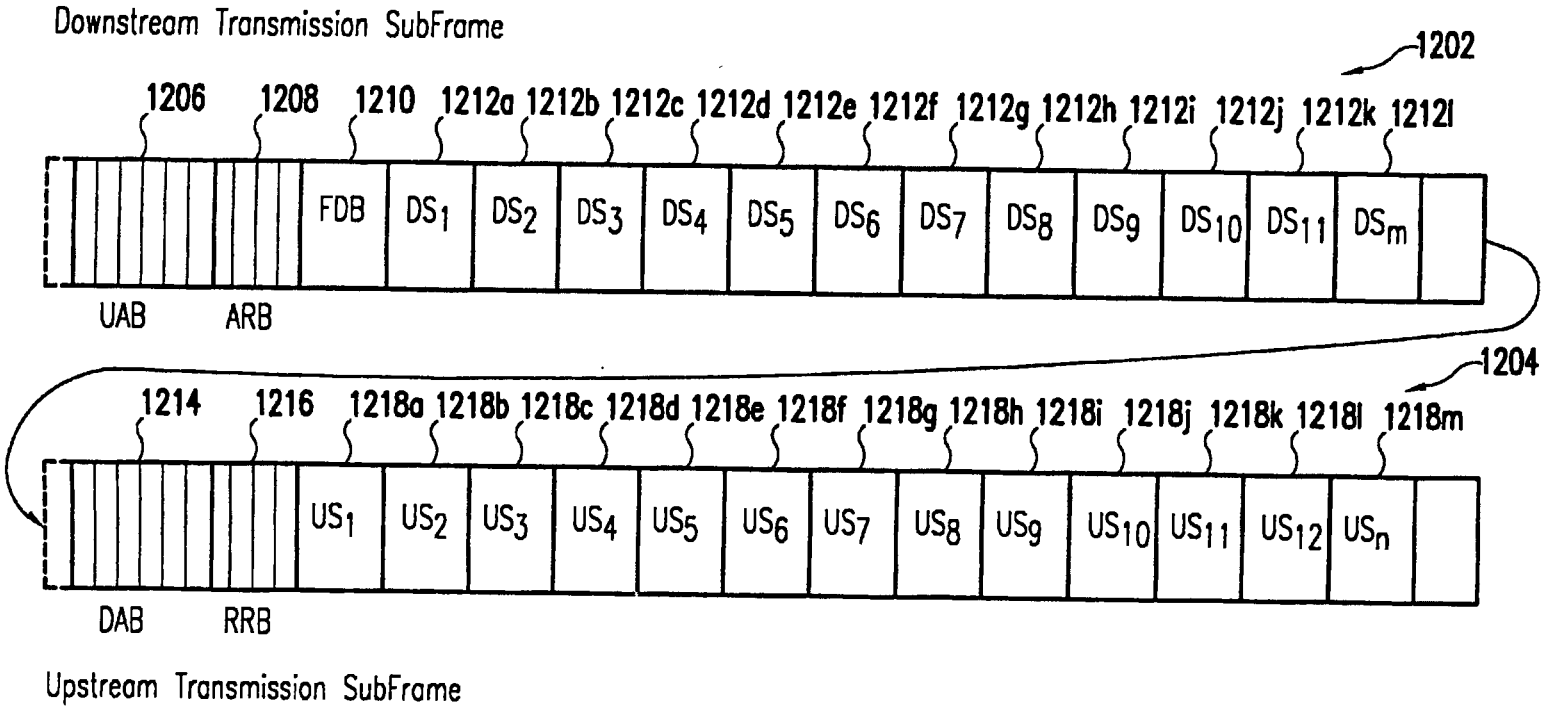

Downstream Transmission SubFrame
1202
1206
1208
1210
1212a
1212b
1212c
1212d
1212e
1212f
1212g
1212h
1212i
1212j
1212k
1212l
FDB
DS1
DS2
DS3
DS4
DS5
DS6
DS7
DS8
DS9
DS10
DS11
DSm
UAB
ARB
1204
1214
1216
1218a
1218b
1218c
1218d
1218e
1218f
1218g
1218h
1218i
1218j
1218k
1218l
1218m
US1
US2
US3
US4
US5
US6
US7
US8
US9
US10
US11
US12
USn
DAB
RRB
Upstream Transmission SubFrame

FIG. 12A

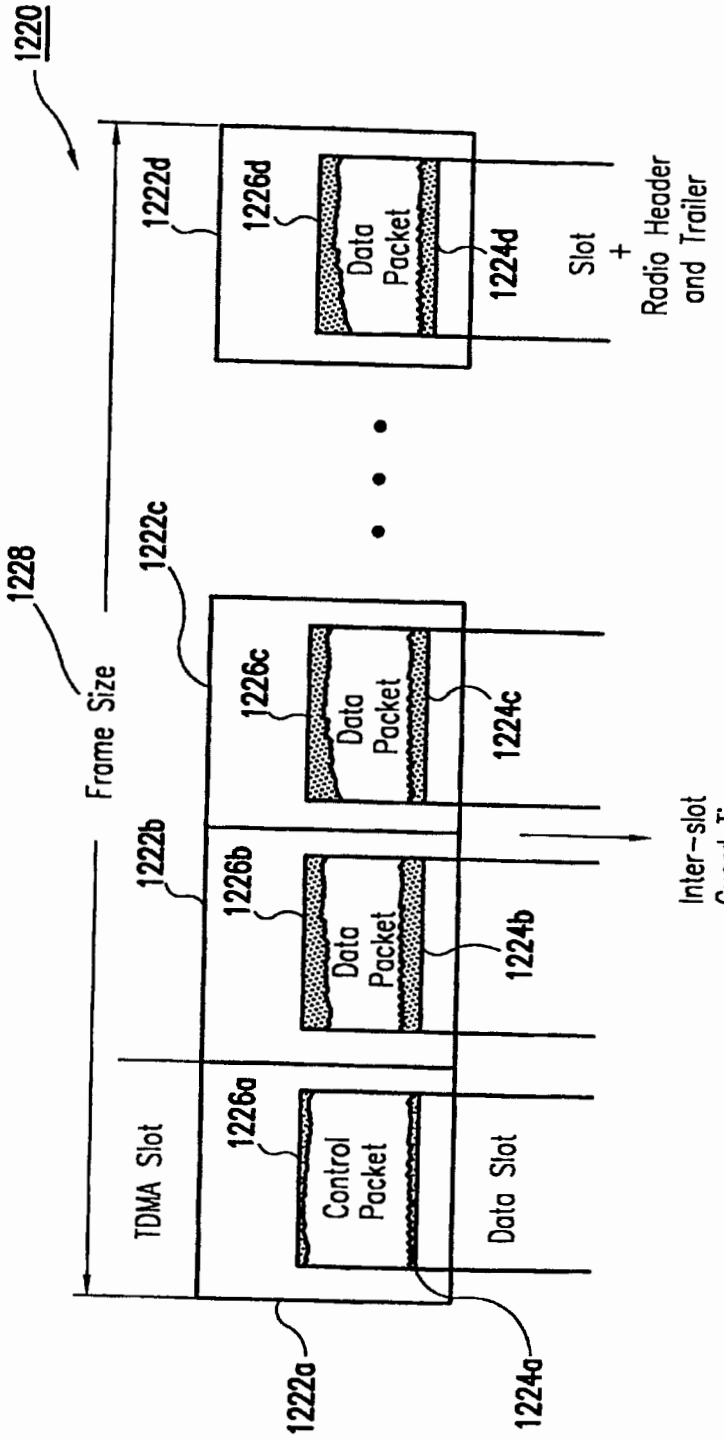

1220
1222d
1226d
Data Packet
1224d
Slot + Radio Header and Trailer
1228
Frame Size
1222c
1226c
Data Packet
1224c
1222b
1226b
Data Packet
1224b
Inter-slot Guard Time
1229
TDMA Slot
1226a
Control Packet
Data Slot
1222a
1224a

FIG. 12B

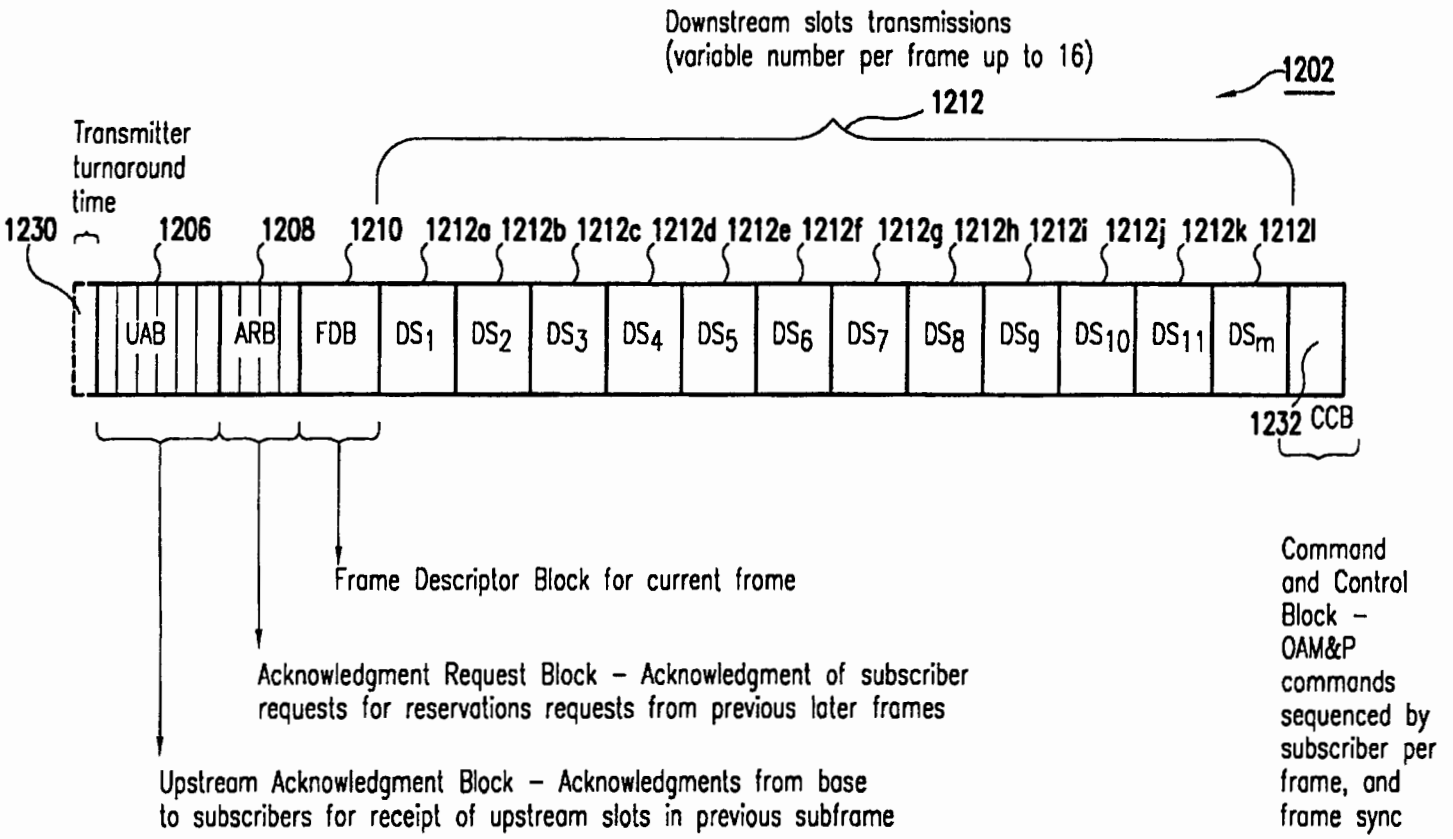

Downstream slots transmissions
(variable number per frame up to 16)
1202
1212
Transmitter
turnaround
time
1230
1206
1208
1210
1212a
1212b
1212c
1212d
1212e
1212f
1212g
1212h
1212i
1212j
1212k
1212l
UAB
ARB
FDB
DS1
DS2
DS3
DS4
DS5
DS6
DS7
DS8
DS9
DS10
DS11
DSm
1232 CCB
Frame Descriptor Block for current frome
Acknowledgment Request Block – Acknowledgment of subscriber requests for reservations requests from previous later frames
Upstream Acknowledgment Block – Acknowledgments from base to subscribers for receipt of upstream slots in previous subframe
Command and Control Block – OAM&P commands sequenced by subscriber per frame, and frame sync

FIG.12C

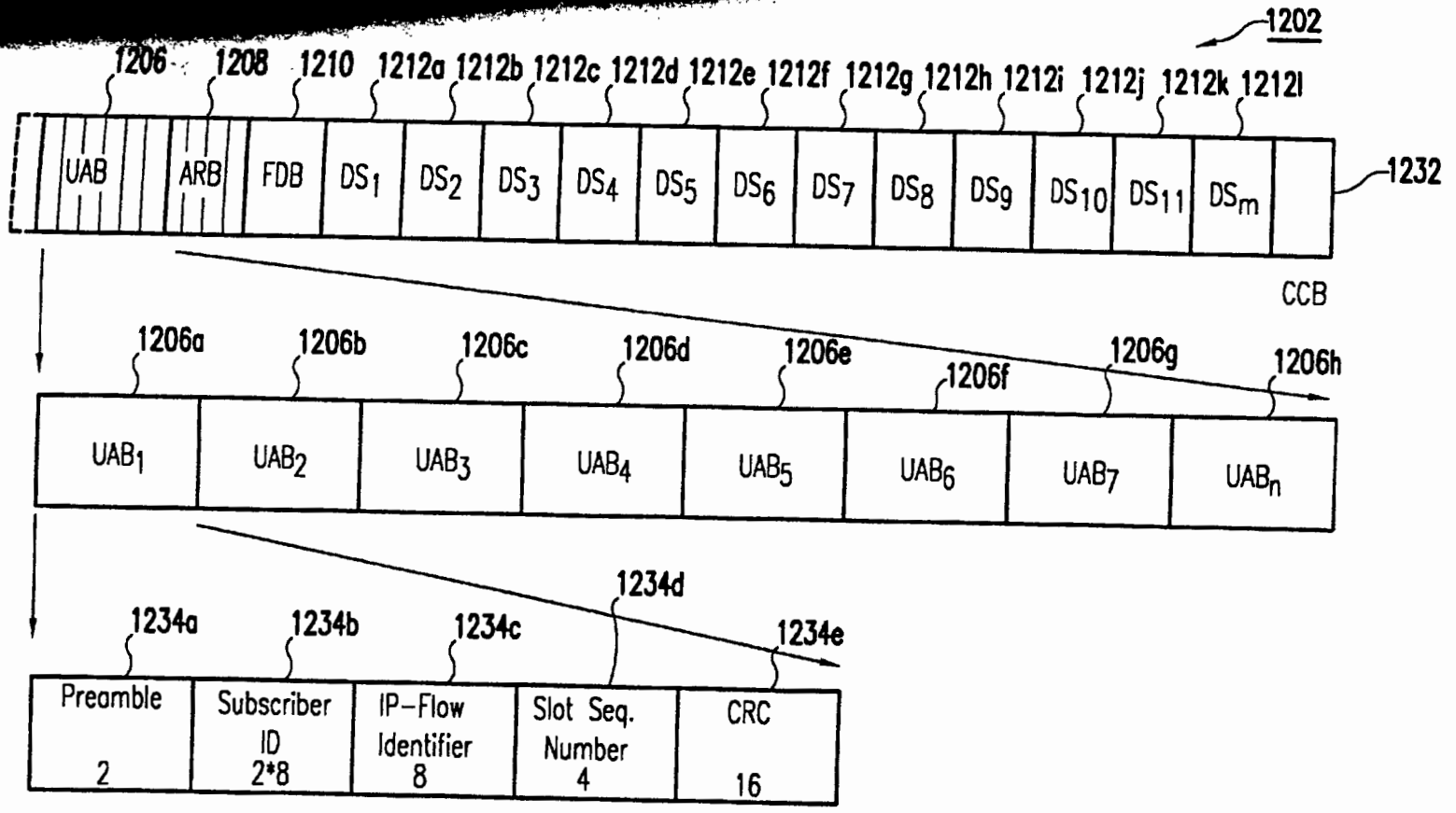

1202
1206
1208
1210
1212a
1212b
1212c
1212d
1212e
1212f
1212g
1212h
1212i
1212j
1212k
1212l
UAB
ARB
FDB
DS1
DS2
DS3
DS4
DS5
DS6
DS7
DS8
DS9
DS10
DS11
DSm
1232
CCB
1206a
1206b
1206c
1206d
1206e
1206f
1206g
1206h
UAB1
UAB2
UAB3
UAB4
UAB5
UAB6
UAB7
UABn
1234d
1234a
1234b
1234c
1234e
Preamble
2
Subscriber ID
2*8
IP–Flow Identifier
8
Slot Seq. Number
4
CRC
16

FIG. 12D

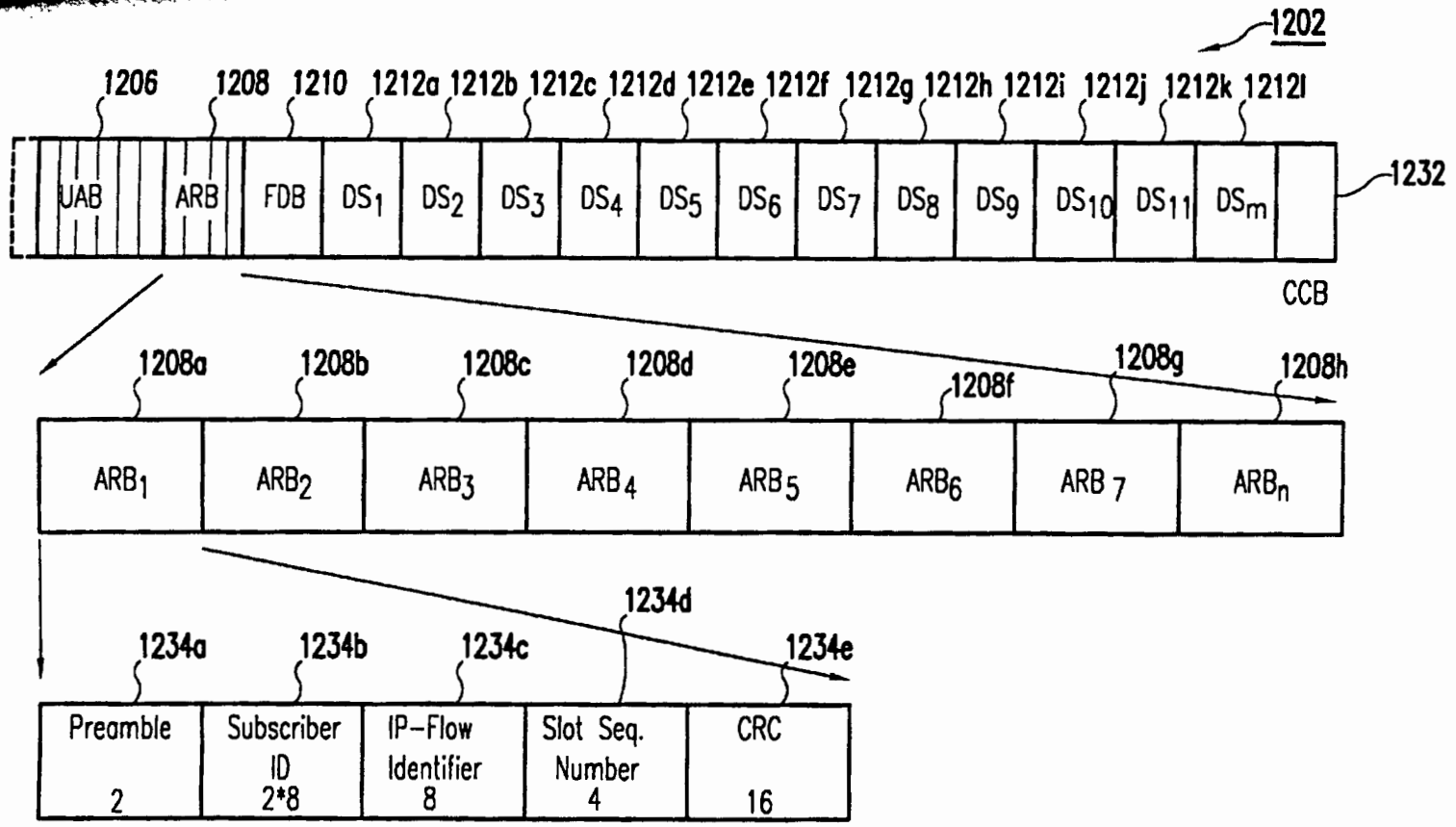

1202
1206
1208
1210
1212a
1212b
1212c
1212d
1212e
1212f
1212g
1212h
1212i
1212j
1212k
1212l
UAB
ARB
FDB
DS1
DS2
DS3
DS4
DS5
DS6
DS7
DS8
DS9
DS10
DS11
DSm
1232
CCB
1208a
1208b
1208c
1208d
1208e
1208f
1208g
1208h
ARB1
ARB2
ARB3
ARB4
ARB5
ARB6
ARB 7
ARBn
1234d
1234a
1234b
1234c
1234e
Preamble
2
Subscriber ID
2*8
IP–Flow Identifier
8
Slot Seq. Number
4
CRC
16

FIG. 12E

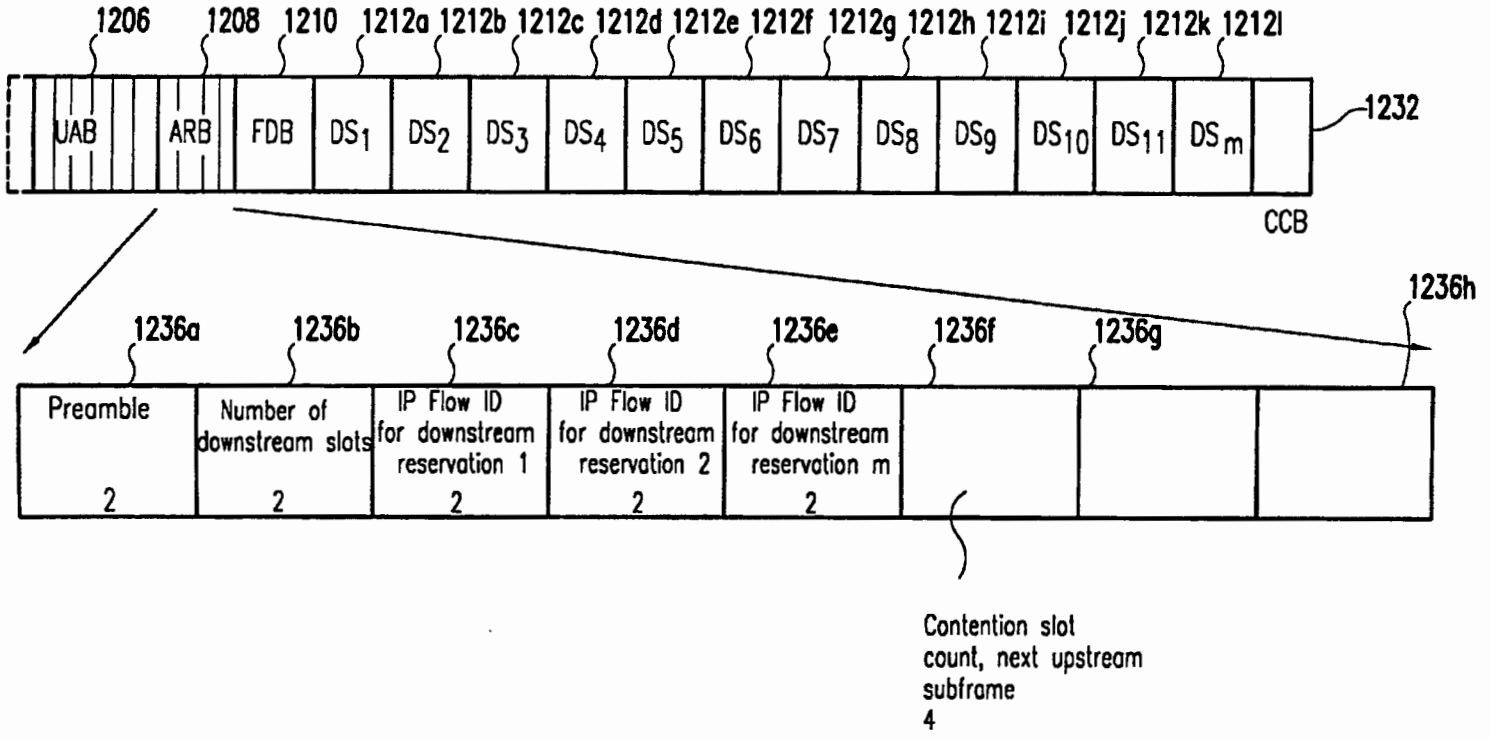

1206
1208
1210
1212a
1212b
1212c
1212d
1212e
1212f
1212g
1212h
1212i
1212j
1212k
1212l
UAB
ARB
FDB
DS1
DS2
DS3
DS4
DS5
DS6
DS7
DS8
DS9
DS10
DS11
DSm
1232
CCB
1236a
1236b
1236c
1236d
1236e
1236f
1236g
1236h
Preamble
2
Number of downstream slots
2
IP Flow ID for downstream reservation 1
2
IP Flow ID for downstream reservation 2
2
IP Flow ID for downstream reservation m
2
Contention slot count, next upstream subframe
4

FIG.12F

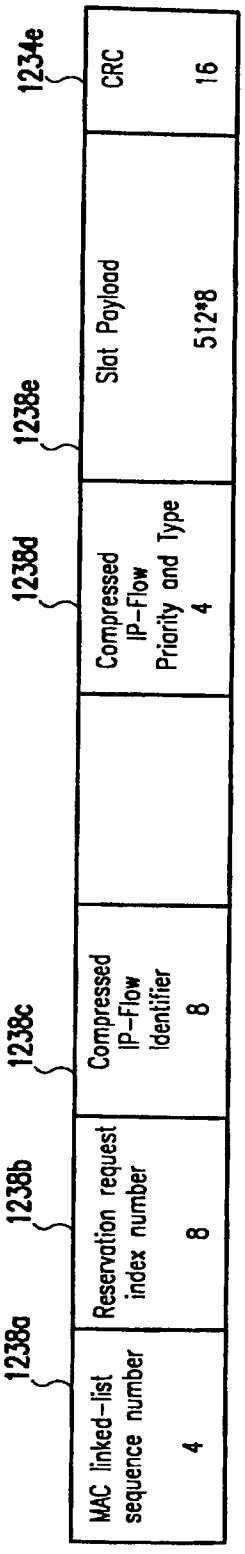

1238a
MAC linked-list sequence number
4
1238b
Reservation request index number
8
1238c
Compressed IP-Flow Identifier
8
1238d
Compressed IP-Flow Priority and Type
4
1238e
Slot Payload
512*8
1234e
CRC
16

FIG. 12G

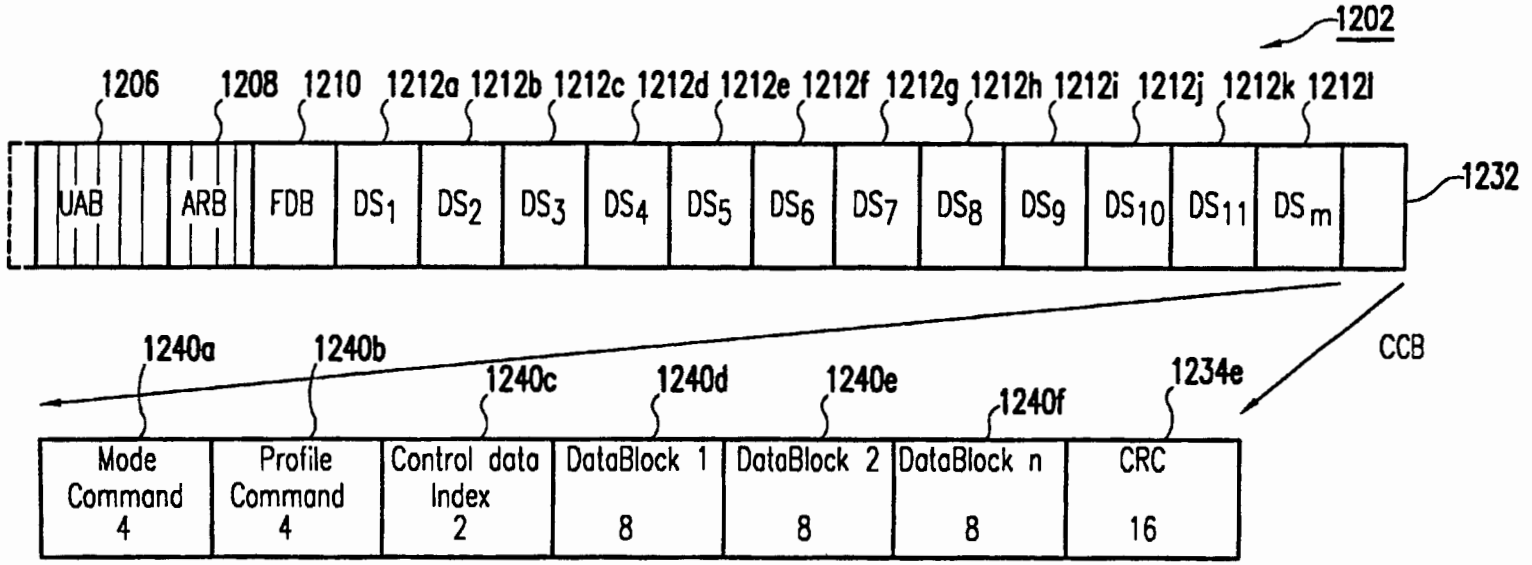

1202
1206
1208
1210
1212a
1212b
1212c
1212d
1212e
1212f
1212g
1212h
1212i
1212j
1212k
1212l
UAB
ARB
FDB
DS1
DS2
DS3
DS4
DS5
DS6
DS7
DS8
DS9
DS10
DS11
DS m
1232
CCB
1240a
1240b
1240c
1240d
1240e
1240f
1234e
Mode Command 4
Profile Command 4
Control data Index 2
DataBlock 1 8
DataBlock 2 8
DataBlock n 8
CRC 16

FIG.12H

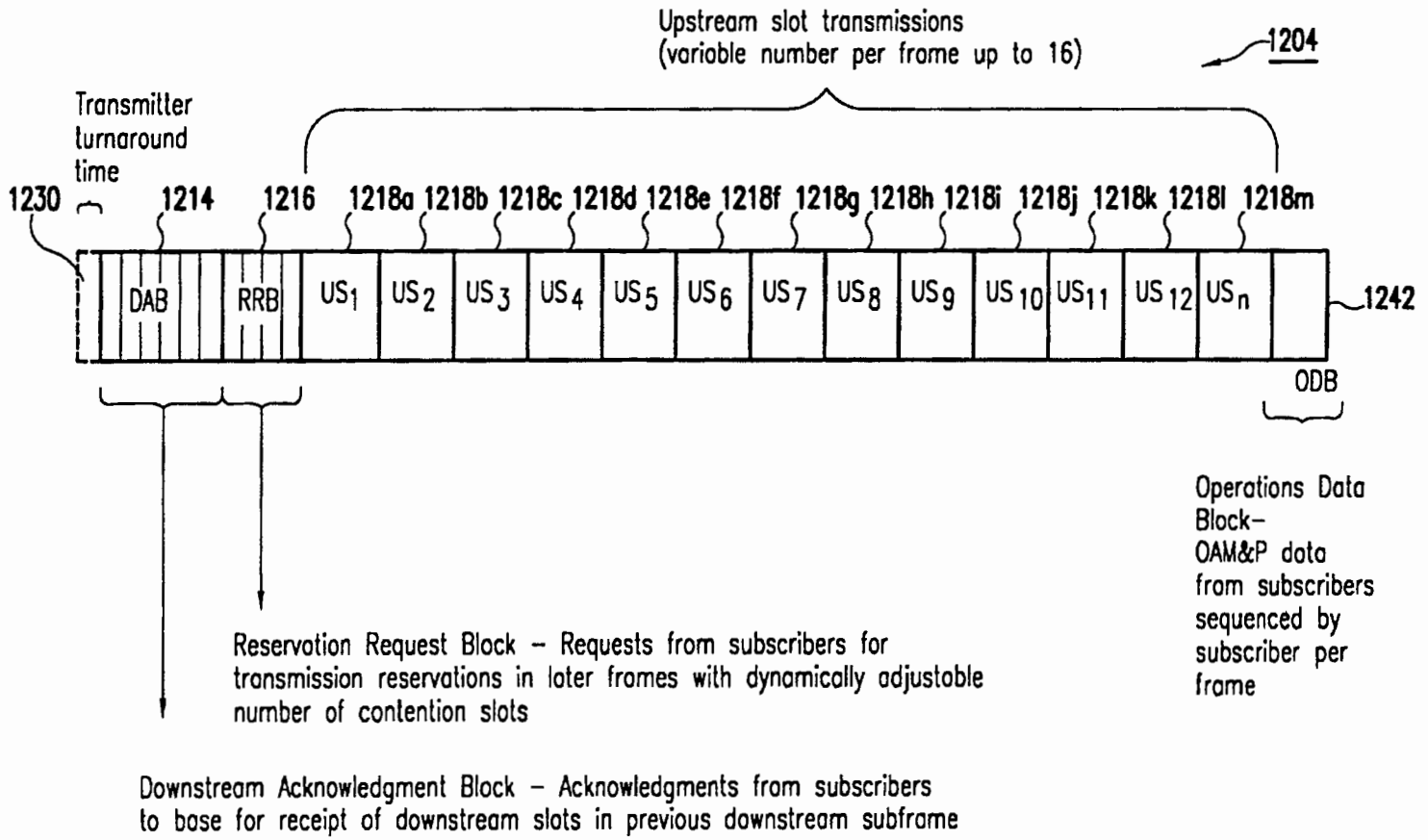

Upstream slot transmissions
(variable number per frame up to 16)
1204
Transmitter
turnaround
time
1230
1214
1216
1218a 1218b 1218c 1218d 1218e 1218f 1218g 1218h 1218i 1218j 1218k 1218l 1218m
DAB
RRB
US1 US2 US3 US4 US5 US6 US7 US8 US9 US10 US11 US12 USn
1242
ODB
Operations Data
Block–
OAM&P data
from subscribers
sequenced by
subscriber per
frame
Reservation Request Block – Requests from subscribers for
transmission reservations in later frames with dynamically adjustable
number of contention slots
Downstream Acknowledgment Block – Acknowledgments from subscribers
to base for receipt of downstream slots in previous downstream subframe

FIG.12I

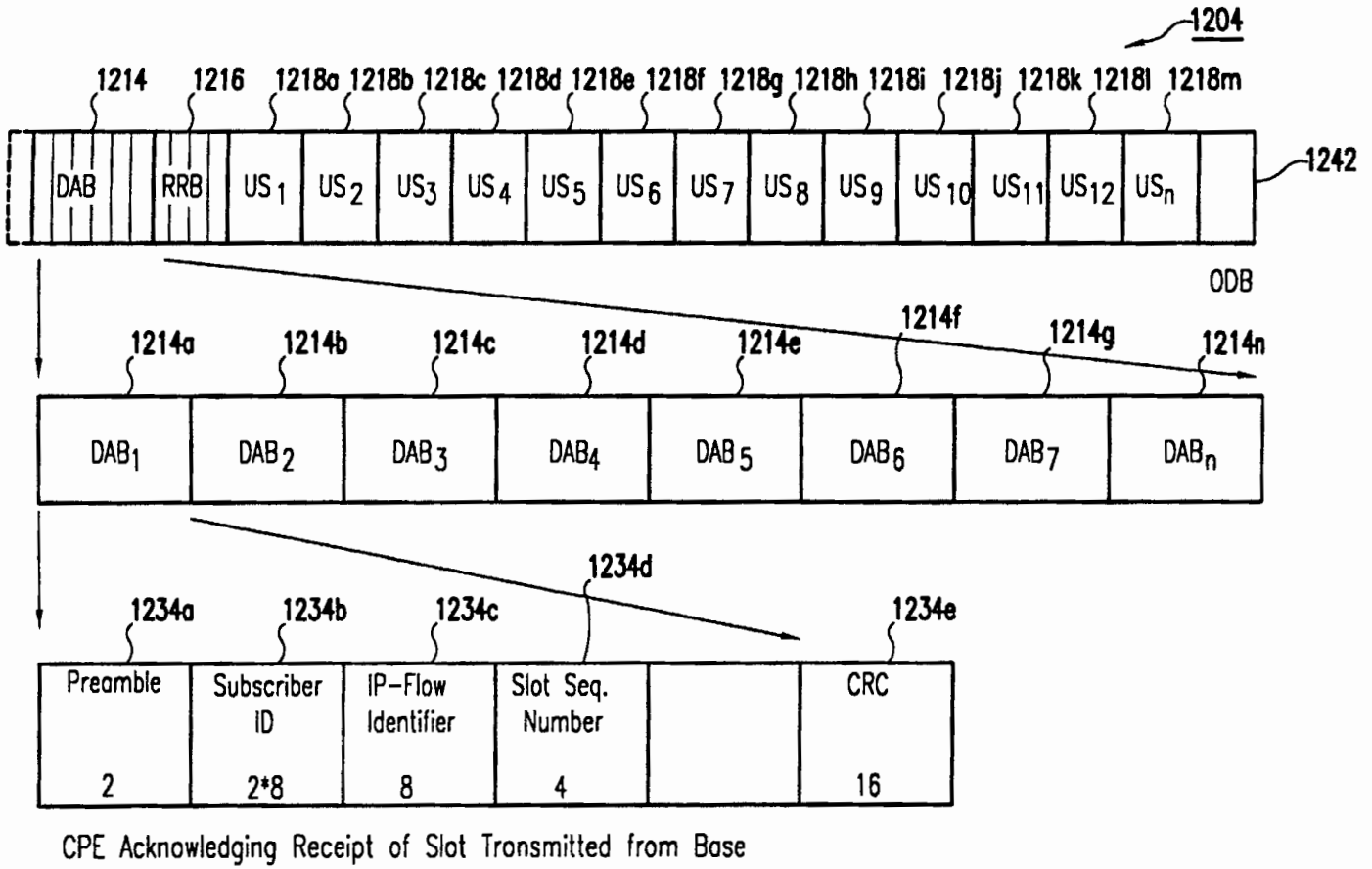
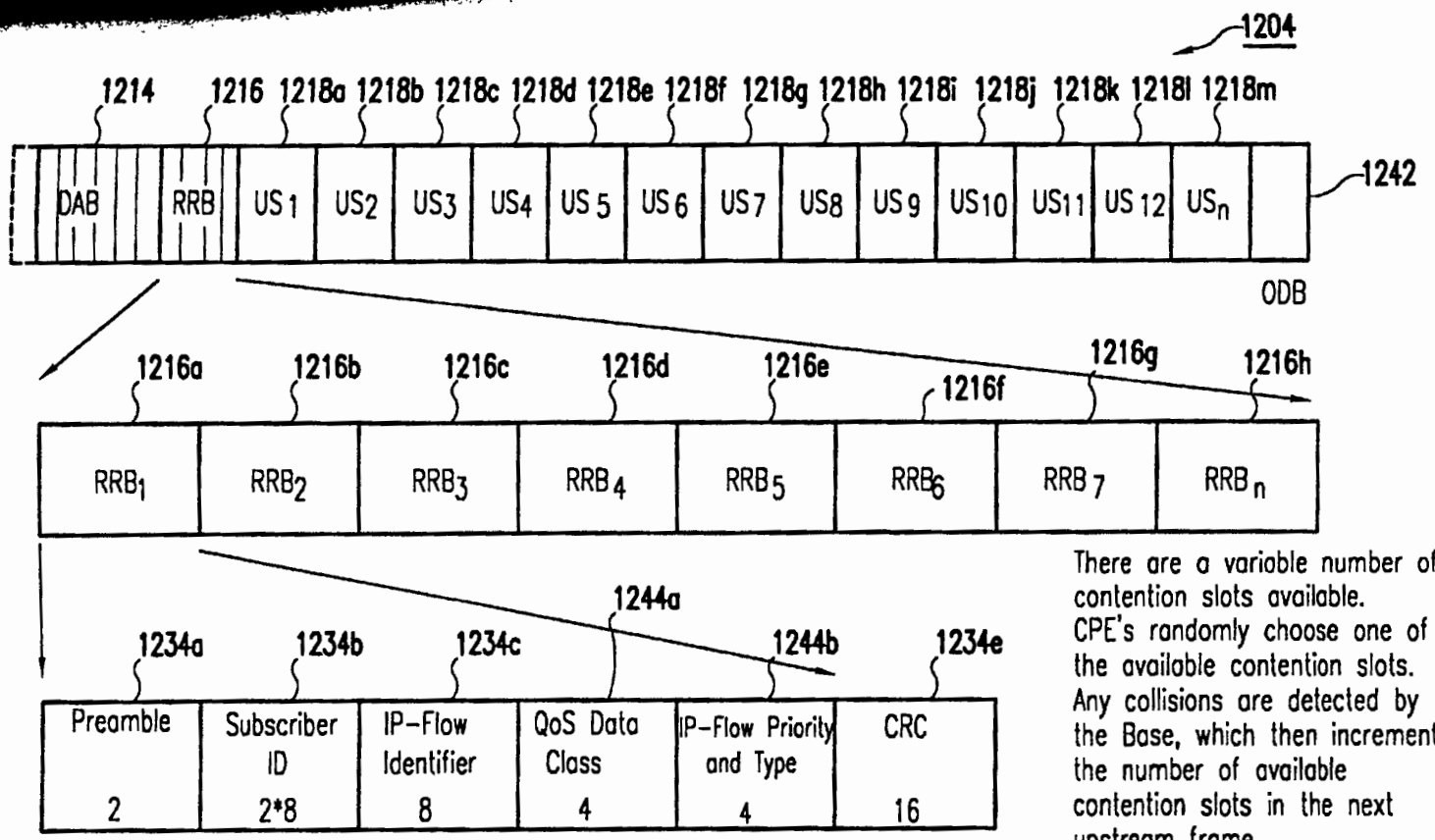

1204
1214
1216
1218a
1218b
1218c
1218d
1218e
1218f
1218g
1218h
1218i
1218j
1218k
1218l
1218m
DAB
RRB
US1
US2
US3
US4
US5
US6
US7
US8
US9
US10
US11
US12
USn
1242
ODB
1214a
1214b
1214c
1214d
1214e
1214f
1214g
1214n
DAB1
DAB2
DAB3
DAB4
DAB5
DAB6
DAB7
DABn
1234d
1234a
1234b
1234c
1234e
Preamble
Subscriber ID
IP-Flow Identifier
Slot Seq. Number
CRC
2
2*8
8
4
16
CPE Acknowledging Receipt of Slot Transmitted from Base

FIG.12J



There are a variable number of contention slots available. CPE's randomly choose one of the available contention slots. Any collisions are detected by the Base, which then increments the number of available contention slots in the next upstream frame.

CPE Requesting a Reservation for Upstream Transmission of Slot

FIG.12K

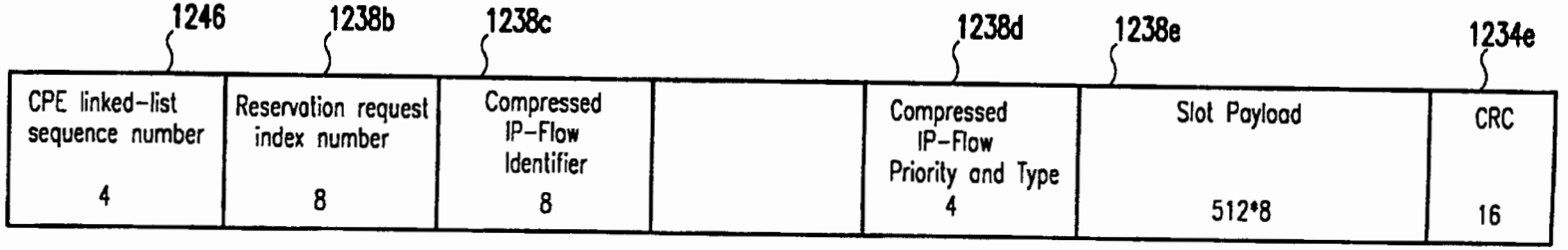

1246
1238b
1238c
1238d
1238e
1234e
CPE linked-list sequence number
4
Reservation request index number
8
Compressed IP-Flow Identifier
8
Compressed IP-Flow Priority and Type
4
Slot Payload
512*8
CRC
16

FIG. 12L

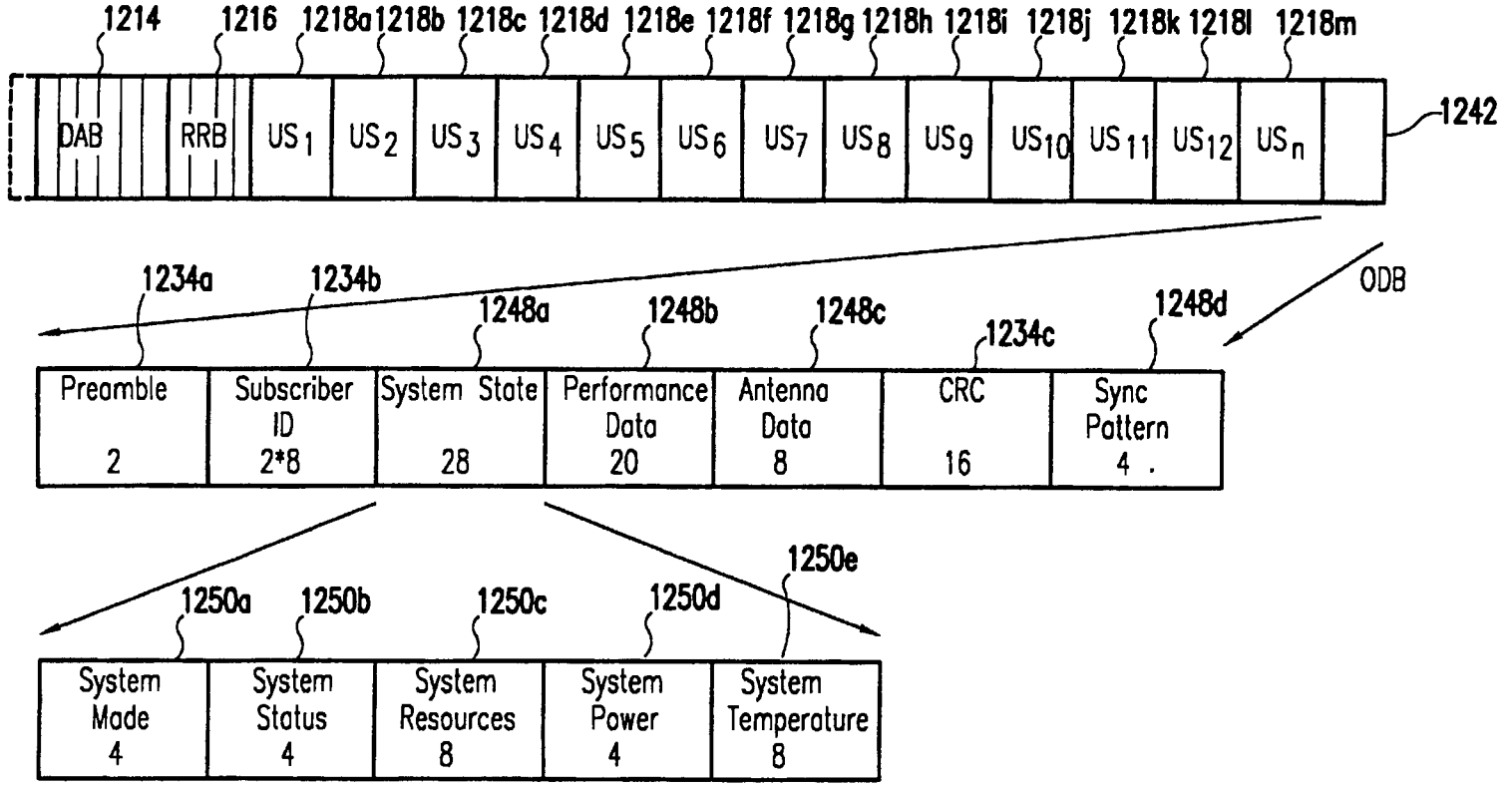

1214
1216
1218a
1218b
1218c
1218d
1218e
1218f
1218g
1218h
1218i
1218j
1218k
1218l
1218m
DAB
RRB
US1
US2
US3
US4
US5
US6
US7
US8
US9
US10
US11
US12
USn
1242
ODB
1234a
1234b
1248a
1248b
1248c
1234c
1248d
Preamble
2
Subscriber ID
2*8
System State
28
Performance Data
20
Antenna Data
8
CRC
16
Sync Pattern
4
1250a
1250b
1250c
1250d
1250e
System Mode
4
System Status
4
System Resources
8
System Power
4
System Temperature
8

FIG.12M

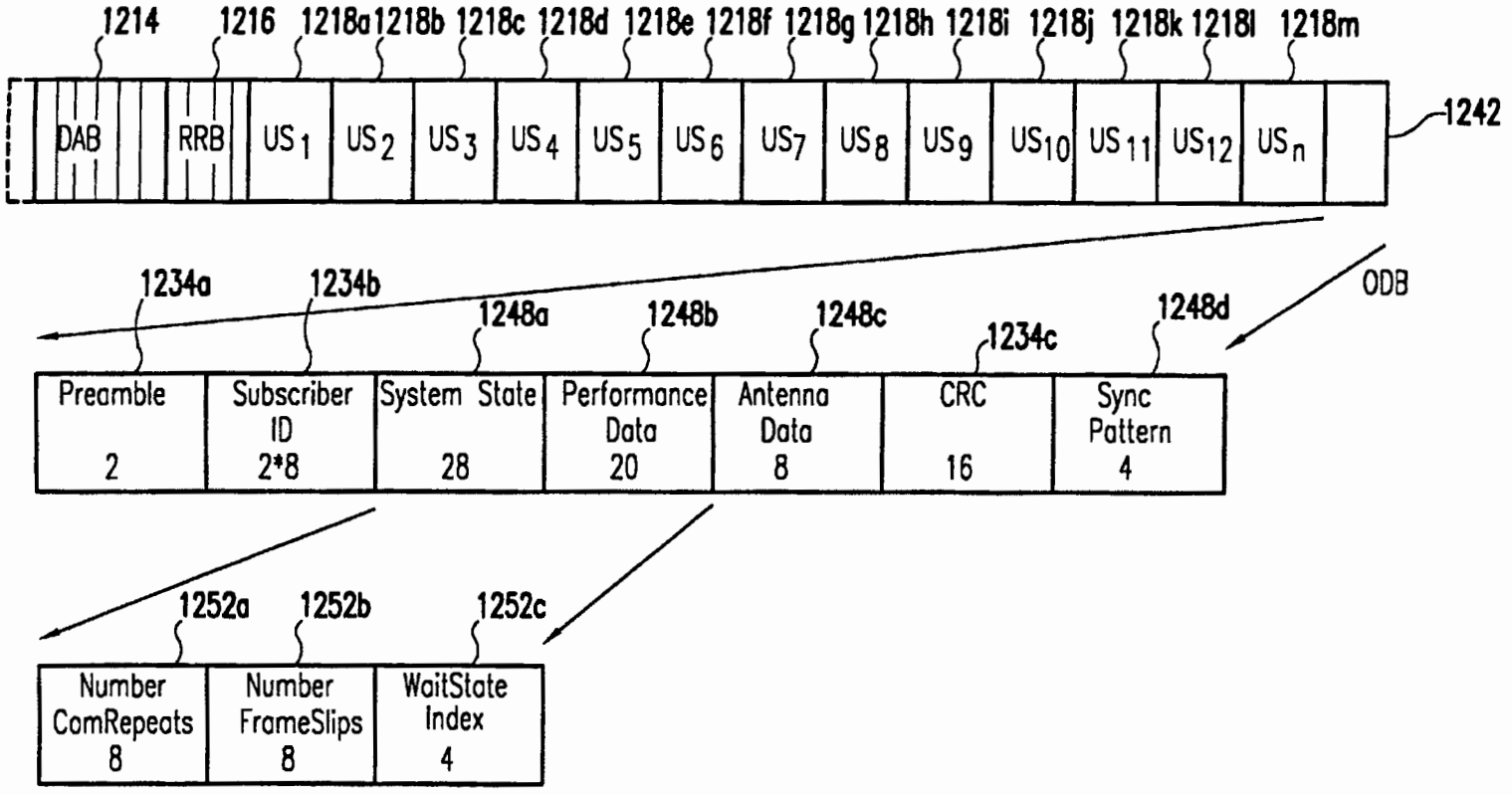

1214
1216
1218a
1218b
1218c
1218d
1218e
1218f
1218g
1218h
1218i
1218j
1218k
1218l
1218m
DAB
RRB
US1
US2
US3
US4
US5
US6
US7
US8
US9
US10
US11
US12
USn
1242
ODB
1234a
1234b
1248a
1248b
1248c
1234c
1248d
Preamble 2
Subscriber ID 2*8
System State 28
Performance Data 20
Antenna Data 8
CRC 16
Sync Pattern 4
1252a
1252b
1252c
Number ComRepeats 8
Number FrameSlips 8
WaitState Index 4

FIG.12N

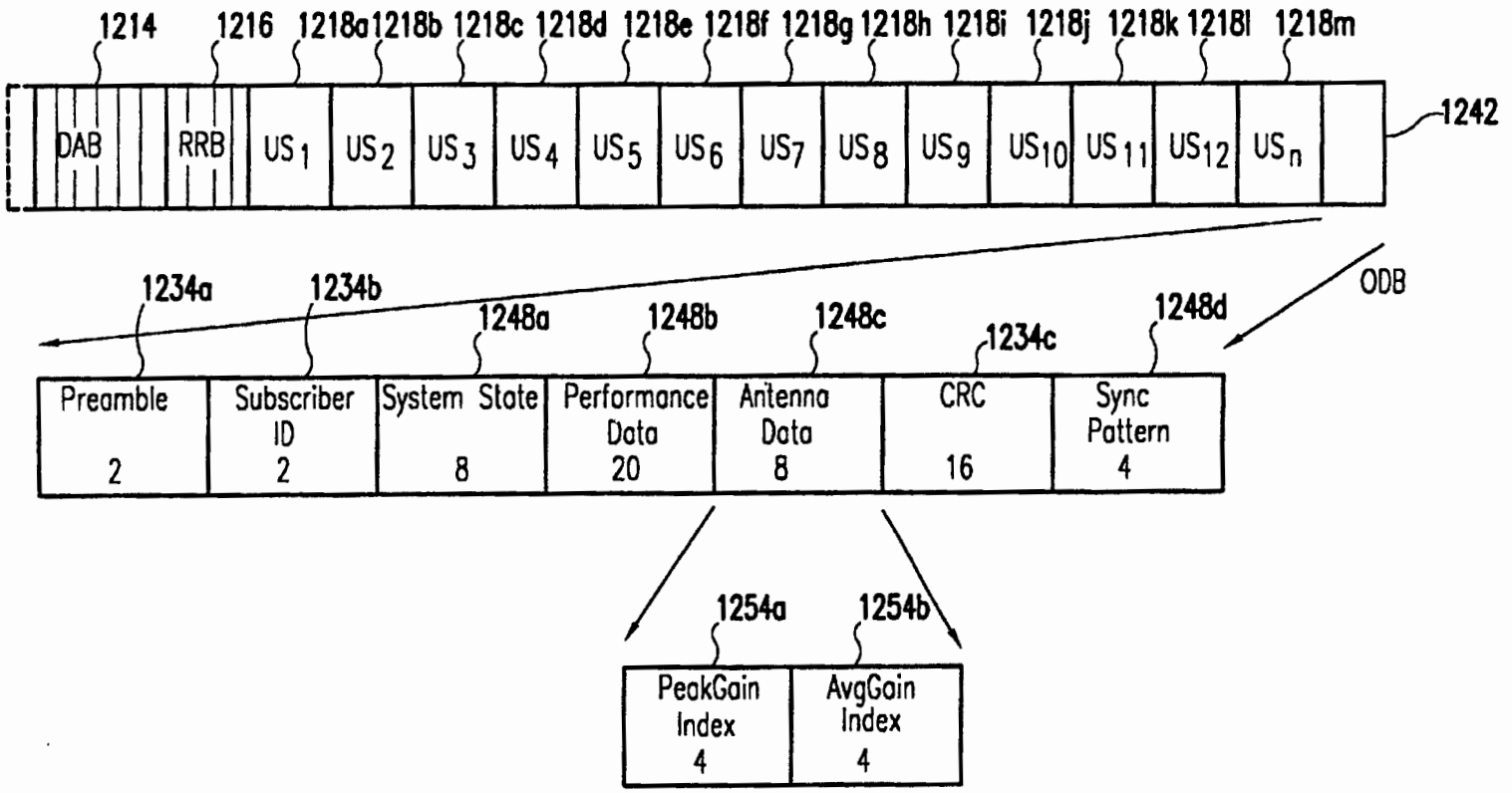

1214
1216
1218a
1218b
1218c
1218d
1218e
1218f
1218g
1218h
1218i
1218j
1218k
1218l
1218m
DAB
RRB
US1
US2
US3
US4
US5
US6
US7
US8
US9
US10
US11
US12
USn
1242
1234a
1234b
1248a
1248b
1248c
1234c
1248d
ODB
Preamble
2
Subscriber ID
2
System State
8
Performance Data
20
Antenna Data
8
CRC
16
Sync Pattern
4
1254a
1254b
PeakGain Index
4
AvgGain Index
4

FIG.120

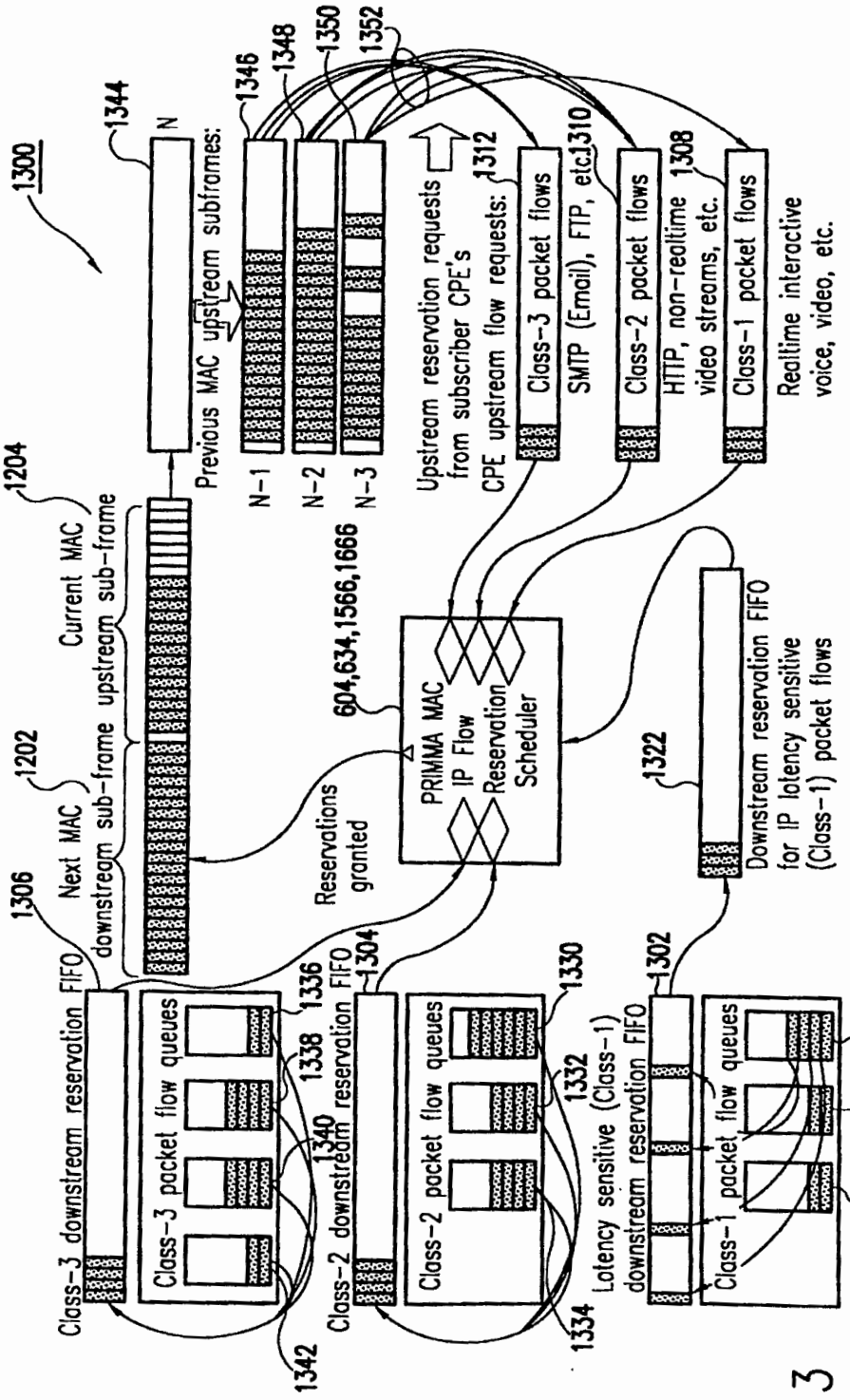

1300
1344
N
1346
1348
1350
1352
Previous MAC upstream subframes:
N-1
N-2
N-3
1204
Current MAC
upstream sub-frame
downstream sub-frame
1202
Next MAC
1306
Class-3 downstream reservation FIFO
Class-3 packet flow queues
1338
1336
1340
Class-2 downstream reservation FIFO
1304
1342
Reservations granted
604,634,1566,1666
PRIMMA MAC
IP Flow
Reservation
Scheduler
Upstream reservation requests from subscriber CPE's
CPE upstream flow requests:
1312
Class-3 packet flows
SMTP (Email), FTP, etc.
1310
Class-2 packet flows
HTTP, non-realtime video streams, etc.
1308
Class-1 packet flows
Realtime interactive voice, video, etc.
1322
Downstream reservation FIFO for IP latency sensitive (Class-1) packet flows
1330
Class-2 packet flow queues
1332
1334
Latency sensitive (Class-1) downstream reservation FIFO
1302
Class-1 packet flow queues
1324
1326
1328

FIG. 13

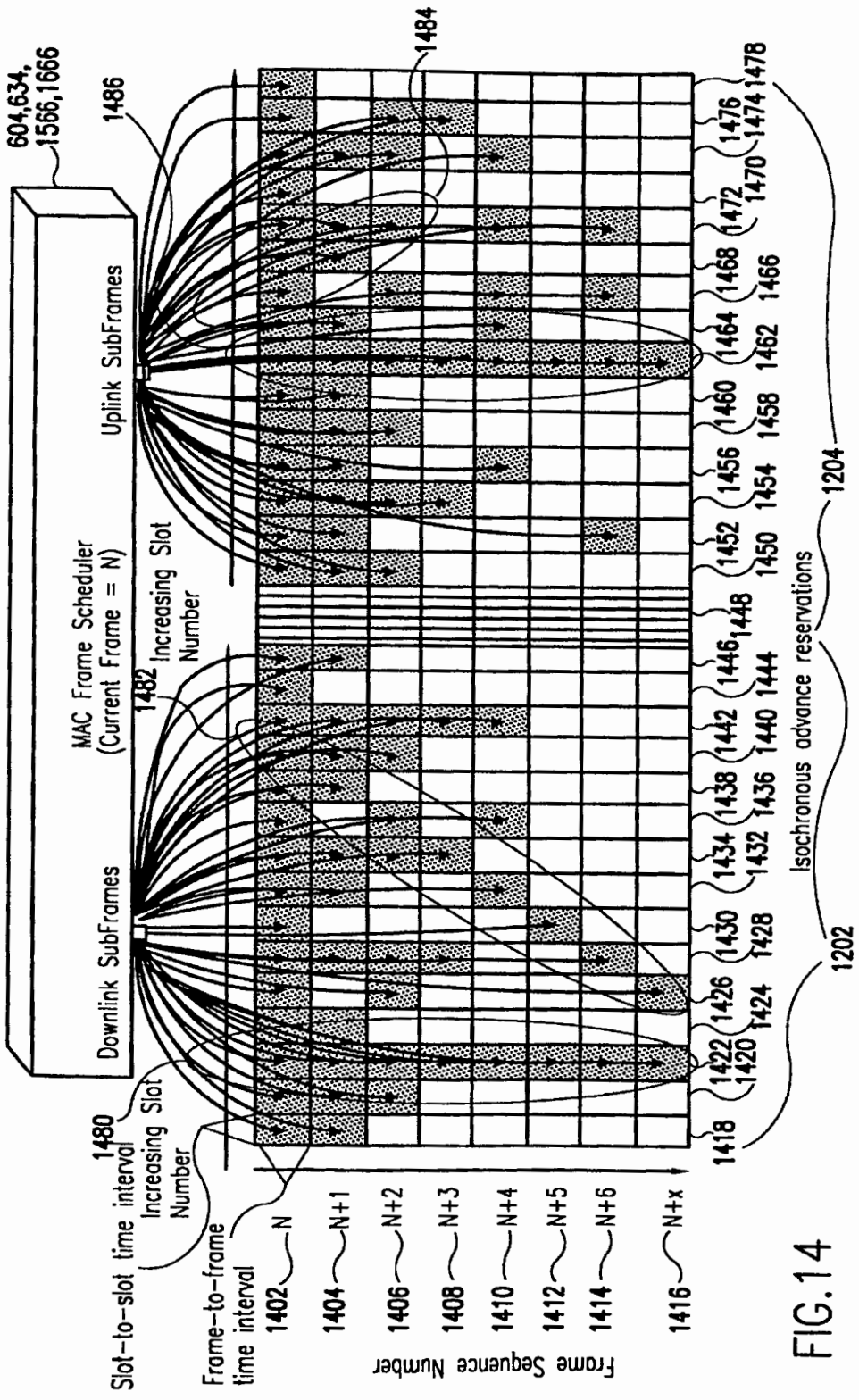

604,634,
1566,1666
1486
1484
1478
1476
1474
1470
1472
1468
1466
1464
1462
1460
1458
1456
1454
1452
1450
1448
1446
1444
1442
1440
1438
1436
1434
1432
1430
1428
1426
1424
1422
1420
1418
1204
1202
Isochronous advance reservations
Uplink SubFrames
Downlink SubFrames
MAC Frame Scheduler
(Current Frame = N)
1482
Increasing Slot
Number
1480
Slot-to-slot time interval
Increasing Slot
Number
Frame-to-frame
time interval
1402
N
1404
N+1
1406
N+2
1408
N+3
1410
N+4
1412
N+5
1414
N+6
1416
N+x
Frame Sequence Number

FIG. 14

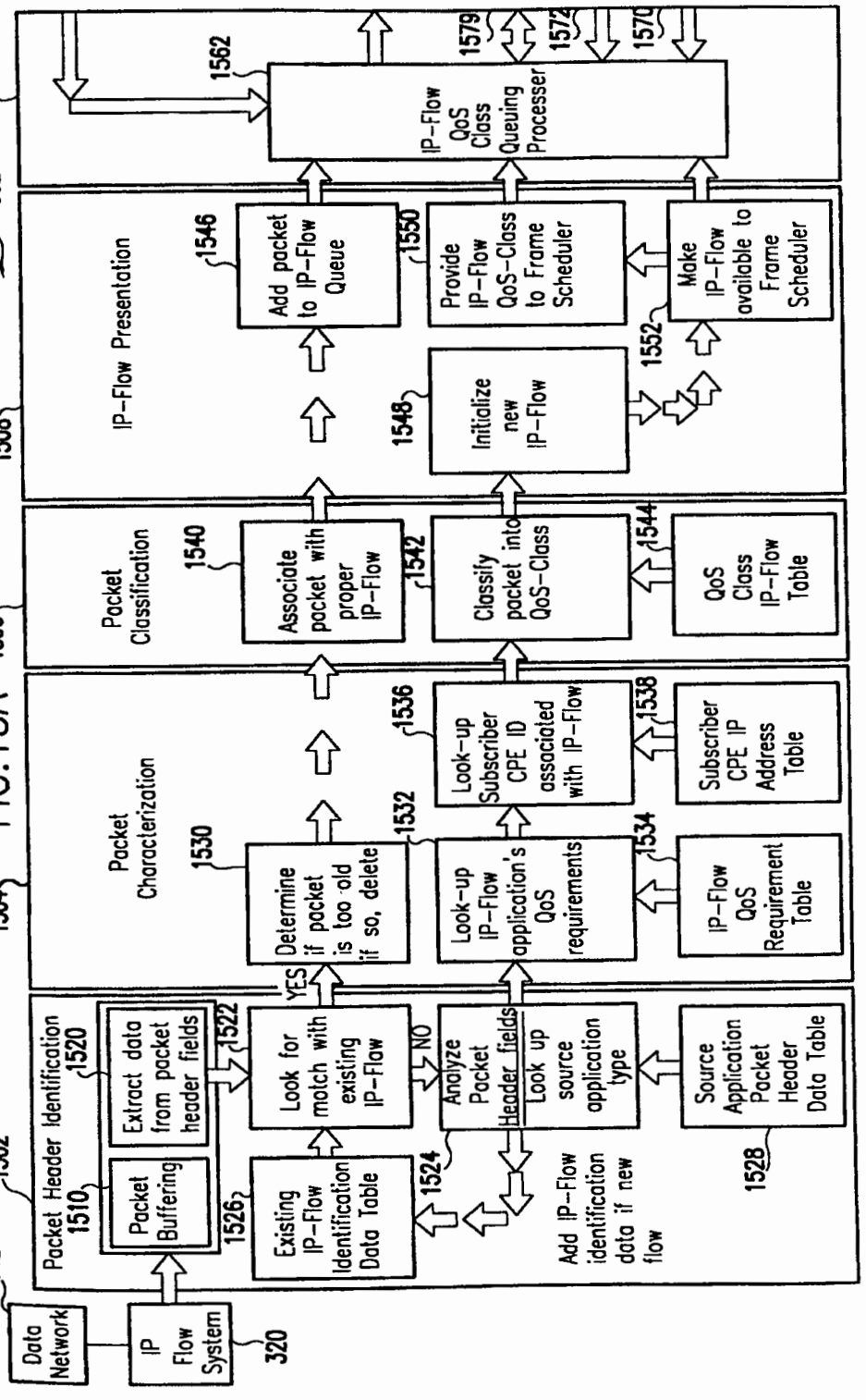

Data Network
142
IP Flow System
320
Packet Header Identification
1502
1520
Extract data from packet header fields
1522
Packet Buffering
1510
1526
Existing IP-Flow Identification Data Table
Look for match with existing IP-Flow
YES
NO
Analyze Packet Header fields
1524
Look up source application type
Add IP-Flow identification data if new flow
Source Application Packet Header Data Table
1528
Packet Characterization
1504
1530
Determine if packet is too old if so, delete
1532
Look-up IP-Flow application's QoS requirements
1534
IP-Flow QoS Requirement Table
Look-up Subscriber CPE ID associated with IP-Flow
1536
1538
Subscriber CPE IP Address Table
Packet Classification
1506
1540
Associate packet with proper IP-Flow
1542
Classify packet into QoS-Class
1544
QoS Class IP-Flow Table
IP-Flow Presentation
1508
1500
602
1546
Add packet to IP-Flow Queue
1548
Initialize new IP-Flow
1550
Provide IP-Flow QoS-Class to Frame Scheduler
1552
Make IP-Flow available to Frame Scheduler
604
1562
IP-Flow QoS Class Queuing Processer
1579
1572
1570

FIG. 15A

FIG.15B

604
1560
604
1578
Link Layer Acknowledgment (ARQ) Processor
1576
Wireless Medium
1566
MAC Downlink Subframe Scheduler
Requests for retransmissions from CPE
290d
292d
RF Transceiver Subscriber Antenna
1562
MAC Downlink Subframes
IP-Flow QoS Class Queues
Frame n 1568a
Frame n+1 1568b
Frame n+2 1568c
Frame n+3 1568d
Frame n+4 1568e
Frame n+5 1568f
Frame n+6 1568g
Frame n+7 1568h
Frame n+8 1568i
Frame n+9 1568j
Frame n+p 1568k
Class 1 1564a
Class 2 1564b
Class 3 1564c
Class 4 1564d
Class 5 1564e
Class m 1564f
Completed Downlink Subframe sent to wireless modem for transmission
Wireless Access Point (WAP) Antenna
IP-Flow QoS Class Queuing Processor
1574
1574
Hierarchical Class-Based Priority Processor
Frames advance in this direction with passage of time
120d
CPE Subscriber Station
1572
1572
VPN DEN Data Table
CPE IP-Flow Queue Depth Status Processor
1582
CPE IP-Flow Queue Depth Messages
1580
294d
Subscriber Workstation
Service Level Agreement (SLA) Priority Data Table
1108
OAM&P
1570
Input data from OAM&P Module

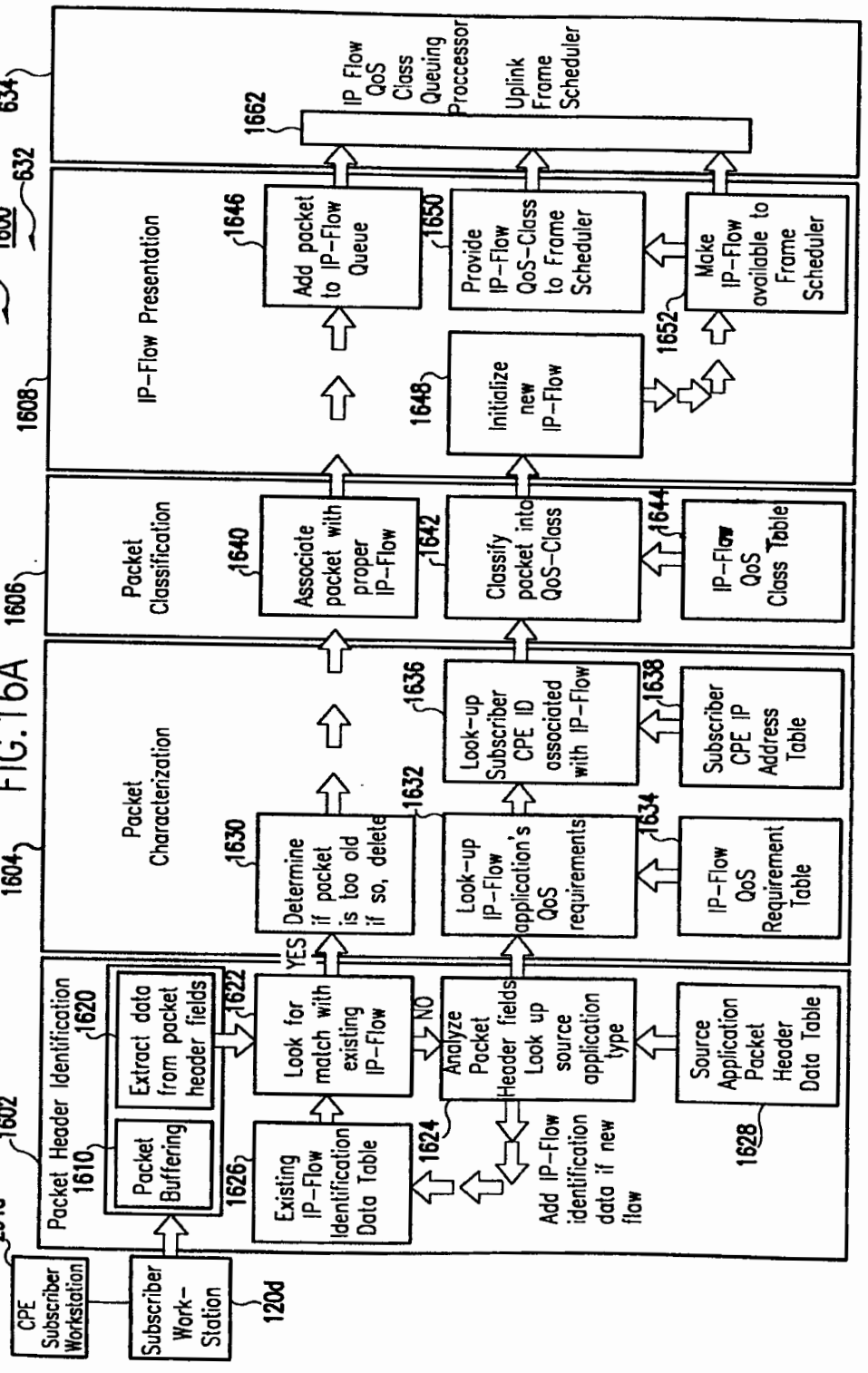

CPE Subscriber Workstation
294d
Subscriber Work-Station
120d
Packet Header Identification
1602
1610
Packet Buffering
1620
Extract data from packet header fields
1622
1626
Existing IP-Flow Identification Data Table
Look for match with existing IP-Flow
YES
NO
1624
Add IP-Flow identification data if new flow
Analyze Packet Header fields
Look up source application type
1628
Source Application Packet Header Data Table
1604
Packet Characterization
1630
Determine if packet is too old if so, delete
1632
Look-up IP-Flow application's QoS requirements
1634
IP-Flow QoS Requirement Table
1636
Look-up Subscriber CPE ID associated with IP-Flow
1638
Subscriber CPE IP Address Table
1606
Packet Classification
1640
Associate packet with proper IP-Flow
1642
Classify packet into QoS-Class
1644
IP-Flow QoS Class Table
1600
1608
IP-Flow Presentation
1646
Add packet to IP-Flow Queue
1648
Initialize new IP-Flow
1650
Provide IP-Flow QoS-Class to Frame Scheduler
1652
Make IP-Flow available to Frame Scheduler
632
634
1662
IP Flow QoS Class Queuing Proccessor
Uplink Frame Scheduler

FIG. 16A

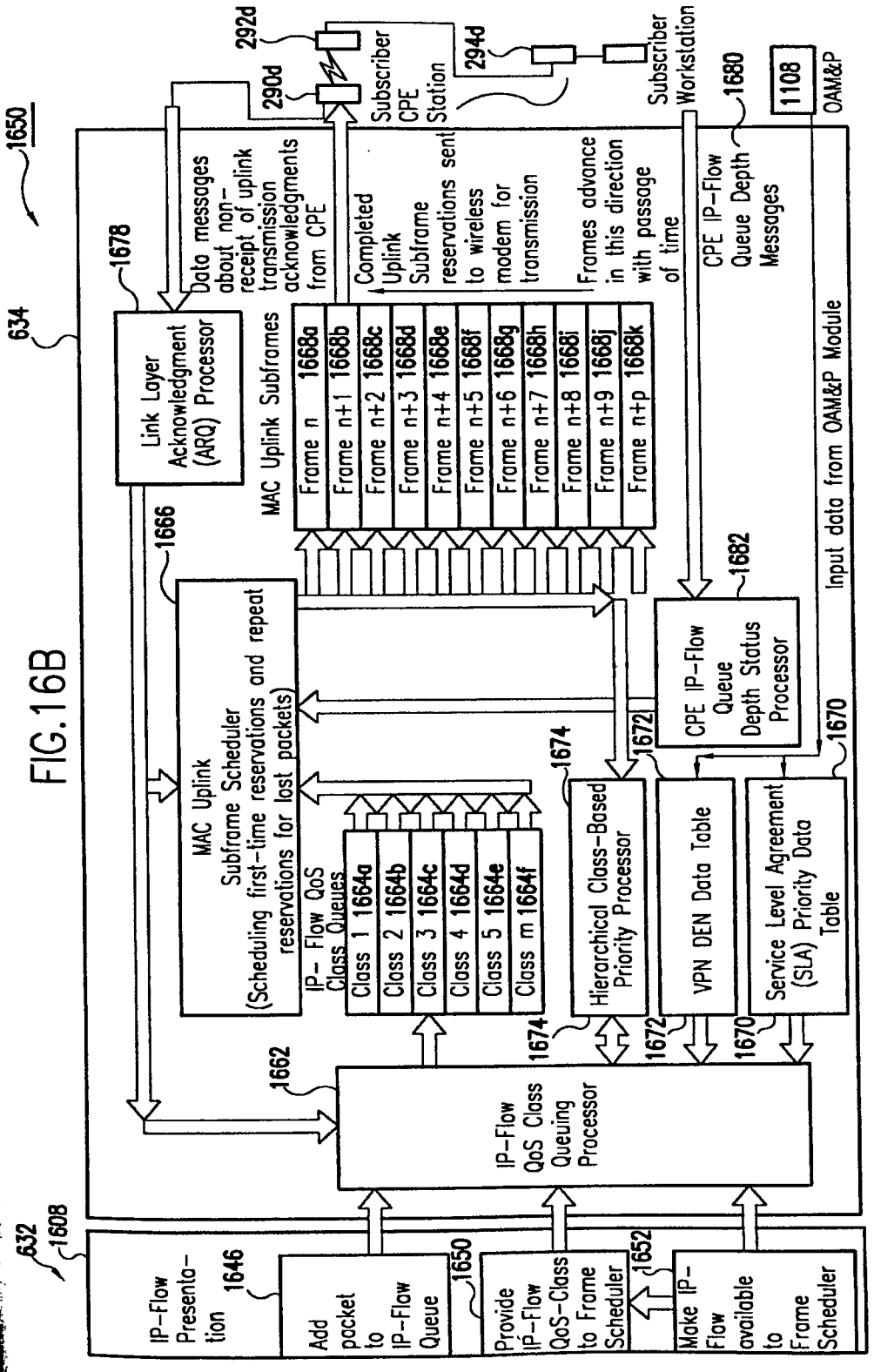

1650
634
1678
Link Layer Acknowledgment (ARQ) Processor
Data messages about non-receipt of uplink transmission acknowledgments from CPE
292d
290d
Subscriber CPE Station
294d
Subscriber Workstation
1680
1108
OAM&P
Completed Uplink Subframe reservations sent to wireless modem for transmission
Frames advance in this direction with passage of time
CPE IP-Flow Queue Depth Messages
MAC Uplink Subframes
Frame n 1668a
Frame n+1 1668b
Frame n+2 1668c
Frame n+3 1668d
Frame n+4 1668e
Frame n+5 1668f
Frame n+6 1668g
Frame n+7 1668h
Frame n+8 1668i
Frame n+9 1668j
Frame n+p 1668k
1666
MAC Uplink Subframe Scheduler (Scheduling first-time reservations and repeat reservations for lost packets)
1682
CPE IP-Flow Queue Depth Status Processor
Input data from OAM&P Module
IP- Flow QoS Class Queues
Class 1 1664a
Class 2 1664b
Class 3 1664c
Class 4 1664d
Class 5 1664e
Class m 1664f
1674
Hierarchical Class-Based Priority Processor
1672
VPN DEN Data Table
1670
Service Level Agreement (SLA) Priority Data Table
1662
IP-Flow QoS Class Queuing Processor
632
1608
IP-Flow Presentation 1646
Add packet to IP-Flow Queue
1650
Provide IP-Flow QoS-Class to Frame Scheduler
1652
Make IP-Flow available to Frame Scheduler

FIG.16B

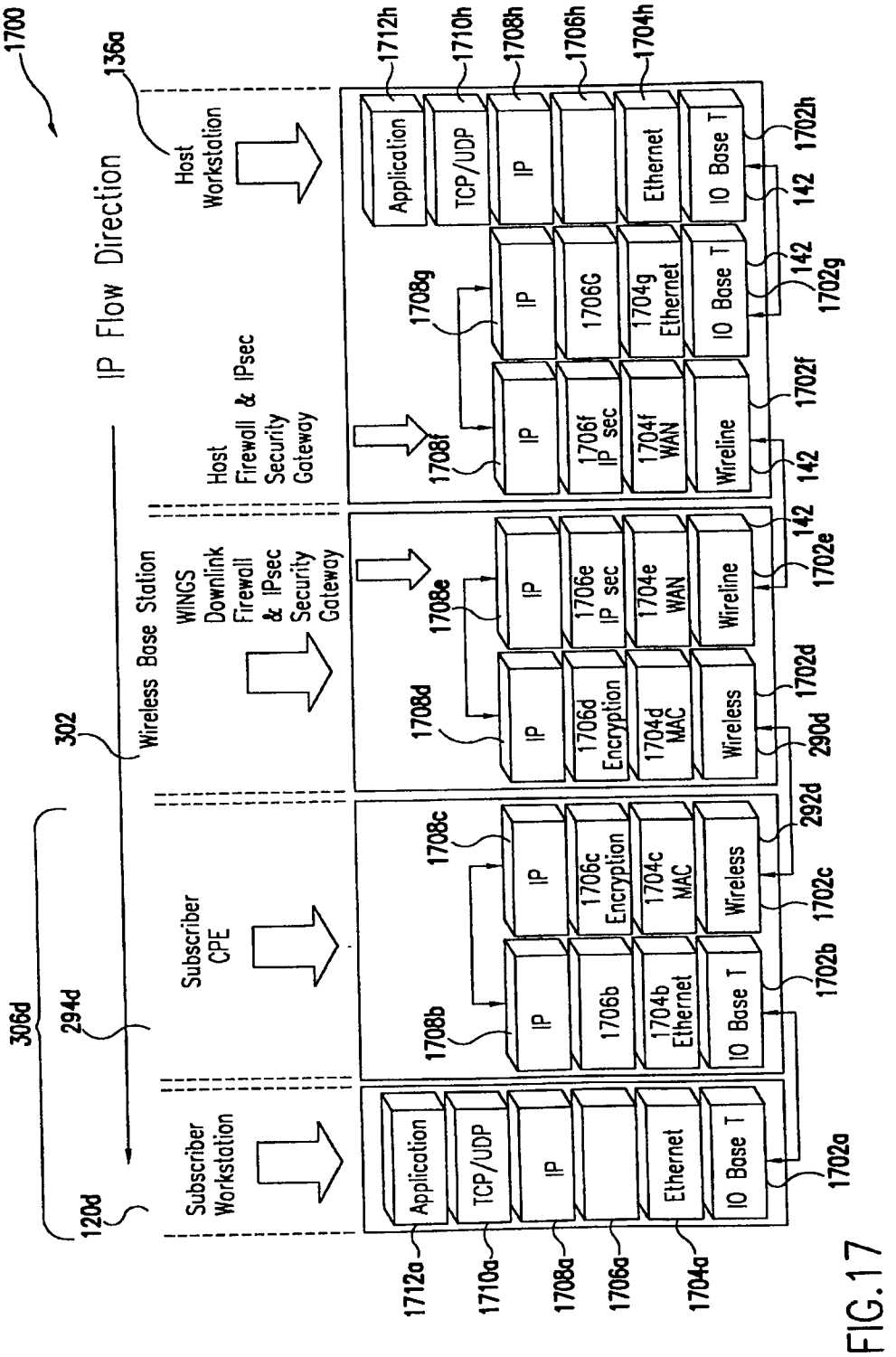

1700
136a
Host Workstation
IP Flow Direction
1712h
1710h
1708h
1706h
1704h
Application
TCP/UDP
IP
Ethernet
10 Base T
1702h
142
1708g
1706G
1704g Ethernet
1702g
Host Firewall & IPsec Security Gateway
1708f
1706f IP sec
1704f WAN
Wireline
1702f
WINGS Downlink Firewall & IPsec Security Gateway
1708e
1706e IP sec
1704e WAN
1702e
1708d
1706d Encryption
1704d MAC
Wireless
1702d
290d
302
Wireless Base Station
Subscriber CPE
1708c
1706c Encryption
1704c MAC
292d
1702c
306d
294d
1708b
1706b
1704b Ethernet
1702b
Subscriber Workstation
120d
1702a
1712a
1710a
1708a
1706a
1704a

FIG. 17

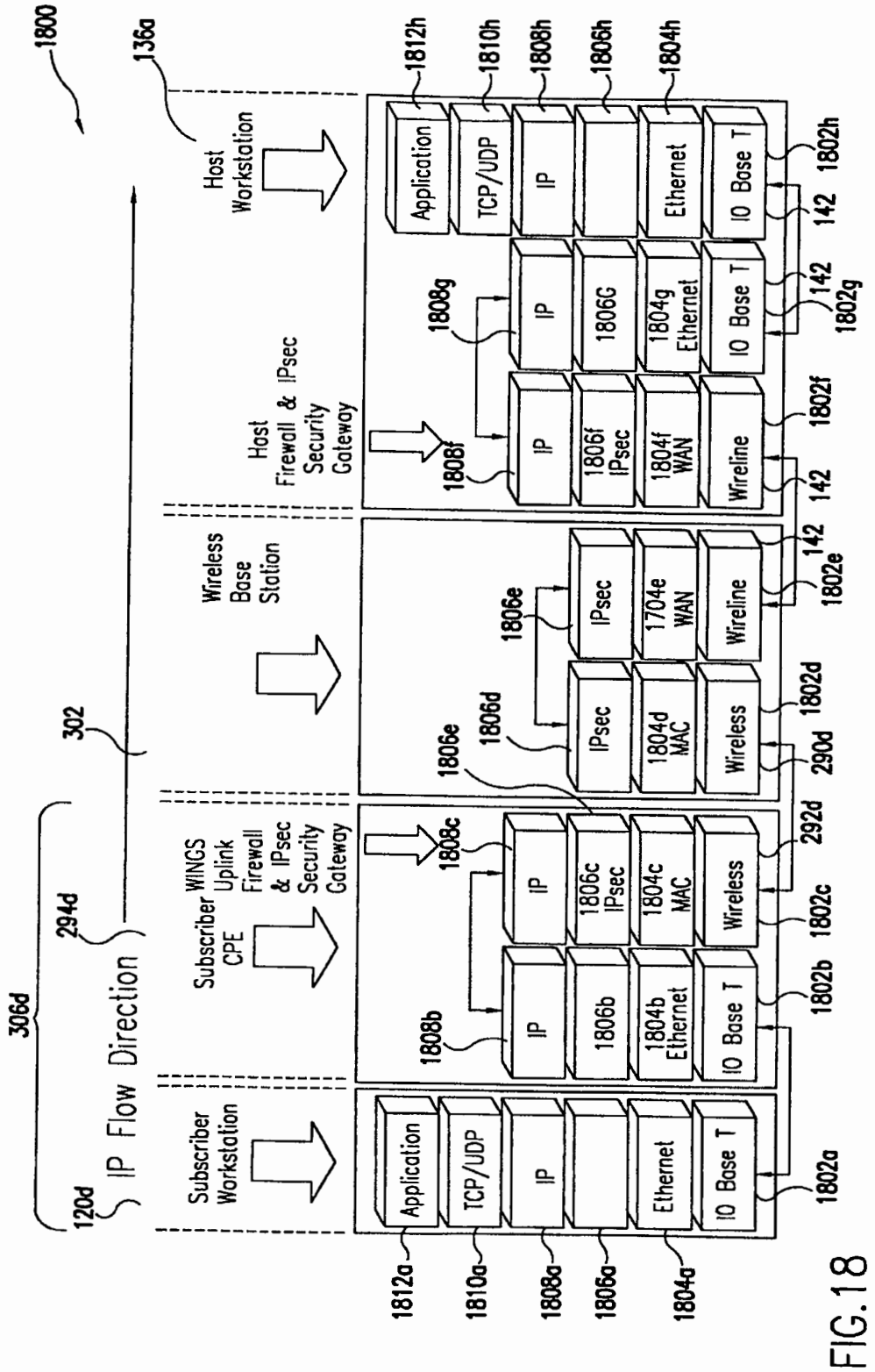

1800
136a
Host Workstation
1812h
1810h
1808h
1806h
1804h
Application
TCP/UDP
IP
Ethernet
10 Base T
1802h
142
Host Firewall & IPsec Security Gateway
1808g
1806G
1804g Ethernet
1802g
1808f
1806f IPsec
1804f WAN
Wireline
1802f
Wireless Base Station
1806e
IPsec
1704e WAN
1802e
1806d
1804d MAC
Wireless
290d
1802d
Subscriber WINGS Uplink Firewall & IPsec Security Gateway
1808c
1806c IPsec
1804c MAC
292d
1802c
Subscriber CPE
1808b
1806b
1804b Ethernet
1802b
Subscriber Workstation
1802a
302
294d
306d
120d
IP Flow Direction
1812a
1810a
1808a
1806a
1804a

FIG. 18

IP-FLOW CLASSIFICATION IN A WIRELESS POINT TO MULTI-POINT (PTMP) TRANSMISSION SYSTEM

This application claims benefit of priority from U.S. Provisional Patent Application No. 60/092,452, filed Jul. 10, 1998.

CROSS-REFERENCE TO OTHER APPLICATIONS

The following applications of common assignee contain common disclosure:

U.S. patent application Ser. No. 09/349,477 entitled "Transmission Control Protocol/Internet Protocol (TCP/IP) Packet-Centric Wireless Point to Multi-Point (PtMP) Transmission System Architecture," filed Jul. 9, 1999,

U.S. patent application Ser. No. 09/349,480 entitled "Quality of Service (QoS)—Aware Wireless Point to Multi-Point (PtMP) Transmission System Architecture," filed Jul. 9, 1999.

U.S. patent application Ser. No. 09/350,126 entitled "Method for Providing Dynamic Bandwidth Allocation Based on IP-Flow Characteristics in a Wireless Point to Multi-Point (PtMP) Transmission System," filed Jul. 9, 1999.

U.S. patent application Ser. No. 09/350,118 entitled "Method for Providing for Quality of Service (QoS)—Based Handling of IP-Flows in a Wireless Point to Multi-Point Transmission System," filed Jul. 9, 1999,

U.S. patent application Ser. No. 09/347,356 entitled "IP-Flow Identification in a Wireless Point to Multi-Point Transmission System," filed Jul. 9, 1999.

U.S. patent application Ser. No. 09/350,150 entitled "IP-Flow Characterization in a Wireless Point to Multi-Point (PtMP) Transmission System," filed Jul. 9, 1999.

U.S. patent application Ser. No. 09/349,476 entitled "IP-Flow Prioritization in a Wireless Point to Multi-Point (PtMP) Transmission System," filed Jul. 9, 1999.

U.S. patent application Ser. No. 09/350,170 entitled "Method of Operation for Providing for Service Level Agreement (SLA) Based Prioritization in a Wireless Point to Multi-Point (PtMP) Transmission System," filed Jul. 9, 1999.

U.S. patent application Ser. No. 09/349,481 entitled "Method for Transmission Control Protocol (TCP) Rate Control With Link-Layer Acknowledgments in a Wireless Point to Multi-Point (PtMP) Transmission System," filed Jul. 9, 1999.

U.S. patent application Ser. No. 09/350,159 entitled "Transmission Control Protocol/Internet Protocol (TCP/IP)—Centric QoS Aware Media Access Control (MAC) Layer in a Wireless Point to Multi-Point (PtMP) Transmission System," filed Jul. 9, 1999.

U.S. patent application Ser. No. 09/347,857 entitled "Use of Priority-Based Scheduling for the Optimization of Latency and Jitter Sensitive IP Flows in a Wireless Point to Multi-Point Transmission System," filed Jul. 9, 1999.

U.S. patent application Ser. No. 09/349,475 entitled "Time Division Multiple Access/Time Division Duplex (TDMA/TDD) Access Method for a Wireless Point to Multi-Point Transmission System," filed Jul. 9, 1999.

U.S. patent application entitled Ser. No. 09/349,483 "Reservation Based Prioritization Method for Wireless Transmission of Latency and Jitter Sensitive IP-Flows in a Wireless Point to Multi-Point Transmission System," filed Jul. 9, 1999.

U.S. patent application Ser. No. 09/349,479 entitled "Translation of Internet-Prioritized Internet Protocol (IP)—

Flows into Wireless System Resource Allocations in a Wireless Point to Multi-Point (PtMP) Transmission System," filed Jul. 9, 1999.

U.S. patent application entitled Ser. No. 09/350,162 "Method of Operation for the Integration of Differentiated services (Diff-serv) Marked IP-Flows into a Quality of Service (QoS) Priorities in a Wireless Point to Multi-Point (PtMP) Transmission System," filed Jul. 9, 1999.

U.S. patent application Ser. No. 09/349,975 entitled "Method for the Recognition and Operation of Virtual Private Networks (VPNs) over a Wireless Point to Multi-Point (PtMP) Transmission System," filed Jul. 9, 1999.

U.S. patent application Ser. No. 09/350,173 entitled "Time Division Multiple Access/Time Division Duplex (TDMA/TDD) Transmission Media Access Control (MAC) Air Frame," filed Jul. 9, 1999.

U.S. patent application Ser. No. 09/349,482 entitled "Application—Aware, Quality of Service (QoS) Sensitive, Media Access Control (MAC) Layer," filed Jul. 9, 1999.

U.S. patent application Ser. No. 09/349,478 entitled "Transmission Control Protocol/Internet Protocol (TCP/IP) Packet-Centric Wireless Point to Point (PtP) Transmission System Architecture," filed Jul. 9, 1999.

U.S. patent application Ser. No. 09/349,474 entitled "Transmission Control Protocol/Internet Protocol (TCP/IP) Packet-Centric Cable Point to Multi-Point (PtMP) Transmission System Architecture," filed Jul. 9, 1999.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates generally to telecommunications and, more particularly, to a system and method for implementing a QoS aware wireless point-to-multi-point transmission system.

2. Related Art

Telecommunication networks such as voice, data and video networks have conventionally been customized for the type of traffic each is to transport. For example, voice traffic is very latency sensitive but quality is less important, so voice networks are designed to transport voice traffic with limited latency. Traditional data traffic, such as, e.g., a spreadsheet, on the other hand is not latency sensitive, but error-free delivery is required. Conventional telecommunications networks use circuit switching to achieve acceptable end user quality of service (QoS). With the advent of new packet switching high bandwidth data networks, different types of traffic can be transported over a data network. Specifically, convergence of separate voice, data and video networks into a single broadband telecommunications network is enabled. To ensure end user satisfaction, a system is desired that provides QoS for various types of traffic to be transported.

Wireless networks present particular challenges over their wireline counterparts in delivering QoS. For example, wireless networks traditionally exhibit high bit error rates (BER) due to a number of reasons. Conventional wireless networks also implement circuit switched connections to provide reliable communications channels. However the use of circuit switched connections allocates bandwidth between communicating nodes whether or not traffic is constantly being transferred between the nodes. Therefore, circuit switched connections use communications bandwidth rather inefficiently.

Packet switching makes more efficient use of available bandwidth than does traditional circuit switching. Packet

switching breaks up traffic into so-called "packets" which can then be transported from a source node to a destination for reassembly. Thus a particular portion of bandwidth can be shared by many sources and destinations yielding more efficient use of bandwidth.

A wireless broadband access telecommunications system is desired which can provide a QoS capability that is comparable to that delivered by wireline broadband access devices. Conventionally, one of the barriers to the deployment of wireless broadband access systems has been the absence of acceptable QoS characteristics, while at the same time delivering bandwidth sufficient to qualify as broadband. Delivery of raw bandwidth over wireless media without acceptable QoS would not benefit end users. Likewise, the delivery of a high level of QoS at the cost of sufficient bandwidth would also not benefit endusers.

Conventional efforts to provide wireless broadband access systems have not granted sufficient priority to QoS as a guiding principle in architecting the wireless systems, resulting in sub-optimal designs. With the rapid emergence of the Internet, the packet switching paradigm, and transmission control protocol/internet protocol (TCP/IP) as a universal data protocol, it has become clear that a new wireless system design has become necessary.

What is needed then is an IP-centric wireless broadband access system with true QoS capabilities.

SUMMARY OF THE INVENTION

The present invention is directed to an IP flow classification system used in a wireless telecommunications system. More specifically, the IP flow classification system groups IP flows in a packet-centric wireless point to multipoint telecommunications system.

The classification system includes: a wireless base station coupled to a first data network; one or more host workstations coupled to the first data network; one or more subscriber customer premise equipment (CPE) stations in wireless communication with the wireless base station over a shared bandwidth using a packet-centric protocol; and one or more subscriber workstations coupled to each of the subscriber CPE stations over a second network; a resource allocation device optimizes end-user quality of service (QoS) and allocates shared bandwidth among the subscriber CPE stations; an analyzing and scheduling device analyzes and schedules internet protocol (IP) flow over the shared wireless bandwidth. The analyzing device includes the above IP flow classifier that classifies the IP flow.

In one embodiment, the classifier includes a means for associating a packet of an existing IP flow with the IP flow. The classifier can include a QoS grouping device that groups a packet of a new IP flow into a QoS class grouping. The QoS grouping device can include a determining device that determines and takes into account QoS class groupings for the IP flow. The QoS grouping device can include an optional differentiated services (Diff Serv) device that takes into account an optional Diff Servs field priority marking for the IP flow. The QoS grouping device can also include an optional type of service (TOS) device that takes into account any optional type of service (TOS) field priority marking for said IP flow.

The cross-referenced applications are incorporated herein by reference in their entireties.

BRIEF DESCRIPTION OF THE FIGURES

The present invention will be described with reference to the accompanying figures, wherein:

FIG. **1A** is a block diagram providing an overview of a standard telecommunications network providing local exchange carrier services within one or more local access and transport areas;

FIG. **1B** depicts an exemplary network including workstations coupled to a data network;

FIG. **1C** illustrates a conventional video network, such as for example a cable television (CATV) network;

FIG. **2A** is a block diagram illustrating an overview of a standard telecommunications network providing both local exchange carrier and interexchange carrier services between subscribers located in different local access and transport areas;

FIG. **2B** illustrates a signaling network in detail;

FIG. **2C** illustrates an exemplary network carrying voice, data and video traffic over a data network;

FIG. **2D** depicts a network including a point-to-multipoint wireless network coupled via a router to a data network;

FIG. **3A** depicts an exemplary perspective diagram of a point-to-multipoint network;

FIG. **3B** depicts a block diagram further illustrating a wireless point-to-multipoint network;

FIG. **4** depicts a wireless Internet protocol network access architecture of the present invention;

FIG. **5A** depicts Internet protocol flows from a subscriber host to a wireless base station, and through a wireline connection to a destination host;

FIG. **5B** illustrates a functional flow diagram including an example functional description of a transmission control protocol adjunct agent performing an outgoing transmission control protocol spoof function;

FIG. **5C** illustrates a functional flow diagram including an exemplary functional description of a transmission control protocol adjunct agent performing an incoming transmission control protocol spoof function;

FIG. **6** illustrates a block diagram representing scheduling of mixed Internet protocol flows;

FIG. **7** illustrates packet header field information which can be used to identify Internet protocol flows and the quality of service requirements of the Internet protocol flows;

FIG. **8A** is a block diagram summarizing an exemplary downlink analysis, prioritization and scheduling function;

FIG. **8B** is a block diagram summarizing an exemplary uplink analysis prioritization and scheduling function;

FIG. **9** illustrates how a downlink flow scheduler can take into account a service level agreement in prioritizing a frame slot and scheduling resource allocation;

FIG. **10** depicts an embodiment of an inventive media access control hardware architecture;

FIG. **11** is an exemplary software organization for a packet-centric wireless point to multi-point telecommunications system;

FIG. **12A** illustrates an exemplary time division multiple access media access control air frame;

FIG. **12B** illustrates an exemplary structure for a time division multiple access/time division duplex air frame;

FIG. **12C** illustrates an exemplary downstream transmission subframe;

FIG. **12D** illustrates an exemplary upstream acknowledgment block field of a downstream transmission subframe;

FIG. **12E** illustrates an exemplary acknowledgment request block field of a downstream transmission subframe;

FIG. 12F illustrates an exemplary frame descriptor block field of a downstream transmission subframe;

FIG. 12G illustrates an exemplary downstream media access control payload data unit of a downstream transmission subframe;

FIG. 12H illustrates an exemplary command and control block of a downstream transmission subframe;

FIG. 12I illustrates an exemplary upstream transmission subframe;

FIG. 12J illustrates an exemplary downstream acknowledgment block of an upstream transmission subframe;

FIG. 12K illustrates an exemplary reservation request block of an upstream transmission subframe **1204**;

FIG. 12L illustrates an exemplary media access control payload data unit of an upstream transmission subframe;

FIGS. 12M, 12N and 12O illustrate an exemplary operations data block of an upstream transmission subframe;

FIG. 13 illustrates how an exemplary flow scheduler for the present invention functions;

FIG. 14 is an exemplary two-dimensional block diagram of an advanced reservation algorithm;

FIG. 15A is an exemplary logical flow diagram for a downlink flow analyzer;

FIG. 15B is an exemplary logical flow diagram for a downlink flow scheduler;

FIG. 16A is an exemplary logical flow diagram for an uplink flow analyzer;

FIG. 16B is an exemplary logical flow diagram for an uplink flow scheduler;

FIG. 17 illustrates Internet protocol flow in a downlink direction, including Internet protocol security encryption; and

FIG. 18 illustrates an uplink direction of Internet protocol security support.

In the figures, like reference numbers generally indicate identical, functionally similar, and/or structurally similar elements. The figure in which an element first appears is indicated by the leftmost digit(s) in the reference number.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

I. An Example Environment

The present invention is described in terms of an example environment. The example environment uses a fixed wireless point-to-multi-point (PtMP) connection to transmit packetized data information including for example, IP telephony, video, data, received from a telecommunications carrier. As used herein, a telecommunications carrier can include US domestic entities (see Definitions below at section II) such as, e.g., ILECs, CLECs, IXCs, NGTs and Enhanced Service Providers (ESPs), as well as global entities such as PTTs and NEs, recognized by those skilled in the art. In addition, as used herein a telecommunications system includes domestic systems used by entities such as, e.g., ILECs, CLECs, IXCs and Enhanced Service Providers (ESPs), as well as global systems recognized by those skilled in the art.

In the preferred embodiment, the traffic arrives from a wide area network (WAN) connection.

Data traffic is received from a data network through a network router and can be demodulated from internet protocol (IP) format to, for example, the point-to-point protocol (PPP). Network routers can include, for example, a general purpose computer, such as the SUN workstation running routing software or a dedicated routing device such as various models from CISCO of San Jose, Calif., ASCEND of Alameda, Calif., NETOPIA of Alameda, Calif., or 3COM of Santa Clara, Calif.

In the alternative, a virtual private networking protocol, such as the point-to-point tunneling protocol (PPTP), can be used to create a "tunnel" between a remote user and a corporate data network. A tunnel permits a network administrator to extend a virtual private network from a server (e.g., a Windows NT server) to a data network (e.g., the Internet).

Although the invention is described in terms of this example environment, it is important to note that description in these terms is provided for purposes of illustration only. It is not intended that the invention be limited to this example environment or to the precise inter-operations between the above-noted devices. In fact, after reading the following description, it will become apparent to a person skilled in the relevant art how to implement the invention in alternative environments.

II. Definitions

Table 1 below defines common telecommunications terminology. These terms are used throughout the remainder of the description of the invention.

TABLE 1

Term	Definition
access tandem (AT)	An AT is a class ¾ switch used to switch calls between EOs in a LATA. An AT provides subscribers access to the IXCs, to provide long distance calling services. An access tandem is a network node. Other network nodes can include, for example, a CLEC, or other enhanced services provider (ESP), an international gateway or global point-of-presence (GPOP), or an intelligent peripheral (IP).
bearer (B) channels	Bearer (B) channels are digital channels used to carry both digital voice and digital data information. An ISDN bearer channel is 64,000 bits per second, which can carry PCM-digitized voice or data.
called party	The called party is the caller receiving a call sent over a network at the destination or termination end.
calling party	The calling party is the caller placing a call over any kind of network from the origination end.
central office (CO)	A CO is a facility that houses an EO homed. EOs are often called COs.
class 1 switch	A class 1 switching office, the Regional Center (RC), is the highest level of local and long distance switching, or "office of last resort" to complete a call.
class 3 switch	A class 3 switching office was a Primary Center (PC); an access tandem (AT) has class 3 functionality.
class 4 switch	A class 4 switching office was a Toll Center (TC) if operators were present or else a Toll Point (TP); an access tandem (AT) has class 4 functionality.
class 5 switch	A class 5 switching office is an end office (EO) or the lowest level of local and long distance switching, a local central office. The switch closest to the end subscriber.
competitive LEC (CLEC)	CLECs are telecommunications services providers of local services that can compete with ILECs. Interprise and Century 21 are examples. A CLEC may or may not handle IXC services as well.
competitive access providers (CAPS)	Teligent and Winstar are examples.
customer premises equipment (CPE)	CPE refers to devices residing on the premises of a customer and used to connect to a telephone network, including ordinary telephones, key telephone systems, PBXs, video conferencing devices and modems.

TABLE 1-continued

Term	Definition
digitized data (or digital data)	Digitized data refers to analog data that has been sampled into a binary representation (i.e., comprising sequences of 0's and 1's). Digitized data is less susceptible to noise and attenuation distortions because it is more easily regenerated to reconstruct the original signal.
egress end office	The egress EO is the node or destination EO with a direct connection to the called party, the termination point. The called party is "homed" to the egress EO.
egress	Egress refers to the connection from a called party or termination at the destination end of a network, to the serving wire center (SWC).
end office (EO)	An EO is a class 5 switch used to switch local calls within a LATA. Subscribers of the LEC are connected ("homed") to EOs, meaning that EOs are the last switches to which the subscribers are connected.
Enhanced Service Provider (ESP)	A network services provider.
equal access	1 + dialing as used in US domestic calling for access to any long distance carrier as required under the terms of the modified final judgment (MFJ) requiring divestiture of the Regional Bell Operating Companies (RBOCs) from their parent company, AT&T.
global point of presence (GPOP)	A GPOP refers to the location where international telecommunications facilities and domestic facilities interface, an international gateway POP.
incumbent LEC (ILEC)	ILECs are traditional LECs in the US, which are the Regional Bell Operating Companies (RBOCs). Bell South and US West are examples. ILEC can also stand for an independent LEC such as a GTE.
ingress end office	The ingress EO is the node or serving wire center (SVC) with a direct connection to the calling party, the origination point. The calling party is "homed" to the ingress EO.
ingress	Ingress refers to the connection from a calling party or origination.
integrated service digital network (ISDN) basic rate interface (BRI) line	An ISDN Basic Rate Interface (BRI) line provides 2 bearer B channels and 1 data D line (known as "2B+D" over one or two pairs) to a subscriber.
integrated services digital network (ISDN)	ISDN is a network that provides a standard for communications (voice, data and signaling), end-to-end digital transmission circuits, out-of-band signaling, and a features significant amount of bandwidth.
inter machine trunk (IMT)	An inter-machine trunk (IMT) is a circuit between two commonly-connected switches.
inter-exchange carrier (IXC)	IXCs are US domestic long distance telecommunications services providers. AT&T, MCI, Sprint, are examples.
internet protocol (IP)	IP is part of the TCP/IP protocols. It is used to recognize incoming messages, route outgoing messages, and keep track of Internet node addresses (using a number to specify a TCP/IP host on the Internet). IP corresponds to the network layer of OSI.
internet service provider (ISP)	An ISP is a company that provides Internet access to subscribers.
ISDN primary rate interface (PRI)	An ISDN Primary Rate Interface (PRI) line provides the ISDN equivalent of a T1 circuit. The PRI delivered to a customer's premises can provide 23B+D (in North America) or 30B+D (in Europe) channels running at 1.544 megabits per second and 2.048 megabits per second, respectively.
local exchange carrier (LEC)	LECs are local telecommunications services providers. Bell Atlantic and US West are examples.
local access and transport area (LATA)	A LATA is a region in which a LEC offers services. There are over 160 LATAs of these local geographical areas within the United States.
local area network (LAN)	A LAN is a communications network providing connections between computers and peripheral devices (e.g., printers and modems) over a relatively short distance (e.g., within a building) under standardized control.
modified final judgment (MFJ)	Modified final judgment (MFJ) was the decision requiring divestiture of the Regional Bell Operating Companies (RBOCs) from their parent company, AT&T.
network node	A network node is a generic term for the resources in a telecommunications network, including switches, DACS, regenerators, etc. Network nodes essentially include all non-circuit (transport) devices. Other network nodes can include, for example, equipment of a CLEC, or other enhanced service provider (ESP), a point-of-presence (POP), an international gateway or global point-of-presence (GPOP).
new entrant (NE)	A new generation global telecommunications.
next generation telephone (NGT)	A new telecommunications services provider, especially IP telephony providers. Examples are Level 3 and Qwest.
packetized voice or voice over a backbone	One example of packetized voice is voice over internet protocol (VOIP). Voice over packet refers to the carrying of telephony or voice traffic over a data network, e.g. voice over frame, voice over ATM, voice over Internet Protocol (IP), over virtual private networks (VPNs), voice over a backbone, etc.
Pipe or dedicated communications facility	A pipe or dedicated communications facility connects an ISP to the internet.
point of presence (POP)	A POP refers to the location within a LATA where the IXC and LEC facilities interface.
point-to-point tunneling protocol (PPTP)	A virtual private networking protocol, point-to-point tunneling protocol (PPTP), can be used to create a "tunnel" between a remote user and a data network. A tunnel permits a network administrator to extend a virtual private network (VPN) from a server (e.g., a Windows NT server) to a data network (e.g., the Internet).
point-to-point (PPP) protocol	PPP is a protocol permitting a computer to establish a connection with the Internet using a modem. PPP supports high-quality graphical front ends, like Netscape.
postal telephone telegraph (PTT)	State regulated telephone companies, many of which are being deregulated. NTT is an example.
private branch exchange (PBX)	A PBX is a private switch located on the premises of a user. The user is typically a private company which desires to provide switching locally.
private line with a dial tone	A private line is a direct channel specifically dedicated to a customer's use between two specificed points. A private line with a dial tone can connect a PBX or an ISP's access concentrator to an end office (e.g. a channelized T1 or PRI). A private line can also be known as a leased line.
public switched telephone network (PSTN)	The PSTN is the worldwide switched voice network.
regional Bell operating companies (RBOCs)	RBOCs are the Bell operating companies providing LEC services after being divested from AT&T.
signaling system 7 (SS7)	SS7 is a type of common channel interoffice signaling (CCIS) used widely throughout the world. The SS7 network provides the signaling functions of indicating the arrival of calls, transmitting routing and destination signals, and monitoring line and circuit status.

TABLE 1-continued

Term	Definition
switching hierarchy or office classification	An office class is a functional ranking of a telephone central office switch depending on transmission requirements and hierarchical relationship to other switching centers. Prior to AT&T's divestiture of the RBOCs, an office classification was the number assigned to offices according to their hierarchical function in the U.S. public switched network (PSTN). The following class numbers are used: class 1 = Regional Center (RC), class 2 = Sectional Center (SC), class 3 = Primary Center (PC), class 4 = Toll Center (TC) if operators are present or else Toll Point (TP), class 5 = End Office (EO) a local central office. Any one center handles traffic from one to two or more centers lower in the hierarchy. Since divestiture and with more intelligent software in switching offices, these designations have become less firm. The class 5 switch was the closest to the end subscriber. Technology has distributed technology closer to the end user, diffusing traditional definitions of network switching hierarchies and the class of switches.
telecommunications carrier	A LEC, a CLEC, an IXC, an Enhanced Service Provider (ESP), an intelligent peripheral (IP), an international/global point-of-presence (GPOP), i.e., any provider of telecommunications services.
transmission control protocol (TCP)	TCP is an end-to-end protocol that operates at the transport and sessions layers of OSI, providing delivery of data bytes between processes running in host computers via separation and sequencing of IP packets.
transmission control protocol/internet protocol (TCP/IP)	TCP/IP is a protocol that provides communications between interconnected networks. The TCP/IP protocol is widely used on the Internet, which is a network comprising several large networks connected by high-speed connections.
trunk	A trunk connects an access tandem (AT) to an end office (EO).
wide area network (WAN)	A WAN is a data network that extends a LAN over the circuits of a telecommunications carrier. The carrier is typically a common carrier. A bridging switch or a router is used to connect the LAN to the WAN.

III. Introduction

A. Quality of Service (QOS) in a Wireless Environment

The concept of quality of service (QoS) is one of the most difficult and least understood topics in data networking. Although a common term in data networking, there are many different usages and definitions for QoS, leading to confusion regarding an exact meaning in precise or quantitative terms. Even further confusion is found when attempts are made to measure or specify numeric quantities sufficient to allow comparison of equipment or network performance with respect to QoS.

The confusion about QoS in general data networking is transferred and magnified when applied to wireless data communications. Wireless transmission has a higher inherent bit error rate (BER) than does wireline transmission. The addition of, e.g., a point-to-multipoint (PtMP) topology for multiple users sharing a wireless medium makes it desirable that QoS be defined in a manner that specifically addresses the multiple complicating factors in wireless data communications.

To provide a non-ambiguous definition of QoS that applies to wireless data communications, the nature of the problem that QoS is meant to solve is helpful. Many of the problems of data communications over wireless are unique and distinct from those of wireline data communications, while some are in fact shared. For wireless broadband access systems, the problems of quality delivery are somewhat more complex than for the wireline analog. Like its wireline counterpart, the problems encountered in wireless delivery of data include, e.g., slow peripheral access, data errors, "drop-outs," unnecessary retransmissions, traffic congestion, out-of-sequence data packets, latency, and jitter. In addition to these problems, wireless delivery adds problems including, e.g., high inherent bit error rates (BERs), limited bandwidth, user contention, radio interference, and TCP traffic rate management. A QoS-aware wireless system is desired to address all these problems.

There are a number of ways in which users or subscribers to a data network experience difficulties. One network difficulty is due to a lack of network availability. Depending on the access technology being used, this can include a "modem no-answer" condition, "network busy" condition, or a sudden unexpected "drop" of a network connection. These conditions would not be described as being consistent with high QoS. Once network connectivity is achieved, slow traffic caused by congestion, local access bottlenecks, and network failures can be experienced as slow web page loading, slow file transfers, or poor voice/video quality in streaming multimedia applications. Poor quality in streaming multimedia applications can instead result from high "jitter," or large and rapid variations in latency, leading to interruptions, distortion, or termination of session. Many different conditions can lead to actual data errors, which in some contexts can be catastrophic, such as in the file transfer of a spreadsheet. It is desirable that these problems of a data communications network be minimized or eliminated.

1. Quality

In data networking, quality usually implies the process of delivering data in a reliable and timely manner. What is reliable and timely is dependent on the nature of the traffic being addressed. These terms may include references to limitations in data loss, expectations of data accuracy, limitations of data latency variations (also known as jitter), and limitations of data retransmissions and limitations of data packet order inversions. Therefore, QoS is a complex concept, which can require a correspondingly complex mechanism to implement it.

QoS can be a relative term, finding different meanings for different users. A casual user doing occasional web browsing, but no file transfer protocol (FTP) file downloads or real time multimedia sessions may have different a different definition of QoS than a power user doing many FTP file downloads of large database or financial files, frequent H.323 video conferencing and IP telephony calls. Also, a user can pay a premium rate (i.e. a so-called service level agreement (SLA)) for high network availability, low latency, and low jitter, while another user can pay a low rate for occasional web surfing only, and on weekends only. Therefore, perhaps it is best to understand QoS as a continuum, defined by what network performance characteristic is most important to a particular user and the user's SLA. Maximizing the end-user experience is an essential component of providing wireless QoS.

2. Service

In data networking, a service can be defined as a type of connection from one end of a network to another. Formerly, this could have been further defined to be protocol specific, such as, e.g., IBM's systems network architecture (SNA), Novell's IPX, Digital's DECnet. However, it appears that TCP/IP (i.e. including user datagram protocol(UDP)) has evolved to become the overwhelming protocol of choice, and will continue to be in the foreseeable future. Therefore, service can be defined to be a particular type of TCP/IP

connection or transmission. Such service types might include, e.g., FTP file transfers, e-mail traffic, hypertext transfer protocol (HTTP) traffic, H.323 videoconferencing sessions. It is desirable that a QoS mechanism deal with these differing types of service, in addition to dealing with the different types of quality as discussed previously.

3. QOS as a Mechanism

QoS can be thought of as a mechanism to selectively allocate scarce networking, transmission and communications resources to differentiated classes of network traffic with appropriate levels of priority. Ideally, the nature of the data traffic, the demands of the users, the conditions of the network, and the characteristics of the traffic sources and destinations all modify how the QoS mechanism is operating at any given instant. Ultimately, however, it is desirable that the QoS mechanism operate in a manner that provides the user with optimal service, in whatever manner the user defines it.

a. Circuit-Switched QoS

In legacy networks created primarily for voice traffic by telephone companies, data transmission was accomplished with reference to a circuit-centric definition of QoS. In this definition, QoS implied the ability to carry asynchronous (i.e. transmission of data through start and stop sequences without the use of a common clock) as well as isochronous (i.e.consistent timed access of network bandwidth for time-sensitive voice and video) traffic. Circuit-switched QoS was accomplished by dedicating an end-to-end circuit for each connection or service, whether it was voice (see FIG. 1A) or data. The circuit-centric QoS mechanism was simply the provision of this circuit for exclusive use by the user. Of course, this approach dedicates the circuit, all transmission channels associated with the circuit, and the transport media itself to a single user for the entire duration of the session, regardless of whether data is actually being transmitted every instant of the session. It was generally believed that only in this manner could true QoS be achieved. Therefore, traditional designs for wireless broadband access systems (see FIG. 2A) also used this approach, dedicating a wireless radio channel to each particular data connection, regardless of the application or whether indeed any data was being transmitted at any given moment. This circuit-centric approach to QoS is fairly expensive, in terms of the cost of the equipment, and the utilization factors for the transmission media itself.

b. Asynchronous Transfer Mode (ATM) QoS

With ATM networking, telephone companies could continue to provide a circuit-centric QoS mechanism with the establishment of permanent virtual connections (PVCs) (i.e. a virtual path or channel connection (VPC or VCC) provisioned for indefinite use) and switched virtual connections (SVCs) (i.e. a logical connection between endpoints established by an ATM network on demand based upon signaling messages received from the end user or another network) in an analogous manner to the legacy voice circuit mechanism. However, several new concepts were needed, including admission policy, traffic shaping, and mechanisms such as, e.g., leaky-buckets, in order to handle traffic that was now categorized as variable bit rate (VBR), constant bit rate (CBR), and unspecified bit rate (UBR).

Virtual circuits were to be established for data transmission sessions, again regardless of the data application or whether data was being transmitted at any given moment. Although ATM provides QoS for broadband network traffic, the underlying assumptions of ATM design include the low BER characteristic of wireline networks, not the high BER of the wireless medium. Without a recognition of the characteristics of the traffic that is being carried by the ATM mechanism and the high inherent BER of wireless, true QoS can not be provided. ATM QoS mechanisms do not address the unique challenges associated with wireless communication.

c. Packet-Switched QoS

Packet-switching is revolutionizing data communications, so conventional circuit-switch and ATM networking concepts and their legacy QoS mechanisms are in need of update. With packet-switched data communications, one cannot dedicate a circuit to a particular data communications session. Indeed, a strength of packet-switching lies in route flexibility and parallelism of its corresponding physical network. Therefore, the QoS mechanism cannot work in the same manner as the legacy circuit-centric QoS mechanism did.

Simply providing "adequate" bandwidth is not a sufficient QoS mechanism for packet-switched networks, and certainly not for wireless broadband access systems. Although some IP-flows are "bandwidth-sensitive," other flows are latency- and/or jitter-sensitive. Real time or multimedia flows and applications cannot be guaranteed timely behavior by simply providing excessive bandwidth, even if it were not cost-prohibitive to do so. It is desirable that QoS mechanisms for an IP-centric wireless broadband access system recognize the detailed flow-by-flow requirements of the traffic, and allocate system and media resources necessary to deliver these flows in an optimal manner.

d. Summary—QoS Mechanisms

Ultimately, the end-user experience is the final arbiter of QoS. It is desirable that an IP-centric wireless broadband access system assign and regulate system and media resources in a manner that can maximize the end-user experience. For some applications such as an initial screen of a Web page download, data transmission speed is the best measure of QoS. For other applications, such as the download or upload of a spreadsheet, the best measure of QoS can be the minimization of transmission error. For some applications, the best measure of QoS can be the optimization of both speed and error. For some applications, the timely delivery of packets can be the best measure of QoS. It is important to note that fast data transmission may not be the same as timely delivery of packets. For instance, data packets that are already "too old" can be transmitted rapidly, but by being too old can be of no use to the user. The nature of the data application itself and the desired end-user experience then can provide the most reliable criteria for the QoS mechanism. It is desired that an IP-centric wireless broadband access system provide a QoS mechanism that can dynamically optimize system behavior to each particular IP flow, and can also adapt to changes with changing network load, congestion and error rates.

4. Service Guarantees and Service Level Agreements (SLAs)

Service guarantees can be made and service level agreements (SLAs) can be entered into between a telecommunications service provider and a subscriber whereby a specified level of network availability can be described, and access charges can be based upon the specified level. Unfortunately, it is difficult to quantify the degree of network availability at any given time, and therefore this becomes a rather crude measure of service performance. It is desired that data delivery rate, error rate, retransmissions, latency, and jitter be used as measures of network availability, but measuring these quantities on a real-time basis can be beyond the capability of conventional network service providers (NSPs).

Another level of service discrimination desired by network service providers is a service level agreement (SLA) that provides for differing traffic rates, network availability, bandwidth, error rate, latency and jitter guarantees. It is desired that an IP-centric wireless broadband access system be provided that can provide for SLAs, enabling service providers to have more opportunities for service differentiation and profitability.

5. Class of Service and Quality of Service

In order to implement a practical QoS mechanism, it is desired that a system be able to differentiate between types of traffic or service types so that differing levels of system resources can be allocated to these types. It is customary to speak of "classes of service" as a means of grouping traffic types that can receive similar treatment or allocation of system and media resources.

Currently, there are several methods that can be used in wireline network devices to implement differentiated service classes. Example methods include traffic shaping, admission control, IP precedence, and differential congestion management. It is desired that an IP-centric wireless broadband access system use all of these methods to differentiate traffic into classes of service, to map these classes of service against a QoS matrix, and thereby to simplify the operation and administration of the QoS mechanism.

B. QoS and IP-Centric Wireless Environment

In a point-to-multipoint (PtMP) wireless system like the present invention, it is desirable that the QoS mechanism cope not only with wireline networking considerations, but also with considerations particular to the wireless environment. As stated earlier, it is desired that the inherent BER of wireless be handled. The high BER can require that error detection, correction, and re-transmission be done in an efficient manner. It is desired that a BER handling mechanism also work efficiently with the re-transmission algorithms of TCP/IP so as to not cause further unnecessary degradation of bandwidth utilization. An additional challenge of wireless is contention among users for limited wireless bandwidth. It is desirable that the system handle service requests from multiple users in a radio medium subject to interference and noise, which can make efficient allocation of radio bandwidth difficult.

As discussed above, the change from circuit-switched and ATM data networks to packet-switched data networks has impacted the definition of QoS mechanisms. The present invention provides a novel QoS mechanism in a point-to-multi-point IP-centric wireless system for packet-switched network traffic. In order for the system to provide optimal QoS performance, it desirable that it include a novel approach to QoS mechanisms. The use of QoS as the underlying guide to system architecture and design constitutes an important, substantial and advantageous difference of the IP-centric wireless broadband access system of the present invention over existing wireless broadband access systems designed with traditional circuit-centric or ATM cell circuit-centric approaches such as those used by Teligent and Winstar.

C. IP-Centric Wireless Broadband Access QoS and Queuing Disciplines

1. Managing Queues

Queuing is a commonly accepted tool required for manipulating data communications flows. In order for packet headers to be examined or modified, for routing decisions to made, or for data flows to be output on appropriate ports, it is desirable that data packets be queued. However, queuing introduces, by definition, a delay in the traffic streams that can be detrimental, and can even totally defeat the intent of queuing. Excessive queuing can have detrimental effects on traffic by delaying time sensitive packets beyond their useful time frames, or by increasing the RTT (Round Trip Time), producing unacceptable jitter or even causing the time-out of data transport mechanisms. Therefore, it is desired that queuing be used intelligently and sparingly, without introducing undue delay in delay-sensitive traffic such as real-time sessions.

In a wireless environment where time division multiple access (TDMA), forward error detection (FEC), and other such techniques can be necessary, it is desirable that queuing be used merely to enable packet and radio frame processing. However, in the case of real-time flows, the overall added delay in real-time traffic can preferably be held to below approximately 20 milliseconds.

The use of queue management as the primary QoS mechanism in providing QoS-based differentiated services is a simple and straight forward method for wireless broadband systems. However, wireless systems are usually more bandwidth constrained and therefore more sensitive to delay than their wireline counterparts. For this reason, it is desirable that QoS-based differentiated services be provided with mechanisms that go beyond what simple queuing can do. However, some queuing can still be required, and the different queuing methods are now discussed.

2. First in, First out (FIFO) Queuing

First in, first out (FIFO) queuing can be used in wireless systems, like wireline systems, in buffering data packets when the downstream data channel becomes temporarily congested. If temporary congestion is caused by bursty traffic, a FIFO queue of reasonable depth can be used to smooth the flow of data into the congested communications segment. However, if the congestion becomes severe in extent, or relatively long in duration, FIFO can lead to the discarding of packets as the FIFO queues are filled to capacity, and the network is not capable of accepting additional packets causing discarding of packets, i.e. so-called "packet-tossing." Although this can have a detrimental effect on QoS in and of itself, the discarding of packets may cause future problems with traffic flow as the TCP protocol causes the retransmission of lost packets in the proper sequence, further exacerbating the problem. The problem of packet discards can be minimized by increasing the size of the FIFO buffers so that more time can pass before discards occur. Unfortunately, eventually the FIFO can become large enough that packets can become too old and the round-trip time (RTT) can increase to the point that the packets are useless, and the data connection is virtually lost.

In a wireless broadband environment, the requirement for FIFO queuing is partially dependent upon the type of RF access method being used. For time division multiple access/time division duplex (TDMA/TDD), it can be desirable that data be queued even for collecting enough data for the construction of data frames for transmission. Frequency division multiple access (FDMA) and code-division multiple access (CDMA) are not as "sequential" in nature as TDMA, and therefore have less of a requirement for FIFO queuing. However, generally for all wireless access techniques, noise and interference are factors that can lead to retransmissions, and therefore further delays and consequent adverse effect on QoS.

Using FIFO queuing, shared wireless broadband systems can uniformly delay all traffic. This can seem to be the "fairest" method, but it is not necessarily the best method if the goal is to provide high QoS to users. By using different types of queue management, a much better base of overall QoS can be achieved.

3. Priority Queuing

The shared wireless broadband environment can include a constricted bandwidth segment as data is transmitted over the RF medium. Therefore, regardless of access technique, these systems can require some amount of queuing. However, using FIFO queuing can result in a constant delay to all traffic, regardless of the priority or type of traffic. Most data communications environments can consist of a mixture of traffic, with combinations of real time interactive data, file and data downloads, web page access, etc. Some of these types of traffic are more sensitive to delay, and jitter, than others. Priority queuing simply reorders data packets in the queue based on their relative priorities and types, so that data from more latency- and jitter-sensitive traffic can be moved to the front of the queue.

Unfortunately, if there is downlink data channel congestion, or congestion caused by an overabundance of high priority traffic, the condition of "buffer starvation" can occur. Because of the relative volume of high priority packets consuming a majority of buffer space, little room is left for lower priority packets. These lower priority packets can experience significant delays while system resources are devoted to the high priority packets. In addition to low priority packets being held in buffers for long periods of time, or never reaching the buffers, resulting in significantly delayed data flows for these packets, the actual applications corresponding to these low priority packets can also be disrupted, and stop working. Because of the nature of this queuing approach, overall latency and jitter and RTT for lower priority packets can be unpredictable, having an adverse effect on QoS.

If queue sizes are small, reordering data within the queues can have little beneficial effect on the QoS. In fact, processing required to examine packet headers in order to obtain the information necessary to reorder the queues may itself add significant delay to the data stream. Therefore, particularly for wireless broadband data environments, priority queuing can be not much better than FIFO queuing as a QoS mechanism.

4. Classed Based Queuing

By allocating queue space and system resources to packets based on the class of the packets, buffer starvation can be avoided. Each class can be defined to include of data flows with certain similar priorities and types. All classes can be given a certain minimum level of service so that one high priority data flow cannot monopolize all system resources. With the classification approach, because no data flow is ever completely shut off, the source application can receive information about the traffic rate, and can be able to provide TCP-mediated transmission rate adjustment supporting smooth traffic flow.

Although this approach can work better than FIFO queuing in wireless broadband systems, latency and jitter sensitive flows can still be adversely affected by high priority flows of large volume.

5. Weighted Fair Queuing

A weighted fair queuing method can attempt to provide low-volume flows with guaranteed queuing resources, and can then allow remaining flows, regardless of volume or priority, to have equal amounts of resource. Although this can prevent buffer starvation, and can lead to somewhat better latency and jitter performance, it can be difficult to attain stable performance in the face of rapidly changing RF downlink channel bandwidth availability.

Providing a high quality of service can require a QoS mechanism that is more sophisticated than simple queue management.

D. IP-Centric Wireless Broadband Access QoS and TCP/IP

1. TCP/IP

The TCP/IP protocol stack has become the standard method of transmitting data over the Internet, and increasingly it is becoming a standard in virtual private networks (VPNs). The TCP/IP protocol stack includes not only internet protocol (IP), but also transmission control protocol (TCP), user datagram protocol (UDP), and internet control message protocol (ICMP). By assuming that the TCP/IP protocol stack is the standard network protocol for data communications, the creation of a set of optimal QoS mechanisms for the wireless broadband data environment is more manageable. QoS mechanisms can be created that can span the entire extent of the network, including both the wireline and the wireless portions of the network. These mechanisms can integrate in a smooth and transparent manner with TCP rate control mechanisms and provide end-to-end QoS mechanisms that are adaptive to both the wireline and wireless portions of the network. Of course, segments of the wireline network that are congested or are experiencing other transport problems cannot be solved by a wireless QoS mechanism. However, a wireless QoS mechanism can optimize data flows in a manner that can enhance the end user experience when there is no severe wireline network congestion or bottleneck present.

2. Differentiation by Class

Data traffic can be handled based on classes of service, as discussed above. To differentiate traffic by class, data traffic (or a sequence of data packets associated with a particular application, function, or purpose) can be classified into one of several classes of service. Differentiation can be done on the basis of some identifiable information contained in packet headers. One method can include analyzing several items in, e.g., an IP packet header, which can serve to uniquely identify and associate the packet and other packets from that packet flow with a particular application, function or purpose. As a minimum, a source IP address, a source TCP or UDP port, a destination IP address, and a destination IP or UDP port can serve to associate packets into a common flow, i.e. can be used to classify the packets into a class of service.

By creating a finite and manageable number of discrete classes of service, multiple IP flows can be consolidated and handled with a given set of QoS parameters by the QoS mechanisms. These classes can be defined to provide common and useful characteristics for optimal management in the combined wireline and wireless network segments.

3. Per-Flow Differentiation

A finite and discrete set of classes of service, can enable QoS mechanisms to be less compute-intensive, to use less memory, fewer state machines, and therefore have better scaleability than having individual QoS mechanisms (or sets of parameters) for each individual IP flow. However, in a network access device such as, e.g., a point to multi-point (PtMP) wireless broadband access system, the total number of simultaneous IP flows typically will not exceed the range of 1000, and therefore the amount of processing overhead that could be required could permit a per-flow QoS differentiation without resorting to classes of service. However, class of service consolidation of IP flows provides advantages related to marketing, billing and administration.

Prior to the present invention, per-flow differentiation has not been used in a wireless environment (including radio frequencies transmitted over coaxial cables and satellite communications).

4. Using IP Precedence for Class of Service

IP precedence bits in a type of service (IP TOS) field, as described in Internet Engineering Task Force (IETF)1992b, can theoretically be used as a means to sort IP flows into classes of service. IETF RFC1349 proposed a set of 4-bit definitions with 5 different meanings: minimize delay; maximize throughput; maximize reliability; minimize monetary cost; and normal service.

These definitions could add significantly to networks, routers and access devices in differentiating different types of flow so that resources could be appropriately allocated, resulting in improved QoS. However, the proposal has not been widely used. Several proposals in the IETF could make use of this field, along with resource reservation protocol (RSVP), to improve network handling of packets.

Although the type of service (TOS) field has been an integral component of the TCP/IP specification for many years, the field is not commonly used. Absent appropriate bits in the field being set by a source processor, the access devices, the network and network routers cannot implement QoS mechanisms.

5. TCP-Mediated Transmission Rate Mechanisms

The manner in which TCP governs transmission rate can be incorporated and managed by an IP-centric wireless QoS mechanism. If a TCP mechanism is not managed, any wireless QoS mechanism can be overwhelmed or countered by wireless bandwidth factors. Before addressing the specific wireless factors that can impact TCP transmission speed, a review of TCP transmission rate mechanism is needed.

TCP can control transmission rate by "sensing" when packet loss occurs. Because TCP/IP was created primarily for wireline environment with its extremely low inherent BER, such as those found over fiber optic lines, any packet loss is assumed by TCP to be due to network congestion, not loss through bit error. Therefore, TCP assumes that the transmission rate exceeded the capacity of the network, and responds by slowing the rate of transmission. However, packet loss in the wireless link segment is due primarily to inherently high BER, not congestion. The difference turns out to be not insubstantial.

TCP can initially cause the transmission rate to ramp-up at the beginning of a packet flow, and is called slow-start mode. The rate can be continuously increased until there is a loss or time-out of the packet-receipt acknowledgment message. TCP can then "back-off", can decrease the transmission window size, and then can retransmit lost packets in the proper order at a significantly slower rate. TCP can then slowly increase the transmission rate in a linear fashion, which can be called congestion-avoidance mode.

If multiple users share a wireless radio link as with the present invention, the inherently high BER of the medium could potentially cause frequent packet loss leading to unproductive TCP retransmission in congestion avoidance mode. Because wireless bandwidth can be a precious commodity, a IP-centric wireless QoS mechanism preferably provides for packet retransmission without invoking TCP retransmission and consequent and unnecessary "whipsawing" of the transmission rate. This, along with several other factors, makes desirable creation of an IP-centric wireless media access control (MAC) layer. One function of an IP-centric MAC layer can be to mediate local retransmission of lost packets without signaling TCP and unnecessarily altering the TCP transmission speed. A primary task of the IP-centric wireless MAC layer is to provide for shared access to the wireless medium in an orderly and efficient manner. The MAC layer according to the present invention, Proactive Reservation-based Intelligent Multimedia-aware Media Access (PRIMMA) layer, available from Malibu Networks Inc., of Calabasas, Calif., can also schedule all packet transmissions across the wireless medium on the basis of, e.g., IP flow type, service level agreements (SLAs), and QoS considerations.

6. TCP Congestion Avoidance in an IP-Centric Wireless System

a. Network Congestion Collapse, Global Synchronization and IP-Centric Wireless TCP Congestion Avoidance

The inherently high bit error rate (BER) of wireless transmission can make an occurrence of problems known as congestion collapse or global synchronization collapse more likely than in a wireline environment. When multiple TCP senders simultaneously detect congestion because of packet loss, the TCP senders can all go into TCP slow start mode by shrinking their transmission window sizes and by pausing momentarily. The multiple senders can then all attempt to retransmit the lost packets simultaneously. Because they can all start transmitting again in rough synchrony, a possibility of creating congestion can arise, and the cycle can start all over again.

In the wireless environment, an occurrence of burst noise can cause packet loss from many IP streams simultaneously. The TCP transmission rate mechanisms of the TCP senders can assume that packet loss was due to congestion, and they can all back-off in synchrony. When the TCP senders restart, the senders can restart in rough synchrony, and indeed can now create real congestion in the wireless link segment. This cyclical behavior can continue for some time, and can possibly cause unpredictable system performance. This can be due in part to overflowing system queues which can cause more packets to be dropped and can cause more unproductive retransmissions. This can degenerate into a "race" state that could take many minutes before re-establishing stability; this can have an obvious negative impact on QoS.

In the wireline world, random early detection (RED) can be used to circumvent global synchronization. By randomly selecting packets from randomly selected packet flows before congestion collapse occurs, global synchronization can be avoided. Queues can be monitored, and when queue depth exceeds a preset limit, RED can be activated, activating a synchronously the TCP senders' transmission rate controllers. This can avoid the initial congestion which would otherwise result in collapse and then global synchronization.

Instead of purely random packet discards, the packets to be discarded can be done with consideration to packet priority or type. While still random, the probability of discard for a given flow can be a function of the by packet priority or type. In a wireless system, weighted random early detection (WRED) can be used without the concern of retransmission and TCP rate reset by preferentially selecting UDP packets of real time IP flows such as streaming audio, and H.323 flows with a more critical packet Time-to-Live parameter. These IP flows are more sensitive to latency and jitter, and less sensitive to packet loss.

In the wireless environment, with an appropriately designed MAC layer, packet loss due to BER that might otherwise trigger congestion collapse and global synchronization can best be managed with local retransmission of lost packets according to the present invention and without RED and the unnecessary retransmission of packets by the TCP sender and the resulting reset of TCP transmission rate. The IP-centric wireless system separately manages the TCP transmission window of the TCP sender remotely by transmitting a packet receipt-acknowledgment before the TCP

sender detects a lost packet and initiates retransmission along with an unnecessary reset of the transmission rate. This IP-centric wireless system TCP transmission window manager communicates with the MAC layer in order to be aware of the status of all packets transmitted over the wireless medium.

b. The Effect of Fractal Self-Similar Network Traffic Characteristics vs. Poisson Distributions on Network Congestion

Conventionally, it has been believed that network traffic can be modeled with a Poisson distribution. Using this distribution leads to the conclusion, through system simulations, that the sum of thousands of individual traffic flows with Poisson distributions results in a uniform overall network traffic distribution. In other words, the overall network can "average-out" the burstiness of individual traffic flows. Using this model, network congestion behavior, burst behavior, and dynamic traffic characteristics have been used to create conventional congestion avoidance strategies, design queue buffer sizes in network devices, and traffic and capacity limitation predictions.

More recent studies have demonstrated that TCP/IP-based traffic causes networks to behave in a fractal, or self-similar fashion. With this model, when the burstiness of individual traffic flows is summed for the entire network, the entire network becomes bursty. The bursty nature of network traffic flow is seen over all time scales and flow scales of the network. This has huge implications both in design of an IP-centric wireless broadband system according to the present invention, and in the design of congestion avoidance strategies in the network as a whole. With this new perspective on network behavior, it has become clear that network routers, switches and transmission facilities in many cases have been "under-engineered." This under-engineering has led to a further exacerbation of the congestion behavior of the network.

The implications for IP-centric wireless system architecture and design range from queue buffer capacity to local congestion avoidance strategies. Because wireless systems have the added burden of a high inherent BER, the effect of network-wide congestion behavior on local (wireless media channel) congestion avoidance strategies must be properly gauged and countered. For this reason, it is desirable that congestion avoidance algorithms of the IP-centric wireless system be crafted to optimize traffic flow with new mathematical and engineering considerations that until very recently were not apparent or available to system designers.

With these considerations in mind, IP-centric wireless system design cannot be done with the conventional wireline system design approaches without resulting in very low system performance characteristics. With traditional design approaches of a circuit-centric wireless system, bandwidth utilization, real time multimedia quality, and overall system QoS provide for a dramatically lower end-user experience.

7. Application-Specific Flow Control in an IP-Centric Wireless System

With a range of data flows, each having different bandwidth, latency and jitter requirements, for the achievement of high QoS as perceived by the end user, it is desirable that the IP-centric wireless system be able to manage QoS mechanism parameters over a wide range, and in real time. The QoS mechanism must be able to alter system behavior to the extent that one or more data flows corresponding to specific applications be switched on and off from appropriate end users in a transparent manner. This approach is in contrast to other QoS mechanisms that seek to achieve high QoS by establishing circuit-centric connections from end to end without regard for an underlying application's actual QoS requirements. By using the present invention, providing a QoS mechanism that is application-specific rather than circuit-specific, scarce wireless bandwidth can be conserved and dynamically allocated where needed by the QoS mechanisms associated with each application type.

B. QoS and IP-Centric Wireless Media Access Control

1. Proactive Reservation-based Intelligent Multimedia-aware Media Access (PRIMMA) MAC Layer

The present invention's proactive reservation-based intelligent multimedia-aware media access (PRIMMA) media access control (MAC) layer provides an application switching function of the IP-centric wireless QoS mechanism. Once the nature and QoS requirements of each IP stream are determined by other portions of the system, this information is communicated to the PRIMMA MAC layer so that the IP flows of each application can be switched to appropriate destinations in a proper priority order.

2. PRIMMA IP Protocol Stack Vertical Signaling

For IP streams that originate from a local user's CPE, application-level information about the nature of the application can be used by the system to assign appropriate QoS mechanism parameters to the IP stream. For IP streams that originate from a non-local host, information about the IP streams for use in configuring the appropriate QoS mechanism parameters can be extracted from packet headers. The information about the IP streams is communicated "vertically" in the protocol stack model from the application layer (i.e. OSI level 7) to the PRIMMA MAC layer (i.e. OSI level 2) for bandwidth reservation and application switching purposes. Although this violates the conventional practice of providing isolation and independence to each layer of the protocol stack, thereby somewhat limiting the degree of interchangeability for individual layers of the stack, the advantages far outweigh the negatives in an IP-centric wireless broadband access system.

3. PRIMMA IP Flow Control and Application Switching

Based on a specific set of QoS requirements of each IP application flow in the IP-centric wireless system, applications are switched in a "proactive" manner by appropriate reservations of bandwidth over the wireless medium. The wireless transmission frames in each direction are constructed in a manner dictated by the individual QoS requirements of each IP flow. By using QoS requirements to build the wireless transmission frames, optimal QoS performance can result over the entire range of applications being handled by the system. For example, latency and jitter sensitive IP telephony, other H.323 compliant IP streams, and real-time audio and video streams can be given a higher priority for optimal placement in the wireless transmission frames. On the other hand, hypertext transport protocol (HTTP) traffic, such as, e.g., initial web page transmissions, can be given higher bandwidth reservation priorities for that particular application task. Other traffic without latency, jitter, or bandwidth requirements such as, e.g., file transfer protocol (FTP) file downloads, email transmissions, can be assigned a lower priority for system resources and placement in the wireless transmission frame.

4. PRIMMA TCP Transmission Rate Agent

Wireless end users are separated from a high speed, low BER wireline backbone by a lower speed, high BER wireless segment which can be subject to burst error events. TCP/IP traffic that traverses the wireless segment can experience frequent packet loss that, without intervention, can create congestion collapse and global synchronization as previously discussed. Therefore, it is desirable that the present invention's IP-centric wireless system make use of a TCP transmission rate agent that can monitor packet loss

over the wireless segment, and can manage the remote TCP transmission rate function by recreating and transmitting any lost packet acknowledgments. The PRIMMA MAC layer can itself retransmit any lost packets over the wireless medium.

The IP-centric wireless TCP transmission rate agent or "adjunct" can also flow-control the IP streams when necessary, and in accordance with the QoS requirements of the IP flows. All IP-centric wireless TCP transmission rate agent functionality can be transparent to both local and remote hosts and applications.

F. Telecommunications Networks

1. Voice Network

a. Simple Voice Network

FIG. **1A** is a block diagram providing an overview of a standard telecommunications network **100** providing local exchange carrier (LEC) services within one or more local access and transport areas (LATAs). Telecommunications network **100** can provide a switched voice connection from a calling party **102** to a called party **110**. FIG. **1A** is shown to also include a private branch exchange **112** which can provide multiple users access to LEC services by, e.g., a private line. Calling party **102** and called party **110** can be ordinary telephone equipment, key telephone systems, a private branch exchange (PBX) **112**, or applications running on a host computer. Network **100** can be used for modem access as a data connection from calling party **102** to, for example, an Internet service provider (ISP) (not shown). Network **100** can also be used for access to, e.g., a private data network. For example, calling party **102** can be an employee working on a notebook computer at a remote location who is accessing his employer's private data network through, for example, a dial-up modem connection.

FIG. **1A** includes end offices (EOs) **104** and **108**. EO **104** is called an ingress EO because it provides a connection from calling party **102** to public switched telephone network (PSTN) facilities. EO **108** is called an egress EO because it provides a connection from the PSTN facilities to a called party **110**. In addition to ingress EO **104** and egress EO **108**, the PSTN facilities associated with telecommunications network **100** include an access tandem (AT) (not shown) at points of presence (POPs) **132** and **134** that can provide access to, e.g., one or more inter-exchange carriers (IXCs) **106** for long distance traffic, see FIG. **2A**. Alternatively, it would be apparent to a person having ordinary skill in the art that IXC **106** could also be, for example, a CLEC, or other enhanced service provider (ESP), an international gateway or global point-of-presence (GPOP), or an intelligent peripheral (IP).

FIG. **1A** also includes a private branch exchange (PBX) **112** coupled to EO **104**. PBX **112** couples calling parties **124** and **126**, fax **116**, client computer **118** and associated modem **130**, and local area network **128** having client computer **120** and server computer **122** coupled via an associated modem **130**. PBX **112** is a specific example of a general class of telecommunications devices located at a subscriber site, commonly referred to as customer premises equipment (CPE).

Network **100** also includes a common channel interactive signaling (CCIS) network for call setup and call tear down. Specifically, FIG. **1** includes a Signaling System 7 (SS7) signaling network **114**. Signaling network **114** will be described further below with reference to FIG. **2B**.

b. Detailed Voice Network

FIG. **2A** is a block diagram illustrating an overview of a standard telecommunications network **200**, providing both LEC and IXC carrier services between subscribers located in different LATAs. Telecommunications network **200** is a more detailed version of telecommunications network **100**. Calling party **102***a* and called party **110***a* are coupled to EO switches **104***a* and **108***a*, respectively. In other words, calling party **102***a* is homed to ingress EO **104***a* in a first LATA, whereas called party **110***a* is homed to an egress EO **108***a* in a second LATA. Calls between subscribers in different LATAs are long distance calls that are typically routed to IXCs. Sample IXCs in the United States include AT&T, MCI and Sprint.

Telecommunications network **200** includes access tandems (AT) **206** and **208**. AT **206** provides connection to points of presence (POPs) **132***a*, **132***b*, **132***c* and **132***d*. IXCs **106***a*, **106***b* and **106***c* provide connection between POPs **132***a*, **132***b* and **132***c* (in the first LATA) and POPs **134***a*, **134***b* and **134***c* (in the second LATA). Competitive local exchange carrier (CLEC) **214** provides an alternative connection between POP **132***d* and POP **134***d*. POPs **134***a*, **134***b*, **134***c* and **134***d*, in turn, are connected to AT **208**, which provides connection to egress EO **108***a*. Called party **110***a* can receive calls from EO **108***a*, which is its homed EO.

Alternatively, it would be apparent to a person having ordinary skill in the art that an AT **206** can also be, for example, a CLEC, or other enhanced service provider (ESP), an international gateway or global point-of-presence (GPOP), or an intelligent peripheral.

Network **200** also includes calling party **102***c* homed to CLEC switch **104***c*. Following the 1996 Telecommunications Act in the U.S., CLECs gained permission to compete for access within the local RBOCs territory. RBOCs are now referred to as incumbent local exchange carriers (ILECs).

i. Fixed Wireless CLECs

Network **200** further includes a fixed wireless CLEC **209**. Example fixed wireless CLECs are Teligent Inc., of Vienna, Va., WinStar Communications Inc., Advanced Radio Telecom Corp. And the BizTel unit of Teleport Communications Group Inc. Fixed wireless CLEC **209** includes a wireless transceiver/receiver radio frequency (RF) tower **210** in communication over an RF link to a subscriber transciever RF tower **212**. Subscriber RF tower **212** is depicted coupled to a CPE box, PBX **112***b*. PBX **112***b* couples calling parties **124***b* and **126***b*, fax **116***b*, client computer **118***b* and associated modem **130***b*, and local area network **128***b* having client computer **120***b* and server computer **122***b* coupled via an associated modem **130***b*.

Network **200** also includes called party **110***a*, a fax **116***a*, client computer **118***a* and associated modem **130***a*, and cellular communications RF tower **202** and associated cellular subscriber called party **204**, all coupled to EO **108***a*, as shown.

EO **104***a*, **108***a* and AT **206**, **208** are part of a switching hierarchy. EO **104***a* is known as a class 5 office and AT **208** is a class 3/4 office switch. Prior to the divestiture of the regional Bell Operating Companies (RBOCs) from AT&T following the modified final judgment, an office classification was the number assigned to offices according to their hierarchical function in the U.S. public switched network (PSTN). An office class is a functional ranking of a telephone central office switch depending on transmission requirements and hierarchical relationship to other switching centers. A class 1 office was known as a Regional Center (RC), the highest level office, or the "office of last resort" to complete a call. A class 2 office was known as a Sectional Center (SC). A class 3 office was known as a Primary Center (PC). A class 4 office was known as either a Toll Center (TC) if operators were present, or otherwise as a Toll Point (TP).

A class 5 office was an End Office (EO), i.e., a local central office, the lowest level for local and long distance switching, and was the closest to the end subscriber. Any one center handles traffic from one or more centers lower in the hierarchy. Since divestiture and with more intelligent software in switching offices, these designations have become less firm. Technology has distributed functionality closer to the end user, diffusing traditional definitions of network hierarchies and the class of switches.

ii. Connectivity to Internet Service Providers (ISPs)

In addition to providing a voice connection from calling party **102***a* to called party **110***a*, the PSTN can provide calling party **102***a* a data connection to an ISP (i.e. similar to client **118***b*).

Network **200** can also include an Internet service provider (ISP) (not shown) which could include a server computer **122** coupled to a data network **142** as will be discussed further below with reference to FIG. **1B**. The Internet is a well-known, worldwide network comprising several large networks connected together by data links. These links can include, for example, Integrated Digital Services Network (ISDN), T1, T3, FDDI and SONET links. Alternatively, an internet can be a private network interconnecting a plurality of LANs and/or WANs, such as, for example, an intranet. An ISP can provide Internet access services for subscribers such as client **118***b*.

To establish a connection with an ISP, client **118***b* can use a host computer connected to a modem (modulator/demodulator) **130***b*. The modem can modulate data from the host computer into a form (traditionally an analog form) for transmission to the LEC facilities. Typically, the LEC facilities convert the incoming analog signal into a digital form. In one embodiment, the data is converted into the point-to-point protocol (PPP) format. (PPP is a well-known protocol that permits a computer to establish a connection with the Internet using a standard modem. It supports high-quality, graphical user-interfaces.) As those skilled in the art will recognize, other formats are available, including, e.g., a transmission control program, internet protocol (TCP/IP) packet format, a user datagram protocol, internet protocol (UDP/IP) packet format, an asynchronous transfer mode (ATM) cell packet format, a serial line interface protocol (SLIP) protocol format, a point-to-point (PPP) protocol format, a point-to-point tunneling protocol (PPTP) format, a NETBIOS extended user interface (NETBEUI) protocol format, an Appletalk protocol format, a DECnet, BANYAN/VINES, an internet packet exchange (IPX) protocol format, and an internet control message protocol (ICMP) protocol format.

iii. Communications Links

Note that FIGS. **1A**, **2A** and other figures described herein include lines which may refer to communications lines or which may refer to logical connections between network nodes, or systems, which are physically implemented by telecommunications carrier devices. These carrier devices include circuits and network nodes between the circuits including, for example, digital access and cross-connect system (DACS), regenerators, tandems, copper wires, and fiber optic cable. It would be apparent to persons having ordinary skill in the art that alternative communications lines can be used to connect one or more telecommunications systems devices. Also, a telecommunications carrier as defined here, can include, for example, a LEC, a CLEC, an IXC, an Enhanced Service Provider (ESP), a global or international services provider such as a global point-of-presence (GPOP), and an intelligent peripheral.

EO **104***a* and AT **206** are connected by a trunk. A trunk connects an AT to an EO. A trunk can be called an inter machine trunk (IMT). AT **208** and EO **108***a* are connected by a trunk which can be an IMT.

Referring to FIG. **1A**, EO **104** and PBX **112** can be connected by a private line with a dial tone. A private line can also connect an ISP (not shown) to EO **104**, for example. A private line with a dial tone can be connected to a modem bay or access converter equipment at the ISP. Examples of a private line are a channelized T1 or integrated services digital network (ISDN) primary rate interface (PRI). An ISP can also attach to the Internet by means of a pipe or dedicated communications facility. A pipe can be a dedicated communications facility. A private line can handle data modem traffic to and from an ISP.

Trunks can handle switched voice traffic and data traffic. For example, trunks can include digital signals DS1–DS4 transmitted over T1 –T4 carriers. Table 2 provides typical carriers, along with their respective digital signals, number of channels, and bandwidth capacities.

TABLE 2

Digital signal	Number of channels	Designation of carrier	Bandwidth in Megabits per second (Mbps)
DS0	1	None	0.064
DS1	24	T1	1.544
DS2	96	T2	6.312
DS3	672	T3	44.736
DS4	4032	T4	274.176

Alternatively, trunks can include optical carriers (OCs), such as OC-1, OC-3, etc. Table 3 provides typical optical carriers, along with their respective synchronous transport signals (STSs), ITU designations, and bandwidth capacities.

TABLE 3

Optical carrier (OC) signal	Electrical signal, or synchronous transport signal (STS)	International Telecommunications Union (ITU) terminology	Bandwidth in Megabits per second (Mbps)
OC-1	STS-1		51.84
OC-3	STS-3	STM-1	155.52
OC-9	STS-9	STM-3	466.56
OC-12	STS-12	STM-4	622.08
OC-18	STS-18	STM-6	933.12
OC-24	STS-24	STM-8	1244.16
OC-36	STS-36	STM-12	1866.24
OC-48	STS-48	STM-16	2488.32

As noted, a private line is a connection that can carry data modem traffic. A private line can be a direct channel specifically dedicated to a customer's use between two specified points. A private line can also be known as a leased line. In one embodiment, a private line is an ISDN/primary rate interface (ISDN PRI) connection. An ISDN PRI connection can include a single signal channel (called a data or D channel) on a T1, with the remaining 23 channels being used as bearer or B channels. (Bearer channels are digital channels that bear voice and data information.) If multiple ISDN PRI lines are used, the signaling for all of the lines can be carried over a single D channel, freeing up the remaining lines to carry only bearer channels.

iv. Telecommunications Traffic

Telecommunications traffic can be sent and received from any network node of a telecommunications carrier. A telecommunications carrier can include, for example, a LEC, a CLEC, an IXC, and an Enhanced Service Provider (ESP). In an embodiment, this traffic can be received from a network

node which is, for example, a class 5 switch, such as EO **104***a*, or from a class 3/4 switch, such as AT **206**. Alternatively, the network system can also be, for example, a CLEC, or other enhanced service provider (ESP), an international gateway or global point-of-presence (GPOP), or an intelligent peripheral.

Voice traffic refers, for example, to a switched voice connection between calling party **102***a* and called party **110***a*. It is important to note that this is on a point-to-point dedicated path, i.e., that bandwidth is allocated whether it is being used or not. A switched voice connection is established between calling party **102***a* and EO **104***a*, then to AT **206** then over an IXC's network such as that of IXC **106***a* to AT **208** and then to EO **108***a* and over a trunk to called party **110***a*. In another embodiment, AT **206** or IXC **106***a* can also be, for example, a CLEC, or other enhanced service provider (ESP), an international gateway or global point-of-presence (GPOP), or an intelligent peripheral.

It is possible that calling party **102***a* is a computer with a data connection to a server over the voice network. Data traffic refers, for example, to a data connection between a calling party **102***a* (using a modem) and a server **122***b* that could be part of an ISP. A data connection can be established, e.g., between calling party **102***a* and EO **104***a*, then to AT **206**, then to CLEC **214**, then over a fixed wireless CLEC **209** link to PBX **112***b* to a modem **130***b* associated with server **122***b*.

c. Signaling Network

FIG. 2B illustrates signaling network **114** in greater detail. Signaling network **114** is a separate network used to handle the set up, tear down, and supervision of calls between calling party **102** and called party **110**. Signaling network **114** in the given example is the Signaling System 7 (SS7) network. Signaling network **114** includes service switching points (SSPs) **236, 238, 240** and **242**, signal transfer points (STPS) **222, 224, 226, 228, 230** and **232**, and service control point (SCP) **234**.

In the SS7 network, the SSPs are the portions of the backbone switches providing SS7 functions. The SSPs can be, for example, a combination of a voice switch and an SS7 switch, or a computer connected to a voice switch. The SSPs communicate with the switches using primitives, and create packets for transmission over the SS7 network.

EOs **104***a*, **108***a* and ATs **206, 208** can be respectively represented in SS7 signaling network **114** as SSPs **236, 238, 240** and **242**. Accordingly, the connections between EOs **104***a*, **108***a* and ATs **206, 208** (presented as dashed lines) can be represented by connections **254, 256, 258** and **268**. The types of these links are described below.

The STPs act as routers in the SS7 network, typically being provided as adjuncts to in-place switches. The STPs route messages from originating SSPs to destination SSPs. Architecturally, STPs can and are typically provided in "mated pairs" to provide redundancy in the event of congestion or failure and to share resources (i.e., load sharing is done automatically). As illustrated in FIG. 2B, STPs can be arranged in hierarchical levels, to provide hierarchical routing of signaling messages. For example, mated STPs **222, 224** and mated STPs **226, 228** are at a first hierarchical level, while mated STPs **230, 232** are at a second hierarchical level.

SCPs provide database functions. SCPs can be used to provide advanced features in an SS7 network, including routing of special service numbers (e.g., 800 and 900 numbers), storing information regarding subscriber services, providing calling card validation and fraud protection, and offering advanced intelligent network (AIN) services. SCP **234** is connected to mated STPs **230** and **232**.

In the SS7 network, there are unique links between the different network elements. Table 4 provides definitions for common SS7 links.

Referring to FIG. 2B, mated STP pairs are connected by C links. For example, STPs **222, 224**, mated STPs **226, 228**, and mated STPs **230, 232** are connected by C links (not labeled). SSPs **236, 238** and SSPs **240, 242** are connected by F links **262** and **264**.

Mated STPs **222, 224** and mated STPs **226, 228**, which are at the same hierarchical level, are connected by B links **270, 272, 244** and **282**. Mated STPs **222, 224** and mated STPs **230, 232**, which are at different hierarchical levels, are connected by D links **266,268,274** and **276**. Similarly, mated STPs **226, 228** and mated STPs **230, 232**, which are at different hierarchical levels, are connected by D links **278, 280, 246** and **248**.

SSPs **236, 238** and mated STPs **222, 224** are connected by A links **254** and **256**. SSPs **240, 242** and mated STPs **226, 228** are connected by A links **258** and **260**.

SSPs **236, 238** can also be connected to mated STPs **230, 232** by E links (not shown). Finally, mated STPs **230, 232** are connected to SCP **234** by A links **250** and **252**.

For a more elaborate description of SS7 network topology, the reader is referred to Russell, Travis, Signaling System #7, McGraw-Hill, New York, N.Y. 10020, ISBN 0-07-054991-5, which is incorporated herein by reference in its entirety.

TABLE 4

SS7 link terminology	Definitions
Access (A) links	A links connect SSPs to STPs, or SCPs to STPs, providing network access and database access through the STPs.
Bridge (B) links	B links connect mated STPs to other mated STPs.
Cross (C) links	C links connect the STPs in a mated pair to one another. During normal conditions, only network management messages are sent over C links.
Diagonal (D) links	D links connect the mated STPs at a primary hierarchical level to mated STPs at a secondary hierarchical level.
Extended (E) links	E links connect SSPs to remote mated STPs, and are used in the event that the A links to home mated STPs are congested.
Fully associated (F) links	F links provide direct connections between local SSPs (bypassing STPs) in the event there is much traffic between SSPs, or if a direct connection to an STP is not available. F links are used only for call setup and call teardown.

d. SS7 Signaled Call Flow

To initiate a call in an SS7 telecommunications network, a calling party using a telephone connected to an ingress EO switch, dials a telephone number of a called party. The telephone number is passed from the telephone to the SSP at the ingress EO of the calling party's local exchange carrier (LEC). First, the SSP can process triggers and internal route rules based on satisfaction of certain criteria. Second, the SSP can initiate further signaling messages to another EO or access tandem (AT), if necessary. The signaling information can be passed from the SSP to STPs, which route the signals between the ingress EO and the terminating end office, or egress EO. The egress EO has a port designated by the telephone number of the called party. The call is set up as a direct connection between the EOs through tandem switches if no direct trunking exists or if direct trunking is full. If the call is a long distance call, i.e., between a calling party and a called party located in different local access transport areas (LATAs), then the call is connected through an inter

exchange carrier (IXC) switch. Such a long distance call is commonly referred to as an inter-LATA call. LECs and IXCs are collectively referred to as the public switched telephone network (PSTN).

Passage of the Telecommunications Act of 1996, authorizing competition in the local phone service market, has permitted CLECs to compete with ILECs in providing local exchange services. This competition, however, has still not provided the bandwidth necessary to handle the large volume of voice and data communications. This is due to the limitations of circuit switching technology which limits the bandwidth of the equipment being used by the LECs, and to the high costs of adding additional equipment.

e. Circuit-Switching

Circuit switching dedicates a channel to a call for the duration of the call. Thus, using circuit switching, a large amount of switching bandwidth is required to handle the high volume of voice calls. This problem is compounded by the use of voice circuits to carry data communications over the same equipment that were designed to handle voice communications.

i. Time Division Multiplexed (TDM) Circuit Switching

TDM circuit switching creates a full-time connection or a dedicated circuit between any two attached devices for the duration of the connection. TDM divides the bandwidth down int fixed time slots in which there can be multiple time slots, each with its own fixed capacity, available. Each attached device on the TDM network is assigned a fixed portion of the bandwidth using one or more time slots depending on the need for speed. When the device is in transmit mode, the data is merely placed in this time slot without any extra overhead such as processing or translations. Therefore, TDM is protocol transparent to the traffic being carried. Unfortunately, however, when the device is not sending data, the time slots remain empty, thereby wasting the use of the bandwidth. A higher-speed device on the network can be slowed down or bottled up waiting to transmit data, but the capacity that sits idle cannot be allocated to this higher priority device for the duration of the transmission. TDM is not well suited for the bursts of data that are becoming the norm for the data needs in today's organization.

2. Data Network

FIG. **1B** depicts an example network **148** including workstations **144** and **146** coupled to data network **142**. Data network **142** can act as a wide area network (WAN) for coupling a plurality of local area networks (LANs) together. Network **148** includes an example local area network including a plurality of host computers such as, e.g., client workstation **138** and server **136**, coupled together by wiring including network interface cards (NICs) and a hub, such as, e.g., an Ethernet hub. The LAN is coupled to data network **142** by a network router **140** which permits data traffic to be routed to workstations **144** and **146** from client **138** and server **136**.

a. Packet-Switching

Unlike voice networks **100** and **200** described above witb reference to FIGS. **1A** and **2A** which transport traffic over circuit-switched connections, data network **148** transports traffic using packet switching.

Currently, internets, intranets, and similar public or private data networks that interconnect computers generally use packet switching technology. Packet switching provides for more efficient use of a communication channel than does circuit switching. Packet switched networks transport packets of information which can include various types of data such as, e.g., digitized voice, data, and video. With packet switching, many different calls can share a communication channel rather than the channel being dedicated to a single call. During a voice call, for instance, digitized voice information might be transferred between the callers only 60% of the time, with silence being transferred the other 40% of the time. With a circuit switched connection, the voice call could tie-up a communications channel that could have 50% of its bandwidth, unused because of the silence. For a data call, information might be transferred between two computers only 10% of the time. With the data call, 90% of the channel's bandwidth may go unused. In contrast, a packet-switched connection would permit the voice call, the data call and possibly other call information to all be sent over the same channel.

Packet switching breaks a media stream into pieces known as, for example, packets, cells or frames. Each packet can then be encoded with address information for delivery to the proper destination and can be sent through the network. The packets can be received at the destination and the media stream is reassembled into its original form for delivery to the recipient. This process is made possible using an important family of communications protocols, commonly called the Internet Protocol (IP).

In a packet-switched network, there is no single, unbroken physical connection between sender and receiver. The packets from many different calls share network bandwidth with other transmissions. The packets can be sent over many different routes at the same time toward the destination, and can then be reassembled at the receiving end. The result is much more efficient use of a telecommunications network's bandwidth than could be achieved with circuit-switching.

b. Routers

Data network **142** can include a plurality of network routers **140**. Network routers are used to route information between multiple networks. Routers act as an interface between two or more networks. Routers can find the best path between any two networks, even if there are several different networks between the two networks.

Network routers can include tables describing various network domains. A domain can be thought of as a local area network (LAN) or wide area network (WAN). Information can be transferred between a plurality of LANs and/or WANs via network routers. Routers look at a packet and determine from the destination address in the header of the packet, the destination domain of the packet. If the router is not directly connected to the destination domain, then the router can route the packet to the router's default router, i.e. a router higher in a hierarchy of routers. Since each router has a default router to which it is attached, a packet can be transmitted through a series of routers to the destination domain and to the destination host bearing the packet's final destination address.

C. Local Area Networks (LANs) and Wide Area Networks (WANs)

A local area network (LAN) can be thought of as a plurality of host computers interconnected via network interface cards (NICs) in the host computers. The NICs are connected via, for example, copper wires so as to permit communication between the host computers. Examples of LANs include an ethernet bus network, an ethernet switch network, a token ring network, a fiber digital data interconnect (FDDI) network, and an ATM network.

A wide area network (WAN) is a network connecting host computers over a wide area. In order for host computers on a particular LAN to communicate with a host computer on another LAN or on a WAN, network interfaces interconnecting the LANs and WANs must exist. An example of a network interface is a router discussed above.

A network designed to interconnect multiple LANs and/or WANs is known as an internet (with a lower case "i"). An internet can transfer data between any of a plurality of networks including both LANs and WANs. Communication occurs between host computers on one LAN and host computers on another LAN via, for example, an internet protocol (IP) protocol. The IP protocol is used to assign each host computer of a network, a unique IP address enabling packets to be transferred over the internet to other host computers on other LANs and/or WANs that are connected to the internet. An internet can comprise a router interconnecting two or more networks.

The "Internet" (with a capital "I") is a global internet interconnecting networks all over the world. The Internet includes a global network of computers which intercommunicate via the internet protocol (IP) family of protocols.

An "intranet" is an internet which is a private network that uses internet software and internet standards, such as the internet protocol (IP). An intranet can be reserved for use by parties who have been given the authority necessary to use that network.

d. Switching vs. Routing

Routing is done at the middle network architecture levels on such protocols as IPX or TCP/IP. Switching is done at a lower level, at layer 2 of the OSI model, i.e. the media access control (MAC) layer.

e. TCP/IP Packet-Centric vs. ATM Circuit-Centric Data Networks

Asynchronous Transfer Mode (ATM) is a fixed-size cell switched circuit-centric data network. ATM implements virtual circuits (VCS), virtual paths (VPs) and transmission paths (TPs). A circuit-centric network like ATM sets up virtual circuits between source and destination nodes which provide QoS by dedicating the virtual circuit to a specific traffic type.

Some networks are packet-centric networks. Unlike a circuit-centric network, a packet-centric network does not use dedicated circuits through which to transfer packets. TCP/IP performs a packetization of user data to be sent between and among the various systems on the IP network. When a large file is sent down the protocol stack, the IP function is responsible for segmentation and packetization of the data. Then a header is placed on the packet for delivery to the data link. The routing and switching of this data is handled at the IP (i.e. network) layer. IP is in a sense a dumb protocol. When a packet is prepared for transmission across the medium, IP does not specifically route the call across a specific channel. Instead, it places a header on the packet and lets the network deal with it. Therefore, the outward bound packets can take various routes to get from a source to a destination. This means that the packets are in a datagram form and not sequentially numbered as they are in other protocols. IP makes its best attempt to deliver the packets to the destination network interface; but it makes no assurances that data will arrive, that data will be free of errors, and that nodes along the way will concern themselves with the accuracy of the data and sequencing, or come back and alert the originator that something is wrong in the delivery mechanism. It is possible that in IP routing of a packet, the packet can be sent along the network in a loop, so IP has a mechanism in its header information to allow a certain number of "hops" or what is called "time to live" on the network. Rather than permit an undeliverable pack to loop around the network, IP has a counter mechanism that decrements every time the packet passes through a network node. If the counter expires, the node will discard the packet. Working together with IP is TCP which provides controls to ensure that a reliable data stream is sent and delivered. At the sending end, TCP puts a byte count header on information that will be delivered to the IP protocol layer and encapsulates it as part of the packet. The receiving end, when it gets packets is responsible for resequencing the packets and ensuring its accuracy. If all of the IP flow is not received correctly, the byte count acknowledgment or nonacknowledgment message can be sent back to the sending end, prompting the sending end to resend the bytes necessary to fill in the remaining portions of the packet flow. TCP buffers additional packets until after resending the nonacknowledged packet.

3. Video Network

FIG. 1C illustrates a conventional video network **150** such as, e.g., a cable television (CATV) network. Video network **150** can include video network **160** coupled to various video capture, distribution links and video output monitors. Video input devices can include, e.g., conference cameras **154** and **158**. Video output devices can include, e.g., televisions **152** and **156**. Video network **160** can include a variety of head end (i.e. the serving end of the cable) and distribution link equipment such as, e.g., coaxial cable television (CATV) and national television standard code (NTSC) tuner equipment for multiplexing various video signals. Standard cable systems have an immense amount of bandwidth available to them.

It is important to note that CATV is a wireless communication method. The frequencies of many video signals are distributed along the cable at the same time. A television tuner selects a particular channel by tuning into a specific frequency or a "frequency band."

Although a cable television CATV video network often includes only one physical cable, a number of channels can simultaneously be present on the cable. This accomplished by sharing the frequency spectrum of the cable and assigning different frequency ranges to different channels using frequency division multiplexing (FDM). A broadband cable communications system can operate exactly like a CATV system. A counter to this FDM technique is division of the cable not divided into frequency bands but into time slots using time-division multiplexing (TDM). With TDM, each transmitting video station can grab the entire bandwidth of the cable, but only for a very short period of time. The cable is currently capable of carrying up to 750 MHz. FDM techniques can be used to divide the channels into a number of dedicated logical channels. Innovations have allowed a time division multiple access (TDMA) within an FDM channel.

A cable system can allow multiplexing on two separate dimensions to achieve data channels over a cable. The channels can be separated by FDM, and in a frequency band the channel can then be shared via TDMA among multiple users. The most common of the TDMA access methods on broadband cable is CSMA/CD developed by XEROX for Ethernet.

Using a single cable, a midsplit arrangement can accommodate two-way simultaneous transmission. Another way to accomodate this is to use a dual cable system.

Broadband is inherently an analog signaling method. Because video cameras, e.g., are also analog devices, a signal from a video camera (or video recorder) can be directly transmitted onto a broadband cable channel in red/green/blue (RGB) format.

G. Convergence of Voice/Data/Video Networks

Recognizing the inherent efficiency of packet-switched data networks such as the Internet, attention has recently focused on the digitization and transmission of voice, data,

video and other information over converged packet-switched data networks. In order to deliver a high quality of service (QoS) end-user experience, the data networks attempt to provide mechanisms to deliver the different types of information timely and with appropriate bandwidth to provide an acceptable end-user experience.

FIG. 2C illustrates an example network **286** carrying voice, data and video traffic over a data network. Network **286** includes calling party **102***b* homed to EO **104***b*, where EO **104***b* is linked to a telephony gateway **288***b*. Network **286** also includes called party **110***c* homed to EO **108***c*, where EO **108***c* is linked to a telephony gateway **288***c*. EOs **104***b* and **108***c* and telephony gateways **288***b* and **288***c* can be linked to signaling network **114**. Telephony gateways **288***b* and **288***c* can also be coupled to data network **142** via routers **140***b* and **140***c*, respectively.

Still referring to FIG. 2C, telephony gateways **288***b* and **288***c* can be used to packetize voice traffic and signaling information into a form appropriate for transport over data network **142**. It would be apparent to those skilled in the art that telephony gateways **288***b* and **288***c* can include various computer devices designed for controlling, setting up and tearing down calls. Voice calls delivered over the data network can include, e.g., voice over packet (VoP), voice over data (VoD), voice over internet protocol (VoIP), voice over asynchronous transfer mode (VoATM), voice over frame (VoF). An example of a telephony gateway **288***b* and **288***c* is a media gateway control protocol (MGCP) compliant gateway available from various vendors such as, e.g., Lucent, of Parsippany, N.J., and CISCO of Palo Alto, Calif. It is important to note that other network devices such as a softswitch available from several member companies of the SoftSwitch Consortium, including Level 3 Communications of Louisville, Colo., could also be necessary to enable transport of, e.g., VoIP.

Network **286** is depicted to include other devices coupled to data network **142**. First, an H.323 compliant video-conferencing system **289** is illustrated including a camera **154***g* and television **152***g* and router **140***g*. Second, a local area network (LAN) **128***a* including a client workstation **138***a* and a server **136***a* are coupled to data network **142** via network router **140***a*. Similarly, LAN **128***f* having a client workstation **138***f* and a server **136***f* are coupled via network router **140***f* to data network **142**.

Data Network **142** can provide for routing of packets of information through network routing devices from source locations to destination locations coupled to data network **142**. For example, data network **142** can route internet protocol (IP) packets for transmission of voice and data traffic from telephony gateway **288***b* to telephony gateway **288***c*. Data Network **142** represents any art-recognized packet centric data network. One well-known data network is the global Internet. Other examples include a private intranet, a packet-switched network, a frame relay network, and an asynchronous transfer mode (ATM) circuit-centric network.

In an example embodiment, data network **142** can be an IP packet-switched network. A packet-switched network such as, e.g., an IP network, unlike a circuit-switched network, does not require dedicated circuits between originating and terminating locations within the packet switched network. The packet-switched network instead breaks a message into pieces known as packets of information. Such packets can then be encapsulated with a header which designates a destination address to which the packet must be routed. The packet-switched network then takes the packets and routes them to the destination designated by the destination address contained in the header of the packet.

Routers **140***a*, **140***b*, **140***c*, **140***d*, **140***e*, **140***f* and **140***g* can be connected to one another via physical media such as, for example, optical fiber link connections, and copper wire connections. Routers **140***a–g* transfer information between one another and intercommunicate according to routing protocols.

Data network **142** could be implemented using any data network such as, e.g., IP networks, ATM virtual circuit-centric networks, frame relay networks, X.25 networks, and other kinds of LANs and WANs. Other data networks could be used interchangeably for data network **142** such as, for example, FDDI, Fast Ethernet, or an SMDS packet switched network. Frame relay and ATM are connection-oriented, circuit-centric services. Switched multi-megabyte data service (SMDS) is a connection-oriented mass packet service that offers speeds up to 45 Mbps.

1. Example Data Networks

a. Asynchronous Transfer Mode (ATM)

ATM is a high-bandwidth, low-delay, fixed-sized cell-based multiplexing network technology. Bandwidth capacity is segmented into 53-byte cells, having a header and payload fields. ATM uses fixed-length cells with the belief that the fixed length cells can be switched more easily in hardware than variable size packets and thus should result in faster transmissions in certain environments.

The ATM environment sets up virtual circuits in a circuit-centric manner. Thus, ATM segments variable length IP packet flows into fixed size cells using a segmentation and resequencing algorithm (SAR).

Each ATM cell contains a 48-byte payload field and a 5-byte header that identifies the so-called "virtual circuit" of the cell. ATM is thought suitable for high-speed combinations of voice, data, and video services. Currently, ATM access can perform at speeds as high as 622 Mbps or higher. ATM has recently been doubling its maximum speed every year.

ATM is defined by a protocol standardized by the International Telecommunications Union (ITU-T), American National Standards Institute (ANSI), ETSI, and the ATM Forum. ATM comprises a number of building blocks, including transmission paths, virtual paths, and virtual channels. Asynchronous transfer mode (ATM) is a cell based switching and multiplexing technology designed to be a general purpose connection-oriented transfer mode for a wide range of telecommunications services. ATM can also be applied to LAN and private network technologies as specified by the ATM Forum.

ATM handles both connection-oriented traffic directly or through adaptation layers, or connectionless traffic through the use of adaptation layers. ATM virtual connections may operate at either a constant bit rate (CBR) or a variable bit rate (VBR). Each ATM cell sent into an ATM network contains a small header including information that establishes a virtual circuit-centric connection from origination to destination. All cells are transferred, in sequence, over this virtual connection. ATM provides either permanent or switched virtual connections (PVCs or SVCs). ATM is asynchronous because the transmitted cells need not be periodic as time slots of data are required to be in synchronous transfer mode (STM).

ATM uses an approach by which a header field prefixes each fixed-length payload. The ATM header identifies the virtual channel (VC). Therefore, time slots are available to any host which has data ready for transmission. If no hosts are ready to transmit, then an empty, or idle, cell is sent.

ATM permits standardization on one network architecture defining a multiplexing and a switching method. Synchro-

nous optical network (SONET) provides the basis for physical transmission at very high-speed rates. ATM can also support multiple quality of service (QoS) classes for differing application requirements by providing separate virtual circuits for different types of traffic, depending on delay and loss performance. ATM can also support LAN-like access to available bandwidth.

Cells are mapped into a physical transmission path, such as the North American DS1, DS3, and SONET; European, E1, E3, and E4; ITU-T STM standards; and various local fiber and electrical transmission payloads. All information is multiplexed and switched in an ATM network via these fixed-length cells.

The ATM cell header field identifies cell type, and priority, and includes six portions. An ATM cell header includes a generic flow control (GFC), a virtual path identifier (VPI), a virtual channel identifier (VCI), a payload type (PT), a call loss priority (CLP), and a header error check (HEC). VPI and VCI hold local significance only, and identify the destination. GFC allows a multiplexer to control the rate of an ATM terminal. PT indicates whether the cell contains user data, signaling data, or maintenance information. CLP indicates the relative priority of the cell, i.e., lower priority cells are discarded before higher priority cells during congested intervals. HEC detects and corrects errors in the header.

The ATM cell payload field is passed through the network intact, with no error checking or correction. ATM relies on higher-layer protocols to perform error checking and correction on the payload. For example, a transmission control protocol (TCP) can be used to perform error correction functions. The fixed cell size simplifies the implementation of ATM switches and multiplexers and enables implementations at high speeds.

When using ATM, longer packets cannot delay shorter packets as in other packet-switched networks, because long packets are separated into many fixed length cells. This feature enables ATM to carry CBR traffic, such as voice and video, in conjunction with VBR data traffic, potentially having very long packets, within the same network.

ATM switches take traffic and segment it into the fixed-length cells, and multiplex the cells into a single bit stream for transmission across a physical medium. As an example, different kinds of traffic can be transmitted over an ATM network including voice, video, and data traffic. Video and voice traffic are very time-sensitive, so delay cannot have significant variations. Data, on the other hand, can be sent in either connection-oriented or connectionless mode. In either case, data is not nearly as delay-sensitive as voice or video traffic. Data traffic, as e.g., spread sheet data requires accurate transmission. Therefore, ATM conventionally must discriminate between voice, video, and data traffic. Voice and video traffic requires priority and guaranteed delivery with bounded delay, while data traffic requires, simultaneously, assurance of low loss. In a converged data network, data traffic can also carry voice traffic, making it also time-dependent. Using ATM, in one embodiment, multiple types of traffic can be combined over a single ATM virtual path (VP), with virtual circuits (VCs) being assigned to separate data, voice, and video traffic.

A transmission path can include one or more VPs. Each VP can include one or more VCs. Thus, multiple VCs can be trunked over a single VP. Switching can be performed on a transmission path, VPs, or at the level of VCs.

The capability of ATM to switch to a virtual channel level is similar to the operation of a private or public branch exchange (PBX) or telephone switch in the telephone world. In a PBX switch, each channel within a trunk group can be switched. Devices which perform VC connections are commonly called VC switches because of the analogy to telephone switches. ATM devices which connect VPs are commonly referred to as VP cross-connects, by analogy with the transmission network. The analogies are intended for explanatory reasons, but should not be taken literally. An ATM cell-switching machine need not be restricted to switching only VCs and cross-connection to only VPs.

At the ATM layer, users are provided a choice of either a virtual path connection (VPC) or a virtual channel connection (VCC). Virtual path connections (VPCs) are switched based upon the virtual path identifier (VPI) value only. Users of a VPC can assign VCCs within a VPI transparently, since they follow the same route. Virtual channel connections (VCCs) are switched upon a combined VPI and virtual channel identifier (VCI) value.

Both VPIs and VCIs are used to route calls through a network. Note that VPI and VCI values must be unique on a specific transmission path (TP).

It is important to note that data network **142** can be any of a number of other data-type networks, including various packet-switched data-type networks, in addition to an ATM network.

b. Frame Relay

Alternatively, data network **142** can be a frame relay network. It would be apparent to persons having ordinary skill in the art, that a frame relay network could be used as data network **142**. Rather than transporting data in ATM cells, data could be transported in frames.

Frame relay is a packet-switching protocol used in WANs that has become popular for LAN-to-LAN connections between remote locations. Formerly frame relay access would top out at about 1.5 Mbps. Today, so-called "high-speed" frame relay offers around 45 Mbps. This speed is still relatively slow as compared with other technology such as ATM.

Frame relay services employ a form of packet-switching analogous to a streamlined version of X.25 networks. The packets are in the form of frames, which are variable in length. The key advantage to this approach it that a frame relay network can accommodate data packets of various sizes associated with virtually any native data protocol. A frame relay network is completely protocol independent. A frame relay network embodiment of data network **142** does not undertake a lengthy protocol conversion process, and therefore offers faster and less-expensive switching than some alternative networks. Frame relay also is faster than traditional X.25 networks because it was designed for the reliable circuits available today and performs less-rigorous error detection.

c. Internet Protocol (IP)

In an embodiment, data network **142** can be an internet protocol (IP) network over an ATM network. It would be apparent to those skilled in the art, that an internet protocol (IP) network over various other data link layer network suchas, e.g., Ethernet, could be used as data network **142**. Rather than transporting data in fixed length ATM circuit-centric cells, data could be transported in variable length IP datagram packet-centric packets as segmented by TCP. The IP data network can lie above any of a number of physical networks such as, for example, a SONET optical network.

2. Virtual Private Networks (VPNs)

A virtual private network (VPN) is a wide area communications network operated by a telecommunications carrier that provides what appears to be dedicated lines when used, but that actually includes trunks shared among all customers as in a public network. Just as a VPN can be provided as a

service through a wireline network, a VPN can be provided in a wireless network. A VPN can allow a private network to be configured within a public network.

VPNs can be provided by telecommunications carriers to customers to provide secure, guaranteed, long-distance bandwidth for their WANs. These VPNs generally use frame relay or switched multi-megabyte data service (SMDS) as a protocol of choice because those protocols define groups of users logically on the network without regard to physical location. ATM has gained favor as a VPN protocol as companies require higher reliability and greater bandwidth to handle more complex applications. VPNs using ATM offer networks of companies with the same virtual security and QoS as WANs designed with dedicated circuits.

The Internet has created an alternative to VPNs, at a much lower cost, i.e. the virtual private Internet. The virtual private Internet (VPI) lets companies connect disparate LANs via the Internet. A user installs either a software-only or a hardware-software combination that creates a shared, secure intranet with VPN-style network authorizations and encryption capabilities. A VPI normally uses browser-based administration interfaces.

3. H.323 Video Conferencing

The H.323 Recommendation for video conferencing will now be briefly overviewed. The H.323 standard provides a foundation for, for example, audio, video, and data communications across IP-based networks, including the Internet. By complying with the H.323 Recommendation, multimedia products and applications from multiple vendors can interoperate, allowing users to communicate without concern for compatibility. H.323 promises to be the foundation of future LAN-based products multimedia applications.

H.323 is an umbrella recommendation from the International Telecommunications Union (ITU) that sets standards for multimedia communications over Local Area Networks (LANs) that do not provide a guaranteed Quality of Service (QoS). These networks dominate today's corporate desktops and include packet-switched TCP/IP and IPX over Ethernet, Fast Ethernet and Token Ring network technologies. Therefore, the H.323 standards are important building blocks for a broad new range of collaborative, LAN-based applications for multimedia communications.

The H.323 specification was approved in 1996 by the ITU's Study Group 16. Version 2 was approved in January 1998. The standard is broad in scope and includes both stand-alone devices and embedded personal computer technology as well as point-to-point and multipoint conferences. H.323 also addresses call control, multimedia management, and bandwidth management as well as interfaces between LANs and other networks.

H.323 is part of a series of communications standards that enable videoconferencing across a range of networks. Known as H.32X, this series includes H.320 and H.324, which address ISDN and PSTN communications, respectively.

The H.323 architecture defines four major components for network-based communications, including terminals, gateways, gatekeepers, and multipoint control units (MCUs).

Terminals are client endpoints on the LAN that provide real-time, two-way communications. All terminals support voice communications; video and data are optional. H.323 specifies the modes of operation required for different audio, video, and/or data terminals to work together. H.323 is the standard of next generation Internet phones, audio conferencing terminals, and video conferencing technologies.

All H.323 terminals also support H.245, which is used to negotiate channel usage and capabilities. Three other components are required: Q.931 for call signaling and call setup, a component called Registration/Admission/Status (RAS), which is a protocol used to communicate with a gatekeeper; and support for RTP/RTCP for sequencing audio and video packets.

Optional components in an H.323 terminal are video codecs, T.120 data conferencing protocols, and MCU capabilities.

A gateway is an optional element in an H.323 conference. An H.323 gateway can provide many services, the most common being a translation function between H.323 conferencing endpoints and other terminal types. This function includes translation between transmission formats (i.e. H.225.0 to H.221) and between communications procedures (i.e. H.245 to H.242). In addition, a gateway also translates between audio and video codecs and performs call setup and clearing on both the LAN side and the switched-circuit network side.

In general, the purpose of the H.323 gateway is to reflect characteristics of a LAN endpoint to an SCN endpoint and vice versa. The primary applications of gateways are likely to be establishing links with analog PSTN terminals, establishing links with remote H.320 compliant terminals over ISDN-based switched-circuit networks, and establishing links with remote H.324-compliant terminals over PSTN networks.

Gateways are not required if connections to other networks are not needed, since endpoints may directly communicate with other endpoints on the same LAN. Terminals communicate with gateways using the H.245 and Q.931 protocols.

With the appropriate transcoders, H.323 gateways 5806 can support terminals that comply with H.310, H.321, H.322, and V.70.

Many gateway functions are left to the designer. For example, the actual number of H.323 terminals that can communicate through the gateway is not subject to standardization. Similarly, the number of SCN connections, the number of simultaneous independent conferences supported, the audio/video/data conversion functions, and inclusion of multipoint functions are left to the manufacturer. By incorporating H.323 gateway technology into the H.323 specification, the ITU has positioned H.323 as the means to hold standards-based conferencing endpoints together.

The gatekeeper is the most important component of an H.323 enabled network. It can act as the central point for all calls within its zone and provides call control services to registered endpoints. In many ways, an H.323 gatekeeper acts as a virtual switch.

Gatekeepers perform two important call control functions. The first is address translation from LAN aliases for terminals and gateways to IP or IPX addresses, as defined in the RAS specification. The second function is bandwidth management, which is also designated within RAS. For instance, if a network manager has specified a threshold for the number of simultaneous conferences on the LAN, the gatekeeper can refuse to make any more connections once the threshold is reached. The effect is to limit the total conferencing bandwidth to some fraction of the total available; the remaining capacity is left for e-mail, file transfers, and other LAN protocols. A collection of all terminals, gateways, and multipoint control units which can be managed by a single gatekeeper are known as an H.323 Zone.

An optional, but valuable feature of a gatekeeper is its ability to route H.323 calls. By routing a call through a gatekeeper, it can be controlled more effectively. Service

providers need this ability in order to bill for calls placed through their network. This service can also be used to re-route a call to another endpoint if a called endpoint is unavailable. In addition, a gatekeeper capable of routing H.323 calls can help make decisions involving balancing among multiple gateways. For instance, if a call is routed through a gatekeeper, that gatekeeper can then re-route the call to one of many gateways based on some proprietary routing logic.

While a gatekeeper is logically separate from H.323 endpoints, vendors can incorporate gatekeeper functionality into the physical implementation of gateways and MCUs.

A gatekeeper is not required in an H.323 system. However, if a gatekeeper is present, terminals must make use of the services offered by gatekeepers. RAS defines these as address translation, admissions control, bandwidth control, and zone management.

Gatekeepers can also play a role in multipoint connections. To support multipoint conferences, users would employ a gatekeeper to receive H.245 control channels from two terminals in a point-to-point conference. When the conference switches to multipoint, the gatekeeper can redirect the H.245 Control Channel to a multipoint controller, the MC. A gatekeeper need not process the H.245 signaling; it only needs to pass it between the terminals or between the terminals and the MC.

LANs which contain gateways could also contain a gatekeeper to translate incoming E.164 addresses into Transport Addresses. Because a Zone is defined by its gatekeeper, H.323 entities that contain an internal gatekeeper can require a mechanism to disable the internal function so that when there are multiple H.323 entities that contain a gatekeeper on a LAN, the entities can be configured into the same Zone.

The Multipoint Control Unit (MCU) supports conferences between three or more endpoints. Under H.323, an MCU consists of a Multipoint Controller (MC), which is required, and zero or more Multipoint Processors (MP). The MC handles H.245 negotiations between all terminals to determine common capabilities for audio and video processing. The MC also controls conference resources by determining which, if any, of the audio and video streams will be multicast.

The MC does not deal directly with any of the media streams. This is left to the MP, which mixes, switches, and processes audio, video, and/or data bits. MC and MP capabilities can exist in a dedicated component or be part of other H.323 components.

The present invention supports multicast for wireless base station **302**, including providing: compatibility with RFC 1112, 1584; recognition and support of multicasting applications, including: multimedia, teleconferencing, database, distributed computing, real-time workgroups; support of broadcasting function over wireless link; preserves bandwidth, retains QoS latency performance; support of IPv6 IGMP and IPv4 IGMP multicast; group membership query, group membership report messages.

Approved in January of 1998, version 2 of the H.323 standard addresses deficiencies in version 1 and introduces new functionality within existing protocols, such as Q.931, H.245 and H.225, as well as entirely new protocols. The most significant advances were in security, fast call setup, supplementary services and T.120/H.323 integration.

G. Packet-Centric QoS-Aware Wireless Point-to-MultiPoint (PtMP) Telecommunications System

1. Wireless Point-to-MultiPoint Telecommunications System

FIG. 2D depicts network **296** including a point-to-multipoint (PtMP) wireless network **298** coupled via router **140***d* to data network **142**. It is important to note that network **296** includes network **286** from FIG. 2C, plus PtMP wireless network **298**. PtMP wireless network **298** enables customer premise equipment (CPE) at a subscriber location to gain access to the various voice, data and video resources coupled to data network **142** by means of wireless connectivity over a shared bandwidth. The wireless PtMP network **298** is a packet switched network which is TCP/IP packet-centric (i.e. no dedicated circuit is created in delivering a communication IP flow) and QoS aware.

Specifically, PtMP wireless network **298** includes a wireless access point (WAP) **290***d* coupled to router **140***d* by, e.g., a wireline connection. A wireless access point **290***e* can be similarly coupled to router **140***e* by a wireline connection. WAP **290***d* is in wireless communication, such as, e.g., radio frequency (RF) communication, with one or more wireless transciever subscriber antennae **292***d* and **292***e*. It would be apparent to those skilled in the art that various wireless communication methods could be used such as, e.g., microwave, cellular, spread spectrum, personal communications systems (PCS), and satellite.

In an alternative embodiment, RF communication is accomplished over cable television (CATV) coaxial cable. As those skilled in the relevant art will understand, a coaxial cable functions as a waveguide over which RF waves propagate. Accordingly, it is possible for the communications link between RF transceiver subscriber antenna **292***d* and WAP **290***d* to be a coaxial cable. Therefore, a coaxial cable connection is analogous to a wireless connection, and is referred to as an alternative form of wireless connection in the present invention.

In another alternative embodiment, RF communication is accomplished over a satellite connection, such as, e.g., a low earth orbit (LEO) satellite connection or a high earth orbit satellite. Taking the example of an LEO satellite connection, WAP **290***d* and RF transceiver subscriber antenna **292***d* function as satellite gateways, with the additional functionalities described in the present invention.

As would be apparent to those skilled in the art, although the present invention has been described in the context of a point-to-multi-point network, the invention is equally applicable to a point-to-point network environment.

Referring to FIG. 3A, in an embodiment of the invention, WAPs **290***d* and **290***e* can be coupled to a wireless base station **302** where "IP flow" traffic can be queued, analyzed, characterized, classified, prioritized and scheduled, as described more fully below with reference to the ensuing figures.

Referring to FIG. 3B, one embodiment of the invention, antennae **292***d* and **292***e* are coupled to subscriber customer premise equipment (CPE) stations **294***d* and **294***e*, respectively (also referred to as CPEs **294***d*, **294***e*). Subscriber CPE stations **294***d* and **294***e* are coupled to various other CPE equipment via wireline or wireless connections. For example, CPE stations **290***d* and **290***e* can be coupled to voice calling parties **124***d*, **124***e*, **126***d* and **126***e*, fax machines **116***d* and **116***e*, video conferencing equipment including video monitors **152***d* and **152***e*, and cameras **154***d* and **154***e*, host computers including client computers **120***d* and **120***e* and servers **122***d* and **122***e*. Various legacy devices such as PBXs can be coupled to CPEs **294***d* and **294***e*. In addition, next generation technologies such as Ethernet phones available from Selsius, a subsidiary of CISCO Systems from San Jose, Calif. and other Internet appliances can be coupled via LAN connections to CPEs **294***d* and **294***e*. Other video conferencing equipment as well as H.323 compliant conferencing equipment can also be coupled to CPEs **294***d* and **294***e*.

In an embodiment of the invention, either of antennae **292***d* and **292***e* can communicate with both WAPs **290***d* and **290***e* for alternate or backup wireless communications paths.

Returning to FIG. **3**A, it depicts an example perspective diagram **300** of a PtMP network of the present invention. Diagram **300** includes a wireless base station **302** shown in wireless communication with subscriber locations **306***a*, **306***b*, **306***c*, **306***d*, **306***e*, **306***f*, **306***g*, **306***h*, **306***i* and **306***j*. Specifically, wireless base station **302** communicates via wireless access point **290***d* to subscriber antennae **292***a–j* of subscriber locations **306***a–j*.

Wireless base station **302** is coupled at interface **320** to network router **140***d* by, e.g., a wireline connection. Network router **140***d* is coupled to data network **142** which includes various other network routers **140***b* for routing traffic to other nodes on data network **142** such as, e.g., telephony gateway **288***b*.

Returning to FIG. **3**B, it depicts block diagram **310** further illustrating the wireless PtMP of the present invention. Diagram **310** includes wireless base station **302** coupled at interface **320** to data network **142**. Also coupled to data network **142** are router **140***d* and telephony gateway **288***b* which is in turn coupled to a class central office (CO) switch at EO **104***b*. IP telephony gateway **288***b* can terminate telephony traffic to PSTN facilities by, e.g., translating packets into time domain multiplexed (TDM) standard telephone signals. Wireless base station **302** is in communication with wireless CPE **294***d* at subscriber location **306***d* via antenna WAP **290***d* and **292***d*. It would be apparent to those skilled in the art that other configurations of CPE **294***d* are possible, such as, e.g., one or more host computers with no telephone devices, one or more telephones with no host computers, one or more host computers and one or more telephone devices, and one or more H.323 capable video-conferencing platforms which could include a host computer with monitor and camera.

CPE **294***d* is shown with several telephone devices **124***d* and **126***d*, e.g., analog phones, and host computers, client **120***d* and server **122***d*. Client **120***d* and server **122***d* can be coupled to CPE **294***d* via a LAN connection such as, e.g., an Ethernet LAN, or via a legacy V.35 device **322***d* providing a high speed data connection. Other Internet appliances capable of attachment to a data network can also be coupled to CPE **294***d*.

2. Networking Protocol Stack Architecture—Wireless IP Network Access Architecture (WINAAR)

FIG. **4** depicts the wireless IP network access architecture (WINAAR) **400** of the present invention. Architecture **400** illustrates the networking protocol stack which is a version of a TCP/IP protocol stack enhanced to support IP-centric, QoS over a packet switched, shared bandwidth, wireless PtMP connection. The networking protocol stack will be described in terms of the Open Systems Interconnect (OSI) 7 layer networking protocol stack standard which includes physical layer (OSI layer 1) **402**, data link layer (OSI layer 2) **404**, network layer (OSI layer 7) **406** and **408**, transport layer (OSI layer 4) **410** and applications layer (OSI layer 7) **412**.

a. Physical Layer

In an example embodiment, physical layer **402** can be implemented using several wireless application specific integrated circuits (wASICs), an off-the-shelf 16QAM/QPSK **416** ASIC; an Interference Mitigation and Multipath Negation (IMMUNE)/RF **418** algorithm ASIC for minimizing and/or eliminating harmful interference; and a frequency hopping (FH) **419** ASIC for providing dynamic and adaptive multi-channel transmission that optimizes data link integrity by changing frequency levels depending on the noise level of a given frequency. Physical layer **402** can include the radio frequency (RF) signal **415**.

b. Data Link Layer

Data link layer **404** lies on top of physical layer **402**. Data link layer **404** can include a media access control (MAC) layer **414** which is depicted graphically in diagram **400** as MAC layer portion **414***a* and proactive reservation-based intelligent multi-media access (PRIMMA) technology portions **414***b* and **414***c*. Arrows **426**, **428** and **430**, respectively, illustrate that MAC layer **414** can read header information from data and multimedia applications **425**, TCP/UDP **427** and IP **429** layers to analyze and schedule an IP packet of an "IP flow." IP packets of the IP flow are identified by analyzing the header information to determine QoS requirements of the IP flow, so that the IP flow can be characterized, classified, presented, prioritized and scheduled.

c. Network Layer

1. Internet Protocol (IP)

Network layer **408** is the Internet protocol (IP) **429**. As will be discussed further below and as already discussed above with reference to data network **142**, IP is a standard protocol for addressing packets of information. Referring now to FIG. **7**, IP header fields **702** can include, e.g., source and destination IP addresses, IP type of service (TOS), IP time to live (TTL), and protocol fields. IP is a datagram protocol that is highly resilient to network failures, but does not guarantee sequence delivery. Routers send error and control messages to other routers using the Internet control message protocol (ICMP). ICMP can also provide a function in which a user can send a "ping" (echo packet) to verify reachability and round trip delay of an IP-addresse host. Another OSI layer 3 protocol is address resolution protocol (ARP) which can directly interface to the data link layer. ARP maps a physical address, e.g., an Ethernet MAC address, to an IP address.

2. Internet Protocol (IP)v4 and IPv6

IP **429** of network layer **408** can be, e.g., an IP version 4 (IPv4) or an IP version 6 (IPv6). IPv6 (sometimes called next-generation internet protocol or IPng) is a backward-compatible extension of the current version of the Internet protocol, IPv4. IPv6 is designed to solve problems brought on by the success of the Internet (such as running out of address space and router tables). IPv6 also adds needed features, including circuiting security, auto-configuration, and real-time services similar to QoS. Increased Internet usage and the allocation of many of the available IP addresses has created an urgent need for increased addressing capacity. IPv4 uses a 32-byte number to form an address, which can offer about 4 billion distinct network addresses. In comparison, IPv6 uses 128-bytes per address, which provides for a much larger number of available addresses.

3. Resource Reservation Protocol (RSVP)

IP **429** of network layer **408** can have RSVP enhancement. Developed to enhance IPv4 with QoS features, RSVP is supposed to let network managers allocate bandwidth based on the bandwidth requirements of an application. Basically, RSVP is an emerging communications protocol that is hoped to signal a router to reserve bandwidth for real-time transmission of data, video, and audio traffic.

Resource reservation protocols that operate on a per-connection basis can be used in a network to elevate the priority of a given user temporarily. RSVP runs end to end to communicate application requirements for special handling. RSVP identifies a session between a client and a server and asks the routers handling the session to give its communications a priority in accessing resources. When the

session is completed, the resources reserved for the session are freed for the use of others.

RSVP unfortunately offers only two levels of priority in its signaling scheme. Packets are identified at each router hop as either low or high priority. However, in crowded networks, two-level classification may not be sufficient. In addition, packets prioritized at one router hop might be rejected at the next.

Accepted as an IETF standard in 1997, RSVP does not attempt to govern who should receive bandwidth, and questions remain about what will happen when several users all demand a large block of bandwidth at the same time. Currently, the technology outlines a first-come, first-served response to this situation. The IETF has formed a task force to consider the issue.

Because RSVP provides a special level of service, many people equate QoS with the protocol. For example, Cisco currently uses RSVP in its IPv4 -based internetwork router operating system to deliver IPv6 -type QoS features. However, RSVP is only a small part of the QoS picture because it is effective only as far as it is supported within a given client/server connection. Although RSVP allows an application to request latency and bandwidth, RSVP does not provide for congestion control or network-wide priority with the traffic flow management needed to integrate QoS across an enterprise. Further, RSVP does not address the particular challenges related to delivering packets over a wireless medium.

The present invention supports RSVP by providing: (1) compatibility with RFC **2205**; (2) recognition and support of RSVP messages, including: Path messages, Reservation (Resv), Path teardown messages, Resv teardown messages, Path error messages, Resv error messages, and Confirmation messages; (3) recognition and support of RSVP objects, including: Null, Session, RSVP_Hop, Time_Values, Style, Flowspec, Sender_Template, Sender_Tspec, Adspec, Error_Spec, Policy_Data, Integrity, and Scope, Resv_Confirm; (4) configurable translation of RSVP Flowspecs for QoS resource allocation in wireless base station **302**.

The present invention provides support of DiffServ and RSVP/int-serv by providing: (1) support of RFC **2474** and **2475**; (2) DiffServ in the core of Internet; (3) RSVP/int-serv for hosts and edge networks; (4) admission control capability for DiffServ compatibility; (5) differentiated services (DSs) (a field marking supported for use by DiffServ, and translation into a wireless base station **302** resource allocation); and (6) support for binding of multiple end-to-end sessions to one tunnel session.

4. Real-time Transport Protocol (RTP) and Real-time Control Protocol (RTCP)

TCP of transport layer **410** can have a RTP and RTCP enhancement. Real-time transport protocol (RTP) is an emerging protocol for the Internet championed by the audio/video transport workgroup of the IETF. Referring to FIG. **7**, RTP and RTCP header fields **708** can include several sub fields of information. RTP supports real-time transmission of interactive voice and video over packet-switched networks. RTP is a thin protocol that provides content identification, packet sequencing, timing reconstruction, loss detection, and security. With RTP, data can be delivered to one or more destinations, with a limit on delay.

RTP and other Internet real-time protocols, such as the Internet stream protocol version 2 (St2), focus on the efficiency of data transport. RTP and other Internet real-time protocols like RTCP are designed for communications sessions that are persistent and that exchange large amounts of data. RTP does not handle resource reservation or QoS control. Instead, RTP relies on resource reservation protocols such as RSVP, communicating dynamically to allocate appropriate bandwidth.

RTP adds a time stamp and a header that distinguishes whether an IP packet is data or voice, allowing prioritization of voice packets, while RSVP allows networking devices to reserve bandwidth for carrying unbroken multimedia data streams.

Real-time Control Protocol (RTCP) is a companion protocol to RTP that analyzes network conditions. RTCP operates in a multi-cast fashion to provide feedback to RTP data sources as well as all session participants. RTCP can be adopted to circumvent datagram transport of voice-over-IP in private IP networks. With RTCP, software can adjust to changing network loads by notifying applications of spikes, or variations, in network transmissions. Using RTCP network feedback, telephony software can switch compression algorithms in response to degraded connections.

5. IP Multi-Casting Protocols

IP **429** of network layer **408** can also support multi-casting protocols. Digital voice and video comprise of large quantities of data that, when broken up into packets, must be delivered in a timely fashion and in the right order to preserve the qualities of the original content. Protocol developments have been focused on providing efficient ways to send content to multiple recipients, transmission referred to as multi-casting. Multi-casting involves the broadcasting of a message from one host to many hosts in a one-to-many relationship. A network device broadcasts a message to a select group of other devices such as PCS or workstations on a LAN, WAN, or the Internet. For example, a router might send information about a routing table update to other routers in a network.

Several protocols are being implemented for IP multi-casting, including upgrades to the Internet protocol itself. For example, some of the changes in the newest version of IP, IPv6, will support different forms of addressing for uni-cast (point-to-point communications), any cast (communications with the closest member of a device group), and multi-cast. Support for IP multi-casting comes from several protocols, including the Internet group management protocol (IGMP), protocol-independent multi-cast (PIM) and distance vector multi-cast routing protocol (DVMRP). Queuing algorithms can also be used to ensure that video or other multi-cast data types arrive when they are supposed to without visible or audible distortion.

Real-time transport protocol (RTP) is currently an IETF draft, designed for end-to-end, real-time delivery of data such as video and voice. RTP works over the user datagram protocol (UDP), providing no guarantee of in-time delivery, quality of service (QoS), delivery, or order of delivery. RTP works in conjunction with a mixer and translator and supports encryption and security. The real-time control protocol (RTCP) is a part of the RTP definition that analyzes network conditions. RTCP provides mandatory monitoring of services and collects information on participants. RTP communicates with RSVP dynamically to allocate appropriate bandwidth.

Internet packets typically move on a first-come, first-serve basis. When the network becomes congested, Resource Reservation Protocol (RSVP) can enable certain types of traffic, such as video conferences, to be delivered before less time-sensitive traffic such as E-mail for potentially a premium price. RSVP could change the Internet's pricing structure by offering different QoS at different prices. Using SLAs, different QoS levels can be provided to users at CPE location stations depending on SLA subscription level.

The RSVP protocol can be used by a host, on behalf of an application, to request a specific QoS from the network for particular data streams or flows. Routers can use the RSVP protocol to deliver QoS control requests to all necessary network nodes to establish and maintain the state necessary to provide the requested service. RSVP requests can generally, although not necessarily, result in resources being reserved in each node along the data path.

RSVP is not itself a routing protocol. RSVP is designed to operate with current and future uni-cast and multi-cast routing protocols. An RSVP process consults the local routing database to obtain routes. In the multi-cast case for example, the host sends IGMP messages to join a multi-cast group and then sends RSVP messages to reserve resources along the delivery paths of that group. Routing protocols determine where packets are forwarded. RSVP is concerned with only the QoS of those packets as they are forwarded in accordance with that routing. The present invention delivers QoS-aware wireless PtMP access to users over a shared wireless bandwidth, and can take into account priority information provided within packet headers of packets in IP flows received for transmission over the wireless base station's bandwidth.

d. VPN Networks (Example Optional Protocols) at Network Layer

Also at network layer **406** are depicted example optional virtual private network (VPN) protocols point to point protocol (PPP) **420** and IPsec **422**, discussed below.

A plurality of protocol standards exist today for VPNs. For example, IP security (IPsec), point-to-point tunneling protocol (PPTP), layer 2 forwarding protocol (L2F) and layer 2 tunneling protocol (L2TP). The IETF has proposed a security architecture for the Internet protocol (IP) that can be used for securing Internet-based VPNs. IPsec facilitates secure private sessions across the Internet between organizational firewalls by encrypting traffic as it enters the Internet and decrypting it at the other end, while allowing vendors to use many encryption algorithms, key lengths and key escrow techniques. The goal of IPsec is to let companies mix-and-match the best firewall, encryption, and TCP/IP protocol products.

IPsec is designed to link two LANs together via an encrypted data stream across the Internet.

1. Point-to-Point Tunneling Protocol (PPTP)

Point-to-point tunneling protocol (PPTP) provides an alternate approach to VPN security than the use of IPsec. Unlike IPsec, which is designed to link two LANs together via an encrypted data stream across the Internet, PPTP allows users to connect to a network of an organization via the Internet by a PPTP server or by an ISP that supports PPTP. PPTP was proposed as a standard to the IETF in early 1996. Firewall vendors are expected to support PPTP.

PPTP was developed by Microsoft along with 3Com, Ascend and US Robotics and is currently implemented in WINDOWS NT SERVER 4.0, WINDOWS NT WORKSTATION 4.0, WINDOWS 95 via an upgrade and WINDOWS 98, available from Microsoft Corporation of Redmond, Wash.

The "tunneling" in PPTP refers to encapsulating a message so that the message can be encrypted and then transmitted over the Internet. PPTP, by creating a tunnel between the server and the client, can tie up processing resources.

2. Layer 2 Forwarding (L2F) Protocol

Developed by Cisco, layer 2 forwarding protocol (L2F) resembles PPTP in that it also encapsulates other protocols inside a TCP/IP packet for transport across the Internet, or any other TCP/IP network, such as data network **112**. Unlike PPTP, L2F requires a special L2F-compliant router (which can require changes to a LAN or WAN infrastructure), runs at a lower level of the network protocol stack and does not require TCP/IP routing to function. L2F also provides additional security for user names and passwords beyond that found in PPTP.

3. Layer 2 Tunneling Protocol (L2TP)

The layer 2 tunneling protocol (L2TP) combines specifications from L2F with PPTP. In November 1997, the IETF approved the L2TP standard. Cisco is putting L2TP into its Internet operating system software and Microsoft is incorporating it into WINDOWS NT 5.0. A key advantage of L2TP over IPsec, which covers only TCP/IP communications, is that L2TP can carry multiple protocols. L2TP also offers transmission capability over non-IP networks. L2TP however ignores data encryption, an important security feature for network administrators to employ VPNs with confidence.

4. IPsec

IP flows using the security encryption features of IPsec **422** are supported by the present invention. The integration of IPsec **422** flows of WINAAR architecture **400** are described below in the downlink and uplink directions with reference to FIGS. **17A** and **17B**, respectively. Wireless base station **302** supports prioritization of IPsec encrypted streams by placing the iS firewall at the wireless base station and unencrypting the datastream and packet header information prior to identification analysis. Through the wireless transmission medium, the frame stream already includes encryption of the frame data and implements frequency hopping.

IPsec provides for secure data transmission for, e.g., VPNs and eCommerce security. IPsec is compatible with RFC 2401–2407. IPsec is supported with IPv4 and IPv6, and also IPsec tunnel mode. Wireless base station **302** security protocol support includes authentication header (AH) and encapsulating security payload (ESP). Wireless base station **302** supports IPsec authentication (MD5), encryption algorithms, and automatic key management (IKE and ISAKMP/Oakley). Wireless base station **302** provides for a choice of transport mode or tunnel mode and selectable granularity of security service, such as, e.g., providing a single encrypted tunnel for all traffic between two hosts, or providing separate encrypted tunnel for each TCP connection between hosts.

e. Transport Layer

1. Transmission Control Protocol/Internet Protocol (TCP/IP) and User Datagram Protocol/Internet Protocol (UDP/IP)

As already discussed, internet protocol (IP) has become the primary networking protocol used today. This success is largely a part of the Internet, which is based on the transmission control protocol/internet protocol (TCP/IP) family of protocols. TCP/IP is the most common method of connecting PCs, workstations, and servers. TCP/IP is included as part of many software products, including desktop operating systems (e.g., Microsoft's Windows 95 or Windows NT) and LAN operating systems.

The most pervasive LAN protocol to date, has been IPX/SPX from Novell's NetWare network operating system (NOS). However, IPX/SPX is losing ground to TCP/IP. Novell now incorporates native IP support into NetWare, ending NetWare's need to encapsulate IPX packets when carrying them over TCP/IP connections. Both UNIX and Windows NT servers can use TCP/IP. Banyan's VINES, IBM's OS/2 and other LAN server operating systems can also use TCP/IP.

Transport layer four **410** can include transmission control protocol (TCP) or user datagram protocol (UDP) **427** part of

the standard TCP/UDP/IP protocol family suite of networking protocols. As will be discussed further below and as already mentioned briely above with reference to data network **142**, TCP is a standard protocol for segmenting traffic into packets, transmitting, reassembling and retransmitting packets of information between a source and destination IP address. Referring now to FIG. **7**, TCP header fields **706** can include, e.g., source and destination port numbers, window size, urgent pointer, flags (SYN, ISN, PSH, RST, FIN), and maximum segment size (MSS). Both TCP and UDP provide a capability for the TCP/IP host to distinguish among multiple applications through port numbers. TCP can provide for a reliable, sequenced delivery of data to applications. TCP can also provide adaptive flow control, segmentation, and reassembly, and prioritization of data flows. UDP only provides unacknowledged datagram capability. The recently defined real time protocol (RTP), RFC 1889, can provide real time capabilities in support of multimedia applications, for example.

TCP uses a window-based flow control. Each TCP source has a dynamically changing transmit window that determines how many packets it can transmit during each successive round-trip time (RTT). The TCP source can continue increasing its transmit window if no packets were lost within the last RTT. Once congestion is detected, the source TCP throttles back its transmission, i.e. it "backs-off," via a multiplicative decrease. An increasing width of the so-called TCP window versus time corresponds to increasingly longer bursts of packets. TCP's window flow-controlled protocol exhibits this effect of increasing throughput and buffer utilization until terminated by loss, followed by a period of rapid backoff.

TCP works over IP to provide end-to-end reliable transmission of data across data network **142**. TCP controls the amount of unacknowledged data in transit by dynamically reducing either window size or segment size. The reverse is also true in that increased window or segment size values achieve higher throughput if all intervening network elements have low error rates, support the larger packets, and have sufficient buffering to support larger window sizes.

f. Application Layer

Applications layer seven **412** can include applications **426** such as, e.g., over TCP, hypertext transport protocol (HTTP), file transfer protocol (FTP), TELNET remote terminal login, and simple simple mail transfer protocol (SMTP); and over UDP, simple network management protocol (SNMP), RPC, NFS, and TFTP. Other applications can also run over the network stack such as, e.g., a world wide web browser such as NETSCAPE NAVIGATOR available from AOL of Reston, Va., a spreadsheet application program such as LOTUS 123 available from IBM of Armonk, N.Y. or a video teleconferencing program such as MS NetMeeting available from MICROSOFT of Redmond, Wash. Packets transmitted from such applications could require special handling and prioritization to achieve an appropriate end-user QoS.

3. PRIMMA-System IP Flow Prioritization

a. Scheduling of Mixed IP Flows

FIG. **6** illustrates block diagram **600** representing scheduling of mixed IP flows. Block diagram **600** shows the scheduling of wireless base station **302**. The functionality of block diagram **600** includes PRIMMA management of Internet, VPN, and realtime IP flows. Referring back to FIG. **3A**, wireless IP flows are coming from data network **142** via network router **140***d* to interface **320** of wireless base station **302**. IP flows are then scheduled for transmission from wireless base station **302** via antenna **290***d* through subscriber location **306***d* via antenna **292***d*.

Referring back to block diagram **600** of FIG. **6**, illustrated therein are the downlink and uplink flows between interface **320** and wireless base station antenna **290***d*. An IP flow, as described herein, refers to a series of related packets of data transmitted from a source to a destination post computer. IP flow **630** from data network **142** (over interface **320**) comprises Internet IP flows **608**, VPN IP flows **610**, and realtime IP flows **612**. IP flow **630** is in the downlink direction.

Downlink IP flow analyzer **602** (hereinafter downlink flow analyzer **602**) analyzes Internet IP flow **608**, VPN IP flow **610** and realtime IP flow **612**. IP flow analyzer **602** is described further below with reference to FIGS. **8A** and **15A**. IP flow analyzer **602** receives packets and analyzes packet header fields to identify new or existing IP flows. IP flow analyzer **602** can also characterize QoS requirements for the IP flow depending on packet header field contents. IP flow analyzer **602** can classify the IP flow and associate a given packet with other packets from an existing IP flow and can group together IP flows with similar QoS requirements. IP flow analyzer **602** can also present the IP flows to a flow scheduler.

Downlink PRIMMA MAC IP flow scheduler **604** (hereinafter downlink flow scheduler **604**) schedules received IP flows **608**, **610**, and **612** for transmission in the downlink direction. Downlink flow scheduler **604** can prioritize the different classes of IP flows. For example, scheduler **604** can reserve slots in downlink frames for latency sensitive IP flows; for FTP type IP flows **608**, scheduler **604** can allocate large amounts of bandwidth for file transfer; and for e-mail type IP flows **608**, a lower priority can be given to packets. In prioritizing allocation of wireless bandwidth frame slots, downlink flow scheduler **604** can take into account the fact that an IP flow **630** is a VPN IP flow **610** from a virtual private network (VPN), such as, e.g., a remote branch office tieing into a corporate network. All traffic from a VPN can be given a higher priority or specific types of VPN traffic can request particular service levels. Downlink flow scheduler **604** can prioritize realtime IP flows **612** such that their arrival at CPEs **294** at CPE subscriber locations **306** will occur as required.

Downlink PRIMMA MAC segmentation and resequencing (SAR) and framer **606** (hereinafter downlink SAR and framer **606**) segments and frames the data packets of received IP flows into frames for transmission over the wireless medium to CPEs **294** at CPE subscriber locations **306**. For example IP flow **616**, **624** can be transmitted to CPE **294***d* at CPE subscriber location **306***d*, via base station antenna **290***d* over a wireless medium to subscriber antenna **292***d* and CPE **294***d* at CPE subscriber location **306***d*. In the present invention, the term wireless medium is used to broadly encompass not only propagation of RF transmissions over cellular communications, but also RF transmissions over satellite communications and cable (e.g., coaxial cable) communications.

In the uplink direction, IP flow **626** from CPE **294***d* at CPE subscriber station **306***d* is received at wireless base station antenna **290***d*. IP flow **626** can include Internet IP flow **618**, VPN IP flow **620** and realtime IP flow **622**. Uplink IP flow analyzer **632** (hereinafter uplink flow analyzer **632**) analyzes Internet IP flow **618**, VPN IP flow **620** and realtime IP flow **622**. Uplink flow analyzer **632** is described further below with reference to FIGS. **8B** and **15B**. In one embodiment, the functionality of IP flow analyzer **632** occurs at the CPE **294***d* at subscriber CPE location **306***d* and sends a request to transmit data up to wireless base station **302**, including information about an IP flow for which CPE **294***d* would like to schedule an uplink slot.

Uplink PRIMMA MAC IP flow scheduler **634** (hereinafter uplink flow scheduler **634**) can schedule the requested IP flow. In one embodiment, the functionality of scheduler **634** can be performed at CPE **294***d* at subscriber CPE location **306***d*. In another embodiment, the functionality of scheduler **634** can be performed at the wireless base station **302**. An advantage of placing uplink flow scheduler **634** at the wireless base station is that this provides efficiencies particularly in a point-to-multi-point architecture. It is more efficient to have one centralized scheduler at the base station **302** rather than to place multiple uplink flow schedulers **634** at CPEs **294** of subscriber CPE locations **306**.

Uplink PRIMMA MAC segmentation and resequencing (SAR) and framer **636** (hereinafter SAR and framer **636**) can segment and frame the data packets of IP flows into frames for transmission over the wireless medium from CPE **294** at CPE subscriber locations **306** to wireless base station **302** for further transmission over data network **142**. IP flow **626** from CPE **294***d* at CPE subscriber location **306***d* can be transmitted to base station antenna **290***d* over a wireless medium such as, e.g., RF communication, cable modem and satellite communication, from subscriber antenna **292***d* coupled to CPE **294***d* at CPE subscriber location **306***d*.

b. Summary of Downlink and Uplink SubFrame Prioritization

Block diagram **800** of FIG. **8**A summarizes an exemplary downlink analysis, prioritization and scheduling function. Similarly, block diagram **830** of FIG. **8**B summarizes an exemplary uplink analysis prioritization and scheduling function. Block diagram **800** and **830** are more detailed views of the function of block diagram **600** of FIG. **6**.

Beginning with block diagram **800** (of FIG. **8**A), it depicts how IP flow prioritization and scheduling of a shared wireless bandwidth is performed in the downlink path, from data network **142**—to router **140***d*—to interface **320**—to wireless base station **302**—WAP **290***d*—over a wireless medium—to wireless transceiver subscriber antenna **292***d*—to subscriber CPE station **294***d* at subscriber CPE location **306***d*.

IP flow analyzer **602** performs the function of identifying, characterizing, classifying, and presenting data packets to a downlink frame scheduler. The functions of identifying, characterizing, classifying and presenting the data packets are described with respect to FIG. **15**A.

During identification, it is determined whether a data packet of an incoming IP data flow is known to the system, i.e. is an "existing IP flow", or rather is the first data packet of a new IP data flow, based on fields in a packet header section. Identification can also include, e.g., determining the source of the packet in order to extrapolate the type of information in the packet payload.

During characterization, a new data packet (of a new IP data flow) previously unknown to the system is characterized based on the packet header information to determine the QoS requirements for the IP data flow, and to identify the subscriber CPE station that will receive the IP data flow.

During classification, the new IP data flow is classified into a communications priority class. Classification can also include grouping together packets from different IP flows having similar characteristics into a single class. Example class groupings of IP flows **630** are illustrated as IP classes **810***a*–**810***g*.

During presentation, the new IP data flow is initialized and presented to a downlink flow scheduler **604**.

Downlink flow scheduler places the data packets of an IP data flow into a class queue based on class queue priorities, and using a set of rules, schedules the data packets for transmission over a wireless medium to a subscriber CPE station **294** at subscriber CPE location **306** with an advanced reservation algorithm. The rules are determined by inputs to the downlink flow scheduler based on, e.g., a hierarchical class-based prioritization, a virtual private network (VPN) directory enabled data priority (such as, for example, directory enabled networking (DEN)), and a service level agreement priority. The advanced reservation algorithm for use in scheduling, e.g., isochronous traffic, is described with respect to FIG. **14** below.

SAR and framer **606** breaks up, sequences, and frames the data packets for wireless transmission from WAP **290***d* over the wireless medium to a wireless transceiver subscriber antenna **292**. Illustrated in block diagram **800** are a number of subscriber applications **820***a*–**820***e* running on devices such as, e.g., subscriber workstation **120***d* (not shown), connected to subscriber CPE stations **294***a*–*e* (not shown) located at subscriber CPE locations **306***a*–**306***e*. Each subscriber CPE location **306** can house one or more subscriber CPE stations **294**, and each subscriber CPE station **294** can receive and transmit one or more IP data flows to and from one or more subscriber workstations **120**. In fact, each application connected to a single CPE station can receive or transmit multiple IP data flows.

Referring to subscriber CPE location **306***a* of FIG. **8**A, a CPE SAR and framer **814***a* resequences the received data and transmits it through CPE flow scheduler **816***a*, and CPE IP flow analyzer **818***a*, to subscriber application **820***a*. CPE IP flow schedulers **816***a*–**816***e* can perform the same function as downlink flow scheduler **604** for uplink traffic. Similarly, CPE IP flow analyzers **818***a*–**818***e* perform the same function as downlink flow analyzer **602**.

In an embodiment of the invention, in downlink mode, CPE IP flow schedulers **816***a*–**816***e* and CPE IP flow analyzers **818***a*–**818***e* perform no function.

Block diagram **800** illustrates the logical functions performed on the downlink path, not necessarily the physical locations of these functions.

The functions of subscriber applications **820***a*–**820***e*, and CPE SAR and framers **814***a*–**814***e* can be performed in the actual subscriber CPE stations **294** connected over a wireless connection to wireless base station **302**.

Block diagram **800** lists an exemplary set of priorities **812** used by downlink flow scheduler **604** to place received data packets into priority class queues. Listed are the following set of example priorities: latency-sensitive UDP prority **812***a*, high priority **812***b*, intermediate priority **812***c*, initial hypertext transfer protocol (HTTP) screens priority **812***d*, latency-neutral priority **812***e*, file transfer protocol (FTP), simple mail transfer protocol (SMTP) and other e-mail traffic priority **812***f* and low priority **812***g*. Persons skilled in the art will recognize that many different priority classes are possible, depending upon the QoS requirements of the end-users. Latency-sensitive UDP priority data can refer to data that has the highest priority because it is sensitive to jitter (i.e., time synchronization is important) and latency (i.e., the amount of time passage between IP data flows in reverse directions). High priority **812***b* can refer to, e.g., premium VPN service, and a high priority SLA service. Intermediate priority **812***c* can refer to, e.g., a value VPN service level and an intermediate level SLA service. HTTP screens priority **812***d* can refer to the download of HTTP data, for example, an initial HTTP screen, which is important for making an Internet user feel as if he has a great deal of bandwidth available for his Internet session. Latency-neutral priority **812***e* can refer to data that is neutral to latency, such as, e.g., e-mail traffic. FTP, SMTP priority **812***f*

data includes data that is insensitive to latency and jitter, but requires a large amount of bandwidth to be downloaded accurately because of the size of a transmission. Finally, low priority data **812***g* can refer to data that can be transmitted over a long period of time, as when one network device transmits its status information to another network device on a 24 hour basis.

Block diagram **830** (of FIG. **8**B) depicts how IP flow analysis, prioritization and scheduling of the shared wireless bandwidth is performed in the uplink path, from subscriber CPE station **294***d*—to wireless transceiver subscriber antenna **292***d*—over the wireless medium—to WAP **290***d*—to wireless base station **302**—to interface **320**—to router **140***d*—to data network **140**.

Block diagram **830** includes uplink flow analyzer **632**, uplink flow scheduler **634** and uplink SAR and framer **636**. These components are similar in function to downlink flow analyzer **602**, downlink flow scheduler **604** and downlink SAR and framer **606**, but instead analyze, schedule and sequence and frame data packets being transmitted from subscriber workstations **120** of subscriber CPE stations **294** (at subscriber CPE locations **306***a*–**306***e*) over the wireless medium, and transmit the data packets to interface **320** for transmission to data network **142**.

Illustrated in FIG. **8**B are subscriber applications **820***a*–**820***e*, which are the same applications shown in FIG. **8**A. Also shown therein are CPE IP flow analyzers **819***a*–**819***e*, CPE IP flow schedulers **817***a*–**817***e*, and CPE SAR and framers **815***a*–**815***e*. These components function analogously to subscriber applications **820***a*–**820***e*, CPE IP flow analyzers **818***a*–**818***e*, CPE IP flow schedulers **816***a*–**816***e*, and CPE SAR and framers **814***a*–**814***e*. However, these components function to analyze, schedule and transmit IP flows in the uplink path, from subscriber CPE stations (at subscriber CPE locations **306***a*–**306***e*) to wireless base station **302** for routing to destination host workstations **136** (not shown).

As noted, multiple applications can be connected to one or more subscriber CPE stations at subscriber CPE locations **306***a*–**306***e*. To prevent collisions between multiple applications contending for a fixed number of bandwidth allocations for uplink communication, in one embodiment of the present invention a reservation scheduling system is used. The bandwidth allocations for data packets are called frame slots, and are described below with respect to FIGS. **12**A–**12**Q, **14**, **16**A and **16**B.

Block diagram **830** illustrates the logical functions performed on the uplink path, not necessarily the physical locations of these functions.

For example, in one embodiment, the analysis function of IP flow analyzer **632** which identifies a packet for uplink, characterizes and classifies the packet, can occur in a preferred embodiment in CPE IP flow analyzers **819***a*–**819***e* at the CPE subscriber stations **294***a*–**294***e* (not shown) at subscriber locations **306***a*–**306***e*.

Also, one embodiment, the functions of CPE IP flow schedulers **817***a*–**817***f* for scheduling uplinks subframe slots can be performed in wireless base station **302** for each of the subscriber CPE stations **294** connected over the wireless connection to wireless base station **302**.

In this embodiment, the scheduling function is performed at uplink flow scheduler **634** at wireless base station **302** based on classification information provided to the wireless base station **302** through an uplink IP flow reservation request from the CPE station. By placing all scheduling function at the wireless base station **302**, overall system quality of service can be optimized by centralizing the control of scheduling.

In another embodiment, however, their respective functions can be performed in the actual subscriber CPE stations.

In the reservation scheduling function of this embodiment, each subscriber CPE station requests the reservation of frame slots for its uplink transmissions using a reservation request block (RRB) of the TDMA airframe, described further below with reference to FIGS. **12**A–**12**O, before it is permitted to communicate in the uplink path with interface **320**. After the reservation request, uplink flow scheduler **634** transmits, as indicated by line **640**, to the requesting subscriber CPE station **294** a description of one or more slots which the CPE station **294** can use to transmit its uplink data packets from source subscriber workstations **120**, over the wireless medium, which are directed toward destination host workstations **136**, over data network **142**.

c. Service Level Requests

FIG. **9** illustrates how PRIMMA MAC IP flow scheduler **604** can also take into account a Service Level Agreement in prioritizing frame slot scheduling and resource allocation. FIG. **9** depicts SLA-mediated IP flow management diagram **900** including prioritization of uplink traffic being transmitted to wireless base station **302** from CPE subscriber locations **306***a*, **306***b*, **306***c* and **306***d*. For example, suppose subscribers of telecommunications services have subscribed to one of four SLA levels, P1 **902***a*, P2 **904***a*, P3 **906***a* and P4 **908***a*. In the illustrated example, suppose IP flows **902***b* are being sent to a subscriber at CPE location **306***a* and have an SLA priority level of P1 **902***a*. Similarly, IP flows **904***b*, **906***b* and **908***b* are being sent to subscribers at CPE locations **306***b*, **306***c* and **306***d* and have SLA priority levels of P2 **904***a*, **906***a* and **908***a*, respectively. PRIMMA MAC scheduler **604**, **634** of wireless base station **302** can take into account SLA-based priorities in allocating available bandwidth to the subscriber CPE IP flows **902***b*, **904***b*, **906***b* and **908***b*. In the example illustration, IP flow **902***b* can be allocated frame slot **902***c* based on SLA priority **902***a*. Frame slots **904***c*, **906***c* and **908***c* can be similarly scheduled taking into account SLA priorities. Uplinked IP flow traffic can then be transmitted on to data network **142**.

SLA-based prioritization can provide a valuable means for a telecommunications provider to provide differentiated services to a variety of customers. For example, it is possible that low priority traffic from a subscriber who has purchased a premium SLA service agreement, can be scheduled at a higher priority than high priority traffic from a subscriber which has only signed up for a value level or low cost SLA service priority.

d. Identification of Headers

FIG. **7** illustrates packet header field information **700** which can be used to identify IP flows and the QoS requirements of the IP flows. Specifically, IP header fields **702** can include, e.g., source and destination IP addresses, helpful in providing application aware preferential resource allocation; IP type of service (TOS), a useful field for assisting PRIMMA MAC in classifying a packet or IP flow; IP time to live (TTL), a useful field for anticipating application packet discards; and protocol fields which can be used in identifying IP flows.

Packet header information **700** also includes UDP header fields **704**. Included in UDP packet header fields **704** are source and destination port numbers.

Packet header information **700** also includes TCP header fields **706**. Included in TCP packet header fields **706** are source and destination port numbers; TCP sliding window size; urgent pointer; SYN, ISN, PSH, RST and FIN flags; and maximum segment size (MSS).

Packet header information **700** also includes realtime protocol RTP and RTCP header fields **708**.

It would be apparent to those skilled in the art that other packet header fields could be useful in identifying an IP flow. The fields have been given by way of example and are not intended to be an exhaustive list of useful packet header fields. Other fields, such as, e.g., fields from IP v6 relating to differentiated services (DIFF SERV) could also be useful to IP flow analyzer **602** and **632** of wireless base station **302**.

e. TDMA MAC Air Frame

FIGS. **12A–12O** illustrate an exemplary time domain multiple access (TDMA) media access control (MAC) transmission air frame. The fields described herein merely refer to one embodiment for the present invention, and are not limiting to the numerous implementations of the present invention.

FIG. **12A** illustrates an entire TDMA MAC transmission air frame. Air frame **1202** includes downstream transmission subframe **1202** and upstream transmission subframe **1204**.

The TDMA MAC air frame of FIG. **12A** includes upstream acknowledgment block (UAB) **1206**, acknowledgment request block (ARB) **1208**, frame descriptor block (FDB) **1210**, data slot $(DS)_1$ **1212***a*, DS_2 **1212***b*, DS_3 **1212***c*, DS_4 **1212***d*, DS_5 **1212***e*, DS_6 **1212***f*, DS_7 **1212***g*, DS_8 **1212***h*, DS_9 **1212***i*, DS_{10} **1212***j*, DS_{11} **1212***k*, DS_m **1212***l*, downstream acknowledgment block (DAB) **1214**, reservation request block (RRB) **1216**, UA_1 **1218***a*, UA_2 **1218***b*, UA_3 **1218***c*, UA_4 **1218U**, UA_5 **1218***e*, UA_6 **1218***f*, UA_7 **1218***g*, UA_8 **1218***h*, UA_9 **1218***i*, UA_{10} **1218***j*, UA_{11} **1218***k*, UA_{12} **1218***l*, and UA_n **1218***m*.

In the embodiment described herein, the type of TDMA used is TDMA/time division duplex (TDMA/TDD). In TDMA/TDD, for one interval of time, transmission is from a CPE station **294** to a wireless base station **302**, and in another instance of time, it is from a wireless base station **302** to a CPE station **194**. Any number of slots can be used for the uplink or for the downlink. The number of slots is dynamically assigned for both the uplink and the downlink. However, because the downlink data rate is usually higher than the uplink data rate, more slots are assigned to the downlink. Although distribution of slots between the downlink and uplink is dynamically assigned, the total number of slots for a frame is fixed in this embodiment.

TABLE 5

MAC Air Frame	Slots	Block/ SubFrame	Name	Description
0	1–8	DAB/ Upstream	Downstream Acknowledgment Request Block	Acknowledgments from subscribers CPE stations to wireless base station of receipt of downstream slots in previous downstream subframe
0	1–8	RRB/ Upstream	Reservation Request Block	Requests from subscriber CPE stations for transmission reservations in later frames with dynamically adjustable number of contentions slots
0	up to 16	US_1-US_{16}/ Upstream	Upstream Slot Transmissions	Data slots in the upstream subframe, which is a variable number per frame (up to 16 in one embodiment)

TABLE 5-continued

MAC Air Frame	Slots	Block/ SubFrame	Name	Description
0	1–3	ODB/ Upstream	Operations Data Block	OA&MP data from subscribers sequenced by a subscriber CPE station per frame
0	0	UAB/ Downstream	Upstream Acknowledgment Block	Acknowledgments from wireless base station to subscriber CPE stations of receipt of upstream slots in a previous subframe
0	0	ARB/ Downstream	Acknowledgment Request Block	Acknowledgments of subscriber CPE requests of having received reservation requests in a previous subframe
0	0	FD/ Downstream	Frame Descriptor Block for current frame	Describes the contents of the downstream transmission subframe
0	up to 16	DS_1-DS_{16}/ Downstream	Downstream Slot Transmission	Data slots in the downstream subframe, which is variable per frame (up to 16 in one embodiment)
0	0	CCB/ Downstream	Command and Control Block	OA&MP commands sequenced by subscribers per frame and frame synchronization

FIG. **12B** is a symbolic illustration of an exemplary TDMA/TDD air frame **1220** of the present invention. TDMA/TDD air frame structure **1220** depicts a frame of frame size **1228**, which can be, e.g., 16 slots or 32 slots. It would be apparent to those skilled in the art that frame structures **1220** having other numbers of slots could be used without departing from the spirit and scope of the invention. Frame structure **1220** includes, e.g., various TDMA slots **1222***a*, **1222***b*, **1222***c* and **1222***d*. Within each TDMA slot **1222***a–c*, can be included a data slot **1224***a*, **1224***b*, **1224***c* and **1224***d* which in turn can contain a control packet **1226***a*, or a data packet **1226***b–d*, respectively.

In the present embodiment the sum of all TDMA slots **1222** within a frame of frame size **1228** is fixed. However, as noted, using the resource allocation methodologies of the present invention it is possible to dynamically allocate a subset of the entire number of TDMA slots **1222** to an uplink direction, where all the uplink TDMA slots are known collectively as an uplink subframe or an upstream transmission subframe **1204**, and to dynamically allocate a subset of the entire number of TDMA slots **1222** to a downlink direction, where all the downlink TDMA slots are known collectively as a downlink subframe or an downlink transmission subframe **1202**. Using the resource allocation method of the present invention, it is possible to allocate all TDMA slots **1222** to a given upstream or downstream direction. It is further possible to allocate all data slots **1224** to a single CPE station. The wireless base station **302** has a state machine, and knows the state of each CPE station **294** having a connection therewith (i.e., having an IP flow recognized by the wireless base station **294**).

Downstream transmission subframe **1202** and upstream transmission subframe **1204** are described in detail below.

1. Downstream Transmission Subframe

FIG. **12C** depicts an exemplary downstream transmission subframe **1202**. The downstream transmission subframe of

FIG. 12C includes transmitter turnaround time **1230**, UAB **1206**, ARB **1208**, FDB **1210**, a variable number of DSs per frame (e.g., **16**) **1212**, and command and control block (CCB) **1232**. The DS transmissions **1212** include DS_1 **1212***a*, DS_2 **1212***b*, DS_3 **1212***c*, DS_4 **1212***d*, DS_5 **1212***e*, DS_6 **1212***f*, $DS_{7\ 1212}g$, DS_8 **1212***h*, DS_9 **1212***i*, DS_{10} **1212***j*, DS_{11} **1212***k*, and DS_m **1212***l*.

FIG. **12D** depicts an exemplary UAB **1206** of a downstream transmission subframe **1202**. The downstream transmission subframe of FIG. **12D** includes UAB **1206**, ARB **1208**, FDB **1210**, DS_1 **1212***a*, DS_2 **1212***b*, DS_3 **1212***c*, DS_4 **1212***d*, DS_5 **1212***e*, DS_6 **1212***f*, DS_7 **1212***g*, DS_n **1212***h*, DS_9 **1212***i*, DS_{10} **1212***j*, DS_{11} **1212***k*, DS_m **1212***l*, and CCB **1232**.

UAB **1206** includes subslots UAB_1 **1206***a*, UAB_2 **1206***b*, UAB_3 **1206***c*, UAB_4 **1206***d*, UAB_5 **1206***e*, UAB_6 **1206***f*, UAB_7 **1206***g*, and UAB_n **1206***h*. UAB_1 **1206***a* includes a preamble **1234***a*, subscriber ID **1234***b*, IP-flow identifier **1234***c*, slot sequence number **1234***d*, and cyclical redundancy check (CRC) **1234***e*.

The UAB field is an acknowledgment by a wireless base station **302** to a CPE station **294** that the slots (e.g., US_1–US_{16}) of an upstream transmission subframe have been received. The reader is referred to the discussion of the upstream transmission subframe below.

In subslot UAB_1 **1206***a* of ARB **1206**: preamble **1234***a* includes data used for link integrity purposes; subscriber ID **1234***b* identifies which CPE station **294** is making the reservation request; IP-flow identifier **1234***c* identifies the IP data flow; quality of service data class **1234***a* identifies the priority class of the IP data flow, if known to the CPE station **294**; IP-flow priority and type **1234***b* is an indicator of a new IP data flow; and CRC **1234***e*, which stands for cyclic redundancy code, provides error checking bits for subslot RRB_1 **1216***a*.

FIG. **12E** depicts an exemplary ARB **1208** of a downstream transmission subframe **1202**. The downstream transmission subframe of FIG. **12E** includes UAB **1206**, ARB **1208**, FDB **1210**, DS_1 **1212***a*, DS_2 **1212***b*, DS_3 **1212***c*, DS_4 **1212***d*, DS_5 **1212***e*, DS_6 **1212***f*, DS_7 **1212***g*, DS_n **1212***h*, DS_9 **1212***i*, DS_{10} **1212***j*, DS_{11} **1212***k*, DS_m **1212***l*, and CCB **1232**.

ARB **1208** includes subslots ARB_1 **1208***a*, ARB_2 **1208***b*, ARB_3 **1208***c*, ARB_4 **1208***d*, ARB_5 **1208***e*, ARB_6 **1208***f*, ARB_7 **1208***g*, and ARB_n **1208***h*. ARB_1 **1208***a* includes a preamble **1234***a*, subscriber ID **1234***b*, IP-flow identifier **1234***c*, slot sequence number **1234***d*, and CRC **1234***e*.

The ARB field is an acknowledgment by a wireless base station **302** to a CPE station **294** that the wireless base station **302** has received an upstream reservation request from the CPE station **294**. The reader is referred to the discussion of the upstream transmission subframe below.

In subslot ARB_1 **1208***a* of ARB **1208**: preamble **1234***a* includes data used for link integrity purposes; subscriber ID **1234***b* identifies which CPE station **294** is making the reservation request; IP-flow identifier **1234***c* identifies the IP data flow;quality of service data class **1234***a* identifies the priority class of the IP data flow, if known to the CPE station **294**; IP-flow priority and type **1234***b* is an indicator of a new IP data flow; and CRC **1234***e*, which stands for cyclic redundancy code, provides error checking bits for subslot RRB_1 **1216***a*.

FIG. **12F** depicts an exemplary FDB **1210** of a downstream transmission subframe **1202**. The downstream transmission subframe of FIG. **12F** includes UAB **1206**, ARB **1208**, FDB **1210**, DS_1 **1212***a*, DS_2 **1212***b*, DS_3 **1212***c*, DS_4 **1212***d*, DS_5 **1212***e*, DS_6 **1212***f*, DS_7 **1212***g*, DS_n **1212***h*, DS_9 **1212***i*, DS_{10} **1212***j*, DS_{11} **1212***k*, DS_m **1212***l*, and CCB **1232**.

The FDB includes detailed information pertaining to the slots (e.g., DS_2–DS_{16}) of the downstream transmission subframe.

FDB **1210** includes a preamble subslot **1236***a*, number of downstream slots subslot, **1236***b*, IP-flow ID for upstream reservation 1 subslot **1236***c*, IP-flow ID for upstream reservation 2 subslot **1236***d*, IP-flow ID for upstream reservation n subslot **1236***e*, and contention slot count for next upstream subframe subslot **1236***f*.

In FDB **1210**, the fields are defined as follows: preamble subslot **1236***a* includes data used for link integrity purposes; number of downstream slots subslot **1236***b* includes the number of downstream slots (DSs), IP-flow ID for downstream reservation subslot **1236***c* includes an IP flow identification for DS_1; IP-flow ID for downstream reservation subslot **1236***d* includes a second IP flow identification for DS_2; IP-flow ID for downstream reservation n subslot **1236***e* includes another IP flow identification for DS_m; contention slot count for next upstream subframe subslot **1236***f* provides a count for the next available upstream subframe.

FIG. **12G** depicts an exemplary downstream MAC payload data unit (PDU). The downstream MAC PDU includes information regarding the actual structure of the payload. The downstream MAC PDU of FIG. **12G** includes MAC linked list sequence number **1238***a* (the sequence number of the MAC linked list), reservation request index number **1238***b* (an index to the downstream IP flow), compressed IP-flow identifier **1238***c*, compressed IP-flow priority and type **1238***d* (identifying the priority and type of a compressed IP flow), slot payload **1238***e* (the amount of data in a downstream data slot), and CRC **1234***e* (error checking information).

FIG. **12H** depicts an exemplary CCB of a downstream transmission subframe **1202**. The CCB comprises OAM&P commands sequenced by subscriber CPE station **294** per frame and frame synchronization. CCB **1232** includes a mode command subslot **1240***a* (includes options of what mode the CPE station is to take), profile command subslot **1240***b* (includes specific system commands, such as a patch for a module), control data index subslot **1240***c* (including download locations and memory requirements or other information needed by the CPE stations to download data), datablock 1 subslot **1240***d* (includes specific system data), datablock 2 subslot **1240***e* (same), datablock n subslot **1240***f* (same), and CRC subslot **1234***e* (error checking information).

2. Upstream Transmission Subframe

FIG. **12I** depicts an exemplary upstream transmission subframe **1204**. The upstream transmission subframe of FIG. **12I** includes transmitter turnaround time **1230**, DAB **1214**, RRB **1216**, a variable number of USs per frame, e.g., 16, **1218**, and operations data block (ODB) **1242**, consisting of OAM&P data from subscribers, sequenced by subscriber per frame. The US transmissions **1218** include US_1 **1218***a*, US_2 **1218***b*, US_3 **1218***c*, US_4 **1218***d*, US_5 **1218***e*, US_6 **1218***f*, US_7 **1218***g*, US_8 **1218***h*, US_9 **1218***i*, US_{10} **1218***j*, US_{11} **1218***k*, $US_{12\ 1218}l$, and US_n **1218***m*.

FIG. **12K** depicts an exemplary RRB **1216** of an upstream transmission subframe **1204**. The upstream transmission subframe of FIG. **12K** also shows DAB **1214**, RRB **1216**, US_1 **1218***a*, US_2 **1218***b*, US_3 **1218***c*, US_4 **1218***d*, US_5 **1218***e*, US_6 **1218***f*, US_7 **1218***g*, US_8 **1218***h*, US_9 **1218***i*, US_{10} **1218***j*, US_{11} **1218***k*, US_{12} **1218***l*, US_n **1218***m*, and ODB **1242**.

RRB **1216** includes subslots RRB_1 **1216***a*, RRB_2 **1216***b*, RRB_3 **1216***c*, RRB_4 **1216***d*, RRB_5 **1216***e*, RRB_6 **1216***f*, RRB_7 **1216***g*, and RRB_n **1216***h*. RRB_1 **1216***a* includes a preamble **1234***a*, subscriber ID **1234***b*, IP-flow identifier **1234***c*, quality of service data class **1244***a*, IP-flow priority and type **1244***b*, and CRC **1234***e*.

A CPE station **294** uses one of the subslots (RRB_1 **1216***a*, RRB_2 **1216***b*, RRB_3 **1216***c*, RRB_4 **1216***d*, RRB_5 **1216***e*,

RRB_6 **1216***f*, RRB_7 **1216***g*, and RRB_n **1216***h*) of RRB **1216** to make a reservation request, which is a request by the CPE station **294** for bandwidth in a future uplink transmission subframe. If two CPE stations **294***d*, **294***e* attempt to access the same subslot in RRB **1216**, which can occur because their pseudorandom number generators select the same subslot, then a "collision" occurs and the data is not readable by wireless base station **302**. The two CPE stations **294***d*, **294***e* are required to try again.

Reservation request slots can be provided on an IP flow basis. Rather than allocate a reservation request slot to every CPE subscriber station, a default number (e.g., 5) are made available as contention slots. If collisions are detected by a greater number of requesting subscribers than the number of reservation request slots, then the slots allocated can be dynamically varied to provide additional RRB slots. (Collisions are analogous to CSMA/CD collisions in Ethernet, where colliding devices on an Ethernet network attempt to retransmit over the bus architecture by retrying at a random time.)

The radio contention method of the present invention builds upon aspects of the "Slotted Aloha" method developed by L. Roberts in 1972, as a refinement of the "Aloha" method developed by N. Abramson in the early 1970's, and so-called bit-mapped reservation protocols. Like the Slotted Aloha method, the present invention provides for discrete slots for transmission of data, rather than allowing the transmission of data at any point. However, instead of transmitting the actual "payload" of data, the present invention advantageously transmits only a "reservation request" describing the actual data payload contents. Also, the number of slots for reservation requests can advantageously be dynamically altered according to the frequency of detected collisions in the recent past.

Unlike various Carrier Sense Multiple Access (CSMA) techniques previously used in wireless, both persistent and non-persistent, the present method advantageously does not require that subscriber CPE station **294***d* "sense" the carrier (the radio channel) before transmission. Instead, a subscriber CPE station **294***d* selects a "subslot" to transmit through a pseudo-random number selection, without a prior carrier sense. If a collision is detected, the subscriber CPE station **294***d* will try again in the next frame using the pseudo-random number process.

Instead of using a bit-map protocol for the resolution of contention, as is used in some reservation protocols, the wireless base station can explicitly grant reservation requests. The standard bit-map protocol can require that all stations can receive signals from all other stations so that the subsequent order of transmission can be implicitly determined from the resulting bit-map pattern. The present method advantageously does not require the receipt of reservation request signals from other CPE subscriber stations **294***d*. This is advantageous because, at higher frequencies (such as, e.g., 2 GHz to 30 GHz) where there may be line-of-sight and distance constraints, the requirement for receipt of the transmissions of other CPE subscriber stations **294***d* could unduly constrain the topology, locations and distances of CPE subscriber stations.

Advantageously, by allowing the wireless base station **302** to explicitly grant the requested reservation, other factors such as relative or dynamic CPE subscriber station **294***d* (or IP-flow) priority factors can be considered. Therefore, the present invention's reservation protocol with a dynamically adjustable number of contention subslots and explicit wireless base station reservation grants, allows a more optimal means of providing for the allocation of wireless, such as, e.g., radio, bandwidth in response to QoS requirements of IP-flows than any prior method.

As noted, RRB_1 **1216***a* includes the following fields: a preamble **1234***a*, subscriber ID **1234***b*, IP-flow identifier **1234***c*, quality of service data class **1244***a*, IP-flow priority and type **1244***b*, and CRC **1234***e*. In subslot RRB_1 **1216***a* of RRB **1216**: preamble **1234***a* includes data used for link integrity purposes; subscriber ID **1234***b* identifies which CPE station **294** is making the reservation request; IP-flow identifier **1234***c* identifies the IP data flow;quality of service data class **1234***a* identifies the priority class of the IP data flow, if known to the CPE station **294**; IP-flow priority and type **1234***b* is an indicator of a new IP data flow; and CRC **1234***e*, which stands for cyclic redundancy code, provides error checking bits for subslot RRB_1 **1216***a*. Optionally, an additional field can be provided in subslot RRB_1 **1216***a* which includes the number of data packets CPE station **294** will transmit in its IP data flow.

FIG. **12***J* depicts an exemplary DAB **1214** of an upstream transmission subframe **1204**, where a CPE acknowledges receipt of a slot from base. The DAB is an acknowledgment from a subscriber CPE station **294** to the wireless base station that downstream slots have been received in a previous subframe.

The DAB **1214** includes subslots DAB_1 **1214***a*, DAB_2 **1214***b*, DAB_3 **1214***c*, DAB_4 **1214***d*, DAB_5 **1214***e*, DAB_6 **1214***f*, DAB_7 **1214***g*, and DAB_n **1214***h*. Subslot DAB_1 **1214***a* includes a preamble **1234***a*, subscriber ID **1234***b*, IP-flow identifier **1234***c*, slot sequence number **1234***d*, and CRC **1234***e*. (These fields have the same information as described with respect to the RRB.)

FIG. **12***L* depicts an exemplary MAC PDU upstream slot. The MAC PDU upstream slot of FIG. **12***L* includes a CPE linked-list sequence number **1246**, reservation request index number **1236***b*, compressed IP-flow identifier **1238***c*, compressed IP-flow priority and type **1238***d*, slot payload **1238***e*, and CRC **1234***e*. The upstream MAC PDU is similar to the downstream MAC PDU, but is used instead for upstream subframe payload information.

FIGS. **12M**, **12N** and **12O** depict an exemplary ODB **1242** in detail. This field is used to store information regarding the connection between the wireless base station **302** and the CPE station **294**. ODB **1242** includes preamble **1234***a* (including link integrity data), subscriber ID **1234***b* (identifies which CPE station **294** is making the reservation request), system state **1248***a* (information about the status of the CPE station **294**), performance data **1248***b* (how full the buffer statistics, cpe processor performance statistics, system state), antenna data **1248***c* (information pertaining to the antenna), CRC **1234***e* (error checking information) and synchronization pattern **1248***d* (error checking information).

Referring to FIG. **12M**, system state subslot **1248***a* comprises system mode **1250***a* (the mode of the CPE station, e.g., command mode, operations mode, or initialization mode of the system), system status **1250***b* (the status of the CPE station), system resources **1250***a* (the mode of the CPE station), system power **1250***b* (the mode of the CPE station), system temperature **1250***a* (the temperature of the CPE station). The CPE stations **294** are required to take turns using ODB **1242** to transmit their information.

Referring to FIG. **12N**, performance data **1248***a* comprises the number of comrepeats **1252***a* (the number of repeats of communication attempts), number of frameslips **1252***b* (the number of frames that have slipped), waitstate index **1252***c* (an index to the waiting state).

f. Exemplary Class-based Frame Prioritization

FIG. **13** shows block diagram **1300**, illustrating how an exemplary flow scheduler for the present invention functions

to schedule products. Block diagram **1300** includes: flow scheduler **604, 634** (which is a combination of downlink flow scheduler **604** and uplink flow scheduler **634**), downlink transmission subframe **1202** (i.e., the next MAC downstream subframe), uplink transmission subframe **1204** (i.e., the current MAC upstream subframe). Block diagram **1300** also includes the following downstream components: downstream reservation first-in-first-out queue **1322**, class 1 downstream queue **1302**, class 2 downstream queue **1304**, and class 3 downstream queue **1306**. Block diagram **1300** also includes the following upstream reservation components: current upstream subframe **1344** (with the current upstream subframe **1204** about to be stored in it), previous upstream subframes **1346, 1348, 1350**, class 1 upstream reservation request queue **1308**, class 2 upstream reservation request queue **1310**, and class 3 upstream reservation request queue **1312**.

In the downlink path, an IP flow QoS class queuing processor (described below with respect to FIGS. **15A** and **15B**) queues the received data packets into class 1 packet flow queues **1324, 1326** and **1328**, class 2 packet flow queues **1330, 1332, 1334**, and class 3 packet flow queues **1336, 1338, 1340** and **1342**.

Based on inputs from a hierarchical class-based priority processor, a virtual private network (VPN) directory enabled (DEN) data table and a service level agreement (SLA) priority data table (described below with respect to FIGS. **15A** and **15B**), the class 1, class 2, and class 3 packet flow queues are respectively assigned to class 1 downstream queue **1302**, class 2 downstream queue **1304**, and class 3 downstream queue **1306**. Flow scheduler **604, 634** schedules these downlink data packets onto the downlink transmission subframe **1202**.

In one embodiment, additional processing is used to minimize latency and jitter. For example, suppose the data packets of class 1 packet flow queue **1324** require jitter-free and latency-free delivery, i.e., delivery of packets must be at constant time intervals and in real-time. Packet flow queue **1324** creates, e.g., 4 equal time spaced slot reservations in future frames, as shown in class 1 downstream queue **1302** and described with respect to FIG. **14** below. The reservations are fed to downstream reservation first-in-first-out queue **1322**, and are scheduled onto a future downstream frame **1202** by flow scheduler **604, 634**.

In the uplink path, reservation requests for future upstream slots arrive at wireless base station **302** as part of the current upstream subframe **1204** received from CPE subscriber stations **294** over the wireless medium. Current upstream subframe **1344** can temporarily store reservation requests for analysis and scheduling of uplink packets in accord with the description of FIG. **8B** above. Previous upstream subframes **1346, 1348, 1350** include upstream reservation requests awaiting upstream frame slot allocations in future upstream subframes **1204**. Reservation request blocks (RRBs), described further above with reference to FIG. **12** * * *, include a request for a number of slots for a single IP flow with an IP flow identifier # and class of flow. The upstream reservation requests (by IP flow and class) are queued onto class 1 upstream reservation request queue **1308**, class 2 upstream reservation request queue **1310**, and class 3 upstream reservation request queue **1312** in an IP flow QoS class queuing processor (described below with respect to FIGS. **16A** and **16B**). Flow scheduler **604** and **606**, and **634** and **1666**, uses these downstream reservations and upstream reservation requests to assign slots to data packets in the next downstream transmission subframe **1202** and upstream transmission subframe **1204**, respectively.

FIG. **14** is an exemplary two-dimensional block diagram **1400** of the advanced reservation algorithm. FIG. **14** includes MAC subframe scheduler **1566, 1666**, frames current frame, n **1402**, and future frames, n+1 1404, n+2 **1406**, n+3 **1408**, n+4 **1410**, n+5 **1412**, n+6 **1414** . . . n+x **1416**, representing frames of data packets to be transmitted at times n, n+1, n+2 . . . n+x. Each frame is divided into a variable length downlink subframe **1202** and a variable length uplink subframe **1204**. The lengths of downlink subframe **1202** and uplink subframe **1204** together comprise the length of an entire frame.

Each frame n **1402** includes a number of slots (**1418–1478**). Slots **1418–1446** comprise the downlink subframe **1202**, and slots **1448–1478** comprise the uplink subframe **1204**. In one embodiment, the slots are fixed in length, with each slot capable of storing a single data packet. The total number of frame slots in a frame remains constant. For example, if a given frame includes 64 frame slots, the slots can be allocated dynamically in either the uplink or downlink directions, such as, e.g., 32 up and 32 down, 64 up and 0 down, 0 up and 64 down. Block diagram **1400** can be thought of as a two dimensional matrix with each slot having a time value (i.e., a slot-to-slot time interval), e.g., 0.01 ms, and each frame having a total frame interval time value (i.e., a frame-to-frame time interval), e.g., 0.5 ms.

In the present invention, an advanced reservation algorithm assigns future slots to data packets based on the priority of the IP data flow with which the packet is associated. Exemplary priorities are described above with respect to FIGS. **8A** and **8B**. For calls that are sensitive to jitter, meaning calls that are time sensitive, it is important to maintain an isochronous (i.e., in phase with respect to time) connection. With such signals, it is important that the data be dispersed in the same slot between frames, or in slots having a periodic variation between frames. For example, vertical reservation **1480** shows a jitter sensitive signal receiving the same slot for downlink communications in each frame. Specifically, the signal is assigned slot **1422** in frames **1402–1416**. If the frame-to-frame interval is 0.5 ms, then a slot will be provided to the IP flow every 0.5 ms. As another example, diagonal reservation **1482** shows a jitter sensitive signal receiving a slot varying by a period of one between sequential frames. Specifically, the signal is assigned slot **1440** in frame **1402**, slot **1438** in slot **1404**, . . . slot **1426** in frame **1416**, to create a "diagonal." If the frame-to-frame interval is 0.5 ms and the slot-to-slot interval is 0.01 ms, then a slot can be provided to the IP flow every 0.5 minus 0.01, equals 0.49 mms. Thus, to decrease the frame interval, a diagonal reservation of positive slope can be used. To obtain an increased frame interval, a diagonal of negative slope such as, e.g., negative slope diagonal uplink reservation **1486**. The diagonal reservation **1482** can also be more pronounced (i.e., using a greater or lesser slope), depending on the period between sequential frames desired. Reservation patterns **1480, 1482, 1484** and **1486** are useful patterns for jitter sensitive communications. Also illustrated is a vertical reservation **1486**, similar to vertical reservation **1480**, useful for a jitter sensitive communication in the uplink direction.

For latency sensitivity, one or more slots can be guaranteed in each frame. For example, for a call that is latency sensitive, but not jitter sensitive, each frame can be assigned one (or more) slots for communications. However, the slot(s) need not be periodic between frames, as with jitter sensitive calls. The greater the number of slots allocated per frame to an IP flow, the greater total bandwidth per frame rate for the IP flow.

For calls that are less latency sensitive, fewer slots per frame can be assigned for the communication. For example, a communication that is less latency sensitive can receive a guaranteed bandwidth of one slot every four frames. A call that is even less latency sensitive can receive, e.g., a single slot every ten frames.

Using these principles, the advanced reservation algorithm can assign the slots from highest priority to lowest priority, exhausting the number of available slots in future frames. IP data flows that are both jitter and latency sensitive can be assigned slots with periodic patterns first (e.g., patterns **1480, 1482, 1484** and **1486**), followed by flows that are highly latency sensitive (but not jitter sensitive), et cetera, until the flows of lowest latency sensitivity are assigned to slots. Prioritization of different classes of IP flows by scheduler **604, 634, 1566, 1666** is described further below with reference to FIGS. **15A, 15B, 16A** and **16B**.

g. Downlink SubFrame Prioritization

1. Overview

FIGS. **15A** and **15B** are exemplary logical flow diagrams for analysis and scheduling of the shared wireless bandwidth for the downlink direction. The logical flow pertains to IP packet flows arriving from data network **140**, at the wireless base station **302**, for transmission down to a subscriber CPE station **294***d* over the wireless medium. FIG. **15A** is an exemplary logical flow diagram **1500** for downlink IP analyzer **602**. FIG. **15B** is an exemplary logical flow diagram **1560** for the downlink flow scheduler **604**.

The functional components for FIGS. **15A** and **15B** are explained by way of method modules, which can be viewed as physical units (e.g., comprising software, hardware, or a combination thereof) or logical vehicles (e.g., used for explanatory purposes only). Those skilled in the art will recognize that the modules are used only to explain an exemplary embodiment, and are not to be considered limiting.

The exemplary logical flow diagram **1500** for downlink IP flow analyzer of FIG. **15A** includes packet header identification component **1502**, packet characterization component **1504**, packet classification component **1506**, and IP flow presentation component **1508**. The functions of these components are explained in detail below.

In one embodiment, downlink IP flow analyzer **602** is physically located in wireless base station **302**, although those skilled in the art will recognize that the same functionality can be located remotely from wireless base station **302**.

FIGS. **2D, 3A** and **3B** are helpful to the reader for an understanding of the downlink IP flow analyzer.

2. Introduction

IP flow analyzer **602** performs the function of identifying, characterizing, classifying, and presenting data packets to a downlink frame scheduler **604**. The functions of identifying, characterizing, classifying and presenting the data packets are respectively performed by packet header identification component **1502**, packet characterization component **1504**, packet classification component **1506** and IP flow presentation component **1508** of downlink IP flow analyzer **602**.

Packet header identification component **1502** determines whether a data packet of an incoming IP data flow is part of an IP flow that is known to the system, or is the first data packet of a new IP data flow, based on the contents of fields of the packet header section. Packet header identification component **1502** also identifies, e.g., the source of the packet using the packet header field contents. Packet characterization component **1504** characterizes a new data packet (of a new IP data flow) to determine the QoS requirements for the IP data flow, and identifies the subscriber CPE station associated with the subscriber workstation that will receive the IP data flow. Packet classification component **1506** classifies the new IP data flow into a communications priority class, grouping the packet together with similar type IP flows. IP data flow presentation **1508** initializes the new IP data flow and presents it to downlink flow scheduler **604**.

Downlink flow scheduler **604** places the data packets of an IP data flow into a class queue, and based on a set of rules, schedules the data packets for transmission over the wireless medium to a subscriber CPE station using, e.g., an advanced reservation algorithm. The rules can be determined by inputs to the downlink flow scheduler from a hierarchical class-based priority processor module **1574**, a virtual private network (VPN) directory enabled (DEN) data table **1572**, and a service level agreement (SLA) priority data table **1570**. The advanced reservation algorithm is described further above with respect to FIG. **14**.

3. Identification

Packet header identification component **1502** identifies the IP flow received from data network **142** at data interface **320** based on the packet header.

An IP flow packet stream from data network **142**, including packets from various IP flows (where each IP flow is associated with a single data "call") is received at packet header identification component **1502**. An IP flow can include packetized data including any type of digital information such as, e.g., packetized voice, video, audio, data, IP flows, VPN flows, and real time flows. The IP flow is transmitted over data network **142** from, e.g., a host workstation **136***d* and arrives at interface **302** of wireless base station **320**. Interface **302** transmits the packets of the IP flow to packet header identification component **1502**. At module **1510**, the received packets are buffered into a storage area. At module **1520**, the contents of the packet header fields are extracted and parsed.

For IP flows known to the system, so-called "existing IP flows," there are entries in a table **1526**. An IP flow is in the system if there is an existing characterized IP data call. In module **1522**, it is determined if there is a match between the incoming packet and an existing IP flow call in an entry in existing IP flow identification table **1526**. If so, then the IP flow is known to the system, and control passes to module **1530** of the packet characterization component **1504**.

If not, meaning that the IP flow is a new IP data flow, then control passes to module **1524**, where the packet header fields are analyzed. Module **1524** analyzes the packet header source field and determines from source application packet header data table **1528** the type of source application making the data call or transmitting the IP packet. The application can be any of the applications described with respect to FIG. **2D** or known to those skilled in the art. Examples include a file transfer protocol (FTP) download from another client workstation **138***f*, an IP voice telephony call (over telephony gateway **288***b*), a voice telephony call from a caller **124***d* (connected over a modem), an e-mail from a LAN **128***a* attacbed host workstation **136***a*, a fax machine call, and a conference call from multiple callers **124***d* and **126***d* (connected over a modem), to name a few. If the IP flow is not known to the system, then the IP flow is given an IP flow identifier number, and control passes to module **1526** where the IP flow identifier number is added to the existing IP flow identification table **1526**.

Once the type source application has been determined by packet header information or by another means, such as direct application identification, then control passes from module **1524** to module **1532** of the packet characterization

component **1504**. In order to identify the type of source application of the IP flow, any type of service (TOS) or differentiated service (DiffServ) field can also be analyzed.

4. Characterization

Packet characterization component **1504** characterizes new IP flows and passes them to packet classification component **1506** for classification.

For an existing IP flow, control passes to module **1530** from module **1522** of the packet header identification component **1502**. If in module **1522** it is determined that the IP data flow is known to the system, in module **1530** it is determined whether the packet is old (i.e., stale). This can include, e.g., determining from a time-to-live field (a field in the IP packet header) the age of the packet, and comparing the field to a threshold age value. If the packet is determined to be stale, it can be discarded. Based on the age of the packet, client application discards can be anticipated. Otherwise, control can pass to module **1540** of the packet classification component **1506**.

For a new IP flow, control passes to module **1532** from module **1524** of the packet header identification component **1502**. If in module **1524** it is determined that the IP flow is not known to the system, in module **1532** the QoS requirements for the application are determined using the source application information identified in modules **1524** and **1528**. Module **1532** performs this operation by looking up the QoS requirements for the identified source application in the QoS requirement table **1534**. Different applications have different QoS requirements in order to provide an acceptable end-user experience. For example, bandwidth allocation (i.e., allocating an appropriate amount of bandwidth) is important to an application performing FTP file transfer downloads, and not jitter (i.e., time synchronizing the received data) and latency (i.e., the amount of time passage between responses). On the other hand, jitter and latency are important to voice telephony and conference calls, while bandwidth allocation is not.

After processing by module **1532**, in module **1536** a destination CPE subscriber station ID lookup from subscriber CPE IP address table **1538**, is performed for the IP flow. Each subscriber CPE station **294***d* can have one or more applications, running on one or more subscriber workstations **120***d*, homed to it. Accordingly, the IP flows can be directed to one or more applications on one or more subscriber workstations of one or more CPE stations **294***d*. A subscriber workstation can be any device coupled to a subscriber CPE station **294***d*. Module **1536** looks up the IP flow in table **1538**, to determine the identity of the subscriber CPE station **294***d* that will receive the packets of the new IP flow from data network **142**. Control then passes from module **1536** to module **1542** of the packet classification component **1506**.

5. Classification

Packet classification component **1506** classifies the IP flow and passes it to IP flow presentation component **1508** for presentment.

For an existing IP flow, control passes to module **1540** from module **1530** of the packet characterization component **1504**. If in module **1530** it is determined that the packet is not stale, then in module **1540** the packet is associated with its existing IP flow. As illustrated in FIG. **15A**, the packet processed herein was determined to be a portion of an IP flow known to the system. Therefore, the QoS processing of modules **1532**, **1536** and **1542** are unnecessary, because the QoS requirements of the present packet are assumed to be the same as for its IP flow. In another embodiment, all packets are characterized and classified. From module **1540**, control can continue with module **1546** of IP flow presentation **1508**.

For the new IP flow, control passes to module **1542** from module **1536** of the packet characterization component **1504**. In module **1542** the packet is classified into a QoS class by performing a table lookup into IP flow QoS class table module **1544**, where the types of QoS classes are stored depending on the QoS requirements for packets. Similar IP flows, (i.e., IP flows having similar QoS requirements) can be grouped together in module **1542**. In classifying packets and IP flows, QoS class groupings, any DiffServ priority markings, and any TOS priority markings can be taken into account. From the module **1542**, control passes to module **1548** of IP flow presentation component **1508**.

6. IP Flow Presentation

IP flow presentation component **1508** prepares and presents the IP flow packets to downlink flow scheduler **604**.

For existing IP flows, control passes to module **1546** from module **1540** of the packet classification component **1540**. In module **1546** the packet is added to the associated existing IP flow queue, which is the queue for the current IP flow. From module **1546**, control passes to IP flow QoS class queuing processor module **1562** of downlink flow scheduler **604**.

For the new IP flow, control passes to module **1548** from module **1542** of the packet classification component **1506**. In module **1548**, this new IP flow can be initialized for presentation to module **1552**. In module **1550**, the IP flow QoS class is presented to frame scheduler **604** to be placed in an appropriate class queue. Module **1552** presents the IP flow (in particular, the data packet) and IP flow identifier to IP flow QoS class queuing processor module **1562** of downlink flow scheduler **604**.

7. Downlink Flow Scheduler

The exemplary logical flow diagram **1560** for the downlink flow scheduler **604** of FIG. **15B** comprises IP flow QoS class queuing processor module **1562**, MAC downlink subframe scheduler module **1566**, hierarchical class-based priority processor module **1574**, VPN DEN data table module **1572**, SLA priority data table **1570**, CPE IP flow queue depth status processor **1582** and link layer acknowledgment processor module **1578**.

Downlink flow scheduler **604** of FIG. **15B** also includes QoS class queues as follows: class 1, **1564***a*; class 2, **1564***b*; class 3, **1564***c*; class 4, **1564***d*; class 5, **1564***e*; and class 6, **1564***f*; and MAC downlink subframes: frame n, **1568***a*; frame n+1, **1568***b*; frame n+2, **1568***c*; frame n+3, **1568***d*; . . . frame n+p, **1568***k*.

In one embodiment, downlink flow scheduler **604** is physically located in wireless base station **302**, although those skilled in the art will recognize that the same functionality can be located remotely from wireless base station **302**.

Downlink flow scheduler **604** is used to schedule the downlink subframe. An entire frame can be divided into an uplink portion (called an uplink subframe) for transmitting uplink frames, and a downlink portion (called a downlink subframe) for transmitting downlink frames.

Also illustrated on FIG. **15B** are WAP antenna, the wireless medium, **290***d*, RF transceiver subscriber antenna **292***d*, subscriber CPE station **294***d* and subscriber workstation **120***d*. WAP antenna **290***d* and RF transceiver subscriber antenna **292***d* respectively provide a wireless connection between wireless base station **302** (where downlink flow scheduler **604** resides in one embodiment) and subscriber CPE station **294***d*, which can transmit an IP flow to an application running on subscriber workstation **120***d*. WAP antenna **290***d* serves as a wireless gateway for data network

142, and RF transceiver subscriber antenna serves as a wireless gateway for subscriber CPE station **294***d*. The connection is also illustrated in FIGS. **2**D and **3**B.

IP flow QoS class queuing processor module **1562** receives the packets from IP flow presentation component **1508**. Module **1562** then creates class queues **1564***a*–**1564***f*, which is a variable number of queues, and places the packets in these class queues. How packets are placed in class queues **1564***a*–**1564***f* is determined by the inputs to module **1562**.

Module **1562** can receive inputs from hierarchical class-based priority processor module **1574**, VPN DEN data table **1572** and service level agreement (SLA) priority data table **1570**. The queuing function of module **1562** can be based on these inputs.

SLA priority data table **1570** can use predetermined service level agreements for particular customers to affect the queuing function. A customer can be provided a higher quality of telecommunications service by, for example, paying additional money to receive such premium service. An algorithm running on module **1562** can increase the queuing priority for messages transmitted to such customers.

Virtual private network (VPN) directory enabled networking (DEN) data table **1572** can provide prioritization for a predetermined quality of service for a VPN for a company that pays for the VPN function. A VPN is understood by those skilled in the relevant art to be a private network, including a guaranteed allocation of bandwidth on the network, provided by the telecommunications service provider. VPN DEN data table **1572** permits module **1562** to provide higher quality of service for customer-purchased VPNs. As with SLA priority data table **1570**, the queuing priority can be increased for such VPNs. For example, a platinum level VPN's lowest priority IP flow classes could also be given a higher priority than a high priority brass level VPN.

Both SLA priority data table **1570** and VPN DEN data table **1572** receive input from operations, administration, maintenance and provisioning (OAM&P) module **1108**. This is a module that is kept off-line, and includes storage and revision of administrative information regarding new customers, or updates of information pertaining to existing customers. For example, the SLA priority of the customers and VPN information is updated from OAM&P module **1108**.

Hierarchical class-based priority processor module **1574** is a module that operates under the principles of hierarchical class-based queuing. Hierarchical class-based queuing was created by Sally Floyd and Van Jacobson, considered early architects of the Internet.

Hierarchical class-based queuing classifies different types of IP flows using a tree structure at the edge access device routers. Each branch of the tree signifies a different class of IP flows, and each class is dedicated a set limited amount of bandwidth. In this manner, different classes of flows are guaranteed minimum bandwidth, so that no single IP data flow within a class, and no single class of IP flows, can use up all available bandwidth. The present invention adds a prioritization feature enabling class based priority reservations to be made using the hierarchical class queue concept, as discussed above with respect to FIGS. **13** and **14**.

MAC downlink subframe scheduler **1566** is a processor module that takes the packets queued in class queues **1564***a*–**1564***f*, and can make frame slot reservations to fill up subframes **1568***a*–**1568***k* based on priorities **1570**, **1572** and **1574**, which is a variable number of frames. In one embodiment, each subframe is scheduled (filled) with up to a predetermined number of packets from each of the classes **1564***a*–**1564***f* according to priorities **1570**, **1572** and **1574**. In another embodiment, the subframes are scheduled according to the inventive advanced reservation algorithm method described with respect to FIGS. **13** and **14** for isochronous reservations. In yet another embodiment, the subframes are scheduled according to a combination of known methods and the advanced reservation algorithm method of the present invention.

The subframes can then be sent to WAP antenna **290***d* for wireless transmission over the wireless medium to RF transceiver subscriber antenna **292***d* coupled to subscriber CPE station **294***d*, which in turn can send the packets contained in the subframes to subscriber workstation **120***d* at CPE subscriber location **306***d*. The subframes can be scheduled from highest priority to lowest priority.

Hierarchical class-based priority (HCBP) processor module **1574** receives as input the subframes that have been scheduled and transmitted from WAP antenna **290***d*. By maintaining awareness of the status of the packets (i.e., by knowing which packets have been sent out), HCBP processor module **1574** knows which packets from which class queues **1564***a*–**1564***f* must yet be scheduled.

Every once in a while, a packet is lost through, e.g., noise. When this situation arises, the subscriber CPE station **294***d* sends a retransmit request **1576** to WAP **290***d*, which transmits the request to link layer acknowledgment (ARQ) processor **1578**. ARQ processor **1578** informs MAC downlink subframe scheduler **1566** of this condition, which in turn reschedules the requested packets from the appropriate class queues **1564***a*–**1564***f* for retransmission. Link layer acknowledgment ARQ processor **1578** also awaits positive acknowledgments from subscriber CPE station **294***d*, to determine that the data packets have been properly received. Only after receiving a positive receipt acknowledgment does MAC downlink subframe scheduler **1566** remove the packet from class queues **1564***a*–**1564***f*.

Each subscriber CPE station **294***d* has a limited amount of memory available for received data packets in an IP flow. When, for example, the devices coupled to the subscriber CPE station **294***d* (e.g., subscriber workstation **120***d*) stop receiving IP data flows (e.g., subscriber workstation **120***d* goes down), the CPE data packet queues in CPE subscriber station **294***d* are quickly filled up. In this scenario, subscriber CPE station **294***d* transmits a CPE IP flow queue depth message **1580** indicating that the queue is filled up, which can be received by CPE IP flow queue depth status processor **1582**. CPE queue depth processor **1582** informs MAC downlink subframe scheduler **1566** of this condition, which stops scheduling downlink subframes directed to subscriber CPE station **294***d*. Processor **1582** can also send messages to MAC downlink subframe scheduler **1566** to flush particular IP flows from class queues **1564***a*–**1564***f*.

h. Uplink SubFrame Prioritization

1. Overview

FIGS. **16**A and **16**B are exemplary logical flow diagrams for the uplink. The logical flow pertains to analysis and scheduling of shared wireless bandwidth to IP packet flows from a subscriber workstation **120***d* coupled to a subscriber CPE station **294***d*, being transmitted over the wireless medium up to the wireless base station **302**, and on to data network **142** for transmission to a destination host workstation **136***a*. FIG. **16**A is an exemplary logical flow diagram **1600** for uplink IP flow analyzer **632**. FIG. **16**B is an exemplary logical flow diagram **1660** for the uplink flow scheduler **634**.

The functional components for FIGS. **16**A and **16**B are explained by way of method modules, which can be viewed

as physical units (e.g., comprising software, hardware, or a combination thereof) or logical vehicles (e.g., used for explanatory purposes only). Those skilled in the art will recognize that the modules are used only to explain an exemplary embodiment, and are not to be considered limiting.

The exemplary logical flow diagram **1600** for uplink IP flow analyzer **632** of FIG. **16**A includes packet header identification component **1602**, packet characterization component **1604**, packet classification component **1606**, and IP flow presentation component **1608**. The functions of these components are explained in detail below.

In one embodiment, uplink IP flow analyzer **632** is physically located in wireless base station **302**, although those skilled in the art will recognize that the same functionality can be located remotely from wireless base station **302**. In a preferred embodiment of the present invention, the function of IP flow analyzer **632** is performed at a subscriber CPE station **294***d* desiring an uplink reservation slot for uplinking a packet/IP flow up to base station **302**. A reservation request block (RRB) request detailing the IP flow identifier, number of packets and classification of the IP flow can be created then by IP flow analyzer **632** and can be uplinked via preferably a contention RRB slot for scheduling by uplink frame scheduler **634** in future uplink subframe slots up at wireless base station **302**.

FIGS. **2**D, **3**A and **3**B are helpful to the reader for an understanding of the uplink IP flow analyzer.

2. Introduction

IP flow analyzer **632** performs the function of identifying, characterizing, classifying, and presenting data packets to an uplink frame scheduler **634**. The functions of identifying, characterizing, classifying and presenting the data packets can be respectively performed by packet header identification component **1602**, packet characterization component **1604**, packet classification component **1606** and IP flow presentation component **1608** of uplink IP flow analyzer **632**.

Packet header identification component **1602** determines whether a packet of an incoming IP flow is known to the system (i.e. is an existing IP flow), or if it is the first data packet of a new IP data flow, and determines the source application based on fields in the header section of the packet. Identification **1602** can include buffering packets and extracting and parsing the header contents. Packet characterization component **1604** characterizes a new data packet (of a new IP flow) to determine the QoS requirements for the IP flow based on the source application, and to identify the subscriber CPE station that will receive the IP flow. Packet classification component **1606** classifies the new IP data flow into one of several priority classes. Classification **1606** can include, e.g., grouping packets having similar QoS requirements. IP data flow presentation **1608** initializes the new IP data flow and presents it to uplink flow scheduler **634**.

Each time a subscriber CPE station **294***d* attempts to communicate in the uplink direction with wireless base station **302**, it requests a reservation by inserting an RRB in the uplink subframe. Uplink frame scheduler **634** then schedules the reservation request in a future uplink subframe and notifies the CPE station **294***d* of the reservation. In a downlink signal, uplink flow scheduler **634** located preferably at wireless base station **302**, transmits a reservation slot for a particular future frame for the requesting subscriber CPE station **294***d* to transmit its uplink data. Uplink flow scheduler **634** assigns the reservation based on the same parameters as the downlink flow scheduler **604** uses in the downlink. In other words, uplink flow scheduler **634** determines the reservation slots based on the queue class priority and based on a set of rules, schedules the reservations for uplink transmissions from subscriber CPE station **294***d* using, e.g., an advanced reservation algorithm. The rules are determined by inputs to the uplink flow scheduler **634** from a hierarchical class-based priority processor module **1674**, a virtual private network (VPN) directory enabled (DEN) data table **1672**, and a service level agreement (SLA) priority data table **1670**. The advanced reservation algorithm is described with respect to FIG. **14**.

3. Identification

Packet header identification component **1602** identifies the IP flow received from a subscriber CPE station **294***d* based on the packet's header contents.

A stream of packets, also known as packets from several IP flows (i.e. each IP flow is associated with a single "call") is received at packet header identification component **1602**. The IP flow in one embodiment is transmitted to subscriber CPE station **294***d* from one or more subscriber workstations **120***d* for uplink to host computers **136***a* coupled to wireless base station **302** by data network **142**. Subscriber CPE station **294***d* can transmit the data packets of the IP flow to packet buffer module **1610** of packet header identification component **1602**. In one embodiment, packet header identification component is within CPE subscriber station **294***d*. At module **1610**, the received packets are buffered in a storage area for transfer to header extraction module **1620**. At module **1620**, the packet header files are extracted and parsed to obtain the contents of the packet header fields.

Relevant fields can include, e.g., source, destination, type of service (TOS) and differentiated service (DiffServ) markings, if any exist.

For IP flows known to the system, there are entries in existing IP flow identification table **1626**. An IP flow is in the system if a previous packet of the IP flow of the existing IP data call has already been identified. In module **1622**, it is determined if there is a match between the incoming IP flow and an entry in table **1626**. If so, then the IP flow is known to the system, and control passes to module **1630** of the packet characterization component **1604**.

If the IP flow is not an existing flow known to the system, meaning that the IP flow is a new IP flow, then control passes to module **1624**, where the packet header fields are analyzed to identify the source application of the IP flow.

Packet header analysis module **1624** determines from source application packet header table **1628** the type of source application making the IP flow. The application can be any of the types of applications described with respect to FIG. **2**D or known to those skilled in the art. Examples include a file transfer protocol (FTP) download from another client workstation **138***f*, a voice telephony call from a caller **124***d* (connected over a modem), a fax machine call, and a conference call from multiple callers **124***d* and **126***d* (connected over a modem), to name a few. If the IP flow is a new IP flow, then the identification information about the new IP flow is added to table **1626**, and control passes from analysis module **1624** to module **1632** of the packet characterization component **1604**.

4. Characterization

Packet characterization component **1604** characterizes the IP flow and passes it to packet classification component **1606** for classification.

If the IP flow is an existing IP flow, control passes to module **1630** from module **1622** of the packet header identification component **1602**. If in module **1622** it is determined that the IP data flow is known to the system, in

module **1630** it is determined whether the packet is old (i.e., stale). This can include determining from a time-to-live field (a field in the IP packet header) the age of the packet, and comparing the field to a threshold age value. If the packet is determined to be stale, it is discarded. Module **1630** can anticipate application packet discards. From module **1630**, control passes to module **1640** of the packet classification component **1606**.

If the IP flow is new, control passes to module **1632** from module **1624** of the packet header identification component **1602**. If in module **1624** it is determined that the application associated with the IP flow application is not known to the system, in IP flow QoS requirements lookup module **1632** the QoS requirements for the application associated with the IP flow are determined. Module **1632** performs this operation by looking up the application in IP flow QoS requirement table **1634**. Different applications have different requirements. For example, bandwidth allocation (i.e., allocating an appropriate amount of bandwidth) is important to an application performing FTP downloads, and not jitter (i.e., time synchronizing the received data) and latency (i.e., the amount of time passage between responses). On the other hand, jitter and latency are important to voice telephony and conference calls, and bandwidth allocation is not.

After processing by module **1632**, control passes to module **163***b*. In CPE subscriber station identifier (ID) lookup module **1636** a subscriber CPE ID lookup is performed for the new IP data flow. Each subscriber CPE station **294***d* can have one or more applications, running on one or more subscriber workstations **120***d*, homed to it. Accordingly, one or many subscribers can generate or receive an IP flow directed from or at a subscriber CPE station **294***d*. A subscriber workstation **120***d* can be any device coupled to a subscriber CPE station **294***d*. Module **1636** looks up the CPE station identifier for the IP flow in table **1638**, to provide the CPE ID in the reservation request block (RRB). Control then passes from module **1636** to module **1648** of the packet classification component **1606**.

5. Classification

Packet classification component **1606** classifies the IP flow and passes it to IP flow presentation component **1608** for presentment.

For existing IP flows, control passes to module **1640** from module **1630** of the packet characterization component **1604**. If in module **1630** it is determined that the packet is not stale, then in module **1640** the packet is associated with its IP flow. As illustrated in FIG. **16**A, the packet processed herein was determined to be a portion of an IP flow known to the system. Therefore, the QoS processing of modules **1632**, **1636** and **1642** are unnecessary, because the QoS requirements of the present packet are the same as for its IP flow.

For new IP flows, control passes to module **1642** from module **1636** of the packet characterization component **1604**. In module **1642** the packet is classified or grouped into a QoS class by performing an IP flow QoS requirement table **1644** lookup where the QoS classes are stored depending on the QoS requirements for packets. From module **1642**, control passes to module **1648** of IP flow presentation component **1608**.

6. IP Flow Presentation

IP flow presentation component **1608** prepares and presents the IP data flow packets to flow scheduler **634**. In one embodiment of the uplink direction, a reservation request block (RRB) is created and uplinked via a contention slot to the wireless base station **302** for scheduling by IP flow scheduler **634**. In another embodiment, the scheduler is located at the CPE station **294***d* so no reservation request is needed.

For existing IP flows, control passes to module **1646** from module **1640** of the packet classification component **1640**. In module **1646**, the packet is added to the IP flow queue, which is the queue for the current existing IP flow. In one embodiment, this can include preparation of a RRB. From module **1646**, control passes to module **1662** of uplink flow scheduler **634**. In one embodiment, this can include uplink of the RRB from CPE **294***d* to wireless base station **302**.

For a new IP flow, control passes to module **1648** from module **1642** of the packet classification component **1606**. In initialize IP flow module **1648**, this new IP flow is initialized for presentation to module **1652**. Module **1652** presents the IP data flow (in particular, the reservation request block data packet) to module **1662** of uplink flow scheduler **634**. In module **1650**, the QoS class for the IP flow is presented to scheduler **634**, preferably by inclusion in a RRB.

7. Uplink Flow Scheduler

The exemplary logical flow diagram for the uplink flow scheduler **634** of FIG. **16**B comprises IP flow QoS class queuing processor module **1662**, MAC uplink subframe scheduler module **1666**, hierarchical class-based priority processor module **1674**, VPN DEN data table module **1672**, SLA priority data table **1670**, CPE IP flow queue depth status processor **1682** and link layer acknowledgment processor module **1678**.

Uplink flow scheduler **634** of FIG. **16**B also includes QoS class queues for class 1, **1664***a*; class 2, **1664***b*; class 3, **1664***c*; class 4, **1664***d*; class 5, **1664***e*; and class 6, **1664***f*; and

MAC uplink subframes: frame n **1668***a*; frame n+1, **1668***b*; frame n+2, **1668***c*; frame n+3, **1668***d*, . . . frame n+p, **1668***k*.

In one embodiment, uplink flow scheduler **634** is physically located in wireless base station **302**, although those skilled in the art will recognize that the same functionality can be located remotely from wireless base station **302**. For example, in another embodiment, uplink flow scheduler **634** can be located at CPE station **294***d* and is in communication with other CPE stations **294** and the wireless base station **302**.

Uplink flow scheduler **634** is used to schedule the uplink subframe. The entire frame is divided into an uplink portion (called an uplink subframe) for transmitting uplink frames, and a downlink portion (called a downlink subframe) for transmitting downlink frames.

Illustrated in FIG. **16**B are WAP antenna **290***d*, the wireless medium, RF transceiver subscriber antenna **292***d*, subscriber CPE station **294***d* and subscriber workstation **120***d*. WAP **290***d* and RF transceiver subscriber antenna **292***d* respectively provide a wireless connection between wireless base station **302** (where uplink flow scheduler **634** resides in one embodiment) and subscriber CPE station **294***d*, which can transmit upstream an IP flow from an application running on client computer **120***d*. WAP **290***d* serves as a wireless gateway for data network **142**, and RF transceiver subscriber antenna **292***d* serves as a wireless gateway for subscriber CPE station **294***d* to uplink the IP flow packet data.

Also illustrated in FIG. **16**B is data interface **320**, which provides a connection from uplink flow scheduler **634** for sending uplinked IP flow packets on to data router **140***d* of data network **142** and on to a destination host computer **136***a*. These connections are also illustrated in FIGS. **2**D and **3**B.

The previous frame includes an uplink reservation request which is received by the wireless base station from a

subscriber CPE station **294***d*. At this point, the reservation request block has been identified, characterized, classified, and presented, preferably at the CPE station **294***d*, and has been transmitted to uplink flow scheduler **634** from uplink flow analyzer **632** at the CPE **294***d*. In particular, the reservation request block is presented to IP flow QoS class queuing processor module **1662** from module **1650**. Module **1662** informs MAC uplink subframe scheduler **1666** of the reservation.

In turn, MAC uplink subframe scheduler **1666** uses a slot in the subframe to acknowledge receipt of the request called the acknowledgment request block (ARB). An exemplary slot used to convey the frame, slot, and IP flow identifier for this reservation is described with respect to FIG. **12**. Scheduler **1666** transmits in this reservation slot the CPE identification data, along with which future slot(s) and frame(s) the requesting subscriber CPE station **294***d* is permitted to use for uplink of the requested data packet IP flow transmissions.

The future slot(s) in the future frame(s) are assigned, e.g., based on inputs from hierarchical class-based priority processor module **1674**, VPN DEN data table **1672** and service level agreement (SLA) priority data table **1670**. These components function in a similar manner to hierarchical class-based priority processor module **1574**, VPN DEN data table **1572** and service level agreement (SLA) priority data table **1570**, described with respect to the downlink flow scheduler **604**.

When IP flow QoS class queuing processor module **1662** receives packets of an existing or new IP flow from IP flow presentation module **1608**, it then creates class queues **1664***a*–**1664***f*, which is a variable number of queues, and places the packets in these class queues. In a preferred embodiment there are between 3 and 10 classes. These queues hold reservation request packets for scheduling. Packets are placed in class queues **1664***a*–**1664***f* according to the contents of the reservation request block for input to module **1662**.

Module **1662** receives inputs from hierarchical class-based priority processor module **1674**, VPN DEN data table **1672** and service level agreement (SLA) priority data table **1670**. The queuing function of module **1662** is based on these inputs. These components function analogously to their counterparts in the downlink flow scheduling method. SLA priority data table **1670** and VPN DEN data table **1672** receive input from operations, administration, maintenance and provisioning (OAM&P) module **1108**. OAM&P module **1108** provides updates to priorities when, e.g., a subscriber modifies its service level agreement or a VPN subscription is changed.

MAC uplink subframe scheduler **1666** takes the requests queued in class queues **1664***a*–**1664***f*, and schedules reservations of slots in frames **1668***a*–**1668***k*, which is a variable number of frames. In one embodiment, each frame is scheduled with up to a predetermined number limit or percentage limit of packets from each of the classes **1664***a*–**1664***f*. The requests can be scheduled as shown in FIG. **13**, taking into account certain priorities. In another embodiment, the frames are scheduled according to the inventive advanced reservation algorithm method for scheduling isochronous type traffic described with respect to FIG. **14**. In yet another embodiment, the frames are scheduled according to a combination of known methods and the advanced reservation algorithm method of the present invention.

The reservation slot schedule can then be sent down to the CPE stations **294** using, e.g., FDB slots such as **1236***g* and **1236***h* of FIG. **12**F. The uplink slots can then be inserted by CPE station **294***d* into the uplink subframe as scheduled. The frame slots are then transmitted up from CPE station **294***d* to wireless base station **302** and are then sent on as packets to their destination addresses. For example, from wireless base station **302** the packets can be transmitted over data network **142** to a host computer **136***a*.

After the uplink packets are received by the wireless base station **302**, the wireless base station **302** sends an upstream acknowledgment data block (UAB) message back down to the transmitting subscriber CPE station **294***d*, to acknowledge receipt of the transmitted data packets.

Every once in a while, a packet is lost through noise or other interference in the wireless medium. When this situation arises, the subscriber CPE station **294***d* determines that it has not received a UAB data acknowledgment, so it sends a retransmit request requesting another uplink reservation slot to wireless base station **302** via WAP **290***d*, which transmits the request to link layer acknowledgment (ARQ) processor **1678**. ARQ processor **1678** informs MAC uplink subframe scheduler **1666** of the need of retransmission (i.e. the need of a frame slot reservation for resending the uplink packet). CPE subscriber station **294***d* can also send to ARQ processor **1678**, other data messages about nonreceipt of uplink transmission acknowledgments. The ARQ **1678** can forward such messages on to the uplink subframe scheduler **1666**. The uplink subframe scheduler **1666** in turn reschedules the requested uplink reservation from the appropriate class queues **1664***a*–**1664***f*. Alternatively, in another embodiment, link layer acknowledgment processor **1678** can also send a positive UAB acknowledgment to the subscriber CPE station **294***d*, to indicate that the data packets have been properly received. Thus uplink scheduler **1666** in addition to scheduling first time reservations, also can schedule repeat reservations for lost packets.

Each subscriber CPE station **294***d* has a limited amount of memory space available for queuing packets received from subscriber workstations **120***d* awaiting reservation slots of uplink from the CPE **294***d* to wireless base station **302**. When, for example, the the queue of subscriber CPE station **294***d* becomes full from a backup of packets awaiting upstream reservations, IP data flows can potentially be lost, or packets may become stale. In this scenario, subscriber CPE station **294***d* transmits a CPE IP flow queue depth message **1680** to the wireless base station **302** indicating that the queue is filled up, which can be received by CPE IP flow queue depth status processor **1682**. Processor **1682** can inform MAC uplink subframe scheduler **1666** of this condition, which can, e.g., increase temporarily the priority of IP flows at subscriber CPE station **294***d* to overcome the backlog or can, e.g., stop transmitting additional downlink packets to the CPE station **294***d* until the queue depth backlog is decreased to an acceptable level again. Processor **1682** can also send messages to MAC uplink subframe scheduler **1666** to flush reservation requests from the subscriber CPE station **294***d* in class queues **1664***a*–**1664***f*.

4. TCP Adjunct Agent

TCP is a reliable transport protocol tuned to perform well in traditional networks where congestion is the primary cause of packet loss. However, networks with wireless links incur significant losses due to bit-errors. The wireless environment violates many assumptions made by TCP, causing degraded end-to-end performance. See for example, Balakrishnan, H., Seshan, S. and Katz, R. H., "Improving Reliable Transport and Handoff Performance in Cellular Wireless Networks," University of California at Berkeley, Berkeley, Calif., accessible over the Internet at URL, http://

www.cs.berkeley.edu/~ss/papers/winet/html/winet.html, dealing more directly with handoffs and bit errors in a narrowband wireless environment, the contents of which are incorporated by reference. Attempts to address this problem have modified TCP in order to overcome it. However, this is not a commercially feasible means of overcoming this challenge. It is impracticable to implement any solution that requires a change to the standard operation of TCP.

The present invention uses an enhanced MAC layer which interfaces with a TCP adjunct agent to intercept TCP layer requests to manipulate the TCP layers at either a source or destination end of a transmission, to modify TCP behavior at the source and destination of the TCP/IP transmission which includes an intermediary wireless link. Packets can be queued at the wireless base station awaiting receipt acknowledgment and the base station can perform local retransmissions across the wireless link to overcome packet loss caused by high bit-error rates. Communication over wireless links is characterized by limited bandwidth, high latencies, sporadic high bit-error rates and temporary disconnections which must be dealt with by network protocols and applications.

Reliable transport protocols such as TCP have been tuned for traditional wired line networks. TCP performs very well on such networks by adapting to end-to-end delays and packet losses caused by congestion. TCP provides reliability by maintaining a running average of estimated round-trip delay and mean deviation, and by retransmitting any packet whose acknowledgment is not received within four times the deviation from the average. Due to the relatively low bit-error rates over wired networks, all packet losses are correctly assumed to be caused by congestion.

In the presence of the high bit-error rates characteristic of wireless environments, TCP reacts to packet losses as it would in the wired environment, i.e. it drops its transmission window size before retransmitting packets, initiates congestion control or avoidance mechanisms (e.g., slow start) and resets its retransmission timer. These measures result in an unnecessary reduction in the link's bandwidth utilization, thereby causing a significant degradation in performance in the form of poor throughput and very high interactive delays.

The present invention maintains packets in class queues awaiting acknowledgment of receipt from the subscriber CPE stations. Unacknowledged data slots can then be resent by having the wireless base station perform local retransmissions to the subscriber CPE station. By using duplicate acknowledgments to identify a packet loss and performing local retransmissions as soon as the loss is detected, the wireless base station can shield the sender from the inherently high bit error rate of the wireless link. In particular, transient situations of very low communication quality and temporary disconnectivity can be hidden from the sender.

For transfer of data from a CPE subscriber host to a wireless base station host, missing packets are detected at the wireless base station and negative acknowledgments can be generated for them. The negative acknowledgments can request that the packet be resent from the CPE subscriber host (the sender). The CPE subscriber host can then process the negative acknowledgment and retransmit corresponding missing packets. Advantageously, no modifications to the sender TCP or receiver TCP is necessary, since the present invention places TCP aware functionality in the MAC layer.

FIG. **5A** illustrates flow **500** depicting IP flows from a source TCP at a subscriber host, down a protocol stack for transmission through a CPE subscriber station, through a wireless medium to a wireless base station, up and through a protocol stack at the wireless base station having an example TCP adjunct agent, then through a wireline connection and through a protocol stack to a destination host. The adjunct TCP agent modifies operation of a TCP sliding window algorithm at the transmitting TCP and in cooperation with proactive reservation-based intelligent multimedia access technology (PRIMMA) media access control (MAC) enables local retransmission over the wireless medium in accord with the present invention.

Specifically, flow **500** illustrates IP packet flow from subscriber workstation **120***d*, through CPE subscriber station **294***d* at CPE subscriber location **306***d*, then over a wireless transmission medium to wireless base station **302**, and eventually over a wireline link over data network **142** to host workstation **136***a*.

TCP adjunct agent **510***e* makes sure transport is reliable by modifying operation of the TCP sliding window algorithm at the transmitting TCP in a manner that optimizes the window for the wireless medium. TCP adjunct agent **510***e* advantageously is transparent to industry standard protocols as agent **510***e* does not require modification of the standard TCP/UDP layer of client subscriber workstation **120***d* or host workstation **136***a*.

Flow **500** includes IP flows from application layer **512***a*, down the protocol stack through TCP/UDP layer **510***a*, through IP layer **508***a*, then through point-to-point (PPP) layer **520***a*, then through data link Ethernet layer **504***a*, then through 10BaseT Ethernet network interface card (NIC) physical layer **502***a*, over a wire line connection to 10BaseT Ethernet NIC physical layer **502***b* of subscriber CPE **294***d*.

Subscriber CPE **294***d* flows packets coming in from NIC **502***b*, back up its protocol stack through Ethernet layer **504***b*, through PPP layers **520***b* and **520***c*, back down through PRIMMA MAC **504***c* to wireless physical layer **502***c* including antenna **292***d*, then over the wireless medium to antenna **290***d* of wireless base station **302**.

Wireless base station **302** flows packet IP flows up from antenna **290***d* at physical layer **502***d* through PRIMMA MAC layer **504***d*, through PPP layer **520***a*, through IP layer **508***d* to TCP adjunct agent **510***e*, which can flow IP flows down through IP layer **508***e*, through PPP layer **520***e*, through wide area network (WAN) layer **504***e*, through wireline physical layer **502***e*, through interface **320**, over routers **140***d*, through data network **142**, via wireline connections to wireline layer **502***f* of WAN host workstation **136***a*.

Host workstation **136***a* flows IP flows from wireline layer **502***f*, up through its protocol stack through WAN layer **504***f*, through PPP layer **520***f*, through IP layer **508***f*, to TCP/UDP layer **510***f* and on to application layer **512***f*.

TCP/UDP layers **510***a* and **510***f* act to provide such transport functions as, e.g., segmentation, managing a transmission window, resequencing, and requesting retransmission of lost packet flows. Normally TCP layers **510***a* and **510***f* would send a window of packets and then await acknowledgment or requests for retransmission. A TCP sliding window algorithm is normally used to vary the transmission flow to provide optimized transport and to back off when congestion is detected by receipt of requests for retransmission. Unfortunately in the wireless environment, due to high bit error rates, not all packets may reach the destination address, not because of congestion, but rather because of high bit error rates, so as to prompt a retransmission request from the destination IP host to the source. Rather than slow transport, TCP adjunct agent **510***e* modifies operation of the TCP sliding window algorithm to optimize operation over wireless. PRIMMA MAC layer **504***d* inter-

acts with TCP adjunct agent **510***e* permitting the agent to intercept, e.g., retransmission requests, from TCP layer **510***a* of subscriber workstation **120***d* intended for host **136***a*, and allowing the wireless base station to retransmit the desired packets or flows to subscriber workstation **120***d* rather than forwarding on the retransmission request to host **136***a*, since the packets could still be stored in the queue of PRIMMA **504***d* and would not be discarded until an acknowledgment of receipt is received from the subscriber CPE. Since retransmission can be performed according to the present invention at the PRIMMA MAC data link layer, i.e. layer 2, retransmission can occur from the base station to the CPE subscriber, rather than requiring a retransmission from all the way over at the transmitting source TCP which would cause TCP to backoff its sliding window algorithm. Thus, by having wireless base station **302** retransmit until receipt is acknowledged over the wireless link, the inherently high bit error rate can be overcome, while maintaining an optimal TCP window.

Recall, a TCP transmitter transmits a TCP sliding window block of packets and alters the size of the window upon detection of congestion. The TCP transmitter transports a block of packets in a window, and then awaits acknowledgment from the receiver. If transmission is going smoothly, i.e. no congestion or lost packets occur, then the transmitter TCP ramps up the transmission rate. This increased transmission rate continues until the transmitting TCP detects congestion or packet loss. When notified of congestion, the transmitting TCP stops transmitting, backs off and sends a smaller block (i.e. a smaller window) of packets.

TCP adjunct agent modifies normal TCP operation by tricking the transmitting TCP and its transmitting window algorithm. The TCP adjunct agent prevents the transmitter from being notified of loss, i.e. receiving congestion notification, from the receiving TCP by, e.g., preventing duplicate retransmission requests. Since the transmitting TCP does not receive such notification, it does not modify the TCP sliding window and transmission continues at the higher rate.

In the event that real congestion occurs, i.e. if the TCP adjunct agent recognizes packets really were lost, then the TCP adjunct agent can let the retransmission request go through to the transmitting TCP. This is advantageously accomplished because the MAC link layer of the present invention is in communication with the higher protocol layers, it is application aware, transport aware and network aware. In this case, because the MAC layer is transport layer aware, PRIMMA MAC layer **504***d* communicates with the TCP adjunct agent **510***e* at layer 4. Since the MAC requires acknowledgment of receipt of wireless transmissions sent to the CPE subscriber station **294***d* for every packet sent from the wireless base station **302**, the MAC layer **504***d* knows whether an inter-TCP layer communication, e.g., a request for retransmission, is sent from a client computer TCP at the CPE station is created because the lost packet was lost in wireless transmission, or because of real congestion.

If PRIMMA MAC **504***d* does not receive an acknowledgment from **504***c*, then the PRIMMA MAC **504***d* of wireless base station **302** can retransmit the contents of the lost packet to the subscriber CPE station **294***d*. If the PRIMMA MAC **504***c* of the subscriber CPE station **294***d* acknowledges receipt and still requests a retransmission, then real congestion could have occurred and the PRIMMA MAC **504***d* of the wireless base station **302** can let the TCP adjunct agent **510***e* know that it should allow the retransmission request to be sent to the transmitting TCP **510***f* of host workstation **136***a*.

Thus, TCP adjunct agent **510***e* of the present invention can modify operation of the TCP sliding window algorithm in a manner that is optimal for the wireless medium, without requiring any change to commercially available TCP layers **510***a* and **510***f* at the receiver and sender hosts. In an embodiment, TCP adjunct agent **510***e* obviates the need for any modification of the TCP layers at either the sending (i.e. transmitting) host or client. In another embodiment the host and client TCP layers are unaware of the modification of operation by the TCP adjunct agent, i.e. it is transparent to source and destination TCP layers. In another embodiment, TCP adjunct agent **510***e* intercepts retransmission requests between a TCP layer of the client computer coupled to the subscriber CPE station and the TCP layer of the host workstation coupled to the data network.

FIG. **5**B illustrates functional flow diagram **522** including an example functional description of TCP adjunct agent **510***e* performing an outgoing TCP spoof function. Referring to FIGS. **5**B and **5**A, diagram **522** assumes that a TCP layer **510***f* at a transmitting host **136***a* has transmitted a windowful of packet data to subscriber workstation **120***d*, and awaits acknowledgment. Diagram **522** illustrates receipt of an outgoing TCP message **524** in TCP adjunct agent **510***e* at wireless base station **302** which has been sent from subscriber workstation **120***d* via subscriber CPE station **294***d*.

In step **526**, the TCP header contents of outgoing TCP message **524** is parsed in order to reveal the contents of the message being sent from subscriber workstation **120***d* through the wireless network toward the transmitting host **136***a*.

In step **528**, it is determined whether the TCP header contents includes a duplicate acknowledgment message from the CPE station. Receiving a duplicate acknowledgment request from the CPE subscriber location could be indicative of a lost message in the wireless medium, or a real congestion problem. If in step **528** the TCP packet is determined to be a duplicate acknowledgment message, then processing can continue with step **532**, if not, then processing can continue with step **530**.

In step **530**, it is determined that there was real congestion, i.e., this was not a duplicate acknowledgment message caused by retransmission attempts at the wireless link layer. Thus, in step **530**, the TCP message is permitted to pass through TCP adjunct **510***e* without modification, and can continue through flow **500** to TCP layer **510***f* of FIG. **5**A.

In step **532**, since there was a duplicate acknowledgment detected in step **528**, it is determined whether the packet was successfully transmitted, or not. Step **532** is performed via intercommunication between TCP adjunct agent **510***e* and PRIMMA MAC layer **504***d*. This is an example of the interactivity between PRIMMA MAC and higher layer protocols illustrated as line **428** in FIG. **4**. PRIMMA MAC layer **504***d* can identify whether a packet was successfully sent from wireless base station **302** to CPE station **294***d* since, as illustrated in FIG. **15**B, requests for retransmission **1576** are received from CPE station **294***d* at link layer acknowledgment (ARQ) processor **1578** to MAC downlink subframe scheduler **1566** alerting the scheduler **1566** to retransmit the lost packet in a future frame **1568**. If in step **532**, it is determined that the packet was successfully transmitted, then processing can continue with step **530**, as described above. If however it is determined that the packet was not successfully transmitted, then processing continues with step **534**.

In step **534**, since the packet was not successfully transmitted, TCP adjunct agent **510***e* can suppress transmission of TCP message **524** since it can be assumed that the

packet was lost in the wireless medium. Processing can continue with step **536**.

In step **536**, TCP adjunct agent **510***e* can wait for notification from PRIMMA MAC **504***d* that a successful link layer retransmission of the lost packet was received at link layer acknowledgment processor **1578**. From step **536**, processing can continue with step **538**.

In step **538**, upon receipt of acknowledgment of a successful PRIMMA MAC **504***d* link layer retransmission, then normal TCP messages can be resumed.

In another step (not shown), TCP adjunct agent and PRIMMA MAC layers can set a limit of a threshold number of retransmission attempts, and if that threshold is reached, then processing can continue with step **530** to permit the TCP message to pass without modification.

FIG. **5**C illustrates functional flow diagram **540** including an example functional description of TCP adjunct agent **510***e* performing an incoming TCP spoof function. Referring to FIGS. **5**C and **5**A, diagram **540** assumes that a TCP layer **510***a* at a transmitting subscriber workstation **120***d* has transmitted a windowful of packet data to host **136***a*, and awaits acknowledgment. Diagram **544** illustrates receipt of an incoming TCP message **542** in TCP adjunct agent **510***e* at wireless base station **302** which has been sent from host workstation **136***a* via data network **142** for transmission over the wireless medium to subscriber CPE **294***d* to subscriber workstation **120***d*.

In step **544**, the TCP header contents of ingoing TCP message **542** is parsed in order to reveal the contents of the message being sent from host **136***a* through the wireless network toward the transmitting subscriber workstation **120***d*.

In step **546**, it is determined whether the TCP header contents includes a duplicate acknowledgment message from host **136***a*. Receiving a duplicate acknowledgment request from the host could be indicative of a lost message in the wireless medium, or a real congestion problem. If in step **546** the TCP packet is determined to be a duplicate acknowledgment message, then processing can continue with step **550**, if not, then processing can continue with step **548**.

In step **548**, it is determined that there was real congestion, i.e., this was not a duplicate acknowledgment message caused by retransmission attempts at the wireless link layer. Thus, in step **548**, the TCP message is permitted to pass through TCP adjunct **510***e* without modification, and can continue through flow **500** to TCP layer **510***a* of FIG. **5**A.

In step **550**, since there was a duplicate acknowledgment detected in step **546**, it can be determined whether the packet was successfully transmitted, or not. Step **550** can be performed via intercommunication between TCP adjunct agent **510***e* and PRIMMA MAC layer **504***d*. This is an example of the interactivity between PRIMMA MAC and higher layer protocols illustrated as line **428** in FIG. **4**. PRIMMA MAC layer **504***d* can identify whether a packet was successfully sent from CPE station **294***d* to wireless base station **302**, as illustrated in FIG. **16**B, requests for retransmission **1676** are received from CPE station **294***d* at link layer acknowledgment (ARQ) processor **1678** to MAC downlink subframe scheduler **1666** alerting the scheduler **1666** to retransmit the lost packet in a future frame **1668**. If in step **550**, it is determined that the packet was successfully transmitted, then processing can continue with step **548**, as described above. If however it is determined that the packet was not successfully transmitted, then processing continues with step **552**.

In step **552**, since the packet was not successfully transmitted, TCP adjunct agent **510***e* can suppress transmission of TCP message **542** since it can be assumed that the packet was lost in the wireless medium. Processing can continue with step **554**.

In step **554**, TCP adjunct agent **510***e* can wait for notification from PRIMMA MAC **504***d* that a successful link layer retransmission of the lost packet was received at link layer acknowledgment processor **1678**. From step **554**, processing can continue with step **556**.

In step **556**, upon receipt of acknowledgment of a successful PRIMMA MAC **504***d* link layer retransmission, then normal TCP messages can be resumed.

In another step (not shown), TCP adjunct agent and PRIMMA MAC layers can set a limit of a threshold number of retransmission attempts, and if that threshold is reached, then processing can continue with step **548** to permit the TCP message to pass without modification.

5. Wireless QoS Aware PRIMMA Media Access Control (MAC) Hardware Architecture

FIG. **10** illustratively depicts an embodiment of PRIMMA MAC hardware architecture **1000**. Architecture **1000** shows data network **142** coupled by a wireline bidirectional connection to WAN interface **320**.

WAN interface **320** is bidirectionally linked to a bidirectional data frame FIFO **1002** which is bidirectionally coupled to both segmentation and resequencing (SAR) **1004** and QoS/SLA rules engine and processor **1008**.

QoS/SLA rules engine and processor **1008** is also bidirectionally coupled to IP flow buffers **1014** and flash random access memory (RAM) **1010**.

SAR **1004** is bidirectionally coupled to IP flow buffers **1014**, flash RAM **1010**, QoS/SLA rules engine and processor **1008** and PRIMA MAC scheduler ASIC **1012**.

PRIMA MAC scheduler ASIC **1012** is also bidirectionally coupled to an RF interface **290**, a static RAM (SRAM) radio cell buffer **1018** and IP blow buffer **1014**.

6. Wireless Base Station Software Organization

FIG. **11** is an exemplary software organization for a packet-centric wireless point to multi-point telecommunications system. The software organization of FIG. **11** includes wireless transceiver and RF application specific integrated circuit (ASIC) module **290**, IP flow control component **1102**, WAN interface management component **1104**, QoS and SLA administration component **1106**, system and OAM&P component **1108**, customer billing and logging component **1110**, directory enabled networking (DEN) component **1112**, and wireless base station **320**.

IP flow control module **1102** includes transmission queuing control module **1102***a*, TCP rate control and class of service module **1102***b*, wireless PRIMMA MAC layer engine **1102***c* and IP flow identification and analysis module **1102***d*.

WAN interface management component **1104** includes WAN ingress/egress queuing control module **1104***a*, WAN interface ports (e.g., for T1, T3, OC3 ports) **1104***b*, firewall and security module **1104***c*, and WAN traffic shaping module **1104***d*.

The IP Flow control component **1102** and WAN interface management component **1104** represent the "core" of the system, where the packet processing, MAC layer scheduling, TCP proxy agent, and WAN I/F control functions are located. Much of the activities of the "non-core" components described above support and control these core components.

QoS and SLA administration component **1106** includes includes QoS performance monitoring and control module

1106*a*, service level agreements module 1106*b*, policy manager module 1106*c* and encryption administration module 1106*d*.

The QoS and SLA administration component 1106 provides the static data needed by the system in order to properly group particular IP-flows into QoS classes. Typically, during the provisioning phase of installing the system, the service provider will (remotely) download pertinent information about the subscriber CPE station 294, including the subscriber CPE stations's SLA, any policy-based information (such as hours of operation or peak data transmission rate allowance.). Encryption keys or "strengths" can also be downloaded, which may be subscriber CPE station or service provider specific.

System OAM&P component 1108 includes SNMP proxy client for WAP module 1108*a*, SNMP proxy clients for CPE module 1108*b*, and system operations, administration, management and provisioning module 1108*c*.

The OAM&P component 1108 allows remote service personnel and equipment to monitor, control, service, modify and repair the system. System performance levels can be automatically monitored, and system traps and traces can be set. Subscriber complaints can be addressed with the use of remote test and debug services controlled by OAM&P component 1108. System capacity limits can be monitored, and proactive provisioning of additional WAN connectivity can occur, as the result of automatic trend analysis functions in OAM&P component 1108.

Customer billing and logging module 1110 includes account logging and database management module 1110*a*, transaction query and processing control module 1110*b*, billing and account control module 111*c*, and user authentication module 1110*d*.

The customer billing and logging component 1110 allows the service provider to receive account, billing and transaction information pertaining to subscribers in the system. For service providers who bill on the basis of usage, cumulative system resource utilization data can be gathered. For specific types of activities (eg. video conferencing, multi-casting, etc.) there may be special billing data that is collected and transmitted to the service provider. This component also controls the availability of the system to subscribers through the operation of the subscriber authentication function. Once a subscriber is authorized to use the system, a new subscriber authentication entry is made (remotely) by the service provider. Likewise, a subscriber can be denied further access to the system for delinquent payment for services, or for other reasons. The service provider can also remotely query the system for specific account-related transactions.

Directory Enabled Networking (DEN) component 1112 includes DEN QoS 1112*a* module, DEN management and provisioning 1112*b* module, DEN IPSEC module 1112*c* and IP-based VPN control and administration module 1112*d*.

The DEN component 1112 allows the service provider the means to input into the system relevant information regarding the operation of DEN-based VPN's of subscribers. Subscriber VPNs need to be "initialized" and "provisioned" so that the system properly allocates system resources to subscribers with these VPNs, and provides for the recognition and operation of these VPNs. Data from DEN component 1112 are utilized by the system to apply the appropriate priorities to IP-flows of the subject subscribers.

The invention's packet-centric wireless base station supports directory enabled networking (DEN), a MICROSOFT, INTEL and CISCO standard for providing a standard structure for how distributed sites manage IP flows. The present invention prioritizes VPN traffic in a lightweight directory access protocol (LDAP)-compliant (LDAP is available from MICROSOFT of Redmond, Wash.) manner which allows remote administration, provisioning and management. The present invention is also LDAP version 2 compliant. The present invention also complies with the X.500 standard promulgated by the international telecommunications union/telecommunications section (ITU/T), and with the RFC 1777.

In one embodiment, DEN provides policy-based network management, IPsec compatible network security, and IPsec based VPNs. The DEN of the wireless base station 302 is planned to be common information model (CIM) 3.0 compatible (once the specification is finalized). The wireless base station 302 can provide native DEN support and supports directory based DEN QoS mechanisms including reservation model (i.e. RSVP, per-flow queuing), and precedence/priority/differentiated model (i.e. packet marking). Wireless base station 302 can plan support of DEN network policy QoS, and until DEN is complete, can support internal QoS and network extensions.

6. IPsec Support

IPsec is introduced above with reference to FIG. 4. IPsec provides a standard method of encrypting packets. In VPN tunnel mode, an entire header can be encoded, i.e. encrypted. In order for the present invention to be able to implement its packet-centric, QoS aware prioritization, during identification of a packet/IP flow, the wireless base station needs to be able to analyze the contents of header fields of the packets. Therefore, analysis of unencrypted packets is desirable.

The present invention already encrypts the data stream prior to transmitting frames over the wireless medium, so IPsec does not really need to be used over the wireless link to provide for encrypted transmission. Where a service provider finds it desirable to use IPsec, IPsec can be used for authentication and secure encapsulation of the header and payload, or just the payload data. IPsec is normally integrated at a firewall. If a service provider desires to implement the present invention and IPsec, then the present invention should be implemented behind the firewall, i.e. the firewall can be moved to the wireless base station. This permits ending the IPsec stream at the base station which can provide the base station access to packet header fields.

FIG. 17 illustrates IP flow in the downlink direction including IPsec encryption. Similarly, FIG. 18 illustratively depicts an uplink direction of IPsec support of the present invention.

FIG. 17 illustrates downlink flow 1700 depicting downlink direction IP flows from a source host workstation 136*a*, down a protocol stack which supports IPsec, for transmission up and through wireless base station 302 which is coupled to data network 142, through encryption layers, then through the wireless link to subscriber CPE 294*d*, up and through a protocol stack at the subscriber CPE 294*d*, then through a wireline connection to data network 142 and up through the protocol stack to the destination subscriber workstation 120*d* at subscriber location 306*d*.

Specifically, flow 1700 illustrates IP packet flow from host workstation 136*a*, through wireless base station 302, then over a wireless transmission link to subscriber CPE 294*d*, and over a wireline link to subscriber workstation 120*d*.

Host workstation 136*a* flows IP flows down from application layer 1712*h*, down through TCP/UDP layer 1710*h*, through IP layer 1708*h*, through optional PPP layer 1706*h*, through Ethernet layer 1705*h*, down through 10BaseT layer 1702*h*, over data network 142 to 10BaseT layer 1702*g*, then up through Ethernet 1704*g*, up its protocol stack through

nal PPP layer **1706***g* to IP layer **1708***g* and **1708***h*, back through Internet firewall and IPsec security gateway down through WAN layer **1704***f*, to wireline layer to data network **142** to wireline physical layer **1702***e*. cline physical layer **1702***e* of wireless base station ows IP flows up the protocol stack through WAN layer through IPsec security gateway **1706***e* and firewall to work layer **1708***e* and **1708***d* and then down through tion layer **1706***d*, PRIMMA MAC layer **1704***d* and to wireless link to subscriber CPE **294***d*.

scriber CPE **294***d* flows packet IP flows up from **292***d* at physical wireless layer **1702***c* up through layer **1704***c*, through encryption layer **1706***c*, through rs **1708***b* and **1708***c*, then down through optional **706***b* to Ethernet layer **1704***b* to 10BaseT connection to 10BaseT connection.

scriber workstation **120***d* flows IP flows up from T layer **1702***a* up through its protocol stack through t layer **1704***a*, through optional PPP layer **1706***a*, l IP layer **1708***a*, to TCP/UDP layer **1710***a* and on up cation layer **1712***a*.

18 illustrates uplink flow **1800** depicting uplink n IP flows from a source TCP at subscriber work-**120***d* at CPE location **306***d*, down a protocol stack for sion through Ethernet coupled CPE subscriber sta-d through wireless medium to wireless base station and through a protocol stack at the wireless base **02** which supports IPsec, then through a wireline on to data network **142** and through a protocol stack ination host.

cally, flow **1800** illustrates IP packet flow from r workstation **120***d*, through subscriber CPE **294***d*, a wireless transmission medium to wireless base **2**, and eventually over a wireline link to host on **136***a*.

1800 includes IP flows from application layer own the protocol stack through TCP/UDP layer rough IP layer **1808***a*, then through optional point-PPP) layer **1806***a*, then through data link Ethernet **4***a*, then through 10BaseT Ethernet network inter-l (NIC) physical layer **1802***a*, over a wire line n to 10BaseT Ethernet NIC physical layer **1802***b* ber CPE **294***d*.

ber CPE **294***d* flows packets coming in from NIC ck up its protocol stack through Ethernet layer ough optional PPP layer **1806***b* to IP layer **1808***b* back down through an Internet firewall and IPsec teway **1806***c*, down through PRIMMA MAC ireless physical layer **1802***c* including antenna over the wireless medium, such as, e.g., RF tion, cable RF, and satellite link, to antenna **290***d* base station **302** at wireless physical layer **1802***d*.

base station **302** flows packet IP flows up from **0***d* at physical wireless layer **1802***d* up through **1804***d*, through IPsec layers **1806***d* and **1806***d*, encapsulate packets and encrypt them. From **1806***e*, IP flows can flow down through WAN and through wireline physical layer **1802***e* over **142**.

physical layer **1802***f* flows IP flows up the ack through WAN layer **1804***f* through IPsec teway **1806***f* and firewall to IP network layer **808***g* and then down through optional PPP layer rnet layer **1804***h* and down through 10BaseT through interface **320**, over routers **140***d*, a network **142**, via wireline connections to ysical layer **1802***h* of host workstation **136***a*.

Host workstation **136***a* flows IP flows up from 10BaseT layer **1802***h* up through its protocol stack through Ethernet layer **1805***h*, through optional PPP layer **1806***h*, through IP layer **1808***h*, to TCP/UDP layer **1810***h* and on to application layer **1812***h*.

IV. Conclusion

While various embodiments of the present invention have been described above, it should be understood that they have been presented by way of example only, and not limitation. Thus, the breadth and scope of the present invention should not be limited by any of the above-described exemplary embodiments, but should be defined only in accordance with the following claims and their equivalents.

What is claimed is:

1. An IP flow classification system that groups IP flows in a packet-centric wireless point to multi-point telecommunications system, said classification system comprising:

a wireless base station coupled to a first data network;

one or more host workstations coupled to said first data network;

one or more subscriber customer premise equipment (CPE) stations in wireless communication with said wireless base station over a shared bandwidth using a packet-centric protocol; and

one or more subscriber workstations coupled to each of said subscriber CPE stations over a second network;

resource allocation means optimizing end-user quality of service (QoS) and allocating shared bandwidth among said subscriber CPE stations; and

means for analyzing and scheduling an internet protocol (IP) flow over said shared wireless bandwidth, wherein said analyzing means comprises:

a classifier that classifies said IP flow.

2. The system of claim **1**, wherein said classifier comprises:

means for associating a packet of an existing IP flow with said IP flow.

3. The system of claim **1**, wherein said classifier comprises:

QoS grouping device that groups a packet of a new IP flow into a QoS class grouping.

4. The system of claim **3**, wherein said QoS grouping device comprises:

determining device that determines and takes into account QoS class groupings for said IP flow.

5. The system of claim **4**, wherein said QoS grouping device comprises:

optional differentiated services (Diff Serv) device that takes into account an optional Diff Servs field priority marking for said IP flow.

6. The system of claim **4**, wherein said QoS grouping device comprises:

optional type of service (TOS) device that takes into account any optional type of service field priority marking for said IP flow.

7. The system according to claim **1**, wherein said packet-centric protocol is transmission control protocol/internet protocol (TCP/IP).

8. The system according to claim **1**, wherein said packet-centric protocol is user datagram protocol/internet protocol (UDP/IP).

9. The system according to claim **1**, wherein said shared wireless bandwidth comprises a wireless communication medium comprising at least one of:

a radio frequency (RF) communications medium;

a cable communications medium; and

a satellite communications medium.

10. The system according to claim **9**, wherein said wireless communication medium further comprises, a telecommunications access method including at least one of:

a time division multiple access (TDMA) access method;

a time division multiple access/time division duplex (TDMA/TDD) access method;

a code division multiple access (CDMA) access method; and

a frequency division multiple access (FDMA) access method.

11. The system according to claim **1**, wherein said first data network comprises at least one of:

a wireline network;

a wireless network;

a local area network (LAN); and

a wide area network (WAN).

12. The system according to claim **1**, wherein said second network comprises at least one of:

a wireline network;

a wireless network;

a local area network (LAN); and

a wide area network (WAN).

13. The system according to claim **1**, wherein said resource allocation means optimizes end-user internet protocol (IP) quality of service (QoS).

14. The system according to claim **1**, wherein said resource allocation means is application aware.

15. The system according to claim **1**, wherein said IP flow includes at least one of:

a transmission control protocol/internet protocol (TCP/IP) flow, and

a user datagram protocol/internet protocol (UDP/IP) flow.

16. The system according to claim **1**, wherein said analyzing and scheduling means further comprises at least one of:

an identifier operative to identify said IP flow,

a characterizer operative to characterize said IP flow, and

a prioritizer device operative to prioritize said IP flow.

17. The system according to claim **16**, wherein said identifier comprises:

an analyzer device operative to analyze one or more header and payload packet fields; and

an identifier device operative to identify a new and an existing IP flow.

18. The system according to claim **17**, wherein said analyzer device comprises:

a buffer operative to buffer packets of a plurality of IP flows;

a data extraction device operative to extract data from said one or more header and payload packet fields of each of said packets; and

a packet field analyzer device operative to analyze said header and payload packet fields.

19. The system according to claim **18**, wherein said data extraction device comprises:

means for determining whether a packet of said IP flow is of version IPv.4 or IPv.6; and

means for parsing said packet of said IP flow.

20. The system according to claim **18**, wherein said packet field analyzer comprises:

determining means for determining a source application type.

21. The system according to claim **18**, wherein said data extraction device comprises:

an IP version determiner; and

a parser operative to parse said packets.

22. The system according to claim **18**, wherein said packet field analyzer comprises:

a source application type determiner operative to determine a source application type of said packets.

23. The system according to claim **20**, wherein said determining means comprises at least one of:

means for storing and retrieving a source application for a source address from a source application table;

means for determining a source application from a type of service (TOS) packet field; and

means for determining a source application from a differentiated services (DiffServ) header field.

24. The system according to claim **17**, wherein said identifier device comprises:

means for storing and retrieving an existing IP flow to and from an IP flow identification data table.

25. The system according to claim **16**, wherein said characterizer comprises:

older determining means for determining whether an age of a packet is older than a threshold age;

means for anticipating client application IP flow discards based on said age of said packet;

QoS determining means for determining a QoS requirement for said IP flow; and

means for determining a subscriber identification for one of said one or more subscriber CPE stations associated with said IP flow.

26. The system according to claim **25**, wherein said older determining means comprises:

means for analyzing a time to live (TTL) packet field for determining said age of said packet.

27. The system according to claim **25**, wherein said QoS determining means determines said QoS requirement based on at least one of:

a source address;

a destination address; and

a UDP port number,

wherein said QoS determining means comprises:

means for storing and retrieving a QoS requirement for an IP flow from an IP flow QoS requirement table.

28. The system according to claim **1**, wherein said classifier comprises:

classifying means for classifying a packet of a new IP flow into a QoS class grouping of a previously classified IP flow.

29. The system according to claim **28**, wherein said classifying means comprises:

means for determining and taking into account QoS class groupings of said previously classified IP flow.

30. The system according to claim **29**, wherein said classifying means comprises:

means for taking into account any optional differentiated services (Diff Serv) field priority marking for said previously classified IP flow.

31. The system according to claim **29**, wherein said classifying means comprises:

means for taking into account any optional type of service (TOS) field priority marking for said previously classified IP flow.

32. The system according to claim 16, wherein said prioritizer device comprises:

means for taking into account hierarchical class based priorities (HCBPs) for said IP flow.

33. The system according to claim 16, wherein said prioritizer device comprises:

means for taking into account virtual private network (VPN) priorities for said IP flow.

34. The system according to claim 16, wherein said prioritizer device comprises:

means for taking into account service level agreement (SLA) based priorities for said IP flow.

35. The system according to claim 16, wherein said prioritizer device comprises:

means for taking into account any type of service (TOS) priorities for said IP flow.

36. The system according to claim 16, wherein said prioritizer device comprises:

means for taking into account any differentiated services (DiffServ) priorities for said IP flow.

37. The system according to claim 16, wherein said identifier comprises:

packet analyzing means for analyzing one or more header and payload packet fields in said IP flow; and

distinguishing means for distinguishing between a new and an existing IP flow.

38. The system according to claim 37, wherein said packet analyzing means is located at each of said one or more subscriber CPE stations for an uplink wireless communication from said each of said one or more subscriber CPE stations to said wireless base station.

39. The system according to claim 37, wherein said distinguishing means is located at each of said one or more subscriber CPE stations for an uplink wireless communication from said each of said one or more subscriber CPE stations to said wireless base station.

40. The system according to claim 37, wherein said packet analyzing means is located at said wireless base station for a downlink wireless communication from said wireless base station to each of said one or more subscriber CPE stations.

41. The system according to claim 37, wherein said distinguishing means is located at said wireless base station for a downlink wireless communication from said wireless base station to each of said one or more subscriber CPE stations.

42. The system according to claim 37, wherein said packet analyzing means comprises:

means for buffering packets of a plurality of IP flows;

extracting means for extracting data from said packet fields of each of said packets; and

second analyzing means for analyzing said packet fields.

43. The system according to claim 42, wherein said extracting means comprises:

means for determining whether said packets are a packet version LPv.4 or IPv.6; and

means for parsing said packet fields of said plurality of IP flows.

44. The system according to claim 42, wherein said second analyzing means comprises:

determining means for determining a source application type.

45. The system according to claim 44, wherein said determining means comprises:

means for storing and retrieving a source application type to and from a source application table.

46. The system according to claim 44, wherein said determining means comprises:

means for determining a source application from a type of service (TOS) packet field..

47. The system according to claim 44, wherein said determining means comprises:

means for determining a source application from a differentiated services (DiffServ) packet field.

48. The system according to claim 44, wherein said determining means comprises:

means for determining a source application from information provided by a direct application conduit.

49. The system according to claim 37, wherein said distinguishing means comprises:

means for storing and retrieving identification information for an existing IP flow to and from an IP flow identification data table.

50. The system according to claim 16, wherein said identifier comprises:

determining means for determining whether said IP flow is known to the system based on a packet received over said shared wireless bandwidth; and

source identifying means for identifying a source application having transmitted said received packet.

51. The system according to claim 50, wherein said determining means is located at said wireless base station for a downlink wireless communication from said wireless base station to said one or more subscriber CPE stations.

52. The system according to claim 50, wherein said determining means is located at each of said one or more subscriber CPE stations for an uplink wireless communication from said each of said one or more subscriber CPE stations to said wireless base station.

53. The system according to claim 50, wherein said identifying means is located at said wireless base station for a downlink wireless communication from said wireless base station to said one or more subscriber CPE stations.

54. The system according to claim 50, wherein said identifying means is located at each of said one or more subscriber CPE stations for an uplink wireless communication from said each of said one or more subscriber CPE stations to said wireless base station.

55. The system according to claim 50, wherein said determining means comprises:

means for buffering said packet;

means for extracting identification information from one or more header and payload packet fields of said packet;

means for performing a lookup of an existing IP flow identifier using said identification information in an existing IP flow data table to determine whether the IP flow is known to the system.

56. The system according to claim 50, wherein said source identifying means comprises:

means for buffering said packet;

means for extracting information from one or more header and payload packet fields of said packet;

means for performing a lookup of a source application type using said information in a source application data table to identify said source application.

57. The system according to claim 16, wherein said characterizer comprises:

age determining means for determining whether an age of a packet is older than a threshold age.

58. The system according to claim 57, wherein said age determining means comprises:

means for analyzing a time to live (TTL) packet field for determining said age of said packet.

59. The system according to claim **57**, wherein said age determining means comprises:

means for anticipating application IP flow discards based on said age of said packet.

60. The system according to claim **16**, wherein said characterizer comprises:

QoS determining means for determining a QoS requirement for said IP flow if said IP flow is a new IP flow.

61. The system according to claim **16**, wherein said characterizer comprises:

means for determining a subscriber CPE identification for said one or more subscriber CPE stations associated with said IP flow if said IP flow is a new IP flow.

62. The system according to claim **60**, wherein said QoS determining means comprises:

means for determining QoS requirements based on at least one of:

a source address,

a destination address, and

a UDP port number.

63. The system according to claim **60**, wherein said QoS determining means comprises:

means for storing and retrieving a QoS requirement for an IP flow from an IP flow QoS requirement table.

64. The system according to claim **16**, wherein said classifier comprises:

means for associating a packet of an existing IP flow with said IP flow.

65. The system according to claim **16**, wherein said classifier comprises:

a QoS grouping device operative to group a packet of a new IP flow into a QoS class grouping.

66. The system according to claim **3**, wherein said QoS grouping device comprises:

a determining device operative to determine and take into account QoS class groupings for said IP flow.

67. The system according to claim **66**, wherein said QoS grouping device comprises:

an optional differentiated services (Diff Serv) device operative to take into account an optional Diff Servs field priority marking for said IP flow.

68. The system according to claim **66**, wherein said QoS grouping device comprises:

an optional type of service (TOS) device operative to take into account any optional type of service (TOS) field priority marking for said IP flow.

69. The system according to claim **16**, wherein said prioritizer comprises:

a hierarchical class based priority (HCBP) prioritizer operative to prioritize said IP flow based on a HCBP priority of said IP flow.

70. The system according to claim **69**, wherein said HCBP prioritizer comprises:

a class based priority limits operative to establish limits for each of said HCBP priorities.

71. The system according to claim **16**, wherein said prioritizer device comprises:

a virtual private network (VPN) prioritizer operative to prioritize a plurality of IP flows based on their source being a VPN.

72. The system according to claim **71**, wherein said virtual private network (VPN) prioritizer grants preferential priority to said plurality of IP flows associated with said VPN.

73. The system according to claim **71**, wherein said virtual private network (VPN) prioritizer grants preferential priority to at least one of:

said VPN IP flows of a particular IP flow type, and

said VPN IP flows from a type of VPN.

74. The system according to claim **73**, wherein said type of VPN comprises:

a directory enabled networking (DEN) table management scheme type.

75. The system according to claim **16**, wherein said prioritizer device comprises:

a service level agreement (SLA) based prioritizer operative to prioritize said IP flow based on an SLA level of a subscriber source of said IP flow.

76. The system according to claim **75**, wherein said SLA level comprises at least one of a premium level, a standard level and a value level.

77. The system according to claim **16**, wherein said prioritizer device comprises:

a type of service (TOS) prioritizer that prioritizes said IP flow based on a TOS marking of a packet of said IP flow.

78. The system according to claim **16**, wherein said prioritizer device comprises:

a differentiated services (DiffServ) prioritizer that prioritizes said IP flow based on a DiffServ marking of a packet of said IP flow.

79. The system according to claim **16**, wherein said prioritizer device comprises:

a weighted fair priority (WFP) prioritizer that ensures fair distribution of said shared bandwidth, that sets reservation policy limits based on IP flow priorities.

80. The system according to claim **75**, wherein said SLA based prioritizer comprises:

means for analyzing said SLA level for said IP flow.

81. The system according to claim **80**, comprising:

means for prioritizing said IP flow based on one or more subscriber-defined parameters.

82. The system according to claim **75**, wherein said SLA level comprises at least one of:

a premium service level;

a normal service level; and

a value service level.

83. The system according to claim **75**, wherein said SLA level is used to provide at least one of:

differing traffic rates between SLA subscribers;

network availability for said SLA subscribers;

increased bandwidth for said SLA subscribers;

decreased error rates for said SLA subscribers;

latency guarantees for said SLA subscribers; and

jitter guarantees for said SLA subscribers.

84. The system according to claim **1**, wherein said resource allocation means comprises:

assigning means for assigning future slots of a transmission frame to a data packet in the transmission frame for transmission over said wireless medium.

85. The system according to claim **84**, wherein said assigning means comprises:

means for applying an advanced reservation algorithm;

first reserving means for reserving a first slot for a first data packet of said IP flow in a future transmission frame based on said advanced reservation algorithm; and

second reserving means for reserving a second slot for a second data packet of said IP flow in a transmission frame subsequent in time to said future transmission frame based on said advanced reservation algorithm,

wherein said second data packet is placed in said second slot in an isochronous manner to the placement of said first data packet in said first slot.

86. The system according to claim **85**, wherein there is a periodic variation between the placement of said first data packet in said first slot and the placement of said second data packet in said second slot.

87. The system according to claim **85**, wherein there is an aperiodic variation between the placement of said first data packet in said first slot and the placement of said second data packet in said second slot.

88. The system according to claim **85**, wherein said advanced reservation algorithm determines whether said IP flow is jitter-sensitive.

89. The system according to claim **1**, wherein said resource allocation means comprises:

means for accounting for hierarchical class based priorities (HCBPs) for said IP flow.

90. The system according to claim **1**, wherein said resource allocation means comprises:

means for accounting for virtual private network (VPN) priorities for said IP flow.

91. The system according to claim **1**, wherein said resource allocation means comprises:

means for accounting for service level agreement (SLA) based priorities for said IP flow.

92. The system according to claim **1**, wherein said resource allocation means comprises:

means for accounting for any type of service (TOS) priorities for said IP flow.

93. The system according to claim **1**, wherein said resource allocation means comprises:

means for accounting for any differentiated services (DiffServ) priorities for said IP flow.

94. The system according to claim **85**, further comprising:

means for providing a periodic variation between the placement of said first data packet in said first slot and the placement of second data packet in said second slot.

95. The system according to claim **85**, further comprising:

means for providing an aperiodic variation between the placement of said first data packet in said first slot and the placement of second data packet in said second slot.

96. The system according to claim **85**, wherein said advanced reservation algorithm comprises:

means for determining whether said IP flow is jitter-sensitive.

97. The system according to claim **85**, comprising: means for providing no periodic variation between successive reservations of succeeding slots.

98. The system according to claim **85**, comprising:

means for providing a periodic variation between successive reservations of succeeding slots.

99. The system according to claim **85**, wherein said advanced reservation algorithm comprises:

means for determining whether said IP flow is jitter-sensitive.

100. The system according to claim **16**, wherein said analyzing and scheduling means comprises an analyzer and a scheduler,

wherein said analyzer is operative to identify IP-priority packet IP flow identification information and to classify said IP flow, and

said scheduler is operative to prioritize said IP flow and operative to take into account said IP priority header identification information.

101. The system according to claim **100**, wherein said IP-priority packet IP flow identification information comprises a determiner operative to determine and to take into account QoS class groupings for said IP flow.

102. The system according to claim **100**, wherein said IP-priority packet IP flow identification information comprises a TOS prioritizer operative to account for any optional type of service (TOS) field priority marking.

103. The system according to claim **102**, wherein said type of service (TOS) field priority marking is compatible with Internet Engineering Task Force (IETF) RFC 1992b.

104. The system according to claim **103**, wherein said type of service (TOS) field priority marking is compatible with IETF RFC 1349.

105. The system according to claim **104**, wherein said marking comprises:

a minimize delay marking;

a maximize throughput marking;

a maximize reliability marking;

a minimize monetary cost marking; and

a normal service marking.

106. The system according to claim **100**, wherein said IP-priority packet header IP flow identification information comprises a DiffServ prioritizer operative to account for any optional differential service (Diff Serv) field priority marking.

107. The system according to claim **106**, wherein said Diff Serv field priority marking is compatible with Internet Engineering Task Force (IETF) RFC 2474.

108. The system according to claim **106**, wherein said Diff Serv field priority marking is compatible with IETF RFC 2475.

109. The system according to claim **100**, wherein said IP-priority packet header IP flow identification information comprises means for taking into account any resource reservation protocol (RSVP) messages and objects.

110. The system according to claim **109**, wherein said RSVP protocol messages include any of the following:

path messages;

reservation (Resv);

path teardown messages;

resv teardown messages;

path error messages; and

confirmation messages.

111. The system according to claim **109**, wherein said RSVP protocol objects include any of the following:

null;

session;

RSVP_hop;

time_values;

style;

flowspec;

sender_template;

sender_Tspec;

Adspec;

Error_Spec;

Policy_data;

Integrity;

Scope; and

Resv_Confirm.

112. The system according to claim 106, wherein said RSVP marking is compatible with Internet Engineering Task Force (IETF) RFC 2205.

113. The system according to claim 33, comprising:

means for analyzing said virtual private network (VPN) priorities for said IP flow.

114. The system according to claim 113, comprising:

means for prioritizing all VPN IP flows.

115. The system according to claim 113, comprising:

means for prioritizing said IP flow based on one or more subscriber-defined parameters.

116. The system according to claim 33, wherein said VPN comprises a directory enabled networking (DEN) table management scheme.

117. The system according to claim 33, wherein said VPN is implemented using a point-to-point tunneling protocol (PPTP).

118. The system according to claim 1, wherein said system is used in a point to point (PtP) telecommunications system.

119. The system according to claim 118, wherein said shared wireless bandwidth comprises a wireless communication medium comprising at least one of:

a radio frequency (RF) communications medium;

a cable communications medium; and

a satellite communications medium.

120. The system according to claim 119, wherein said wireless communication medium further comprises, a telecommunications access method including at least one of:

a time division multiple access (TDMA) access method;

a time division multiple access/time division duplex (TDMA/TDD) access method;

a code division multiple access (CDMA) access method; and

a frequency division multiple access (FDMA) access method.

121. The system according to claim 118, wherein said first data network comprises at least one of:

a wireline network;

a wireless network;

a local area network (LAN); and

a wide area network (WAN).

122. The system according to claim 118, wherein said second network comprises at least one of:

a wireline network;

a wireless network;

a local area network (LAN); and

a wide area network (WAN).

123. The system according to claim 118, said resource allocation means comprises a resource allocator that allocates shared bandwidth between said wireless base station and one of said subscriber CPE stations.

124. The system according to claim 123, wherein said resource allocator optimizes end-user internet protocol (IP) quality of service (QoS).

125. The system according to claim 123, wherein said resource allocator is application aware.

126. The system according to claim 1, wherein said system is a broadband coaxial cable telecommunications system wherein said wireless medium comprises a coaxial cable communications medium.

127. The system according to claim 1, wherein the system is a wireless local area network (LAN) point to multi-point (PMP) system.

128. The system according to claim 1, wherein said wireless base station is a wireless access point.

129. The system according to claim 126, wherein said resource allocation means comprises a coaxial cable resource allocator for allocating shared bandwidth between said wireless base station and said subscriber CPE stations.

130. The system according to claim 129, wherein said coaxial cable resource allocator optimizes end-user internet protocol (IP) quality of service (QoS).

131. The system according to claim 126, wherein said coaxial cable communications medium comprises a radio frequency data communication over a coaxial cable, wherein one or more cable modems modulate and demodulate signals transmitted over said coaxial cable communications medium.

132. The system according to claim 131, wherein said cable modem is DOC/SYS compliant.

133. The system according to claim 130, wherein said end-user IP QoS optimized coaxial cable resource allocator system comprises:

an IP flow identifier;

an IP flow characterizer;

an IP flow classifier; and

an IP flow prioritizer.

134. The system according to claim 129, wherein said coaxial cable communications medium comprises, a telecommunications access method including at least one of:

a time division multiple access (TDMA) access method;

a time division multiple access/time division duplex (TDMA/TDD) access method;

a code division multiple access (CDMA) access method; and

frequency division multiple access (FDMA) access method.

135. The system according to claim 126, wherein said first data network comprises at least one of:

a wireline network;

a wireless network;

a local area network (LAN); and

a wide area network (WAN).

136. The system according to claim 126, wherein said second network comprises at least one of:

a wireline network;

a wireless network;

a local area network (LAN); and

a wide area network (WAN).

137. The system according to claim 129, wherein said coaxial cable resource allocator is application aware.

138. The system according to claim 129, wherein the system is used in a point to point (PtP) network.

139. The system according to claim 1, wherein said resource allocation means is a part of a media access control (MAC) layer.

140. The system according to claim 9, wherein said wireless communication medium further comprises a telecommunications access method comprising a time division multiple access/time division duplex (TDMA/TDD) access method and wherein the system further comprises a TDMA/TDD media access control (MAC) transmission frame, comprising:

one or more dynamically allocatable IP flow control slots for providing IP flow control information over a wireless medium between said wireless base station and said one or more subscriber customer premises equipment (CPE) stations; and

one or more dynamically allocatable IP flow data slots for providing IP flow data information over said wireless communication medium between said wireless base station and said one or more subscriber customer premises equipment (CPE) stations.

141. The system according to claim **140**, wherein said control slots comprise at least one of:

a downstream acknowledgment slot;

a reservation request slot;

an operations data slot;

an upstream acknowledgment slot;

an acknowledgment request slot;

a frame descriptor slot; and

a command and control slot.

142. The system according to claim **140**, wherein said data slots comprise at least one of:

uplink data slots for transmission in an uplink direction from each of said one or more subscriber CPE stations to said wireless base station; and

downlink data slots for transmission in a downlink direction from said wireless base station to each of said one or more subscriber CPE stations.

143. The system according to claim **140**, wherein said time division multiple access/time division duplex (TDMA/TDD) transmission media access method involves:

a downlink subframe for use over said wireless medium from said wireless base station and said one or more subscriber customer premises equipment (CPE) stations; and

an uplink subframe for use over said wireless medium from said subscriber CPE stations to said wireless base station,

wherein a bandwidth is dynamically allocated between said downlink subframe and said uplink subframe for transmission of Internet protocol (IP) flow information so as to optimize end-user IP quality of service (QOS).

144. The system according to claim **143**, wherein multiple slots in said downlink subframe are scheduled for one of said one or more subscriber CPE stations for a single internet protocol (IP) flow.

145. The system according to claim **143**, wherein multiple slots in said downlink subframe are scheduled for one of said one or more subscriber CPE stations for a plurality of Internet protocol (IP) flows.

146. The system according to claim **143**, wherein multiple slots in said uplink subframe are scheduled for one of said one or more subscriber CPE stations for a single internet protocol (IP) flow.

147. The system according to claim **143**, wherein multiple slots in said uplink subframe are scheduled for one of said one or more subscriber CPE stations for a plurality of internet protocol (IP) flows.

148. The system according to claim **143**, further comprising:

one or more dynamically allocatable reservation request contention slots for addressing contentions between reservation requests for available slots in said uplink subframe between said wireless base station and each of said one or more subscriber CPE stations for transmission of IP flows.

149. The system according to claim **143**, wherein said contention slots are dynamically allocated according to the frequency of detected collisions between said reservation requests.

150. The system according to claim **143**, further comprising:

a frame descriptor block for transmitting one or more reservation slots in said downlink subframe defining where each of said one or more subscriber CPE stations requesting a reservation will place uplink data thereof.

151. The system according to claim **140**, wherein said IP flow control slots comprise at least one of:

a downstream acknowledgment slot;

an operations data slot;

an upstream acknowledgment slot;

an acknowledgment request slot; and

a frame descriptor slot.

152. The system of claim **1**, further comprising a TCP adjunct system that prevents operation of a transmission control program (TCP) sliding window algorithm that controls a TCP transmission rate in said packet-centric wireless point to multi-point telecommunications system, said TCP adjunct system comprising:

a TCP adjunct agent that takes into account application awareness, guarantees enduser quality of service (QoS), and prevents operation of a TCP sliding window algorithm that controls a TCP transmission rate in a manner that optimizes for a wireless communication medium.

153. The system of claim **152**, wherein said TCP adjunct agent obviates modification of a source TCP layer at a first of said one or more host workstations and a destination TCP layer at one of said one or more subscriber workstations.

154. The system of claim **153**, wherein said source and destination TCP layers are unaware of operation modification by said TCP adjunct agent.

155. The system of claim **152**, wherein said TCP adjunct agent is configured to intercept retransmission requests between a TCP layer of one of said subscriber workstations coupled to a first subscriber CPE station and a TCP layer of at least one of a host workstation and said wireless base station.

156. An IP flow classification system that groups IP flows in a packet-centric wireless point to multi-point telecommunications system, said classification system comprising:

a wireless base station coupled to a first data network;

one or more host workstations coupled to said first data network;

one or more subscriber customer premise equipment (CPE) stations in wireless communication with said wireless base station over a shared wireless bandwidth using a packet-centric protocol over a wireless communication medium;

one or more subscriber workstations coupled to each of said subscriber CPE stations over a second network;

a resource allocator operative to optimize end-user quality of service (QoS) and allocating shared bandwidth among said subscriber CPE stations; and

an analyzer and scheduler operative to analyze and schedule an internet protocol (IP) flow over said shared wireless bandwidth, wherein said analyzer and scheduler comprises:

a classifier that classifies said IP flow.

157. The system of claim **156**, wherein said classifier comprises:

an association device that associates a packet of an existing IP flow with said IP flow.

158. The system of claim **156**, wherein said classifier comprises:

QoS grouping device that groups a packet of a new IP flow into a QoS class grouping.

159. The system of claim 158, wherein said QoS grouping device comprises:

determining device that determines and takes into account QoS class groupings for said IP flow.

160. The system of claim 159, wherein said QoS grouping device comprises:

optional differentiated services (Diff Serv) device that takes into account an optional Diff Servs field priority marking for said IP flow.

161. The system of claim 159, wherein said QoS grouping device comprises:

optional type of service (TOS) device that takes into account any optional type of service (TOS) field priority marking for said IP flow.

162. The system according to claim 156, wherein the wireless point to multi-point telecommunications system is a wireless local area network (LAN) system.

163. The system according to claim 156, wherein the wireless point to multi-point telecommunications system is a wireless wide area network (WAN) system.

* * * * *

US006466985B1

(12) **United States Patent**
Goyal et al.

(10) **Patent No.: US 6,466,985 B1**
(45) **Date of Patent: Oct. 15, 2002**

(54) **METHOD AND APPARATUS FOR PROVIDING QUALITY OF SERVICE USING THE INTERNET PROTOCOL**

(75) Inventors: **Pawan Goyal**, Mountain View, CA (US); **Gisli Hjalmtysson**, Gillette, NJ (US)

(73) Assignee: **AT&T Corp.**, New York, NY (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/288,617**

(22) Filed: **Apr. 9, 1999**

Related U.S. Application Data

(60) Provisional application No. 60/081,479, filed on Apr. 10, 1998.

(51) **Int. Cl.**[7] **G06F 13/00**

(52) **U.S. Cl.** **709/238**; 709/236; 709/246; 370/236

(58) **Field of Search** 709/230, 232, 709/234, 235, 236, 237, 238, 245, 246, 250, 311; 370/230, 231, 235, 236

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,920,705 A * 7/1999 Lyon et al. 370/409
5,963,555 A * 10/1999 Takase et al. 370/395
6,084,855 A * 7/2000 Soirinsuo et al. 370/235

* cited by examiner

Primary Examiner—Viet D. Vu

(57) **ABSTRACT**

A method and apparatus for communicating information in a network is described. A packet for the information is generated at a first network device. The first network device assigns a flow label to the packet. The flow label indicates that the packet is part of a particular sequence of packets. The first network device also assigns a direction to the packet by, for example, setting a bit in the flow label. The packet is then sent to a second network device through at least one intermediate network device. This process is continued for the entire sequence of packets. The intermediate network device actually routes the packets to the second network device. The intermediate network device receives the packets at an input port. A flow label is identified for each packet. The intermediate network device determines whether a flow table has an entry for the flow label. If there is no present entry for the flow label in the flow table, an entry for the flow label is created. If there is an entry for the flow label, an output port associated with the flow label is obtained. The intermediate network device then sends the packet to the output port. This continues at each intermediate network device until each packet reaches the second network device.

15 Claims, 2 Drawing Sheets

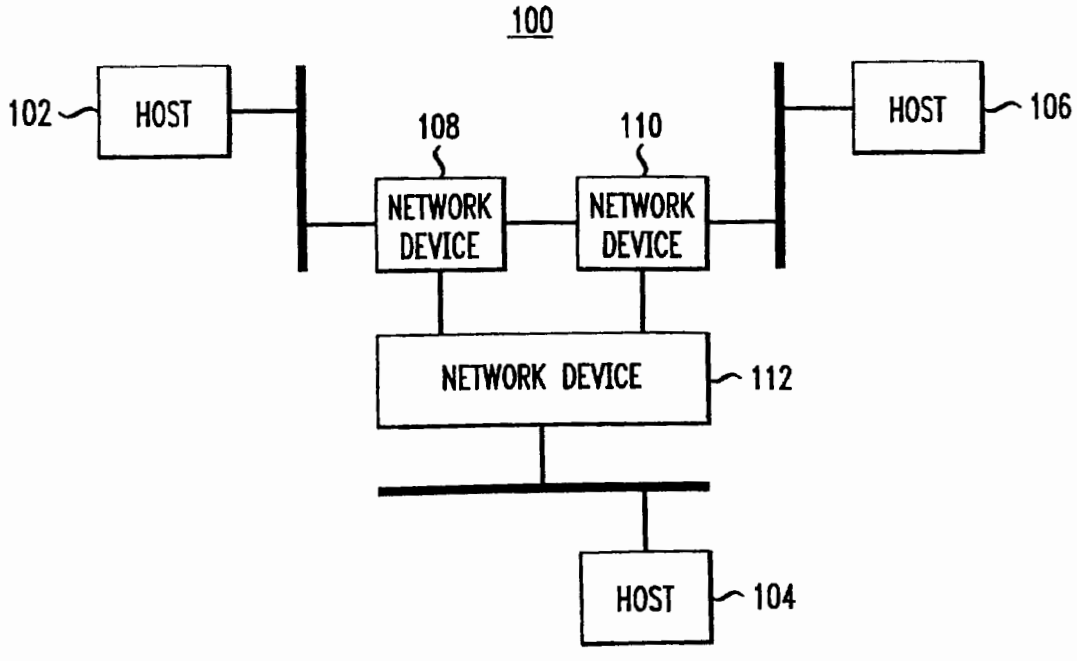


FIG. 1

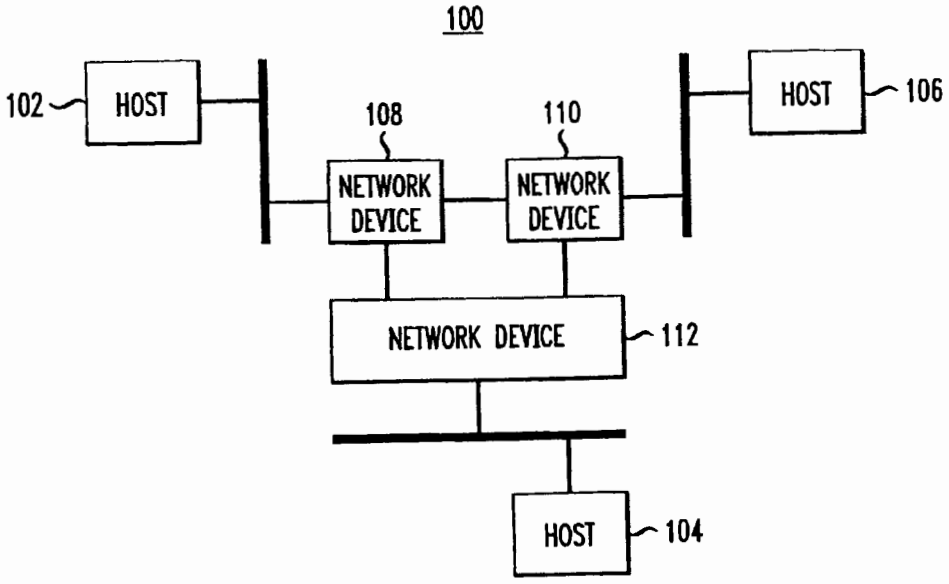
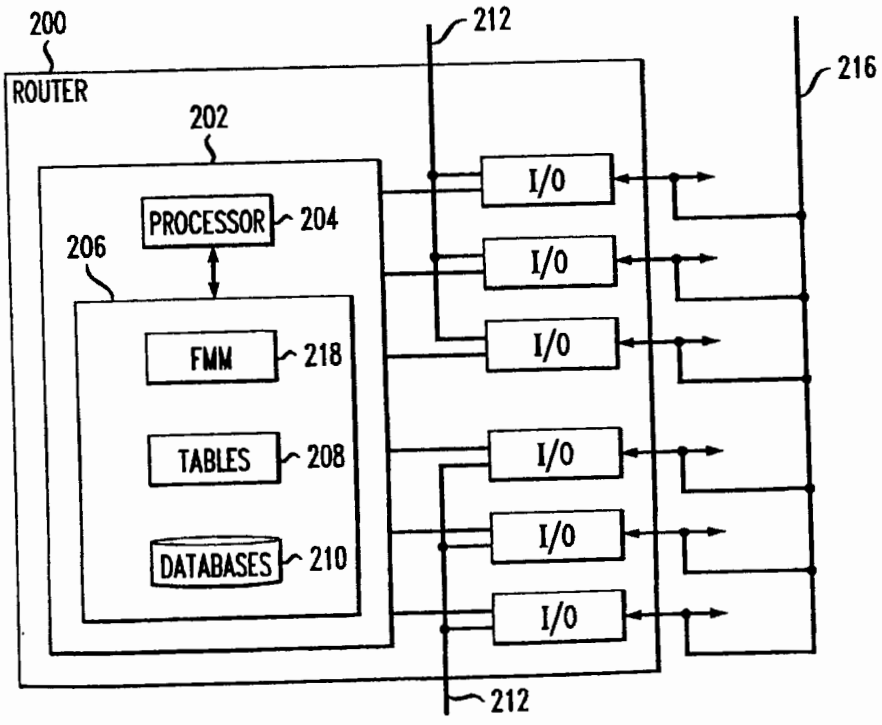

100
102
HOST
108
NETWORK DEVICE
110
NETWORK DEVICE
HOST
106
NETWORK DEVICE
112
HOST
104

FIG. 2


200
ROUTER
212
216
202
PROCESSOR
204
206
FMM
218
TABLES
208
DATABASES
210
I/O
I/O
I/O
I/O
I/O
I/O
212

METHOD AND APPARATUS FOR PROVIDING QUALITY OF SERVICE USING THE INTERNET PROTOCOL

CROSS-REFERENCE TO RELATED APPLICATION

This application claims the benefit of U.S. Provisional Application No. 60/081,479, filed Apr. 10, 1998, entitled "Provision of Quality Services Using the Internet Protocol," the entire disclosure of which is hereby incorporated by reference.

FIELD OF THE INVENTION

The embodiments of the invention relate to communications in general. In particular, the embodiments of the invention relate to a method and apparatus for providing quality of service using the Internet Protocol (IP).

BACKGROUND OF THE INVENTION

With the explosive growth of the Internet and new network applications almost exclusively written for the Internet Protocol (IP), it has become essential to optimize protocols and network management for the IP. Originally designed for data networking, the Internet is increasingly being used for audio and video applications. Whereas using a single network level technology may potentially simplify network management, providing sufficient service quality for multimedia applications over the Internet remains a significant challenge. After almost a decade of emphasis on resource reservations and end-to-end Quality of Service (QoS), as part of both the design of Asynchronous Transfer Mode (ATM) networks and standardization of the Resource Reservation Protocol (RSVP), there is now a significant backlash against these state rich, fine grained QoS models. This is in part based on the observation that rather than unavailability of bandwidth, service instability is causing problems to multimedia applications. Current efforts on differentiated services are an attempt to develop a service model that improves the service quality of the Internet while acting at aggregate levels. However ensuring stable service level requires richer traffic management facilities than currently available in the Internet.

The two most essential characteristics of the IP that have contributed to its success and distinguish it from connection oriented networks are the softness of state inside the network, and the aggregation properties of this state. Apart from tbe routing database, for best effort destination based routing, (cached) state is used purely for performance enhancement, but is not essential for correctly delivering packets to destination. In particular this state can be lost, or removed at routers discretion without affecting the validity of state elsewhere in the network. With single class destination based routing, prefix matching effectively aggregates the forwarding information for multiple destinations into a single entry per prefix. Even with nodal service differentiation (ToS bits) this property is retained.

In contrast, the strengths of the state rich telephony network, and derived connection-oriented models (e.g., ATM), are their service quality assurances. In part, the quality assurances are achieved through resource reservations and tight channel scheduling, based on declared or inferred user objectives. In part, the consistent quality is achieved through network management; exploiting intra-domain knowledge about network load and conditions. In addition connection oriented models enhance stability, as the time-scale of (load) change becomes that of connection duration, rather than that of routing updates and packet interarrivals. Whereas traditional methods exploit mechanisms for connectivity and reservations to achieve quality of service and implement traffic management, the coupling of these distinct mechanisms has contributed to the perceived complexity of connection oriented networks.

In view of the foregoing, it can be appreciated that a substantial need exists for introducing the QoS advantages of connection-oriented networks into connectionless networks (e.g., using IP) without losing the advantages given by connectionless networks.

SUMMARY OF THE INVENTION

One embodiment of the invention comprises a method and apparatus for communicating information in a network. A packet for the information is generated at a first network device such as an end system. The first network device assigns a flow label to the packet. The flow label indicates that the packet is part of a particular sequence of packets. The first network device also assigns a direction to the packet by, for example, setting a bit in the flow label. The packet is then sent to a second network device (e.g., another end system) through at least one intermediate network device (e.g., a router or switch). This process is continued for the entire sequence of packets for a given flow.

The intermediate network device actually routes the packets to the second network device. The intermediate network device receives the packets at an input port. A flow label is identified for each packet. The intermediate network device determines whether a flow table has an entry for the flow label. If there is no present entry for the flow label in the flow table, an entry for the flow label is created. If there is an entry for the flow label, an output port associated with tbe flow label is obtained. The intermediate network device then sends the packet to the output port. This continues at each intermediate network device until each packet of a given flow reaches the second network device.

With these and other advantages and features of the invention that will become hereinafter apparent, the nature of the invention may be more clearly understood by reference to the following detailed description of the invention, the appended claims and to the several drawings attached herein.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a block diagram of a network suitable for practicing one embodiment of the invention.

FIG. **2** is a block diagram of a router suitable for practicing one embodiment of tbe invention.

FIG. **3** is a block flow diagram of the steps performed by a first network device in accordance with one embodiment of the invention.

FIG. **4** is a block flow diagram of the steps performed by an intermediate network device in accordance with one embodiment of the invention.

DETAILED DESCRIPTION

The embodiments of the invention are directed to enhancing the consistency in service quality on the Internet. Whereas the soft-state and scalability have been key to the success of the Internet, its service quality is wanting. The embodiments borrow some of the concepts from connection-oriented networks, without compromising on the essential characteristics of IP. The embodiments are optimized for

carrying IP "flows," and may be implemented as part of lower level (layer 2) protocols, including ATM or Multiprotocol Label Switching (MPLS). One particular advantageous embodiment of the invention creates the capability of running the IP directly on top of the optical layer of a network.

The embodiments of the invention provide QoS capability using the IP, particularly as set forth in Request For Comments (RFC) 1883 titled "Internet Protocol, Version 6 (IPv6) Specification," dated December 1995 ("IPv6"), which is incorporated by reference herein. By introducing QoS characteristics using the IP, the need for carrying IP datagrams in lower layer protocol units A ATM cells) is avoided. This reduces network complexity by removing a network layer, which in turn simplifies network and service management. Furthermore, using IP also avoids the complexity of mapping customer requests (at the network interface) onto requests on the physical backbone infrastructure that may have conflicting service models.

The embodiments of the invention explicitly identify a group of IP packets as belonging to a "flow," and ensures that packets of a flow traverse the same sequence of routers. This creates some unique advantages, such as load balancing, enhanced management and accountability, the ability to assign attributes to a flow, amortizing costly (in terms of bandwidth) operations over a sequence of packets, and providing symmetric and reverse path routing.

A first advantage is load balancing. In current IP, all the packets from a source intended for a given destination traverse the same path. This technique does not optimally utilize the network resources. Furthermore, if a link on the path gets overloaded, a router may reroute either all or part of the traffic to a given destination. This leads to either routing instability or route fluttering, neither of which is desirable for a large class of applications. This can be overcome using "route pinning." Route pinning involves ensuring that packets for a particular flow traverse the same sequence of intermediate network devices (e.g., routers). Route pinning enables the network to better utilize its resources and avoid the route-flapping problem while ameliorating routing instability.

A second advantage is that the embodiments of the invention enhance management and accountability within the network. Recording of network usage, whether for billing or off-line diagnostic analysis, is an important part of providing network services. This is particularly important in a network that provides some assurance regarding quality. Labeled flows traversing a fixed path enable this functionality.

A third advantage is that certain attributes, such as QoS attributes, can be assigned to designated flows. For a number of reasons, it may be desirable to assign attributes to distinguished flows, for example to reserve resources along the path. This requires a mechanism to declare a path as "special" and then to describe (and possibly negotiate) the path attributes. RSVP is designed for this purpose. The use of RSVP with the current IP (e.g., IPv4), however, is particularly complex without some mechanism to declare a path as "special." This problem is overcome somewhat using the flow label already defined for IPv6.

A fourth advantage is that costly operations in terms of bandwidth can be amortized over a sequence of packets. Assignment of flow attributes (for example, reservations), requires some mechanism to establish state and share that state across sequence of datagrams. State sharing fails if the forwarding path is not stable across multiple packets. Installing state is expensive, both in latency (end-to-end consistency) and in processing (admission control, and negotiations). Thus, it is desirable that forwarding path of a flow be stable. Stable forwarding paths also enable other optimizations. For example, though explicit routes can be specified in every IPv6 packet, significant bandwidth savings can be accrued when stability of a flow path can be assumed, as with using, for example, the concept of route pinning. The same concept applies more generally to extension headers.

In addition, the embodiments of the invention offer the advantage of symmetric and reverse path routing. Symmetric routing is desirable because it simplifies algorithms and provides benefits to some services. Reverse path routing is necessary for flow level control where it must be possible to send control messages on the reverse forwarding path. Part of the complexity of RSVP is due to mechanisms to provide a control path on the reverse data path. Explicit labeling of flows will enable routing symmetry.

The embodiments of the invention provide the fundamental strengths of connections while retaining the softness of state and aggregation properties of connectionless networks. The embodiments of the invention makes use of flows that can be designated as "special" or "distinguished" through the use of a unique identifier or label for each flow. A distinguished flow can then be pinned to a route, support reverse path routing, and may have associated (QoS) attributes.

A new flow can be established by using a previously undeclared flow name. Assignment of a name declares the intent to use this flow for something special, and enables the end-system to refer to the flow for later attribute assignment. An unknown flow name is interpreted as a request for a new flow.

In addition to the flow name, the first packet of a flow (effectively the flow-request) contains a datagram of a network level protocol for which the network node can do routing (e.g., IPv6). Although subsequent packets may contain arbitrary datagrams, softness of state is achieved when all the transferred packets are of such "known" protocol. In that case, if the state is lost, the next packet of the named flow is processed as if it were the first packet of a new flow. To optimize the transport network for IPv6, the network nodes simply support IPv6 routing. The flows defined herein incur no call setup delay. Moreover, adopting (or assuming) a "use it or loose it" state invalidation policy, there is no need for explicit tear-down. A flow may be uni- or bidirectional.

There are four aspects of constructing a flow: (1) declaring a name; (2) pinning the route, (3) enabling reverse path routing, and (4) assigning attributes (such as QoS). Abstractly, current network nodes maintain two tables, a routing table and a forwarding table. In the case of a traditional router the forwarding table corresponds to the routing cache. On an ATM switch or an MPLS Label Switch Router (LSR), the forwarding table is respectively the Virtual Channel (VC) lookup table or the label lookup table. To support the flows used in the embodiments of the invention, a traditional router would be augmented with an additional forwarding table for mapping flow names to flow state (including the output port), as discussed in more detail with reference to FIG. 2.

A flow request may be interpreted as an implicit request for route pinning. If not, route pinning may be requested subsequently in a separate message. Without route pinning, the entry in the flow cache simply points to the corresponding entry (in the regular cache) for the destination address. When route pinning is requested this entry is copied and thus becomes independent of changes in the default destination based route.

Reverse path routing on a flow requires the node to record the incoming port as a part of the flow forwarding state. In addition this information must be conveyed to the output port. This given, however, reverse path routing may be achieved either by constructing a new path in the reverse direction (ie., a path association) or by a naming convention allowing a name to be resolved in the reverse path name space.

Other flow attributes are signaled separately, and can be processed with specialized software using a conventional processor, or by a dedicated attribute control processor. Flows maintain a control mapping separate from the forwarding map, thus supporting control paradigms allowing service specific controllers, potentially installed on demand, to process the attribute messages.

To aggregate state, a node (e.g., a backbone border node) may aggregate smaller flows and tunnel the aggregate flow to a particular node in the network (an egress router for example). After exiting the tunnel, the data packets would then be routed to their respective destinations as if they had originated at the tunnel end. To construct the tunnel, the router precedes the user packets with a "tunnel request" packet, a datagram of the "known" protocol, distinguished as a tunnel request. In the case of the IP as the known protocol, explicit routing could be specified using the source routing option (extension header for IPv6). Of course this is complementary to the use of normal tunneling, which themselves might exploit flows as well.

The IPv6 flow label can be used to implement a flow in an IPv6 network. To define a name an end-system sets a locally unique flow label on a packet intended for that flow. Hop-by-hop extension headers are used to assign attributes to the route. For example, the hop-by-hop router alert extension can be used, although additional extension headers are possible as well.

For network centric flows in IPv6, a router may identify a sequence of packets whose flow label is not set and aggregate them into a tunneled flow. The tunnel request packet is an IPv6 datagram carrying the assigned flow label. The router then sets the flow label of subsequent packets, which are then nullified on exit from the tunnel. To aggregate labeled flows we use traditional IP tunneling with flows.

The embodiments of the invention modify the current definition of the IPv6 flow label for reverse path forwarding. The first bit of the flow label specifies whether it is source or destination unique, with a zero (0) implying a destination unique flow label, and a one (1) declaring a source unique flow label. To send on the reverse path, the receiver flips the first bit of the flow label. Since the source and destination are also swapped (as compared to the received packet), the same address is used with the flow label to uniquely identify the flow.

Implementing the flow concept in MPLS is similar to that for IPv6. A sender uses a label to define a flow name. As labels are "link local" this amounts to upstream label allocation. Flow pinning is implemented as with IPv6. Attributes are signaled using hop-by-hop router alerts. Reverse path is accomplished in a manner similar to IPv6, that is, dividing the namespace on each link into two, with a direction distinguished by the leading bit. It is also possible, however, to use a separate flow name to associate the reverse path to the corresponding label path. Furthermore, this flow association might be maintained only at the higher level, and not be explicit at the label path level. This could for example be the case if running IPv6 over MPLS.

The embodiments of the invention aggregate separate flows to a tunnel in MPLS by using a label stack. A label is pushed on the stack at the entry of the tunnel, and popped off on exit. As the tunnel end may in effect be a multiplexing point (i.e., a virtual termination of many tunnels) the tunnel label must be assigned by the tunnel termination node (downstream allocation), to ensure that the enclosed labels further down on the stack are uniquely resolved. To avoid subsequent round trip delays, however, the first request may yield two labels, allowing the entry point node to maintain a cache one label for subsequent tunnel construction.

Referring now in detail to the drawings wherein like parts are designated by like reference numerals throughout, there is illustrated in FIG. **1** a network suitable for practicing one embodiment of the invention. FIG. **1** illustrates an exemplary network **100** having multiple hosts and multiple intermediate network devices connected as shown. Network **100** shown in FIG. **1** represents one possible network configuration and will be used to describe the operation of the invention. Specifically, three intermediate network devices **108, 110** and **112** are coupled to one another as shown. FIG. **1** also illustrates three host devices **102, 104** and **106**. Each host is coupled to a particular intermediate network device using an interface **120** (not shown). Interface **120** may be any type of interface circuit, including a network, capable of coupling one or more hosts to an intermediate network device. Alternatively, interface **120** may be omitted, and the host (or hosts) coupled directly to the intermediate network device. To simplify the illustration, only one host device is shown coupled to each intermediate network device. Those skilled in the art will appreciate that multiple hosts may be coupled to a single intermediate network device and a single host may be coupled to multiple intermediate network devices.

It can be appreciated that the particular configuration shown in FIG. **1** is chosen as an example only and is not limitive of the type of network on which the present invention can work. The number of configurations that networks can take are virtually limitless and techniques for setting up these configurations are well known to those skilled in the art. The embodiments of the present invention can operate on any of these possible configurations.

Furthermore, both the host device and intermediate network device can represent several types of devices. An example of a host device would be an end system (ES). An ES is a device attached to a network or subnetwork that is used to support end-user applications or services (e.g., a personal computer). An example of an intermediate network device would be a router, ATM switch or LSR. In this embodiment of the invention, a router is used as an example to demonstrate the principles described herein. Furthermore, the router utilizes IPv6 to route individual packets between hosts or end systems.

FIG. **2** is a block schematic diagram of a router suitable for practicing one embodiment of the invention. A router **200** is capable of incorporating the teachings of the present invention and includes a routing engine **202** having a processor **204** and a storage device **206**. Storage device **206** may be any suitable computer readable memory device, such as one or more dynamic random access memory (DRAM) devices, disk drives, or other mechanism for storing data.

Routing engine **202** includes in storage device **206** various computer program segments that when executed by a processor (e.g., processor **204**) performs the functionality for the various embodiments of the invention. In one embodiment of the invention, the computer program segments are combined into a single flow management module

(FMM) **218**. It can be appreciated, however, that the functions performed by this module can be separated into more modules, or be distributed throughout the system, and still fall within the scope of the invention. Furthermore, although this embodiment of the invention implements the functionality of this module in software, it can be appreciated that the functionality of this module may be implemented in hardware, software, or a combination of hardware and software, using well-known signal processing techniques.

Routing engine **202** includes also includes various tables **208** and databases **210** contained within storage device **206**. Tables **208** and databases **210** maintain information necessary for router **200** to properly forward data. Tables **208** may include a Routing Table and a Flow Table. Databases **210** may include a Link State Database and a Forwarding Database. Routing engine **202** is capable of calculating paths through a network based on information contained in tables **208** and databases **210**, as well as the functionality provided by FMM **218**.

Input/Output (I/O) interfaces **212** are coupled to routing engine **202** and provide a physical connection to one or more network links **216**. I/O interfaces **212** may be any suitable means for controlling communication signals between objects using a desired set of protocols, services and operating procedures, such as IPv6. In this embodiment of the invention, I/O interfaces **212** are bidirectional, that is, signals can be sent and received using any particular I/O interface. Those skilled in the art, however, will recognize that uni-direction interfaces can also be used and fall within the scope of the invention. Furthermore, those skilled in the art will understand that the communication signals may be received over any suitable medium such as twisted-pair wire, co-axial cable, fiber optics, radio-frequencies, and so forth.

Processor **204** may be any general purpose microprocessor having sufficient speed to implement the functionality described herein, such as the Pentium®, Pentium Pro, or Pentium II processors made by Intel Corporation.

It can be appreciated that although router **200** is used as an example to describe this embodiment of the invention, those skilled in the art will appreciate that various types of routers and other intermediary network devices may be used with the invention described herein.

FIG. **3** is a block flow diagram of the steps performed by a first network device in accordance with one embodiment of the invention. As shown in FIG. **3**, a packet for the information is generated at a first network device at step **302**. A flow label is assigned to the packet at step **304**. A direction is assigned to the packet using the flow label at step **306**. The packet is sent to a second network device through the network at step **308**.

With respect to step **306**, an example of assigning a direction includes a flow label having a plurality of bits. The first network device would indicate the direction for the packet by modifying the flow label. The first bit of the flow label specifies whether it is source or destination unique, with a zero (0) implying a destination unique flow label, and a one (1) declaring a source unique flow label. To send on the reverse path, the receiving network device flips the first bit of the flow label. Since the source and destination are also swapped (as compared to the received packet), the same address is used with the flow label to uniquely identify the flow.

FIG. **4** is a block flow diagram of the steps performed by an intermediate network device in accordance with one embodiment of the invention. As shown in FIG. **4**, a first packet is received at a first input port of an intermediate network device at step **402**. A flow label for the first packet is identified at step **404**. Whether a flow table has an entry for the flow label is determined at step **406**. An entry for the flow label is created if no present entry at step **408**. An output port associated with the flow label is obtained from the table at step **410**. The first packet is sent to the output port at step **412**.

An entry for the flow label is added to the flow table if there is no present entry at step **408**. Whether the flow label has an associated routing attribute is determined. A pointer associated with the flow label is stored in the flow table, the pointer pointing to an output port in a routing table for the intermediate network device, if the flow label does not have a routing attribute associated with it. An output port from the routing table is stored in the flow table associated with the flow label, if the flow label does have a routing attribute associated with it.

The steps described with reference to FIG. **4** may be better understood using the following example. In this embodiment of the invention, the intermediate network device is router **200**. Router **200** receives a first packet at a first input port. FMM **218** of router **200** identifies a flow label for the first packet. FMM **218** searches a flow table stored with tables **208** to determine whether the flow table has an entry for the flow label. FMM **218** creates an entry for the flow label if no entry is currently in the flow table for the flow label. FMM **218** then obtains an output port associated with the flow label from the table. Routing engine **202** sends the first packet to the output port.

Router **200** performs the above steps with each packet that it receives. Packets received subsequent to the first packet and having the same flow label would be processed more efficiently since an entry for the flow label will already be present. If a packet does not have a flow label, FMM **218** can be programmed to ignore the packet in terms of processing it for flow control, or to assign a flow label if needed for more efficient routing (e.g., such as for tunneling).

If there is no entry for the flow label present in the flow table, FMM **218** creates an entry. FMM **218** determines whether any attributes have been assigned to the flow label, such as route pinning. If a route pinning attribute has not been assigned to the flow label, a pointer associated with the flow label is stored in the flow table. The pointer points to an output port in the routing table for router **200**. If a route pinning attribute has been assigned to the flow label, by FMM **218** or the first network device, FMM **218** copies the output port stored in the routing table having the same destination as the first packet in the flow table. The output port is associated with the flow label. In this manner, subsequent packets sharing the same flow label will traverse the path designated by the routing table if route pinning is not set, or will traverse the same path as the first packet if route pinning is set. In the latter case, a stable forwarding path is maintained since each packet within a flow will traverse the same sequence of routers.

To enable reverse path forwarding, the first input port where the first packet was received must also be stored in the flow table in association with the flow label for the first packet. If router **200** receives a second packet, FMM **218** identifies a flow label for the second packet. Further, FMM **218** also determines a direction for the second packet by examining the first bit of the flow label. If the first bit of the flow label indicates that the second packet is from the second network device back to the first network device, then routing engine **202** sends the packet to the I/O interface port where

the first packet was initially received. This can be accomplished since I/O interfaces **212** are bidirectional. Those skilled in the art could appreciate that reverse path routing could also be implemented using unidirectional ports if necessary.

The embodiments of the invention can also apply the aggregation properties already currently used in IPv4 and IPv6 for individual packets to the concept of flows. For example, if router **200** receives a second packet at an input port, it first identifies its flow label. Then it determines whether the first packet and the second packet should be aggregated based on their flow labels and either associated attributes or intelligence of routing engine **202**. If the first and second packets should be aggregated, they are aggregated using conventional tunneling techniques.

The use of flows in MPLS is similar to IPv6, with specific modifications to take advantage of the underlying mechanisms for MPLS. This embodiment of the invention assumes a "neighbor discover protocol" that can (be augmented to) carry capability information such as label space. It also assumes that the network can do routing on demand inside the label network (this is virtually implied by on demand routes).

There are two types of allocations in this embodiment of the invention. The first is upstream allocated on-demand label paths of three sub-types: (1) without any acknowledgments (acks); (2) with hop-by-hop ack's; and for tunnels. The second general allocation type is downstream allocated on-demand label paths. These are allocated using, for example, the methods and apparatus set forth in U.S. patent application Ser. No. 09/015,496, filed on Jan. 29, 1998, entitled "An Architecture For Lightweight Signaling In ATM Networks", the entire disclosure of which is hereby incorporated by reference. One of the down sides of downstream allocation as compared to upstream allocation is that with tunnels the hop-by-hop latency (over the tunnel) may be O (end-to-end) latency. One solution could be to place an IPv4 header into a single ATM cell.

In both the upstream allocation and downstream allocation, the label path setup can be accomplished in the following steps: (1) the end system initiates path setup; (2) the path (channel, featherweight flow); (3) can subsequently assign additional parameters to the flow. One solution is to use a higher level protocol for routing and forwarding of non-switched packets

As part of boot strapping nodes exchange capabilities and setup a default path that is subsequently used for out of band signaling. If a node understands multiple higher level protocols (i.e., can route using the rules of many protocols) e.g., IPv4, IPv6, ATM UNI, a different signaling path can be established for each of these protocols.

The label distribution protocol uses a short label, which is valid for a limited time and created on demand. This provides manageability, as the forwarding map can be customized based on network conditions, the need of the flow, and so forth. Softness of state is maintained for those protocol types for which a default path (and/or a signaling channel) is defined, and whose protocol data units carry enough information to establish (recover) the forwarding state. This follows as the default paths per protocol assume that the LSR per protocol processing is sufficient to reach inside the MPLS encapsulation and parse the (header) information needed to (re)create the state. As a consequence, no explicit signaling is needed. The soft state is relatively efficient, as it is "use it or loose it," thus averting the need for keep-alive messages to maintain the state.

As part of neighborhood discovery each node informs its upstream neighbor of the label space it is willing to accept. It is assumed that by standardization some small number of labels may be taken for granted (say 1 or 10). The upstream node may subsequently ask for the namespace to be expanded (this is done on signaling connections from controller, to controller).

Labels are assigned for one-way connections by default. New extensions allow for bidirectional label assignment. This can be done either by mechanisms to make a label valid both ways, or alternatively by associating a new label path going on a reverse route with a particular existing forward path.

To create a new label path from a source (S) to a destination (D), S creates a datagram (or otherwise the protocol transfer unit of the protocol used on the default (signaling) path), allocates a new label and issues a path-request communicating a path request, the new label and the datagram.

The path request could be a new router alert option (an extension header in IPv6), or could be communicated implicitly either on a signaling channel, or by interpreting a new (unknown) label as a path request. The last one offers particular advantages and is described in more detail below.

With respect to the new label, potentially more than a single label will be pushed on the stack, for example when constructing a tunnel. The new label(s) may be pushed on top of already existing labels.

The datagram may encode information about routes (e.g., explicit routes), type of service, or desired service quality as the richness of the protocol on the default path allows.

An intermediate node receiving a path-request processes the message as follows. First, a new label entry is created in its forwarding table(s) unique on the pair (input port, label). This could be achieved by having a separate forwarding table per input port. Note that input port here may be an abstract input port, e., an end of a tunnel. The forwarding function on the enclosed datagram is then performed, yielding an output port determination. A new label is allocated on the outgoing port. The label forwarding table is then updated, recording the outgoing port and outgoing label. The request is then forwarded to the appropriate output port. If an acknowledgment is requested, then the intermediate node replies with an ack. If ack's are requested, the information could be a semi-static and negotiated as part of neighbor discovery. This could also, however, be done on demand by having the datagrams carry path construction attribute objects.

If a bidirectional path is requested then instead of replying with an just an ack, a label is allocated a label on the reverse path, the forwarding table is updated, the reverse label is recorded, and an ack is sent using the new upstream label.

Some of the possible errors include "Reject—sent on a well-known (signaling) channel." This would also include a reason code.

To implement flows with a soft state, label paths can be created for arbitrary data streams. For example, an access node from a frame-relay network to a LSR network capable of routing IP datagrams only, would simply create a label path by creating and sending an IP datagram as a path request, and encapsulating the frame-relay packets. When new (unknown) labels are interpreted as label requests, however, the state constituting labels paths used by well know protocols (i.e., one for which the LSR cloud can do routing) becomes soft. This happens because the state may be locally managed, and in particular lost, as the next

datagram arriving with a label that is no longer recognized will serve as a path request and effectively reconstruct the state. In this case there is no call setup delay. Moreover, by adopting (or assuming) a "use it or loose it" state invalidation policy, there is no need for explicit tear-down.

RSVP is designed to support reservations for individual end-to-end flows on the Internet, in particular IPv4. An RSVP session is identified by a destination address and transport level protocol, and optionally the destination ("generalized") port. A session is further classified into flows by receiver specified filters. Whereas in principle these filters may be applied to any fields in the IP- or protocol level header (even potentially application level headers), current specifications and implementations of RSVP limit filters to a source address, and optionally the source port. Reservations are on simplex streams and are exclusively receiver initiated.

A destination address may be a multicast address, with the multicast session having multiple senders and receivers (multipoint-to-multipoint). Receiver initiated reservations may result in different reservations in different segments of the distribution of the multicast (variegated multicast trees). Moreover, in multicast sessions with multiple senders, receivers may use the three different reservation "styles" (one of wild-card, fixed, or shared exclusive) to make reservations at even a finer level of the flow than specified by the filters.

The two principal messages of QoS management in RSVP are the path message, sent from senders towards receivers, and the reservation messages, sent from receivers towards senders. Path messages establish flow identification state along the downstream path. This state includes filters and the traffic description (T-spec). Messages are processes at each hop before forwarding. Reservations messages carry reservation requests (R-specs), and styles. Significant complexity is incorporated into RSVP to ensure that the RSVP signaling messages are forwarded to the same path (forward and reverse) as data is being forwarded. To make the RSVP state "soft" and to cope with route changes and changes in the topology of multicase distribution trees, path state and reservation state must be refreshed periodically.

Implementing RSVP using this embodiment of the invention simplifies RSVP in several ways. First, this embodiment of the invention already does a flow classification, thus subsuming most of the filtering mechanisms of RSVP. In addition to the benefit of separation of mechanisms, using this embodiment of the invention allows for rich filtering at the edge of the network (to classify the incoming datastream into flows), but very simple flow identification (explicit, or very trivial) inside the network. Filtering of finer grained subflows and RSVP reservation styles that apply on subflows (fixed, or shared exclusive) could be implemented by performing a nodal classification, or by defining a new flow for each of the subflows. This embodiment of the invention supports variegated trees similar to that of RSVP.

RSVP messages are effectively signaled in-band on the established flow, distinguished with router alert option (hop-by-hop extension header). Whereas this could be the standard router alert options, the new "CC" extension header could also be used, further improving efficiency by allowing the RSVP messages to be forwarded on the output ports before nodal processing takes place. The latter is a departure from current RSVP semantics. Using this embodiment of the invention, the RSVP path message serves to advertise the T-spec, and possibly a filter for subflow classification. The previous hop information is not needed. As the message is forwarded without processing the "Adspec" does not serve a useful purpose in this case. The reserve messages, however, must be processed at every branch point of a multicast flow. This is achieved using bidirectional flows and a blocking router alert option. Point-to-point connections, however, can exploit the in-band signaling for reservation messages, thus allowing the reservations to be processed in parallel. Other RSVP messages, for errors, tear-down, and confirmation are processed by the RSVP processor in a standard manner, but are transmitted in-band on the established flow. The RSVP processor must in addition process a tear-down commands from the forwarding engine, to invalidate reservation state for flows that have become invalid (at the forwarding level).

Using the support for bidirectional flows, forwarding on the reverse path may be moved out of the RSVP control process, and handled at forwarding level. To allow for softness of the reservation state and to allow for adjustments in reservations in multicast flows as membership (and thus topology) changes, state refresh may still be needed. The stability of the connections, however, can be exploited and therefore the need for frequent state refresh may be reduced. In particular, it is feasible to have the data traffic refresh the state, limiting refresh only to "keep-alives" during extended inactivity. This is particularly true for point-to-point flows. The need for state refresh is further reduced if the route of the flow is pinned. Therefore, removing the connectivity issues from the RSVP processing, and benefiting from the stability caused by connections, RSVP is simplified and may be more optimized for common cases (e.g., point-to-point flows) while retaining the essential qualities of RSVP.

It is worthy to note that any reference in the specification to "one embodiment" or "an embodiment" means that a particular feature, structure, or characteristic described in connection with the embodiment is included in at least one embodiment of the invention. The appearances of the phrase "in one embodiment" in various places in the specification are not necessarily all referring to the same embodiment.

Although various embodiments are specifically illustrated and described herein, it will be appreciated that modifications and variations of the present invention are covered by the above teachings and within the purview of the appended claims without departing from the spirit and intended scope of the invention. For example, although a router was used in certain embodiments of the invention, those skilled in the art will appreciate that the principles described herein can also be applied to other network devices such as ATM switches or LSRs.

What is claimed is:

1. A method for communicating information in a network, comprising:

generating a packet for the information at a first network device;

assigning a flow label to said packet;

assigning a direction to said packet using said flow label, and wherein said flow label is reverse path forwarding enabled; and

sending said packet to a second network device through the network.

2. The method of claim 1, wherein said flow label is comprised of a plurality of bits, and wherein said step of assigning said direction comprises the step of setting one of said bits in said flow label.

3. A method for communicating information in a network, comprising:

receiving a first packet at a first input port of an intermediate network device;

identifying a flow label for said first packet;

determining whether a flow table has an entry for said flow label;

creating an entry for said flow label if no entry is present;

obtaining an output port associated with said flow label from said table; and

sending said first packet to said output port.

4. The method of claim **3**, wherein said step of creating an entry comprises the steps of:

adding an entry for said flow label to said flow table;

determining whether said flow label has an associated routing attribute;

storing a pointer associated with said flow label in said flow table, said pointer pointing to an output port in a routing table for said intermediate network device, if said flow label does not have a routing attribute associated with it; and

storing said output port from said routing table in said flow table associated with said flow label, if said flow label does have a routing attribute associated with it.

5. The method of claim **3**, further comprising the step of storing said first input port in said flow table associated with said flow label.

6. The method of claim **5**, wherein said input and output ports for said intermediate network device are bidirectional, further comprising the steps of:

receiving a second packet at a second input port of said intermediate network device;

identifying a flow label for said second packet;

determining a direction for said second packet using said flow label; and

obtaining said first input port associated with said flow label from said flow table; and

sending said first packet to said first input port.

7. The method of claim **3**, further comprising the steps of:

receiving a second packet at an input port of said intermediate network device;

identifying a flow label for said first packet;

determining whether said first and second packets should be aggregated; and

aggregating said first and second packets in accordance with said determination.

8. A method for requesting a connection-oriented packet flow between a first network device and a second network device in a packet network, comprising

allocating a flow label to the packet flow that has not been previously declared;

assigning the flow label to a header in a first packet of the packet flow;

adding any desired flow attributes to the header, and

sending the packet to the second network device through the network, where in the flow label that has not been previously declared is interpreted by the second network device as a request for a new packet flow and the second network device can utilizes the flow attributes in establishing state for the packet flow, and wherein the flow label includes an indication of direction for the flow.

9. The method of claim **8** wherein the second network device treats the flow label as a request to pin the route between the first network device and the second network device.

10. The method of claim **8** wherein the second network device may send another packet in a reverse path by using a second flow label including a different indication of direction for the flow.

11. The method of claim **10** wherein the flow attributes reflect quality of service attributes for the packet flow.

12. A network router comprising:

a processor;

a plurality of input/output (I/O) interfaces connected to the processor; and

one or more storage devices, connected to the processor, further comprising a routing table, a flow table, and a computer program which when executed by the processor performs a method of establishing a packet flow between the network router and a second network router comprising the steps of:

receiving a packet from the second network router with a flow label that has not been previously declare;

interpreting the flow label as a request for a new packet flow between the network router and the second network router; and

creating an entry for the flow label in the flow table, and wherein the flow label includes an indication of direction for the flow.

13. The network router of claim **12** wherein the request for a new packet flow is treated by the network router as an implicit request for route pinning between the network router and the second network router.

14. The network router of claim **12** wherein the network router may send another packet in a reverse path to the second network router by using a second flow label including a different indication of direction for the flow.

15. The network router of claim **14** wherein the packet includes quality of service attributes for the packet flow and wherein the network router can establish state based on the quality of service attributes.

* * * * *

US006453360B1

(12) **United States Patent**
Muller et al.

(10) **Patent No.: US 6,453,360 B1**
(45) **Date of Patent: Sep. 17, 2002**

(54) **HIGH PERFORMANCE NETWORK INTERFACE**

(75) Inventors: **Shimon Muller**, Sunnyvale, CA (US); **Denton E. Gentry, Jr.**, Fremont, CA (US); **John E. Watkins**, Sunnyvale, CA (US); **Linda T. Cheng**, San Jose, CA (US)

(73) Assignee: **Sun Microsystems, Inc.**, Santa Clara, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/259,765**

(22) Filed: **Mar. 1, 1999**

(51) **Int. Cl.**[7] **G06F 13/00**

(52) **U.S. Cl.** **709/250**; 709/232; 709/236; 709/238; 370/235; 370/392; 370/401

(58) **Field of Search** 709/106, 201, 709/230, 232, 235, 236, 238, 245, 246, 250, 249; 370/231, 235, 389, 392, 396, 401

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,414,704	A	5/1995	Spinney	370/60

(List continued on next page.)

FOREIGN PATENT DOCUMENTS

EP	0 447 725	9/1991	G06F/15/16
EP	0 573 739	12/1993	H04L/12/56
EP	0 853 411	7/1998	H04L/29/06
EP	0 865 180	9/1998	H04L/12/56
WO	WO 95/14269	5/1995	G06F/7/08
WO	WO 97/28505	8/1997	G06F/13/14
WO	WO 99/00737	1/1999	G06F/13/00
WO	WO 99/00945	1/1999	H04L/12/46
WO	WO 99/00948	1/1999	H04L/12/56
WO	WO 99/00949	1/1999	H04L/12/56

OTHER PUBLICATIONS

Peter Newman, et al., "IP Switching and Gigabit Routers," *IEEE Communications Magazine*, vol. 335, No. 1, Jan. 1997, pp. 64–69.

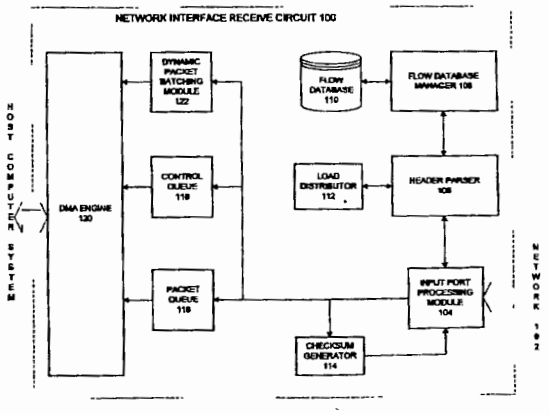
(List continued on next page.)

Primary Examiner—Viet D. Vu
(74) *Attorney, Agent, or Firm*—Park, Vaughan & Fleming LLP

(57) **ABSTRACT**

A high performance network interface is provided for receiving a packet from a network and transferring it to a host computer system. A header portion of a received packet is parsed by a parser module to determine the packet's compatibility with, or conformance to, one or more pre-selected protocols. If compatible, a number of processing functions may be performed to increase the efficiency with which the packet is handled. In one function, a re-assembly engine re-assembles, in a re-assembly buffer, data portions of multiple packets in a single communication flow or connection. Header portions of such packets are stored in a header buffer. An incompatible packet may be stored in another buffer. In another function, a packet batching module determines when multiple packets in one flow are transferred to the host computer system, so that their header portions are processed collectively rather than being interspersed with headers of other flows' packets. In yet another function, the processing of packets through their protocol stacks is distributed among multiple processors by a load distributor, based on their communication flows. A flow database is maintained by a flow database manager to reflect the creation, termination and activity of flows. A packet queue stores packets to await transfer to the host computer system, and a control queue stores information concerning the waiting packets. If the packet queue becomes saturated with packets, a random packet may be discarded. An interrupt modulator may modulate the rate at which interrupts associated with packet arrival events are issued to the host computer system.

62 Claims, 49 Drawing Sheets-



U.S. PATENT DOCUMENTS

5,566,170	A	* 10/1996	Bakke et al.	370/392
5,583,940	A	12/1996	Vidrascu et al.	380/49
5,684,954	A	11/1997	Kaiserswerth et al.	395/200.2
5,748,905	A	5/1998	Hauser et al.	395/200.79
5,758,089	A	5/1998	Gentry et al.	395/200.64
5,778,180	A	7/1998	Gentry et al.	395/200.42
5,778,414	A	7/1998	Winter et al.	711/5
5,787,255	A	7/1998	Parlan et al.	395/200.63
5,793,954	A	8/1998	Baker et al.	395/200.8
5,870,394	A	2/1999	Oprea	370/392
6,014,567	A	* 1/2000	Budka	455/453
6,044,079	A	* 3/2000	Calvignac et al.	370/395
6,094,435	A	* 7/2000	Hoffman et al.	370/414
6,163,539	A	* 12/2000	Alexander et al.	370/392
6,172,980	B1	* 1/2001	Flanders et al.	370/401
6,246,683	B1	* 6/2001	Connery et al.	370/392
6,253,334	B1	* 6/2001	Amdahl et al.	714/4

OTHER PUBLICATIONS

Francois Le Faucheur, "IETF Multiprotocol Label Switching (MPLS) Architecture," *IEEE International Conference*, Jun. 22, 1998, pp. 6–15.

F. Hallsall, "Data Communications, Computer Networks and Open Systems", *Electronic Systems Engineering Series*, 1996, pp. 451–452.

R. Cole, et al., "IP Over ATM: A Framework Document," *IETF Online*, Apr. 1996, pp. 1–31.

Toong Shoon Chan and Ian Gorton, *Parallel Architecture Support for High–Speed Protocol Processing*, Feb. 1, 1997, Microprocessors and Microsystes, GB, IPC, vol. 20, No. 6, pp. 325–339.

Pending U.S. patent application Ser. No. 09/259,445, entitled "Method and Apparatus for Distributing Network Processing on a Multiprocessor Computer," by Shimon Muller et al., filed Mar. 1, 1999 (Attorney Docket SUN–P3481–JTF).

Pending U.S. patent application Ser. No. 09/260,367, entitled "Method and Apparatus for Suppressing Interrupts in a High–Speed Network Environment," by Denton Gentry, filed Mar. 1, 1999 (Attorney Docket SUN–P3482–JTF).

Pending U.S. patent application Ser. No. 09/259,736, entitled "Method and Apparatus for Modulating Interrupts in a Network Interface," by Denton Gentry et al., filed Mar. 1, 1999 (Attorney Docket SUN–P3483–JTF).

Pending U.S. patent application Ser. No. 09/260,618, entitled "Method and Apparatus for Classifying Network Traffic in a High Performance Network Interface," by Shimon Muller et al., filed Mar. 1, 1999 (Attorney Docket SUN–P3486–JTF).

Pending U.S. patent application Ser. No. 09/259,932, entitled "Method and Apparatus for Managing a Network Flow in a High Performance Network Interface," by Shimon Muller et al., filed Mar. 1, 1999 (Attorney Docket SUN–P3487–JTF).

Pending U.S. patent application Ser. No. 09/260,324, entitled "Method and Apparatus for Dynamic Packet Batching with a High Performance Network Interface," by Shimon Muller et al., filed Mar. 1, 1999 (Attorney Docket SUN–P3488–JTF).

Pending U.S. patent application Ser. No. 09/258,952, entitled "Method and Apparatus for Early Random Discard of Packets," by Shimon Muller et al., filed Mar. 1, 1999 (Attorney Docket SUN–P3490–JTF).

Pending U.S. patent application Ser. No. 09/260,333, entitled "Method and Apparatus for Data Re–Assembly with a High Performance Network Interface," by Shimon Muller et al., filed Mar. 1, 1999 (Attorney Docket SUN–P3507–JTF).

Pending U.S. patent application Ser. No. 09/258,955, entitled "Dynamic Parsing in a High Performance Network Interface," by Denton Gentry, filed Mar. 1, 1999 (Attorney Docket SUN–P3715–JTF).

Pending U.S. patent application Ser. No. 09/259,936, entitled "Method and Apparatus for Indicating an Interrupt in a Network Interface," by Denton Gentry et al., filed Mar. 1, 1999 (Attorney Docket SUN–P3814–JTF).

Sally Floyd & Van Jacobson, *Random Early Detection Gateways for Congestion Avoidance*, Aug., 1993, IEEE/ACM Transactions on Networking.

U.S. patent application Ser. No. 08/893,862, entitled "Mechanism for Reducing Interrupt Overhead in Device Drivers," filed Jul. 11, 1997, inventor Denton Gentry.

* cited by examiner

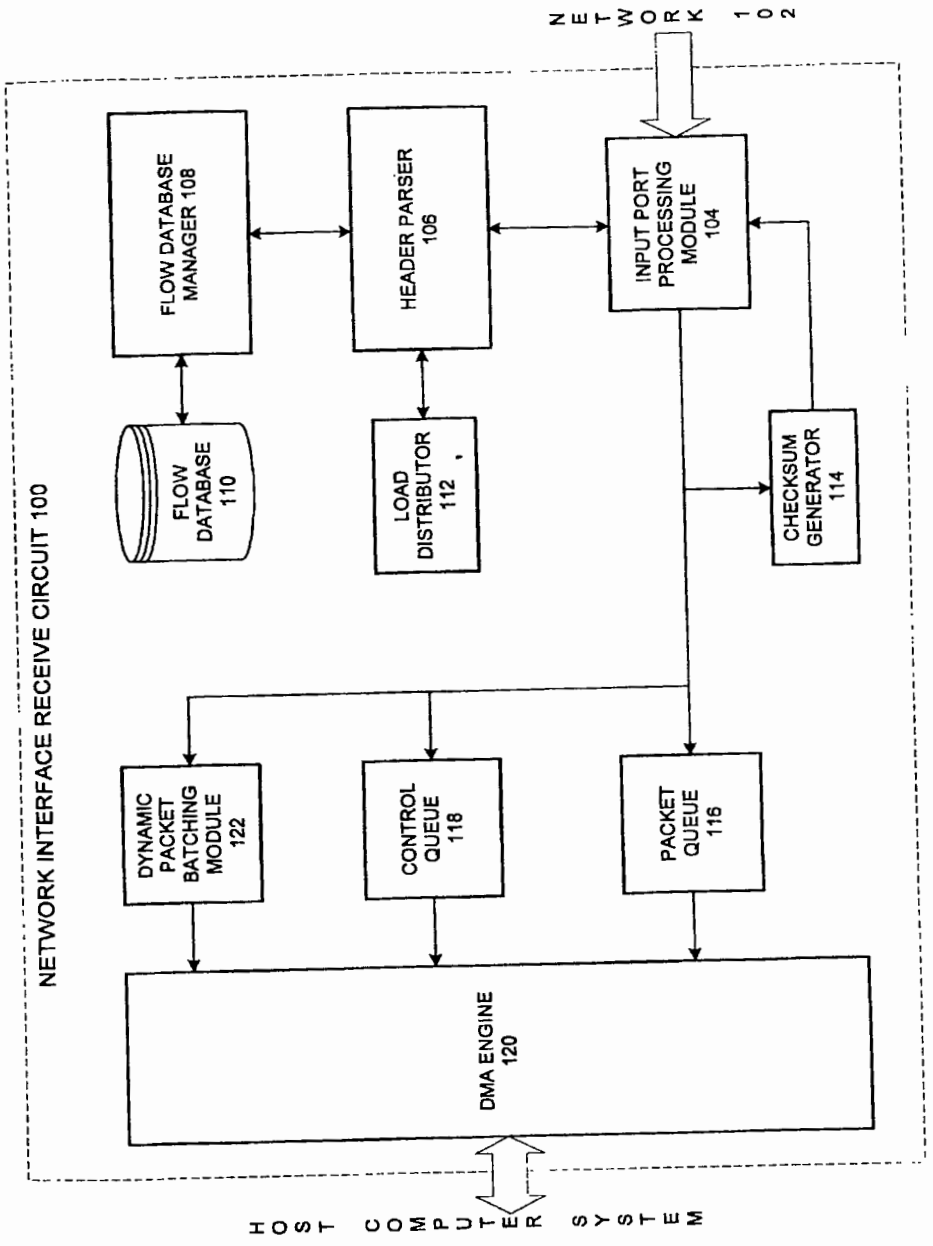

NETWORK 102
NETWORK INTERFACE RECEIVE CIRCUIT 100
FLOW DATABASE MANAGER 108
HEADER PARSER 106
INPUT PORT PROCESSING MODULE 104
FLOW DATABASE 110
LOAD DISTRIBUTOR 112
CHECKSUM GENERATOR 114
DYNAMIC PACKET BATCHING MODULE 122
CONTROL QUEUE 118
PACKET QUEUE 116
DMA ENGINE 120
HOST COMPUTER SYSTEM

FIG. 1A

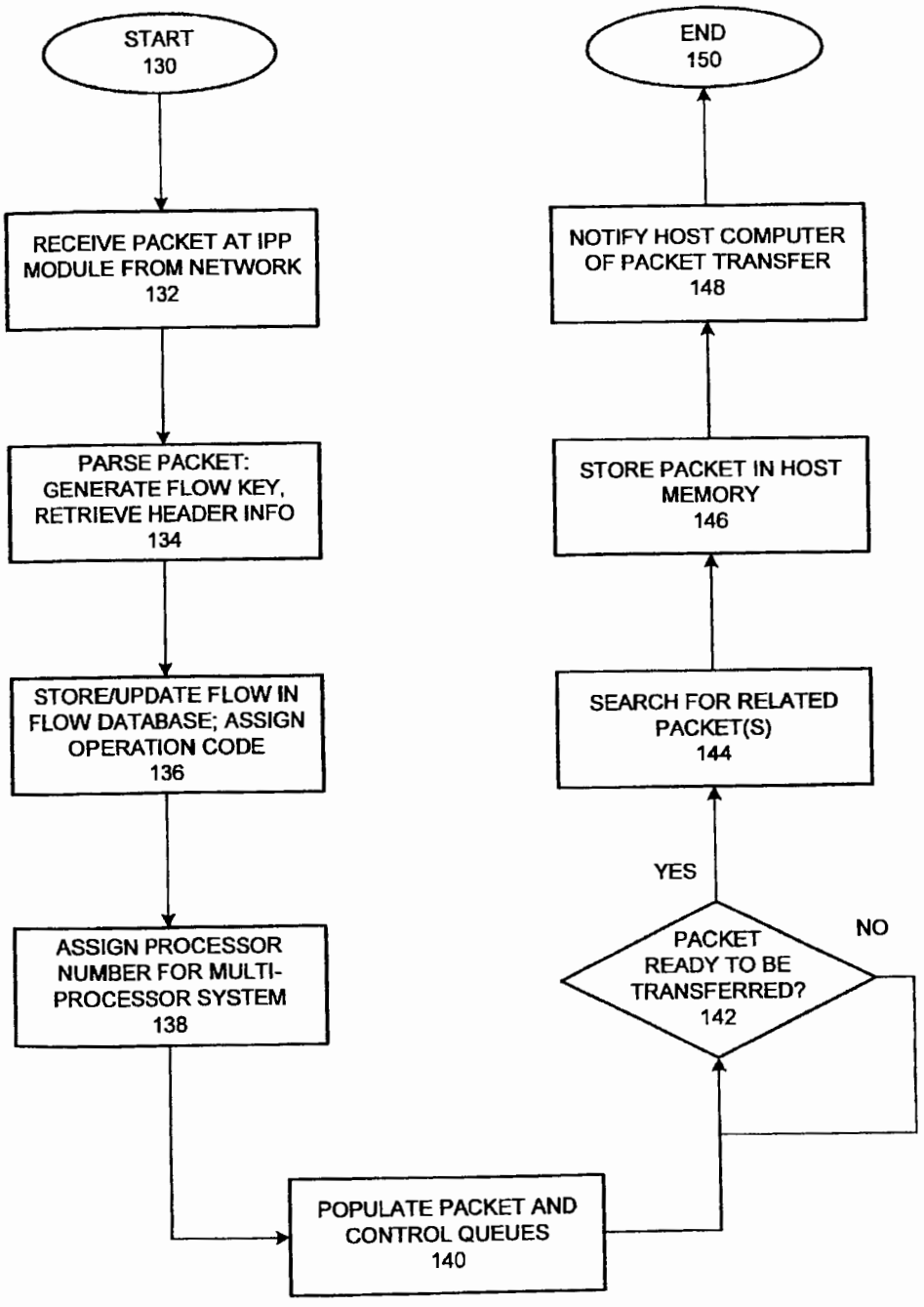

START
130
RECEIVE PACKET AT IPP MODULE FROM NETWORK
132
PARSE PACKET: GENERATE FLOW KEY, RETRIEVE HEADER INFO
134
STORE/UPDATE FLOW IN FLOW DATABASE; ASSIGN OPERATION CODE
136
ASSIGN PROCESSOR NUMBER FOR MULTI-PROCESSOR SYSTEM
138
POPULATE PACKET AND CONTROL QUEUES
140
PACKET READY TO BE TRANSFERRED?
142
YES
NO
SEARCH FOR RELATED PACKET(S)
144
STORE PACKET IN HOST MEMORY
146
NOTIFY HOST COMPUTER OF PACKET TRANSFER
148
END
150

FIG. 1B

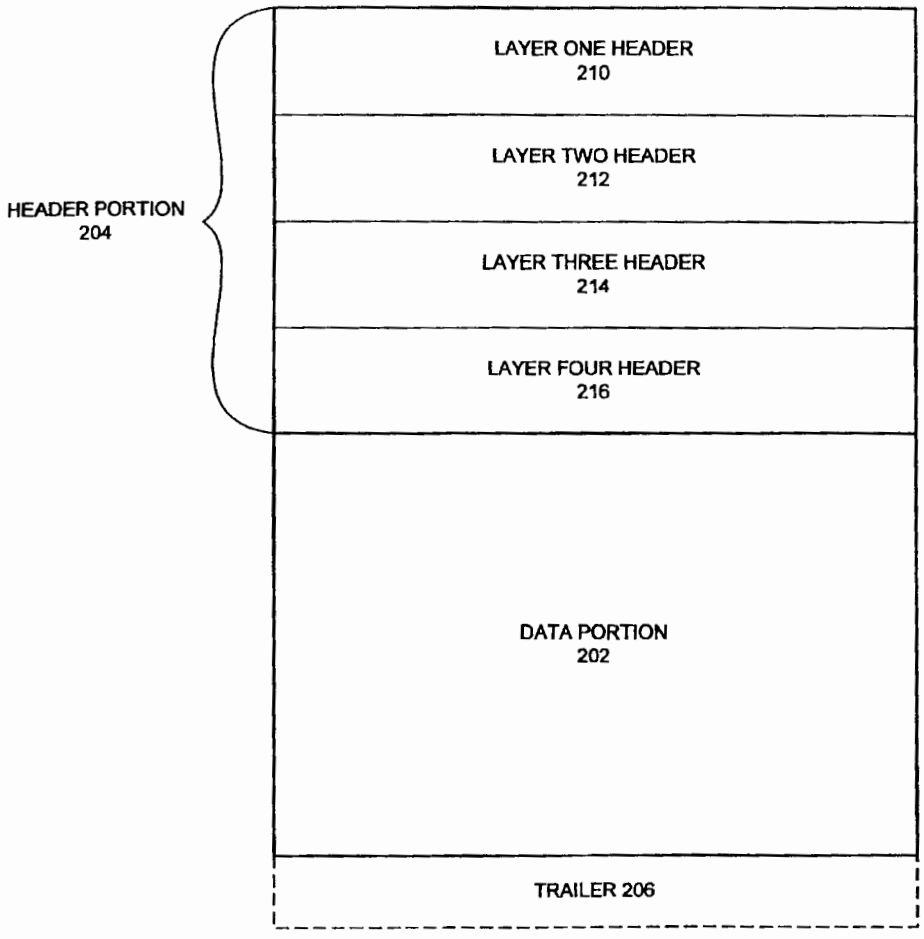

LAYER ONE HEADER
210
LAYER TWO HEADER
212
HEADER PORTION
204
LAYER THREE HEADER
214
LAYER FOUR HEADER
216
DATA PORTION
202
TRAILER 206
PACKET 200

FIG. 2

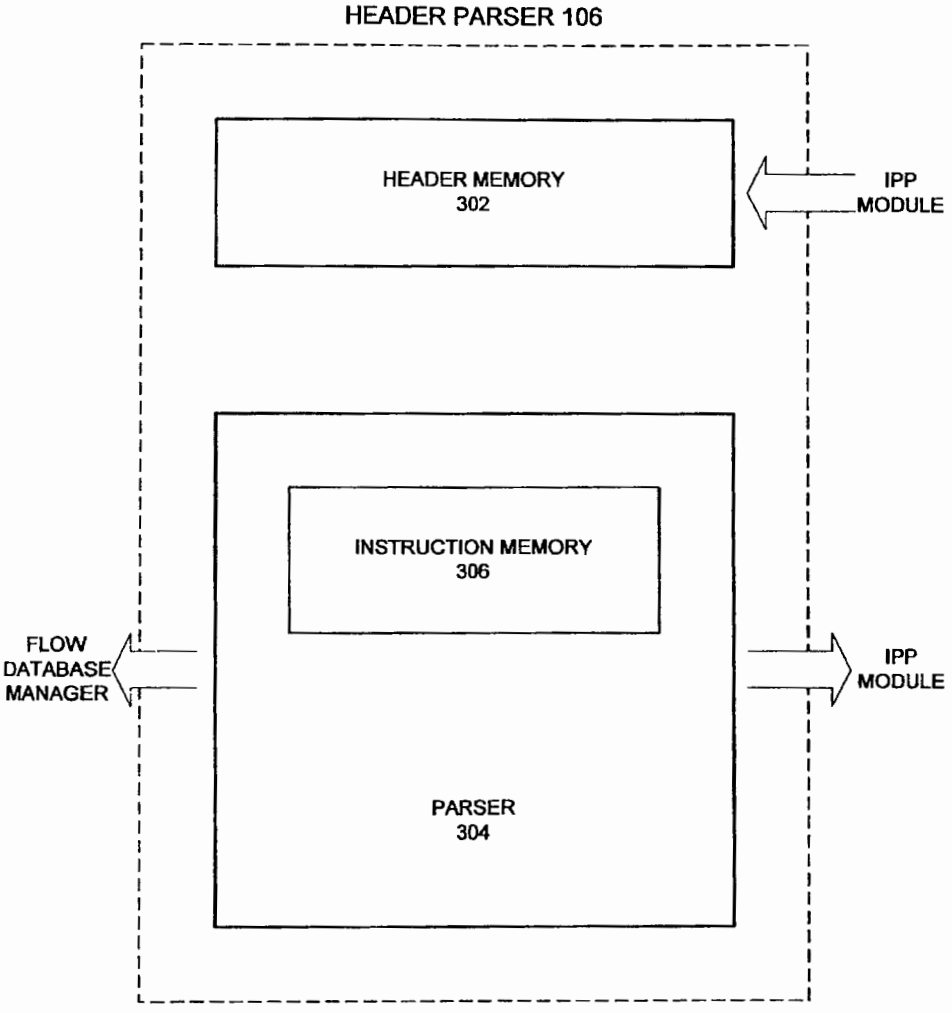

HEADER PARSER 106
HEADER MEMORY
302
IPP
MODULE
INSTRUCTION MEMORY
306
FLOW
DATABASE
MANAGER
IPP
MODULE
PARSER
304

FIG. 3

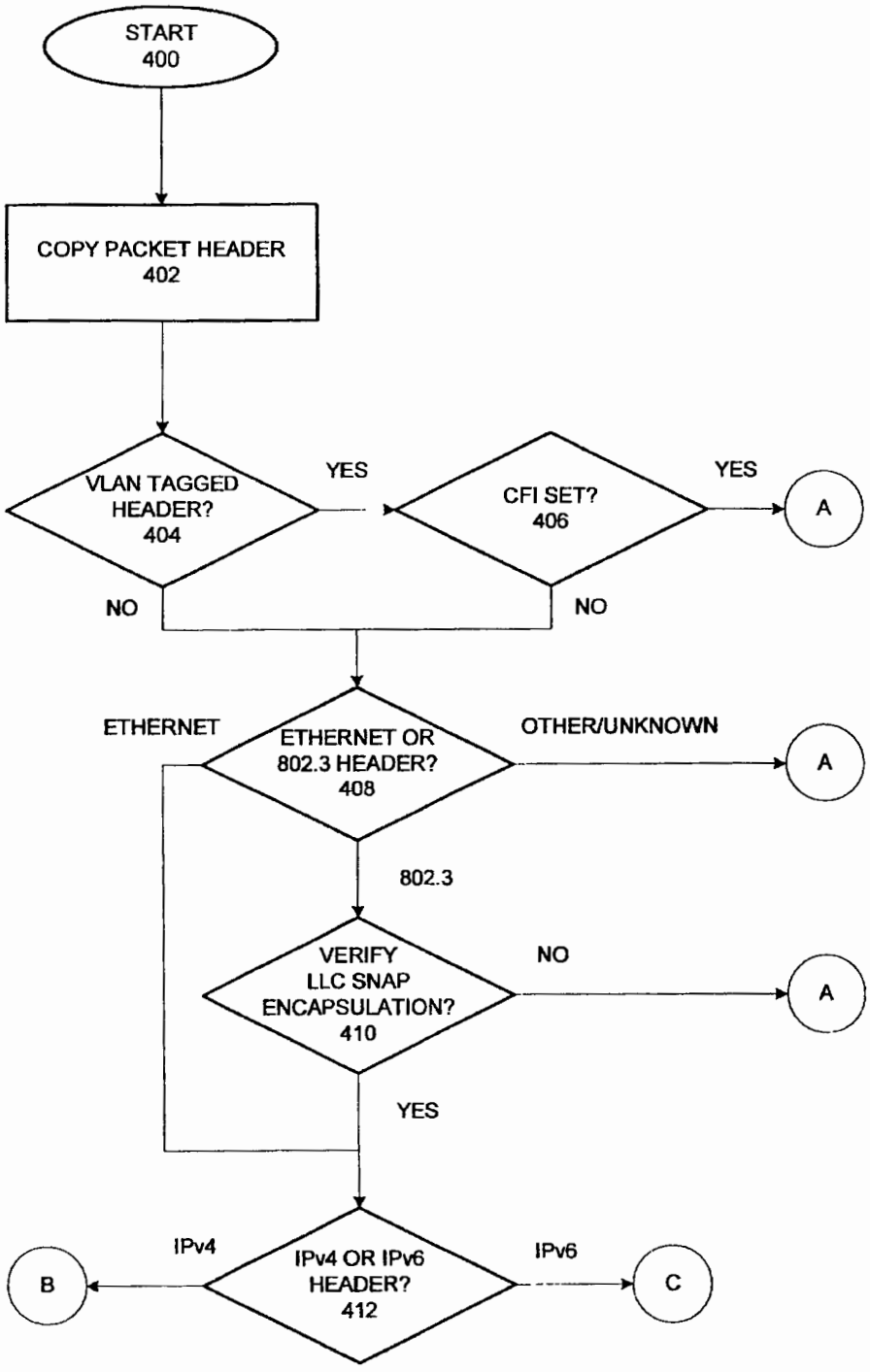

START
400
COPY PACKET HEADER
402
VLAN TAGGED HEADER?
404
YES
CFI SET?
406
YES
A
NO
NO
ETHERNET
ETHERNET OR 802.3 HEADER?
408
OTHER/UNKNOWN
A
802.3
VERIFY LLC SNAP ENCAPSULATION?
410
NO
A
YES
IPv4
IPv4 OR IPv6 HEADER?
412
IPv6
B
C

FIG. 4A

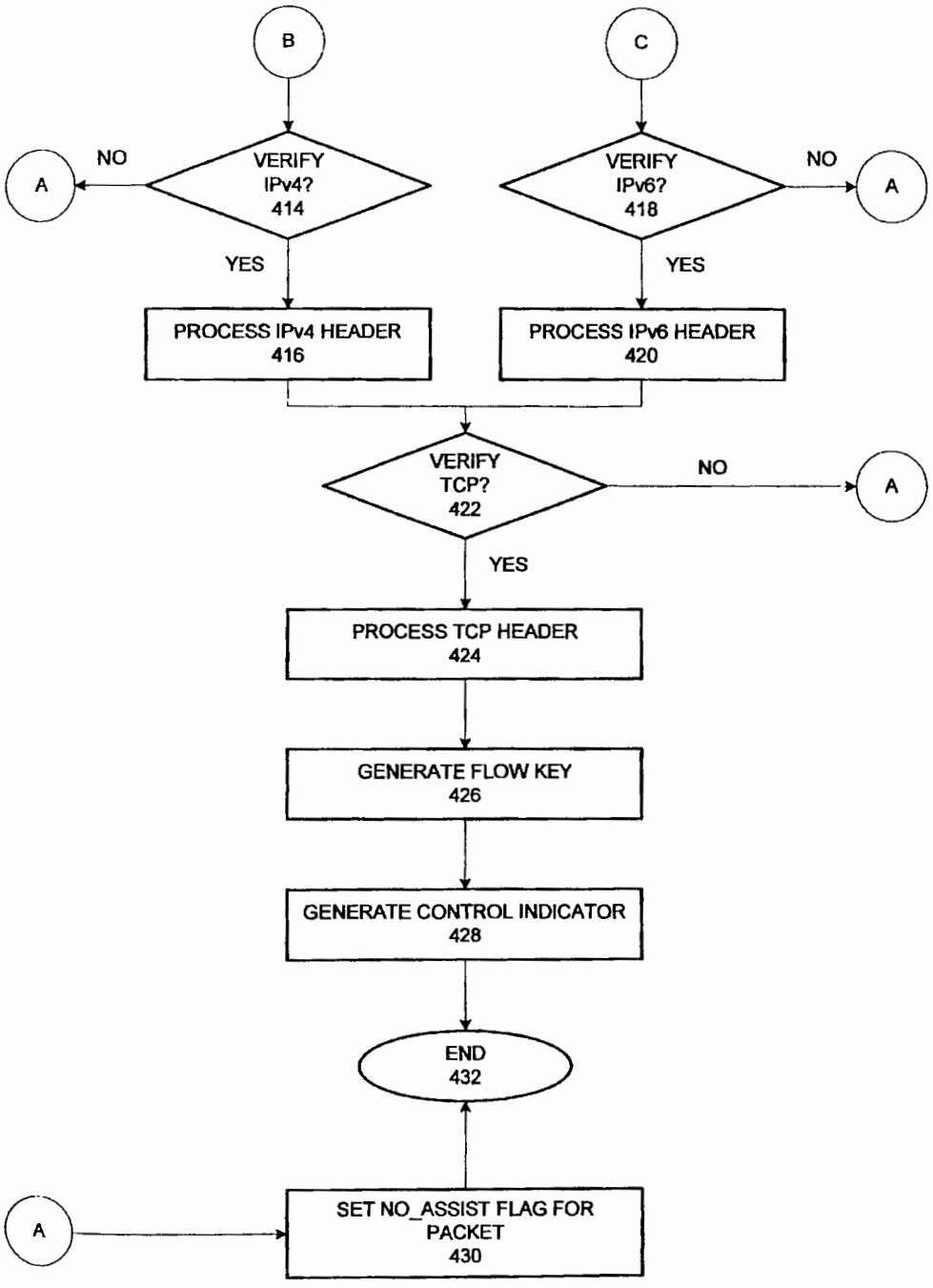

B
C
VERIFY IPv4? 414
NO
A
YES
VERIFY IPv6? 418
NO
A
YES
PROCESS IPv4 HEADER 416
PROCESS IPv6 HEADER 420
VERIFY TCP? 422
NO
A
YES
PROCESS TCP HEADER 424
GENERATE FLOW KEY 426
GENERATE CONTROL INDICATOR 428
END 432
A
SET NO_ASSIST FLAG FOR PACKET 430

FIG. 4B

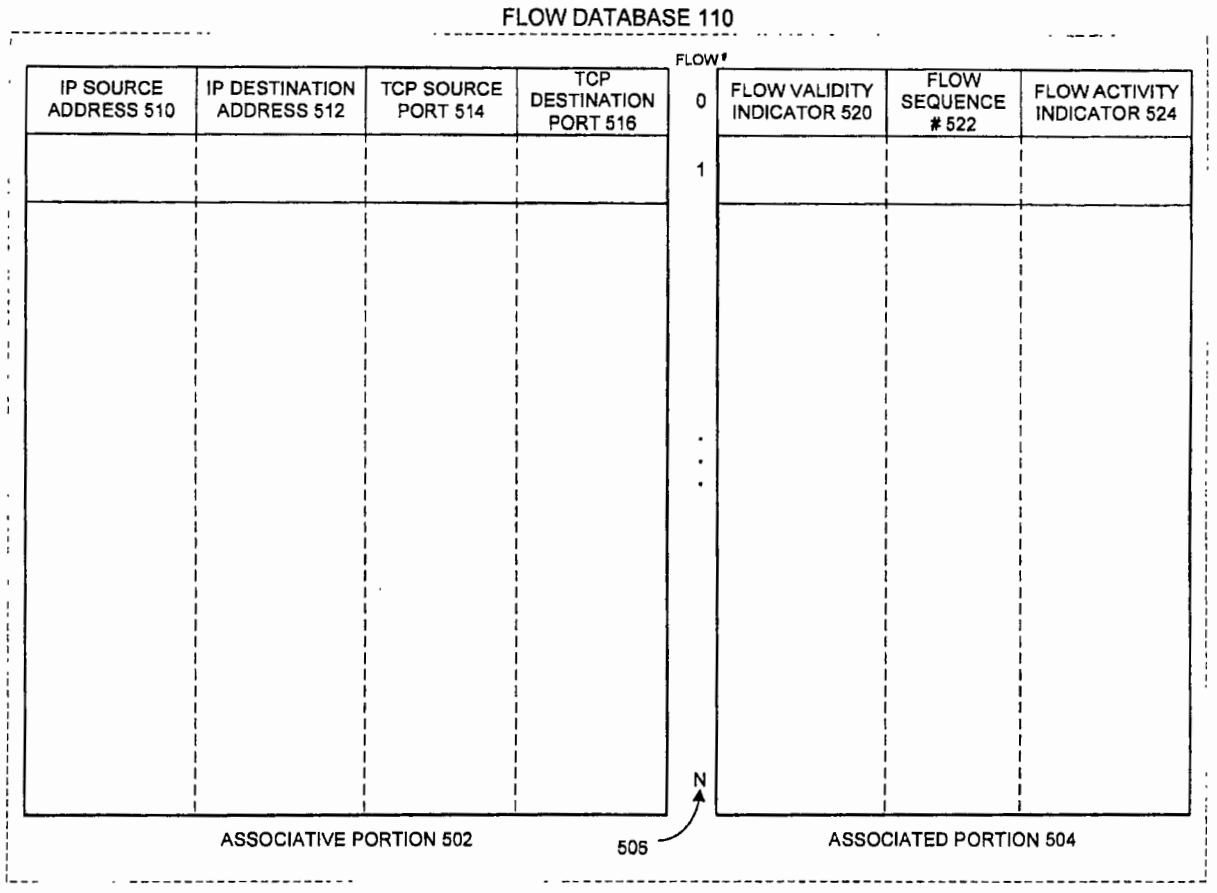

FLOW DATABASE 110
FLOW #
IP SOURCE ADDRESS 510
IP DESTINATION ADDRESS 512
TCP SOURCE PORT 514
TCP DESTINATION PORT 516
FLOW VALIDITY INDICATOR 520
FLOW SEQUENCE # 522
FLOW ACTIVITY INDICATOR 524
0
1
N
505
ASSOCIATIVE PORTION 502
ASSOCIATED PORTION 504

FIG. 5

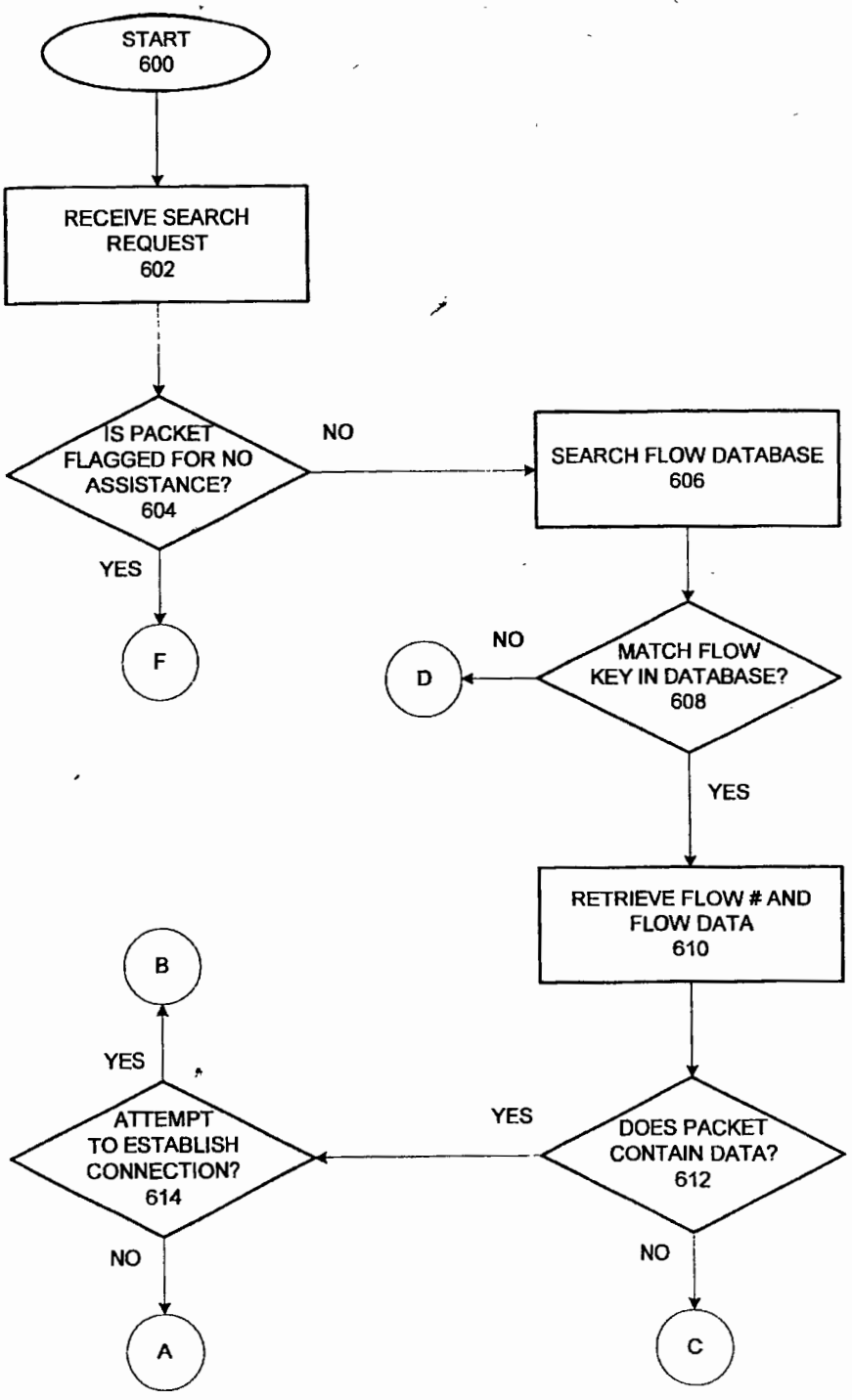

START
600
RECEIVE SEARCH REQUEST
602
IS PACKET FLAGGED FOR NO ASSISTANCE?
604
NO
SEARCH FLOW DATABASE
606
YES
F
NO
D
MATCH FLOW KEY IN DATABASE?
608
YES
RETRIEVE FLOW # AND FLOW DATA
610
B
YES
ATTEMPT TO ESTABLISH CONNECTION?
614
YES
DOES PACKET CONTAIN DATA?
612
NO
NO
A
C

FIG. 6A

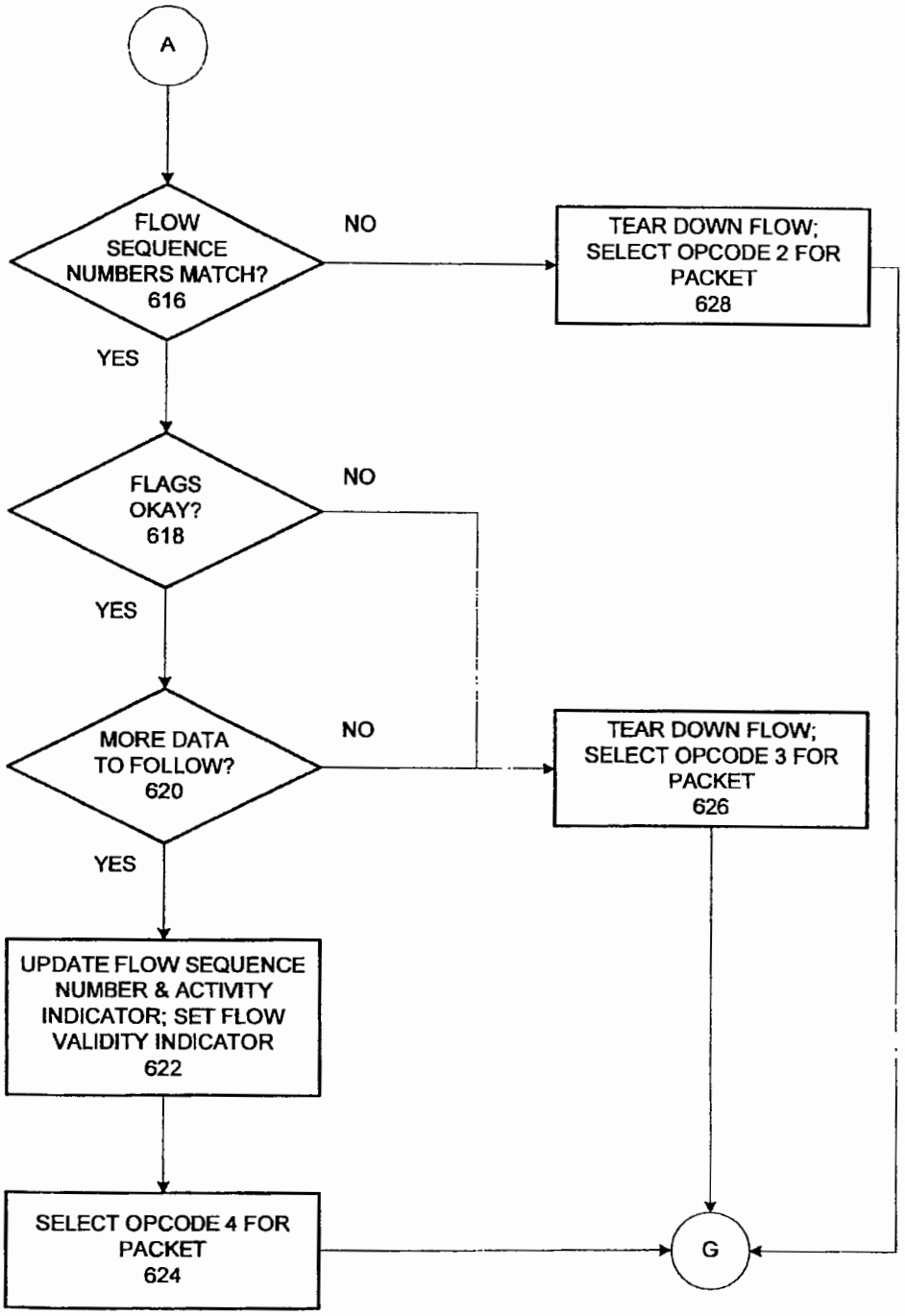

A
FLOW SEQUENCE NUMBERS MATCH? 616
NO
TEAR DOWN FLOW; SELECT OPCODE 2 FOR PACKET 628
YES
FLAGS OKAY? 618
NO
YES
MORE DATA TO FOLLOW? 620
NO
TEAR DOWN FLOW; SELECT OPCODE 3 FOR PACKET 626
YES
UPDATE FLOW SEQUENCE NUMBER & ACTIVITY INDICATOR; SET FLOW VALIDITY INDICATOR 622
SELECT OPCODE 4 FOR PACKET 624
G

FIG. 6B

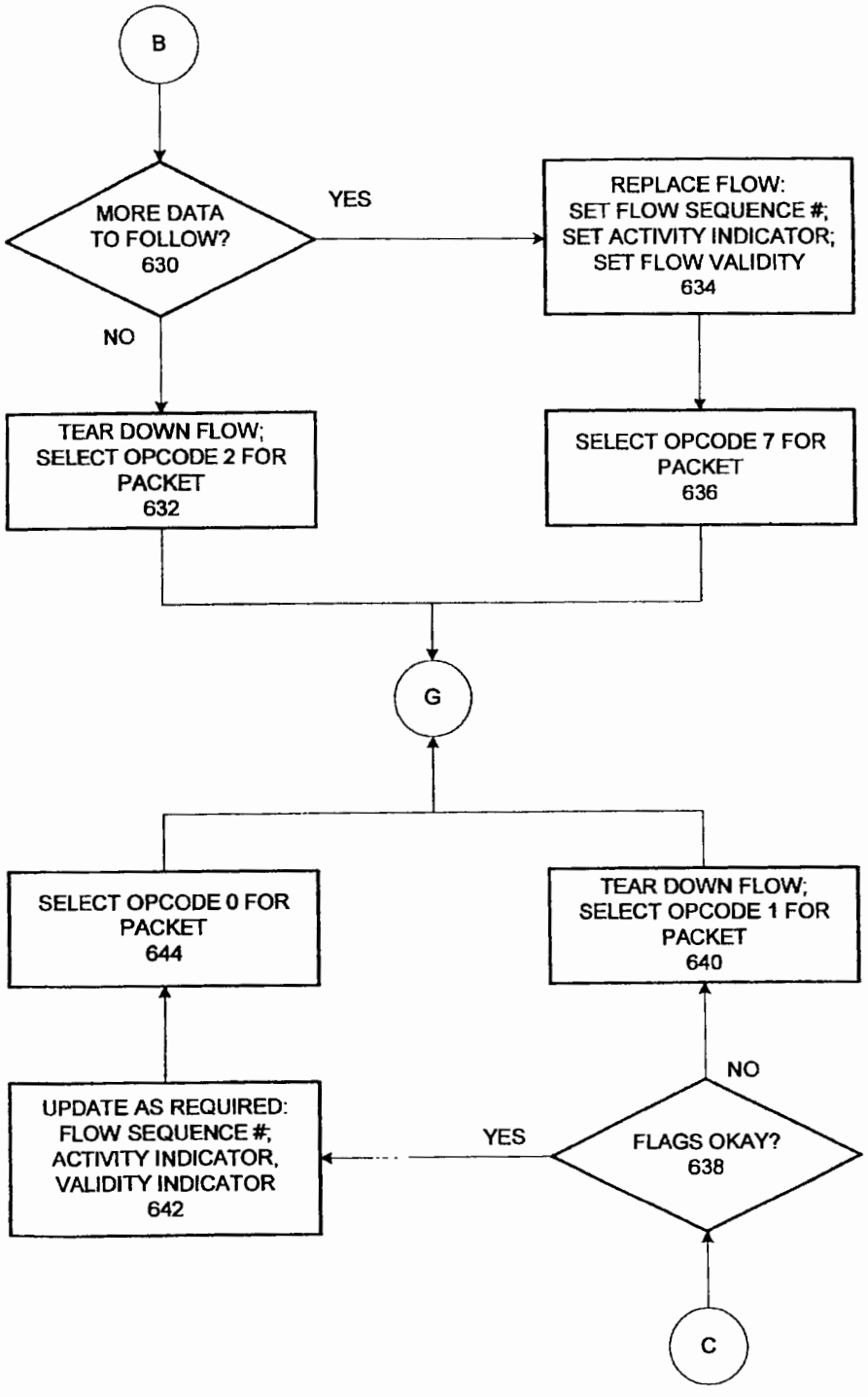

B
MORE DATA TO FOLLOW? 630
YES
NO
REPLACE FLOW: SET FLOW SEQUENCE #; SET ACTIVITY INDICATOR; SET FLOW VALIDITY 634
TEAR DOWN FLOW; SELECT OPCODE 2 FOR PACKET 632
SELECT OPCODE 7 FOR PACKET 636
G
SELECT OPCODE 0 FOR PACKET 644
TEAR DOWN FLOW; SELECT OPCODE 1 FOR PACKET 640
NO
UPDATE AS REQUIRED: FLOW SEQUENCE #; ACTIVITY INDICATOR, VALIDITY INDICATOR 642
YES
FLAGS OKAY? 638
C

FIG. 6C

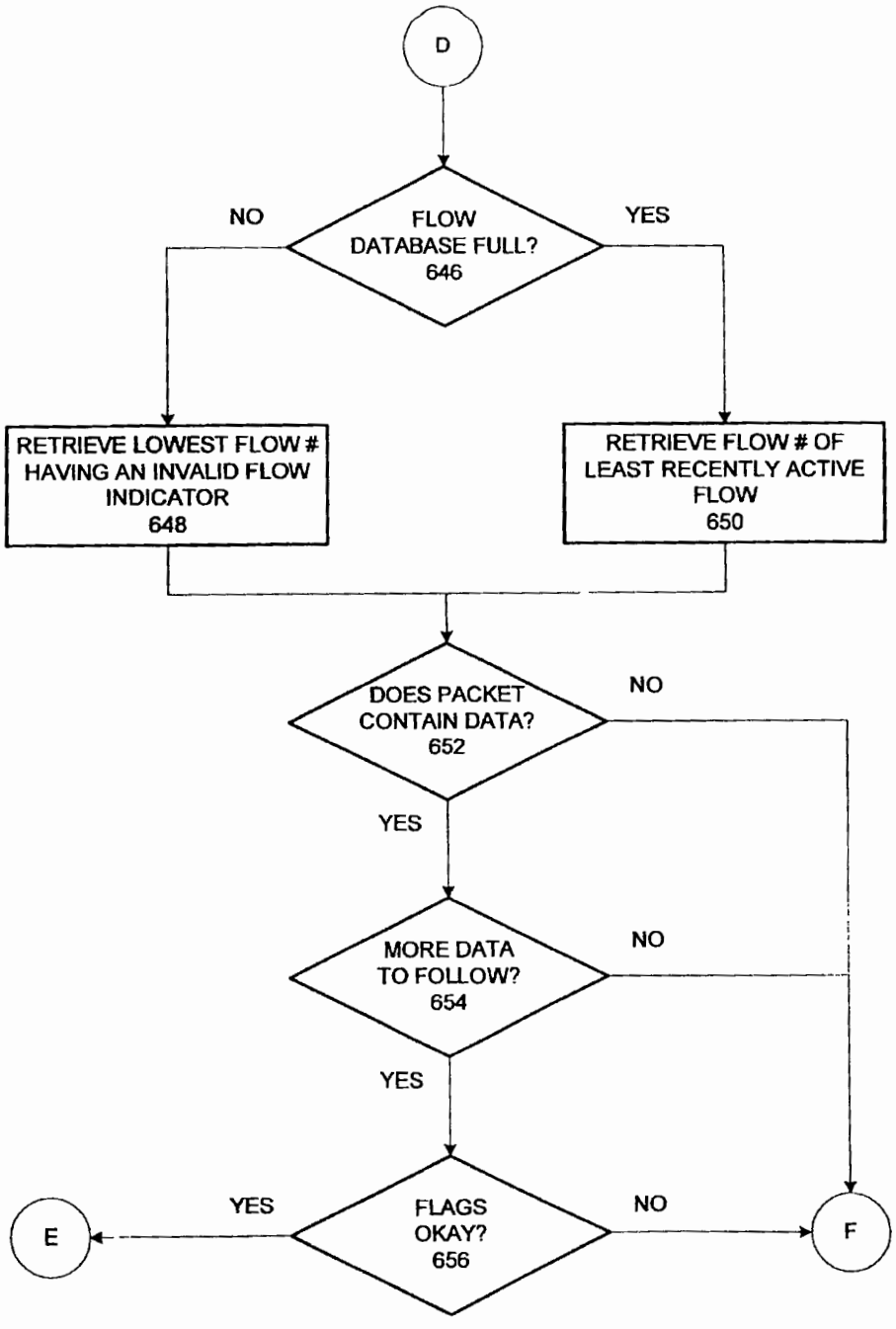

D
FLOW DATABASE FULL? 646
NO
YES
RETRIEVE LOWEST FLOW # HAVING AN INVALID FLOW INDICATOR 648
RETRIEVE FLOW # OF LEAST RECENTLY ACTIVE FLOW 650
DOES PACKET CONTAIN DATA? 652
NO
YES
MORE DATA TO FOLLOW? 654
NO
YES
FLAGS OKAY? 656
YES
NO
E
F

FIG. 6D

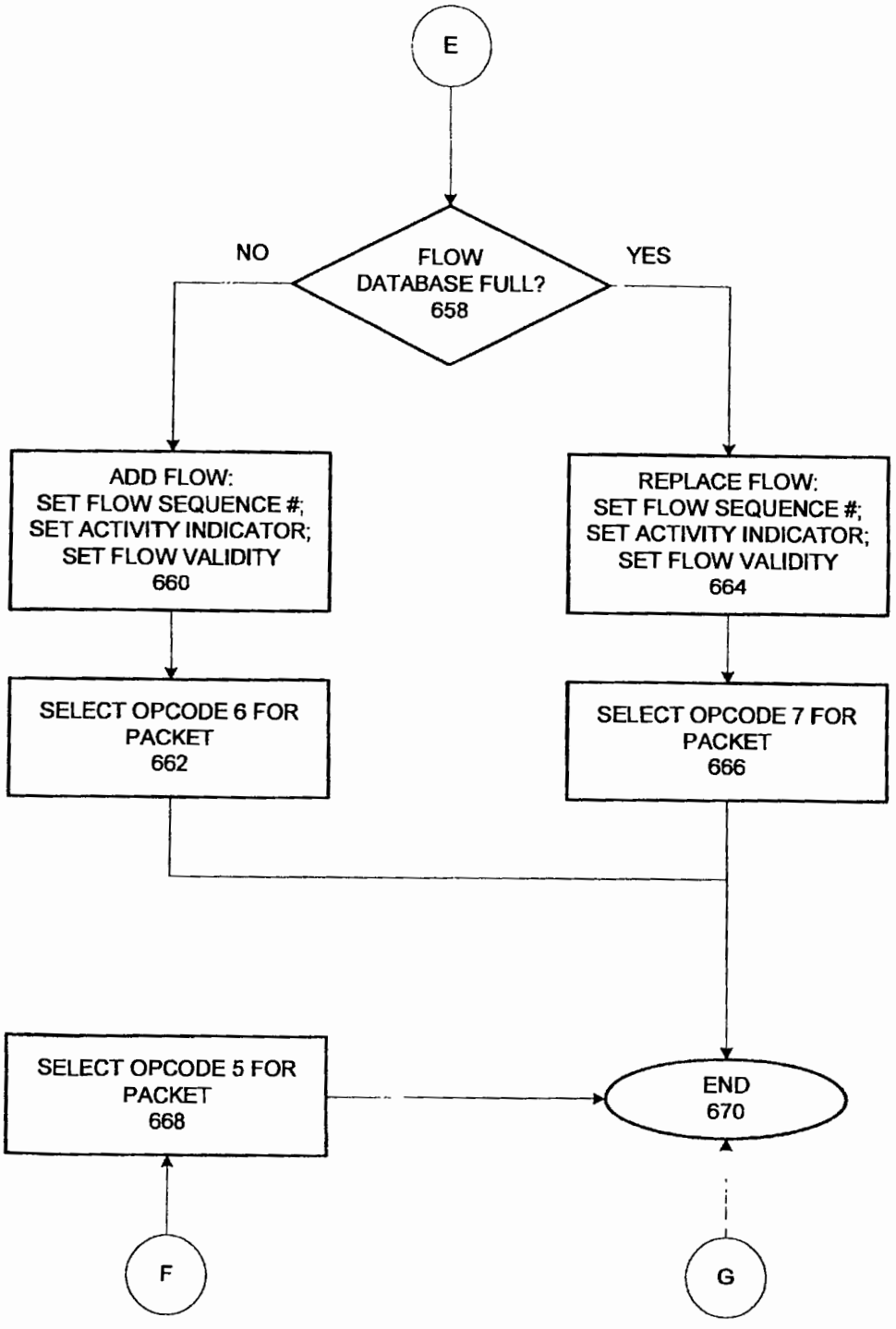

E
FLOW DATABASE FULL? 658
NO
YES
ADD FLOW: SET FLOW SEQUENCE #; SET ACTIVITY INDICATOR; SET FLOW VALIDITY 660
REPLACE FLOW: SET FLOW SEQUENCE #; SET ACTIVITY INDICATOR; SET FLOW VALIDITY 664
SELECT OPCODE 6 FOR PACKET 662
SELECT OPCODE 7 FOR PACKET 666
SELECT OPCODE 5 FOR PACKET 668
END 670
F
G

FIG. 6E

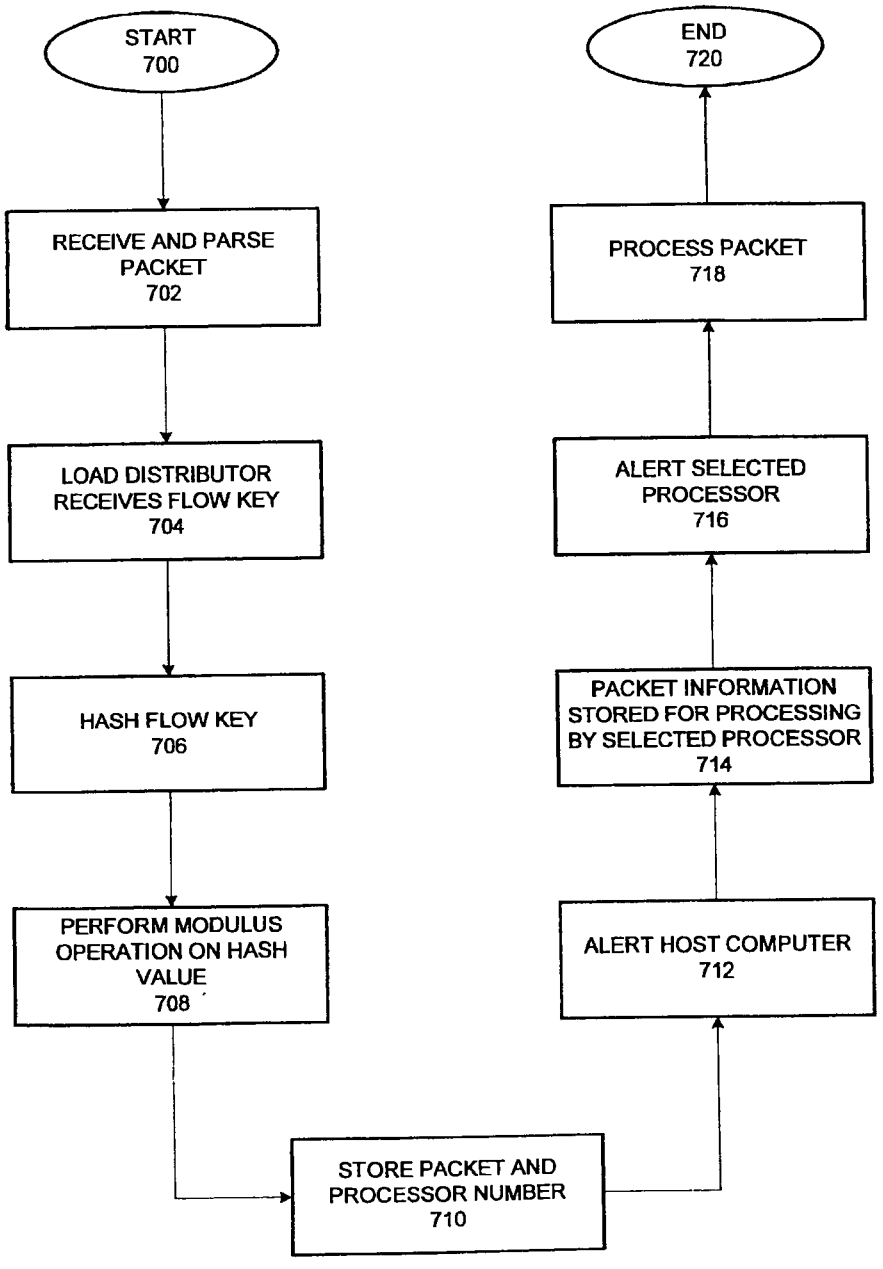

START
700
RECEIVE AND PARSE PACKET
702
LOAD DISTRIBUTOR RECEIVES FLOW KEY
704
HASH FLOW KEY
706
PERFORM MODULUS OPERATION ON HASH VALUE
708
STORE PACKET AND PROCESSOR NUMBER
710
ALERT HOST COMPUTER
712
PACKET INFORMATION STORED FOR PROCESSING BY SELECTED PROCESSOR
714
ALERT SELECTED PROCESSOR
716
PROCESS PACKET
718
END
720

FIG. 7

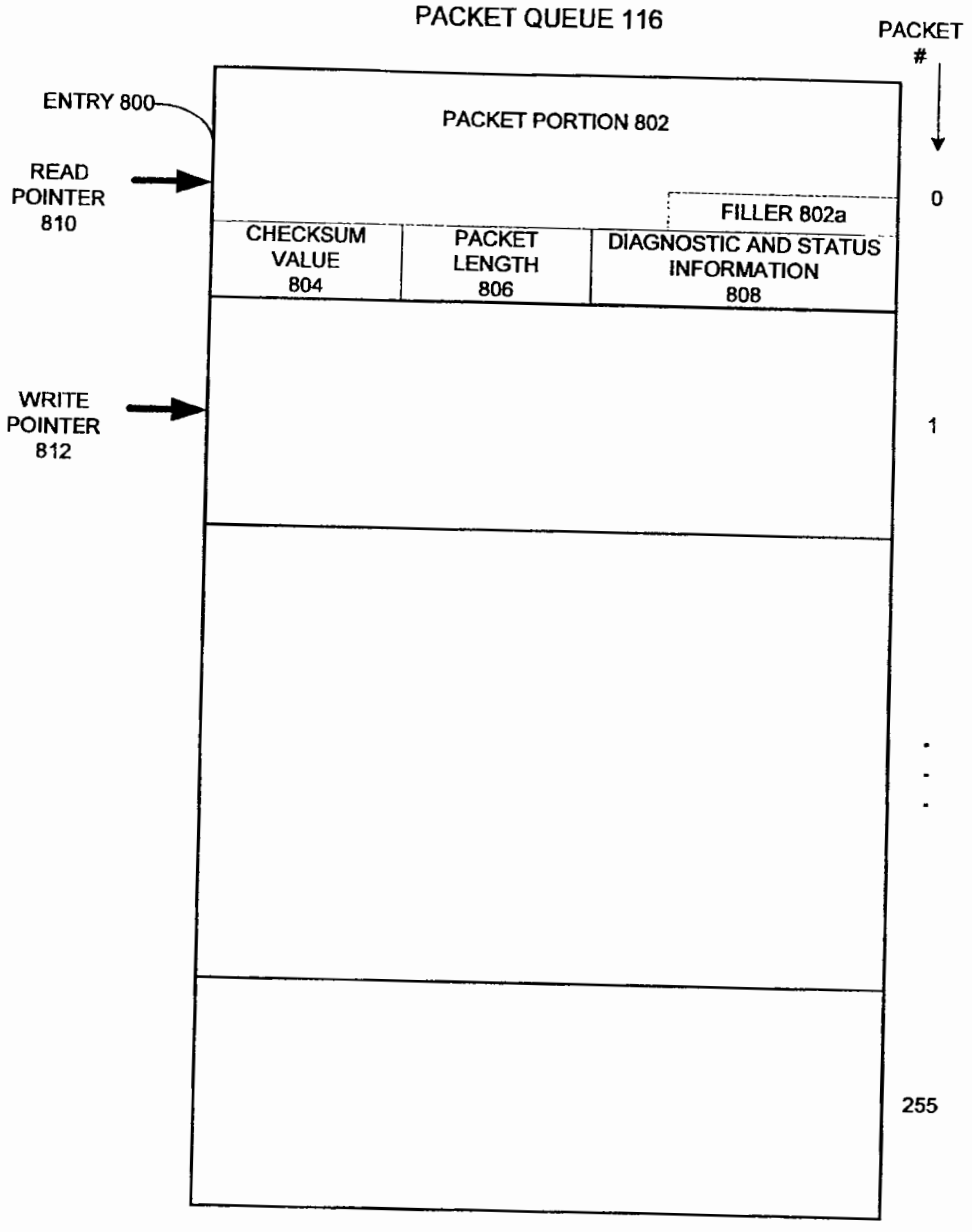

PACKET QUEUE 116
PACKET #
ENTRY 800
PACKET PORTION 802
READ POINTER 810
FILLER 802a
0
CHECKSUM VALUE 804
PACKET LENGTH 806
DIAGNOSTIC AND STATUS INFORMATION 808
WRITE POINTER 812
1
255

FIG. 8

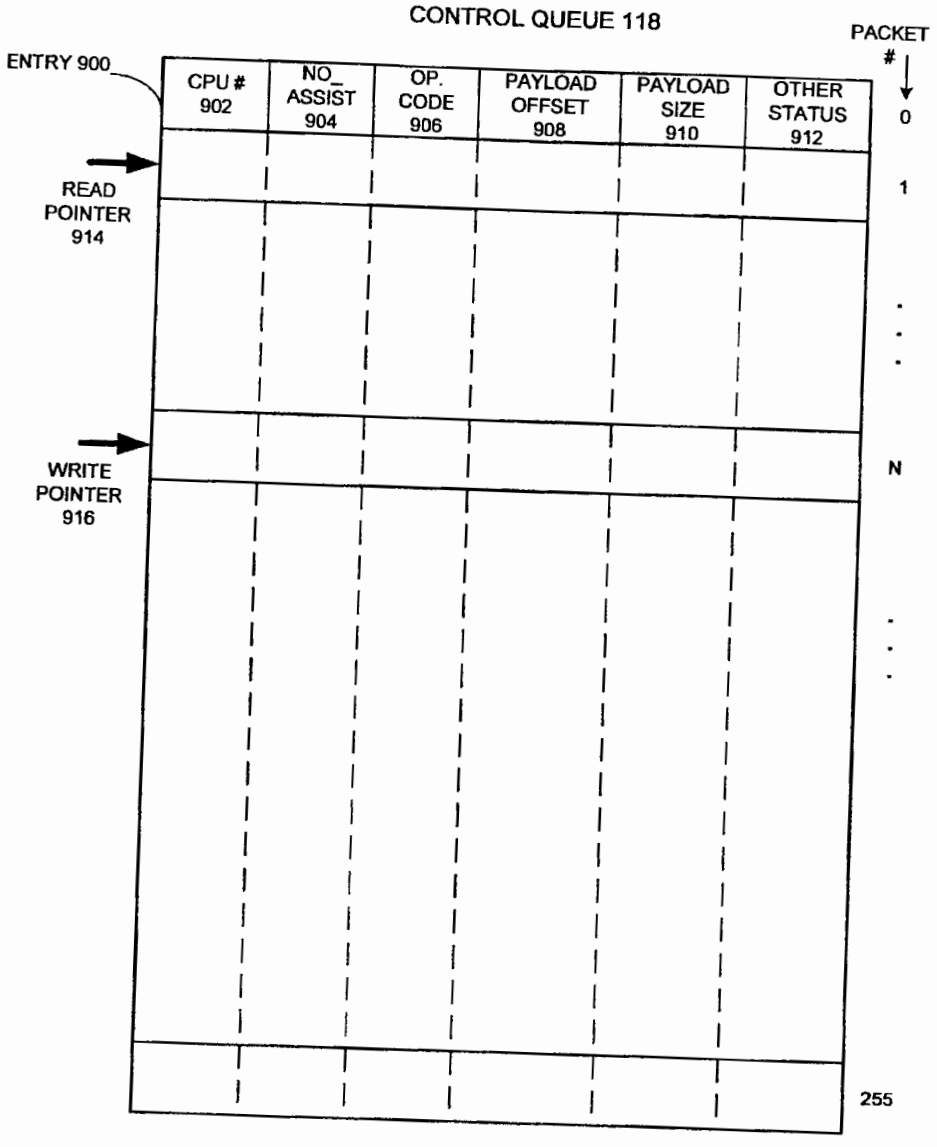

CONTROL QUEUE 118
PACKET #
ENTRY 900
CPU # 902
NO_ ASSIST 904
OP. CODE 906
PAYLOAD OFFSET 908
PAYLOAD SIZE 910
OTHER STATUS 912
0
READ POINTER 914
1
WRITE POINTER 916
N
255

FIG. 9

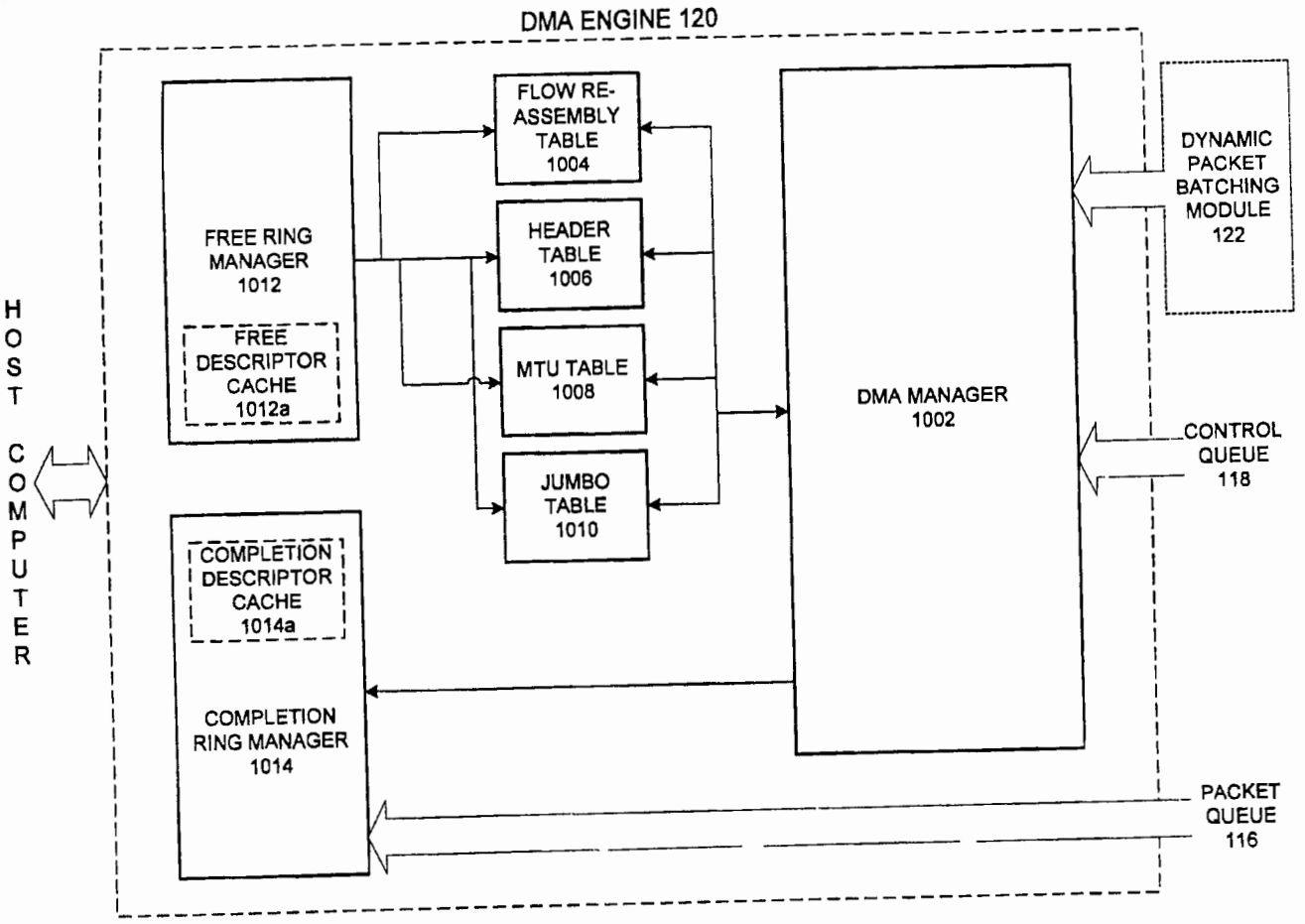

DMA ENGINE 120
FLOW RE-ASSEMBLY TABLE 1004
HEADER TABLE 1006
MTU TABLE 1008
JUMBO TABLE 1010
FREE RING MANAGER 1012
FREE DESCRIPTOR CACHE 1012a
COMPLETION DESCRIPTOR CACHE 1014a
COMPLETION RING MANAGER 1014
DMA MANAGER 1002
DYNAMIC PACKET BATCHING MODULE 122
CONTROL QUEUE 118
PACKET QUEUE 116
HOST COMPUTER

FIG. 10

FLOW RE-ASSEMBLY TABLE 1004

VALIDITY INDICATOR 1106	NEXT ADDRESS 1104	FLOW RE-ASSEMBLY BUFFER INDEX 1102	PACKET #
			0
			. . .
			63

HEADER TABLE 1006

VALIDITY INDICATOR 1116	NEXT ADDRESS 1114	HEADER BUFFER INDEX 1112

MTU TABLE 1008

VALIDITY INDICATOR 1126	NEXT ADDRESS 1124	MTU BUFFER INDEX 1122

JUMBO TABLE 1010

VALIDITY INDICATOR 1136	NEXT ADDRESS 1134	JUMBO BUFFER INDEX 1132

FIG. 11

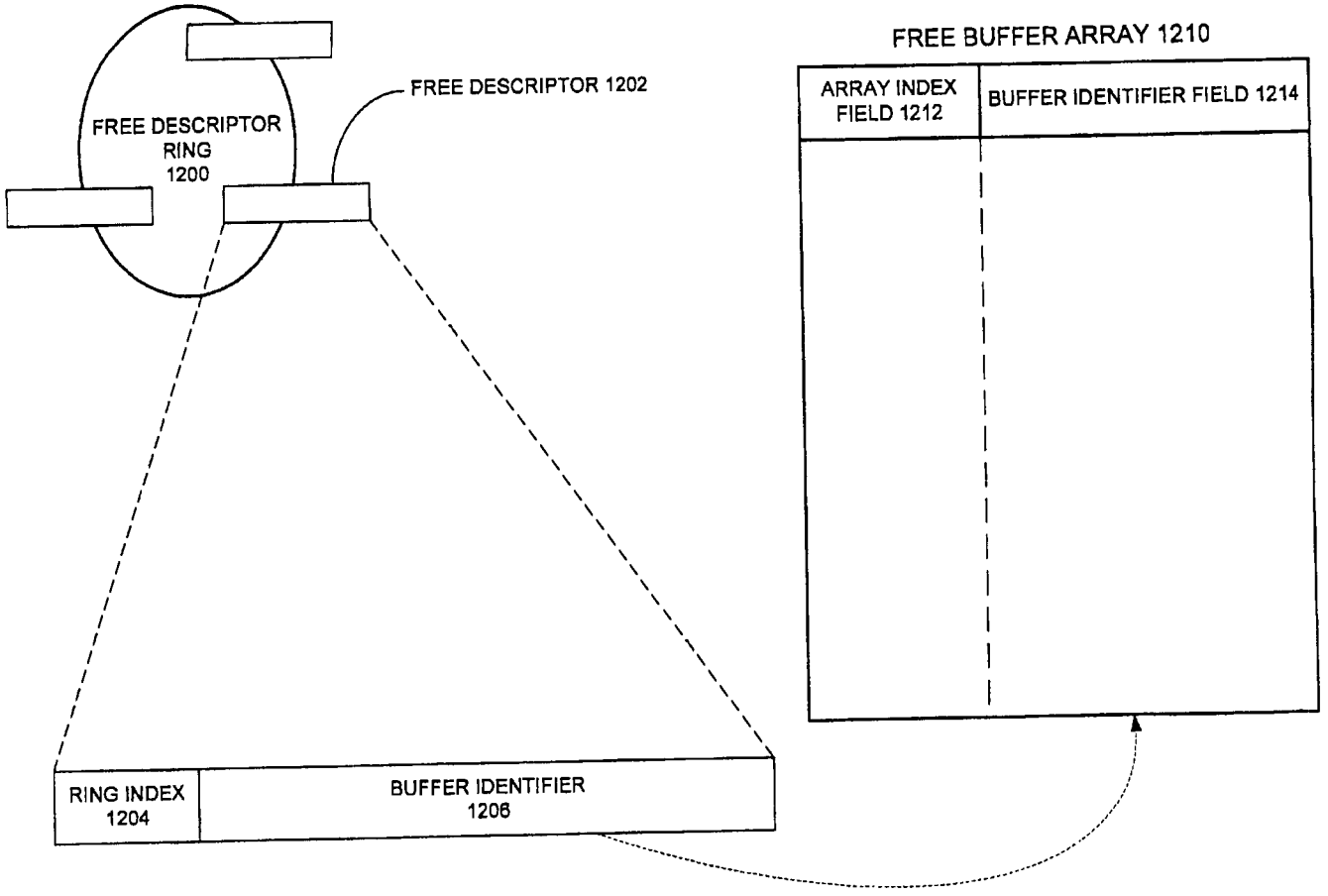

FREE DESCRIPTOR RING 1200
FREE DESCRIPTOR 1202
FREE BUFFER ARRAY 1210
ARRAY INDEX FIELD 1212
BUFFER IDENTIFIER FIELD 1214
RING INDEX 1204
BUFFER IDENTIFIER 1206

FIG. 12A

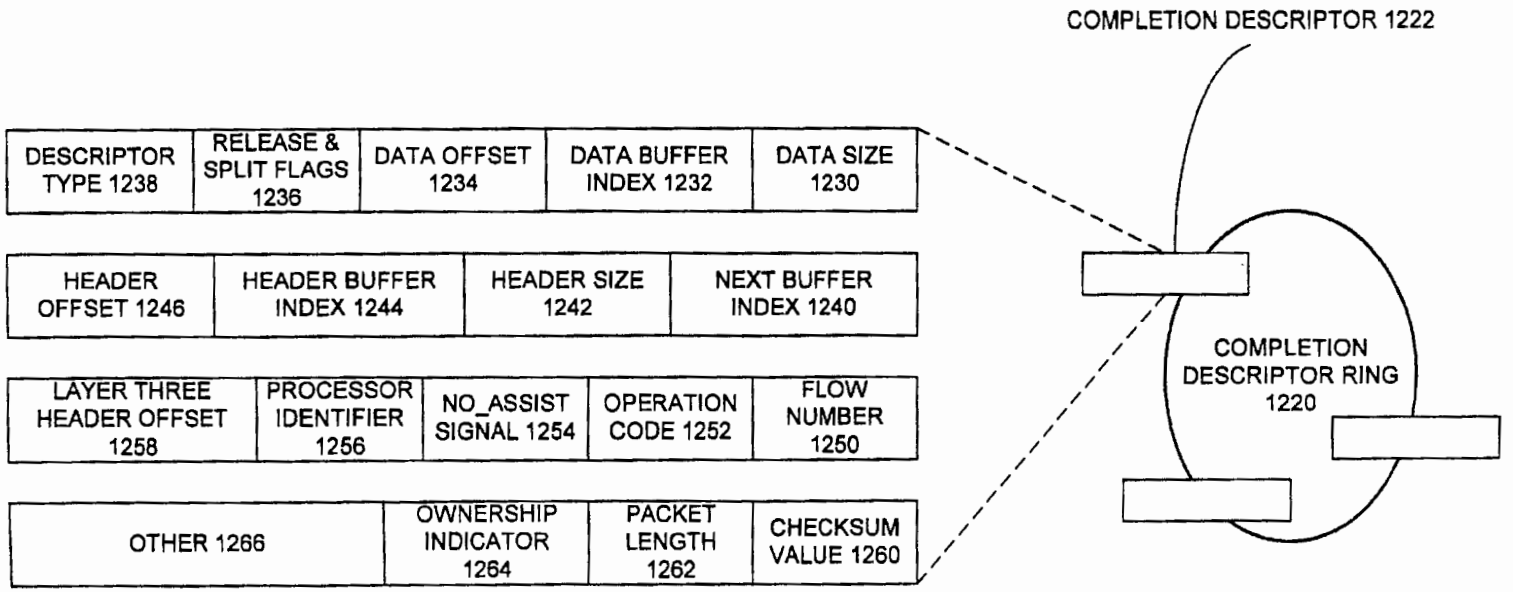

COMPLETION DESCRIPTOR 1222
DESCRIPTOR TYPE 1238
RELEASE & SPLIT FLAGS 1236
DATA OFFSET 1234
DATA BUFFER INDEX 1232
DATA SIZE 1230
HEADER OFFSET 1246
HEADER BUFFER INDEX 1244
HEADER SIZE 1242
NEXT BUFFER INDEX 1240
LAYER THREE HEADER OFFSET 1258
PROCESSOR IDENTIFIER 1256
NO_ASSIST SIGNAL 1254
OPERATION CODE 1252
FLOW NUMBER 1250
OTHER 1266
OWNERSHIP INDICATOR 1264
PACKET LENGTH 1262
CHECKSUM VALUE 1260
COMPLETION DESCRIPTOR RING 1220

FIG. 12B

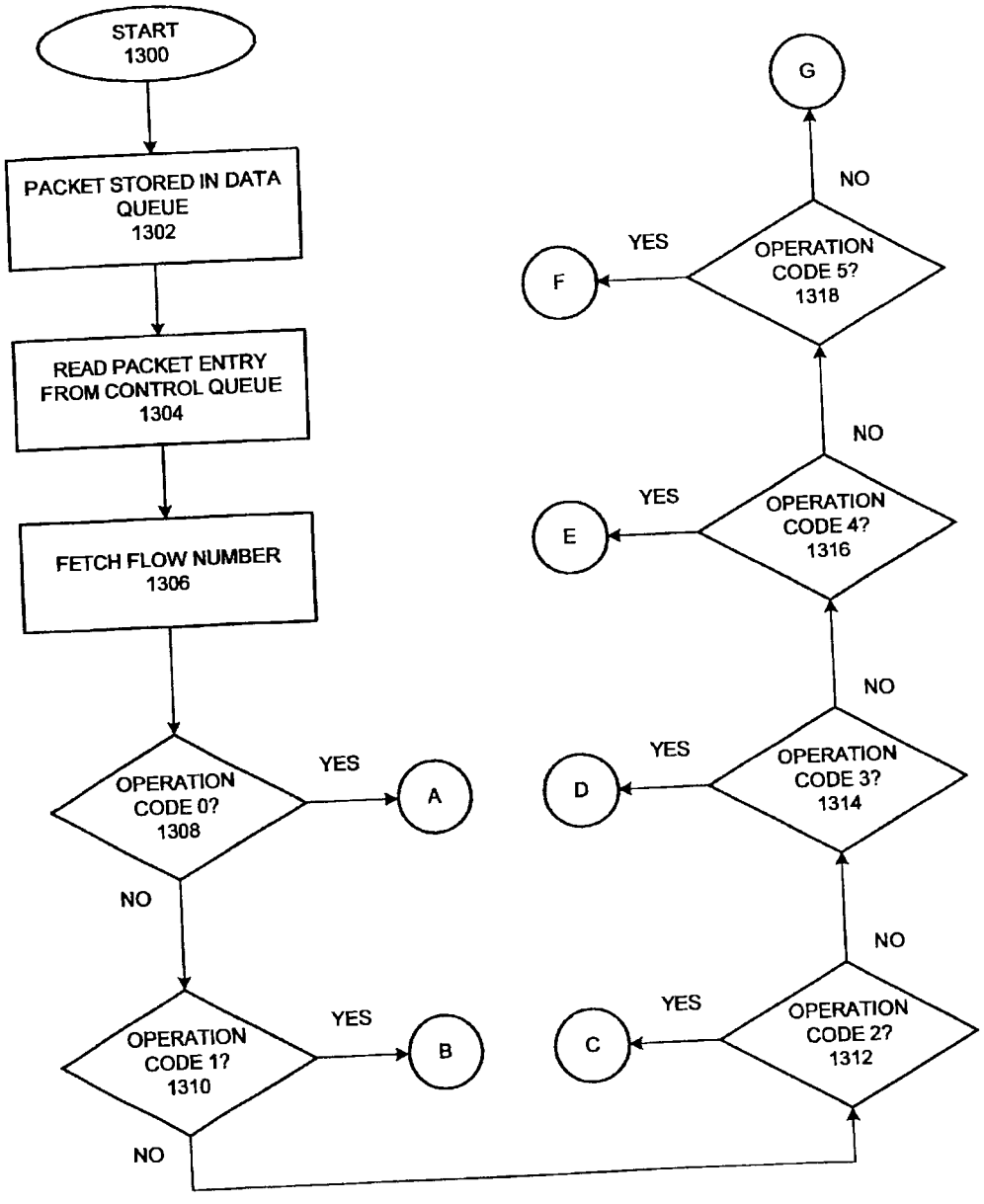

START
1300
PACKET STORED IN DATA QUEUE
1302
READ PACKET ENTRY FROM CONTROL QUEUE
1304
FETCH FLOW NUMBER
1306
OPERATION CODE 0?
1308
YES
A
NO
OPERATION CODE 1?
1310
YES
B
NO
OPERATION CODE 2?
1312
YES
C
NO
OPERATION CODE 3?
1314
YES
D
NO
OPERATION CODE 4?
1316
YES
E
NO
OPERATION CODE 5?
1318
YES
F
NO
G

FIG. 13

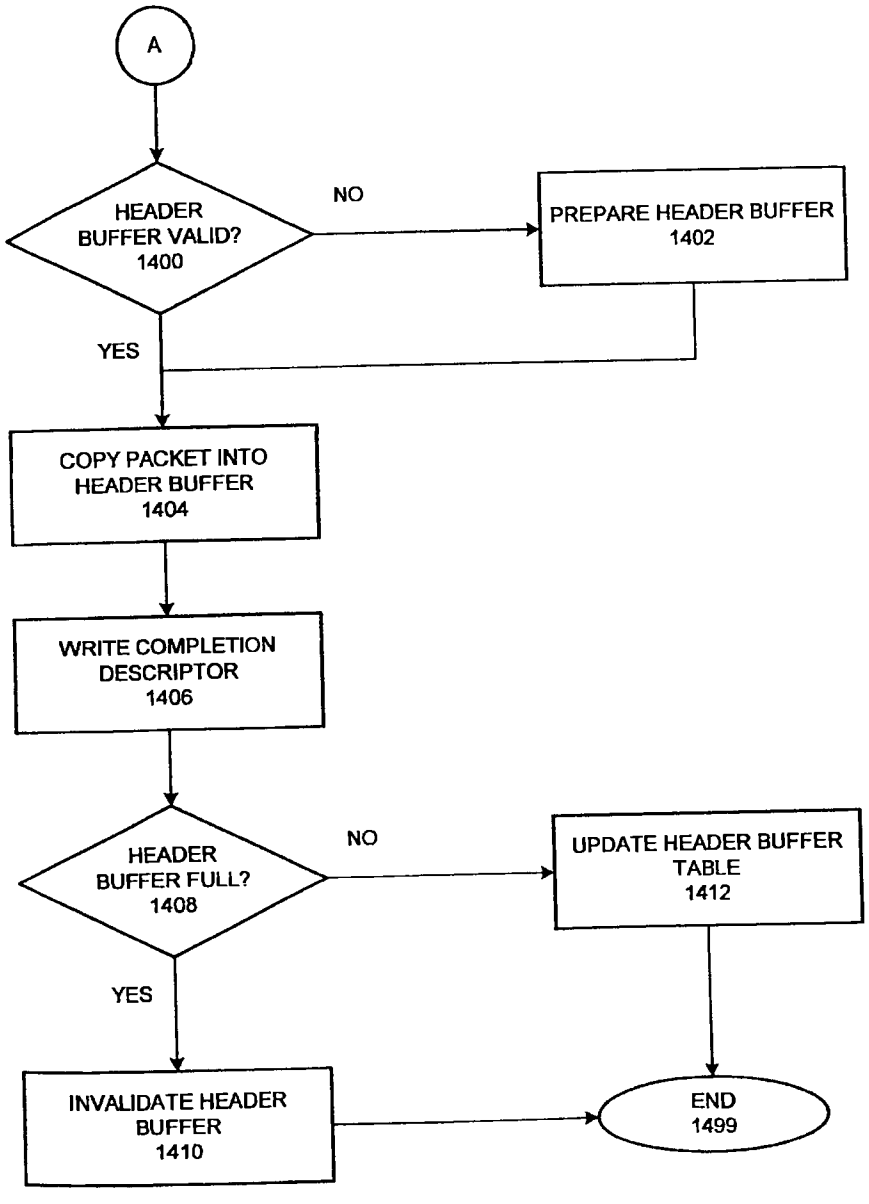

A
HEADER BUFFER VALID? 1400
NO
PREPARE HEADER BUFFER 1402
YES
COPY PACKET INTO HEADER BUFFER 1404
WRITE COMPLETION DESCRIPTOR 1406
HEADER BUFFER FULL? 1408
NO
UPDATE HEADER BUFFER TABLE 1412
YES
INVALIDATE HEADER BUFFER 1410
END 1499

FIG. 14

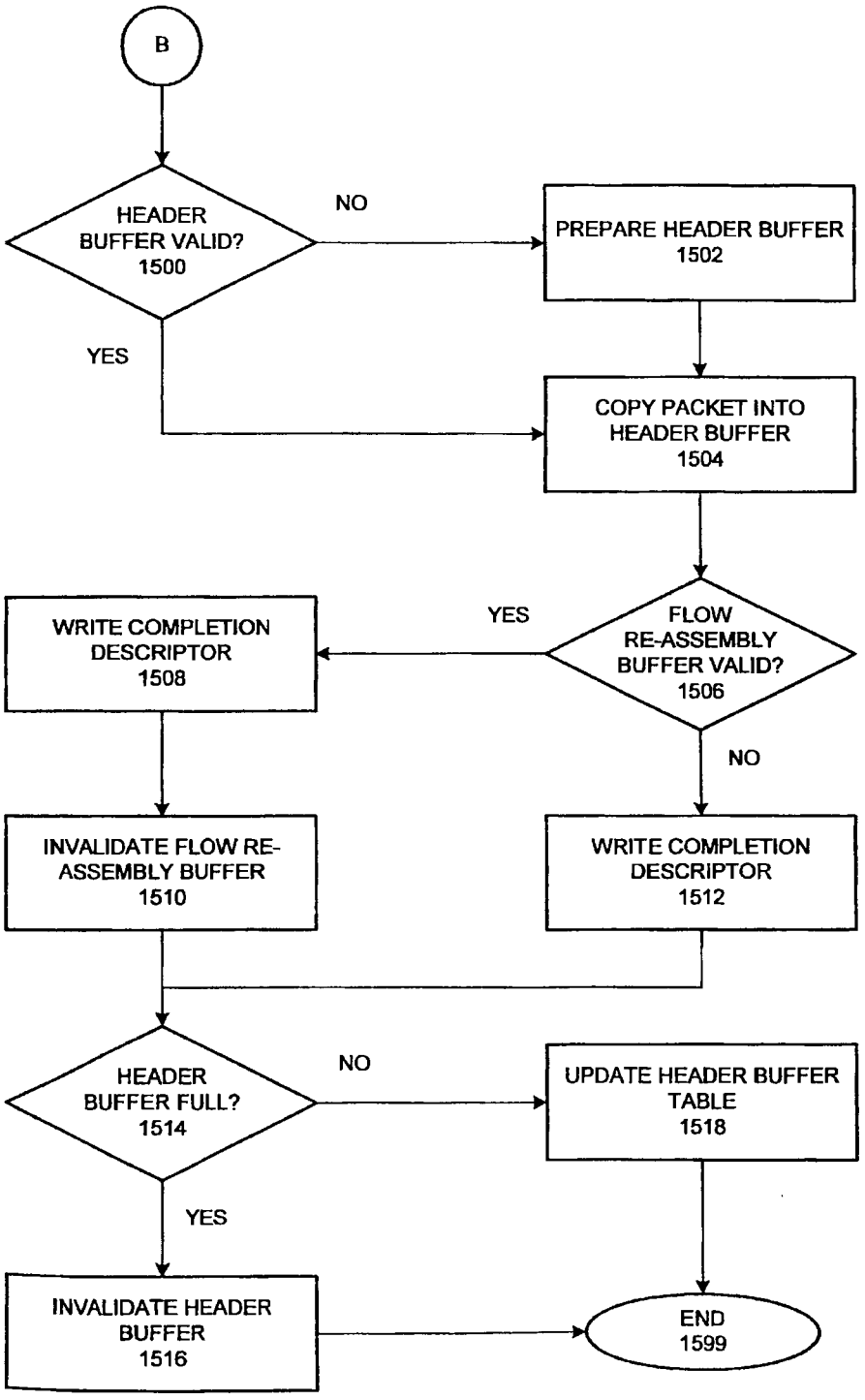

B
HEADER BUFFER VALID? 1500
NO
PREPARE HEADER BUFFER 1502
YES
COPY PACKET INTO HEADER BUFFER 1504
FLOW RE-ASSEMBLY BUFFER VALID? 1506
YES
WRITE COMPLETION DESCRIPTOR 1508
NO
INVALIDATE FLOW RE-ASSEMBLY BUFFER 1510
WRITE COMPLETION DESCRIPTOR 1512
HEADER BUFFER FULL? 1514
NO
UPDATE HEADER BUFFER TABLE 1518
YES
INVALIDATE HEADER BUFFER 1516
END 1599

FIG. 15

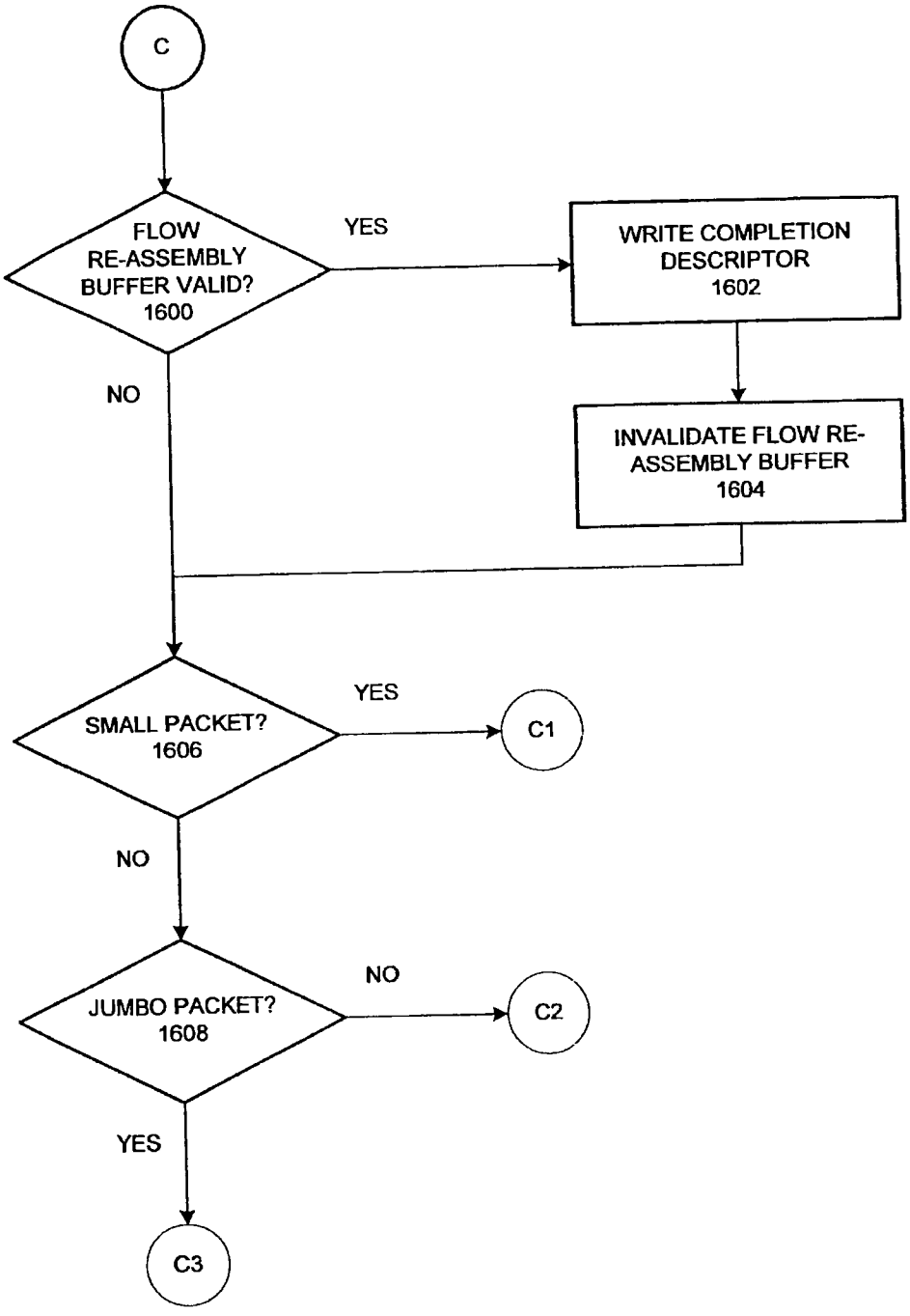

C
FLOW RE-ASSEMBLY BUFFER VALID? 1600
YES
WRITE COMPLETION DESCRIPTOR 1602
NO
INVALIDATE FLOW RE-ASSEMBLY BUFFER 1604
SMALL PACKET? 1606
YES
C1
NO
JUMBO PACKET? 1608
NO
C2
YES
C3

FIG. 16A

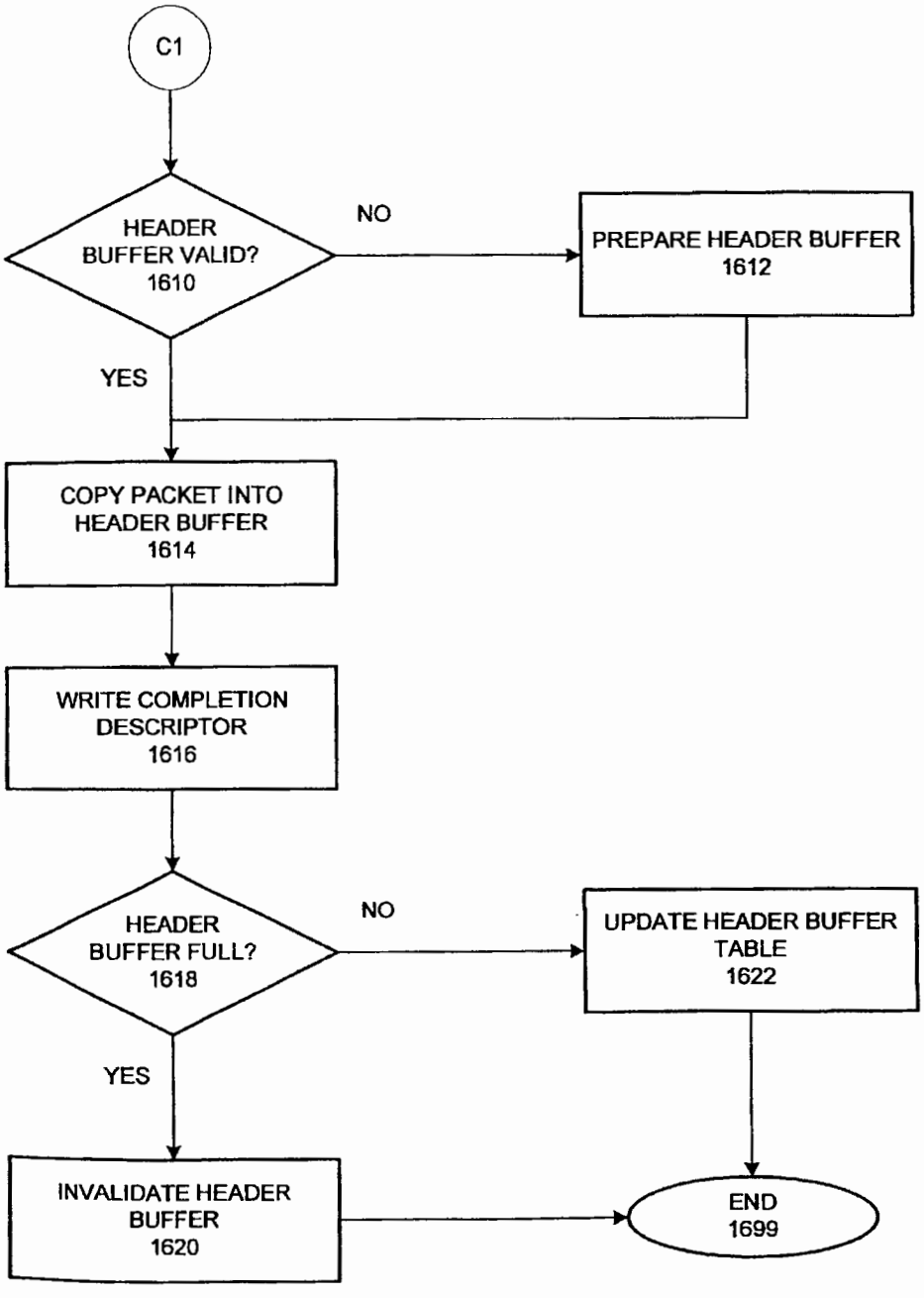

C1
HEADER BUFFER VALID? 1610
NO
PREPARE HEADER BUFFER 1612
YES
COPY PACKET INTO HEADER BUFFER 1614
WRITE COMPLETION DESCRIPTOR 1616
HEADER BUFFER FULL? 1618
NO
UPDATE HEADER BUFFER TABLE 1622
YES
INVALIDATE HEADER BUFFER 1620
END 1699

FIG. 16B

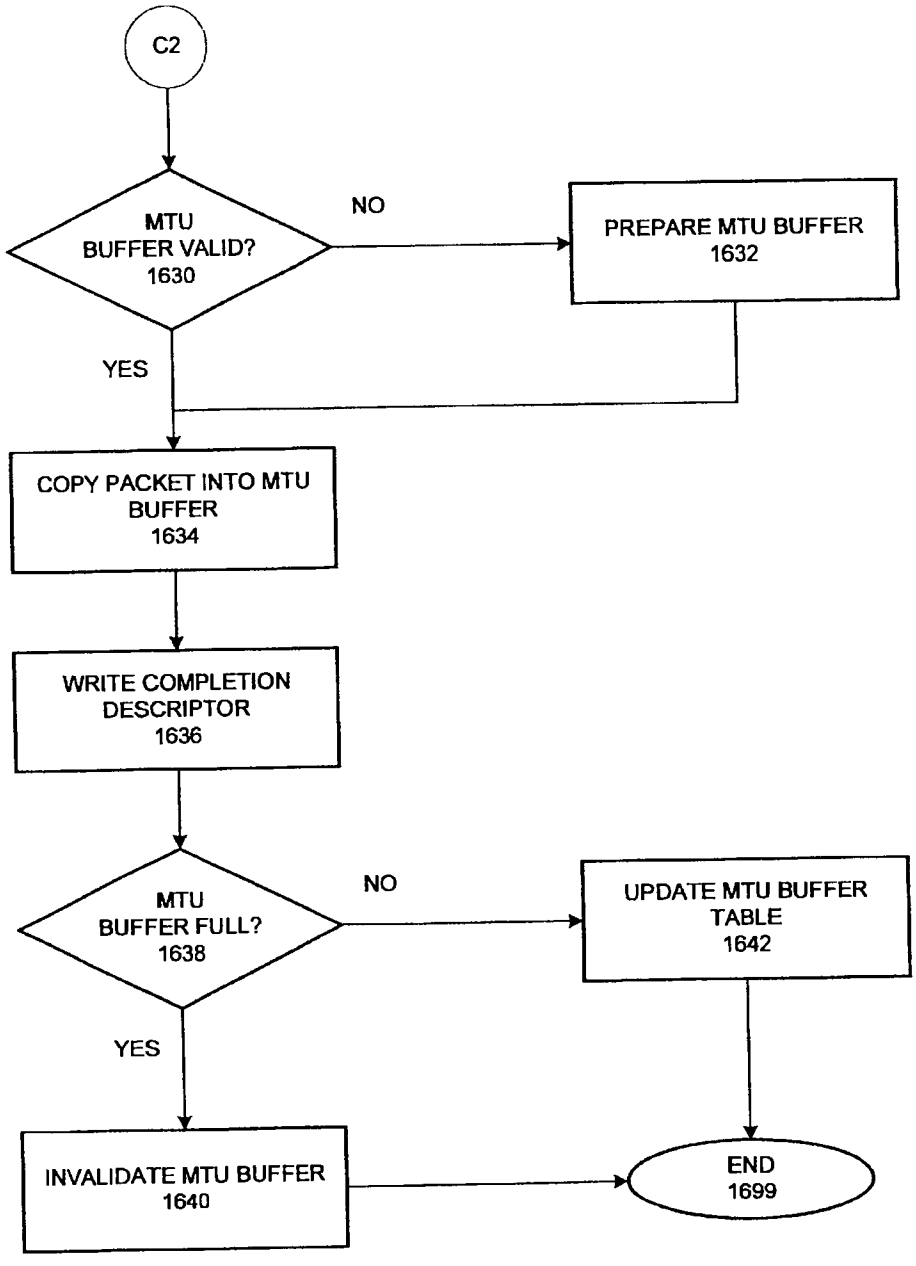

C2
MTU BUFFER VALID? 1630
NO
PREPARE MTU BUFFER 1632
YES
COPY PACKET INTO MTU BUFFER 1634
WRITE COMPLETION DESCRIPTOR 1636
MTU BUFFER FULL? 1638
NO
UPDATE MTU BUFFER TABLE 1642
YES
INVALIDATE MTU BUFFER 1640
END 1699

FIG. 16C

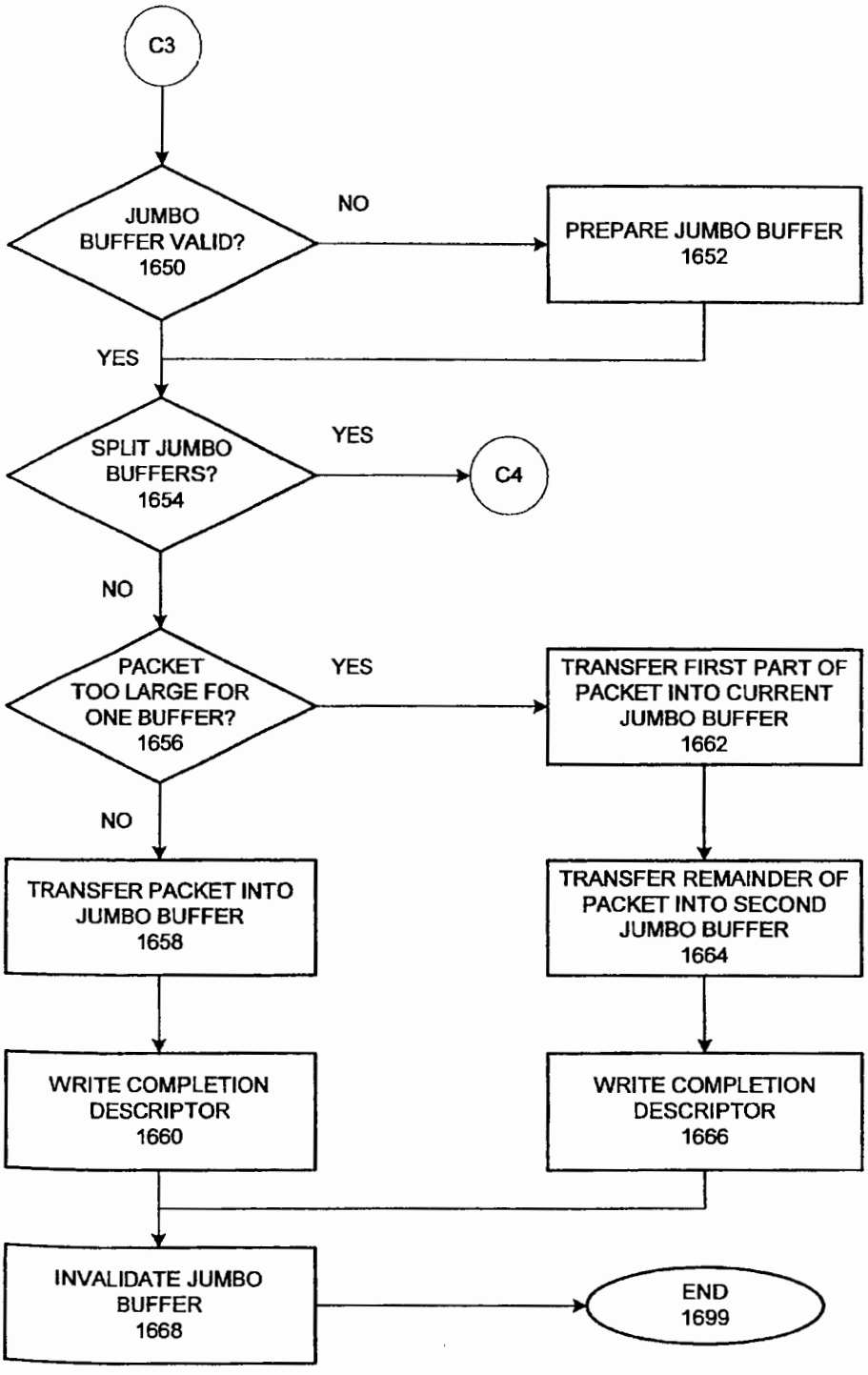

C3
JUMBO BUFFER VALID? 1650
NO
PREPARE JUMBO BUFFER 1652
YES
SPLIT JUMBO BUFFERS? 1654
YES
C4
NO
PACKET TOO LARGE FOR ONE BUFFER? 1656
YES
TRANSFER FIRST PART OF PACKET INTO CURRENT JUMBO BUFFER 1662
NO
TRANSFER PACKET INTO JUMBO BUFFER 1658
TRANSFER REMAINDER OF PACKET INTO SECOND JUMBO BUFFER 1664
WRITE COMPLETION DESCRIPTOR 1660
WRITE COMPLETION DESCRIPTOR 1666
INVALIDATE JUMBO BUFFER 1668
END 1699

FIG. 16D

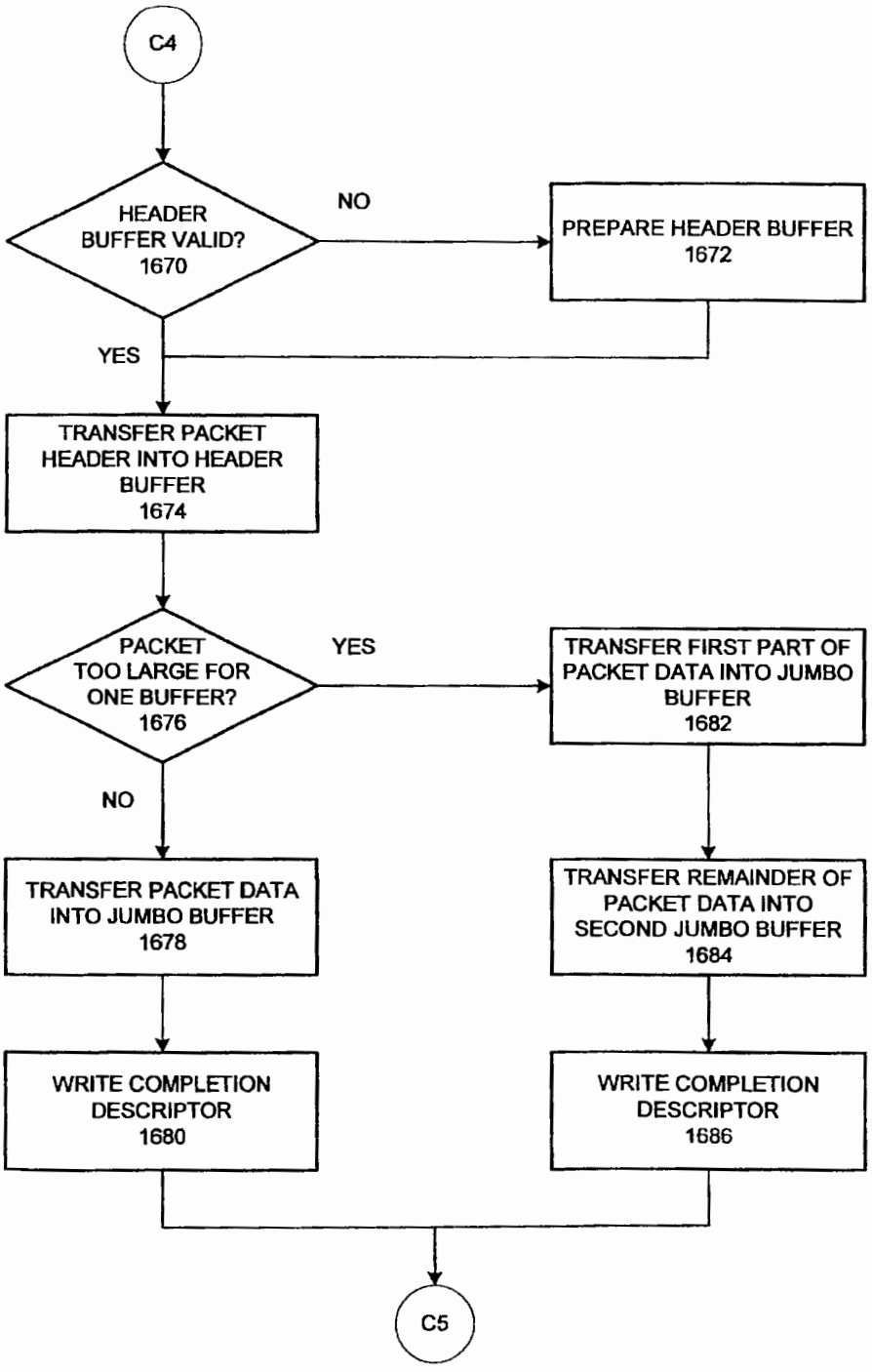

C4
HEADER BUFFER VALID? 1670
NO
PREPARE HEADER BUFFER 1672
YES
TRANSFER PACKET HEADER INTO HEADER BUFFER 1674
PACKET TOO LARGE FOR ONE BUFFER? 1676
YES
TRANSFER FIRST PART OF PACKET DATA INTO JUMBO BUFFER 1682
NO
TRANSFER PACKET DATA INTO JUMBO BUFFER 1678
TRANSFER REMAINDER OF PACKET DATA INTO SECOND JUMBO BUFFER 1684
WRITE COMPLETION DESCRIPTOR 1680
WRITE COMPLETION DESCRIPTOR 1686
C5

FIG. 16E

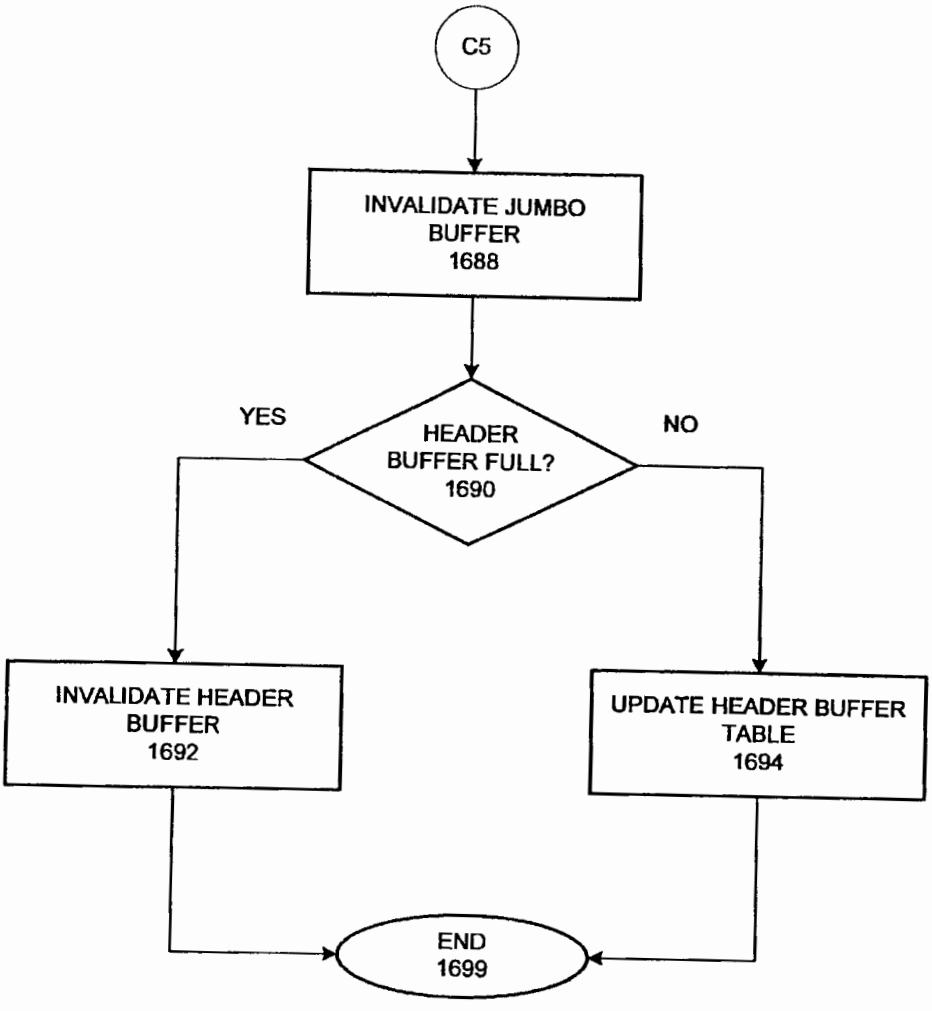

C5
INVALIDATE JUMBO BUFFER 1688
HEADER BUFFER FULL? 1690
YES
NO
INVALIDATE HEADER BUFFER 1692
UPDATE HEADER BUFFER TABLE 1694
END 1699

FIG. 16F

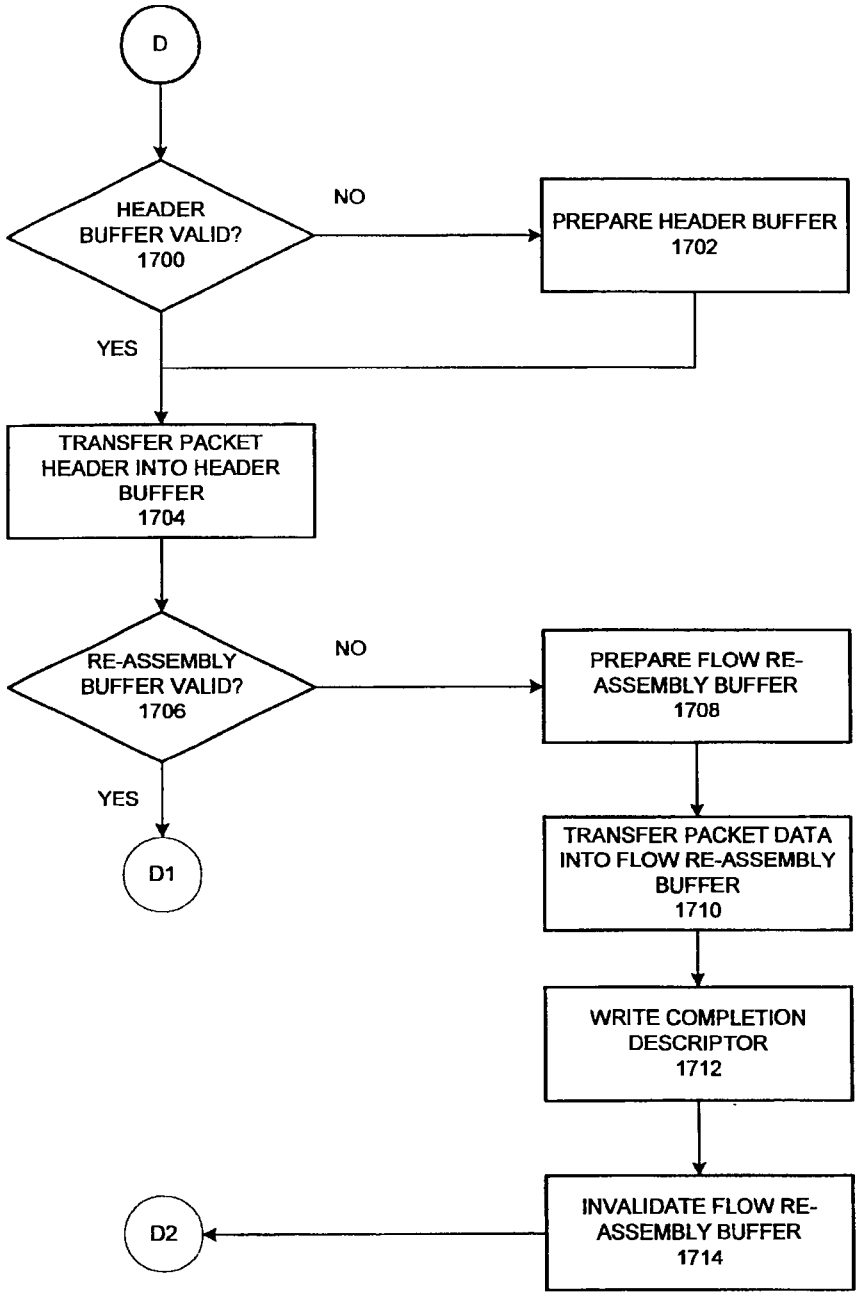

D
HEADER BUFFER VALID? 1700
NO
PREPARE HEADER BUFFER 1702
YES
TRANSFER PACKET HEADER INTO HEADER BUFFER 1704
RE-ASSEMBLY BUFFER VALID? 1706
NO
PREPARE FLOW RE-ASSEMBLY BUFFER 1708
YES
D1
TRANSFER PACKET DATA INTO FLOW RE-ASSEMBLY BUFFER 1710
WRITE COMPLETION DESCRIPTOR 1712
INVALIDATE FLOW RE-ASSEMBLY BUFFER 1714
D2

FIG. 17A

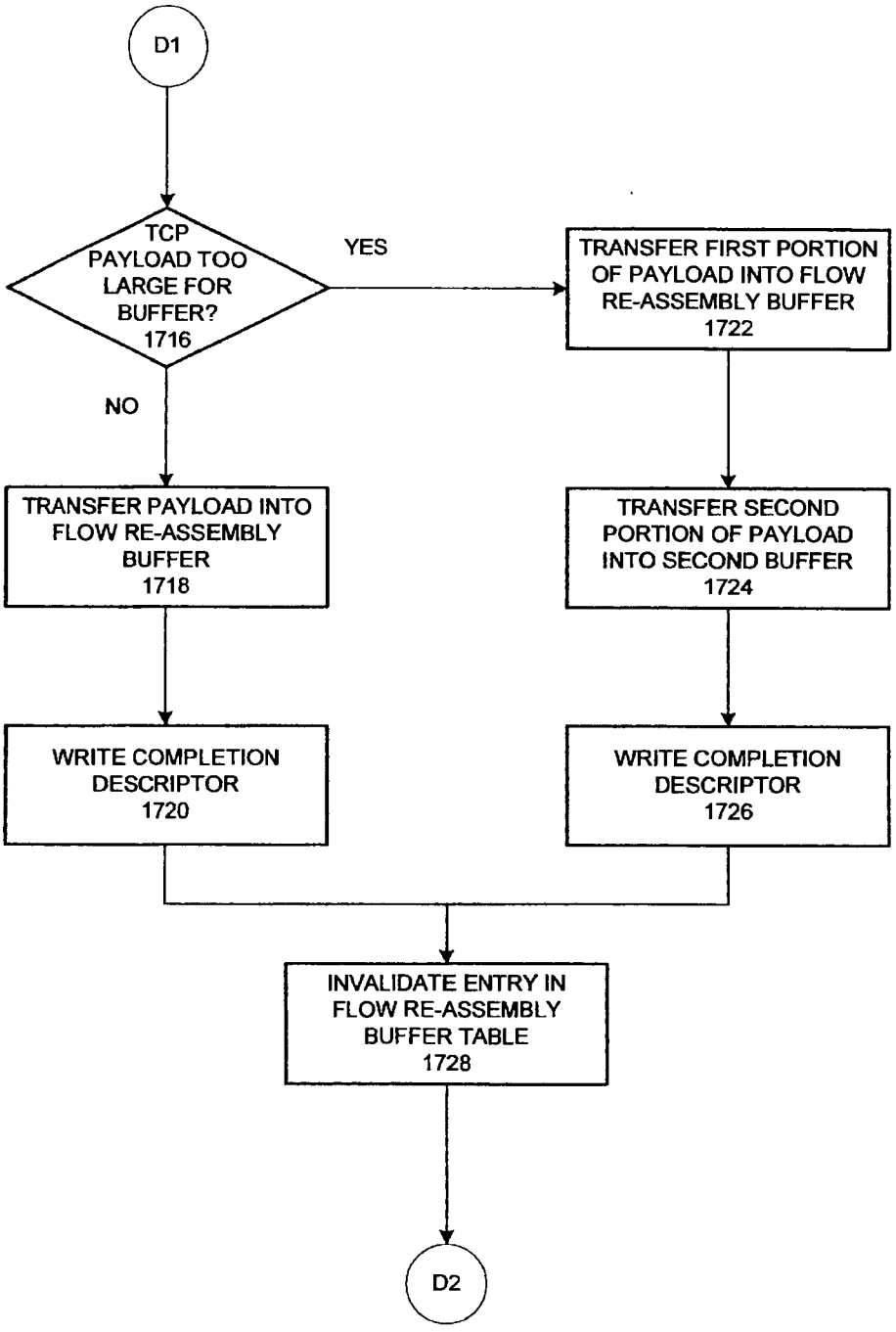

D1
TCP PAYLOAD TOO LARGE FOR BUFFER? 1716
YES
NO
TRANSFER FIRST PORTION OF PAYLOAD INTO FLOW RE-ASSEMBLY BUFFER 1722
TRANSFER PAYLOAD INTO FLOW RE-ASSEMBLY BUFFER 1718
TRANSFER SECOND PORTION OF PAYLOAD INTO SECOND BUFFER 1724
WRITE COMPLETION DESCRIPTOR 1720
WRITE COMPLETION DESCRIPTOR 1726
INVALIDATE ENTRY IN FLOW RE-ASSEMBLY BUFFER TABLE 1728
D2

FIG. 17B

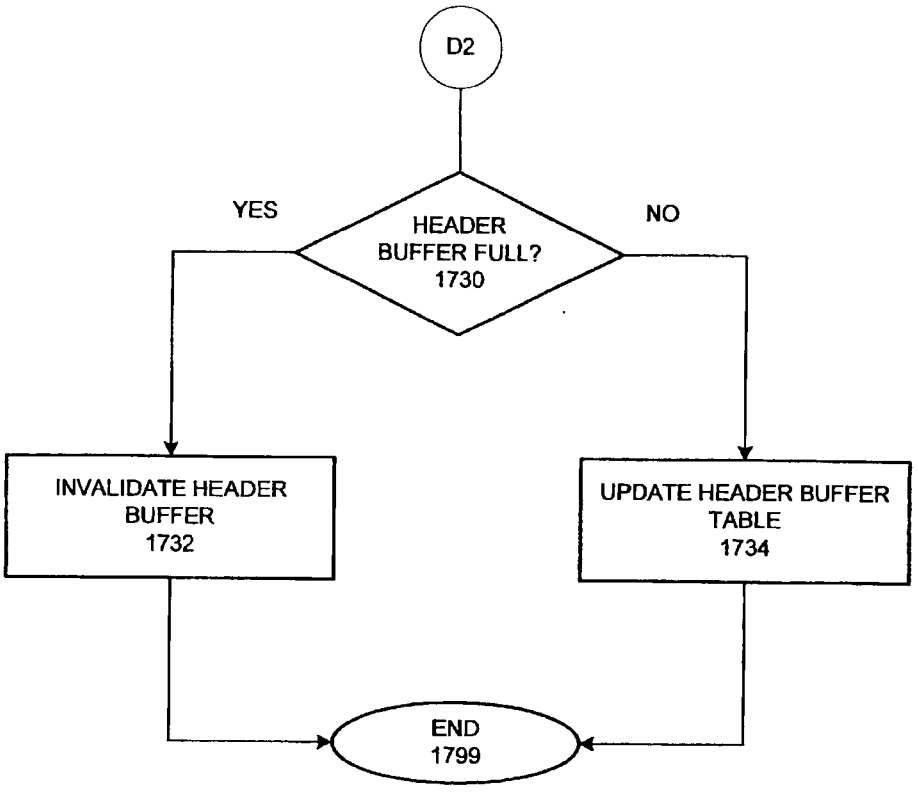

D2
YES
HEADER BUFFER FULL? 1730
NO
INVALIDATE HEADER BUFFER 1732
UPDATE HEADER BUFFER TABLE 1734
END 1799

FIG. 17C

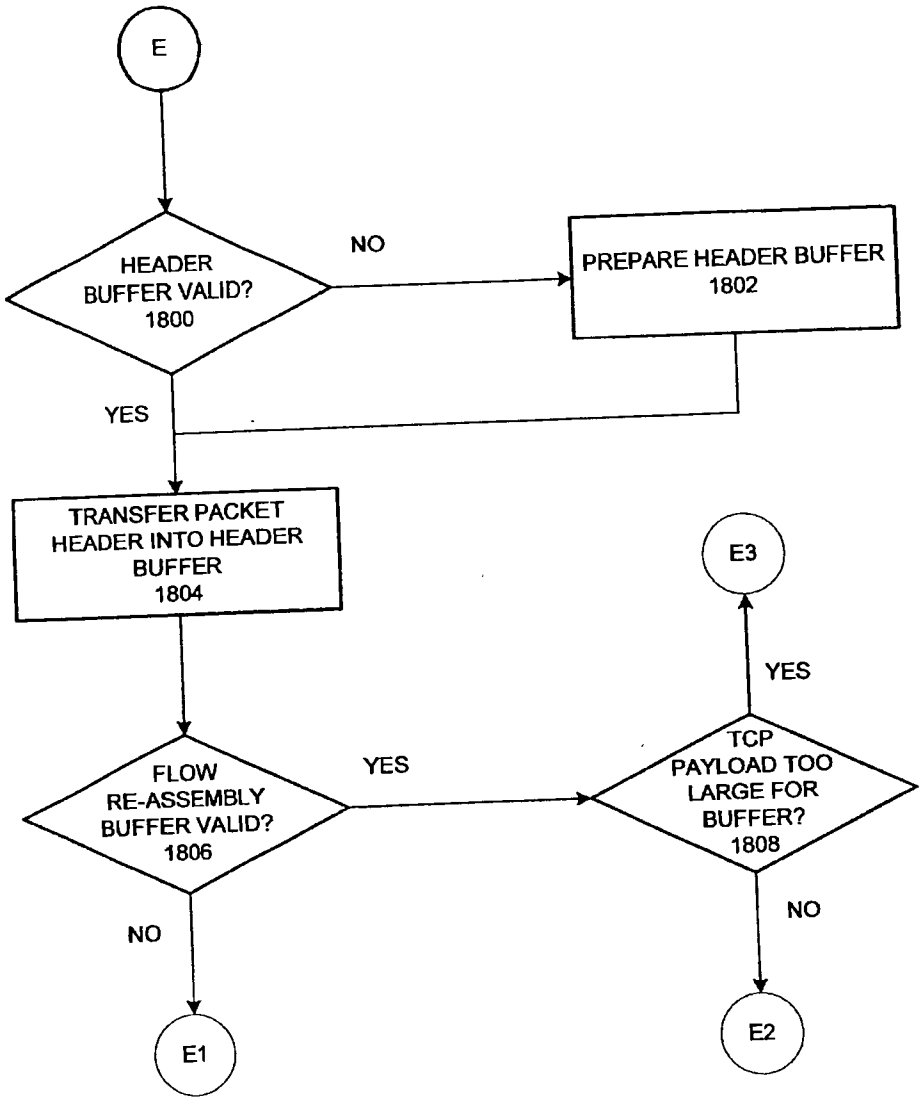

E
HEADER BUFFER VALID? 1800
NO
PREPARE HEADER BUFFER 1802
YES
TRANSFER PACKET HEADER INTO HEADER BUFFER 1804
FLOW RE-ASSEMBLY BUFFER VALID? 1806
YES
TCP PAYLOAD TOO LARGE FOR BUFFER? 1808
YES
E3
NO
E2
NO
E1

FIG. 18A

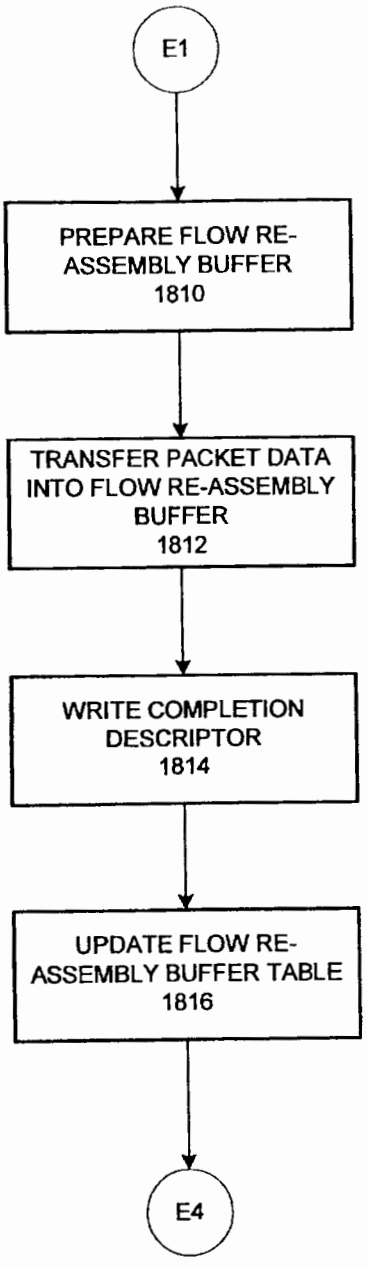

E1
PREPARE FLOW RE-ASSEMBLY BUFFER 1810
TRANSFER PACKET DATA INTO FLOW RE-ASSEMBLY BUFFER 1812
WRITE COMPLETION DESCRIPTOR 1814
UPDATE FLOW RE-ASSEMBLY BUFFER TABLE 1816
E4

FIG. 18B

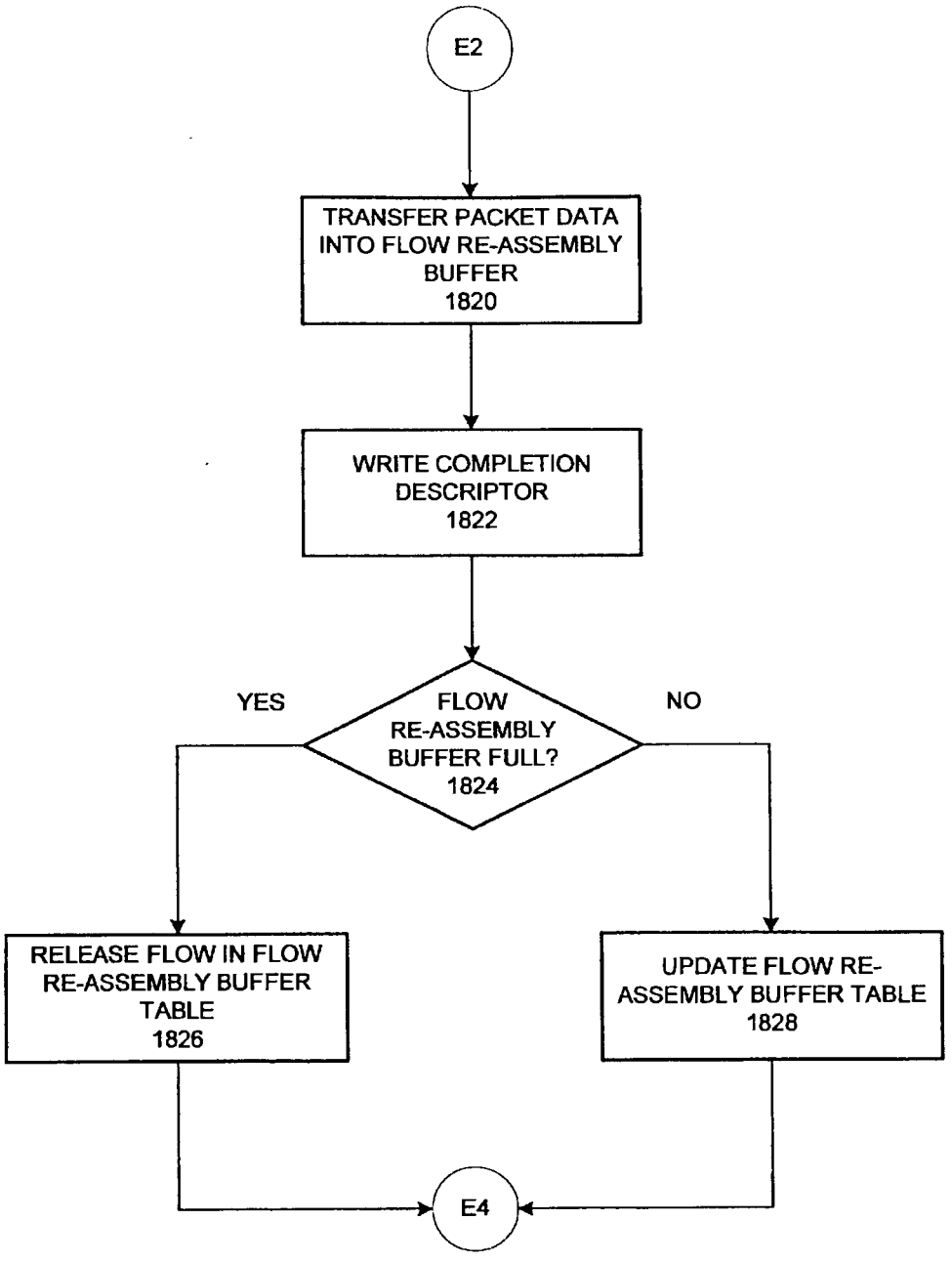

E2
TRANSFER PACKET DATA INTO FLOW RE-ASSEMBLY BUFFER 1820
WRITE COMPLETION DESCRIPTOR 1822
YES
FLOW RE-ASSEMBLY BUFFER FULL? 1824
NO
RELEASE FLOW IN FLOW RE-ASSEMBLY BUFFER TABLE 1826
UPDATE FLOW RE-ASSEMBLY BUFFER TABLE 1828
E4

FIG. 18C

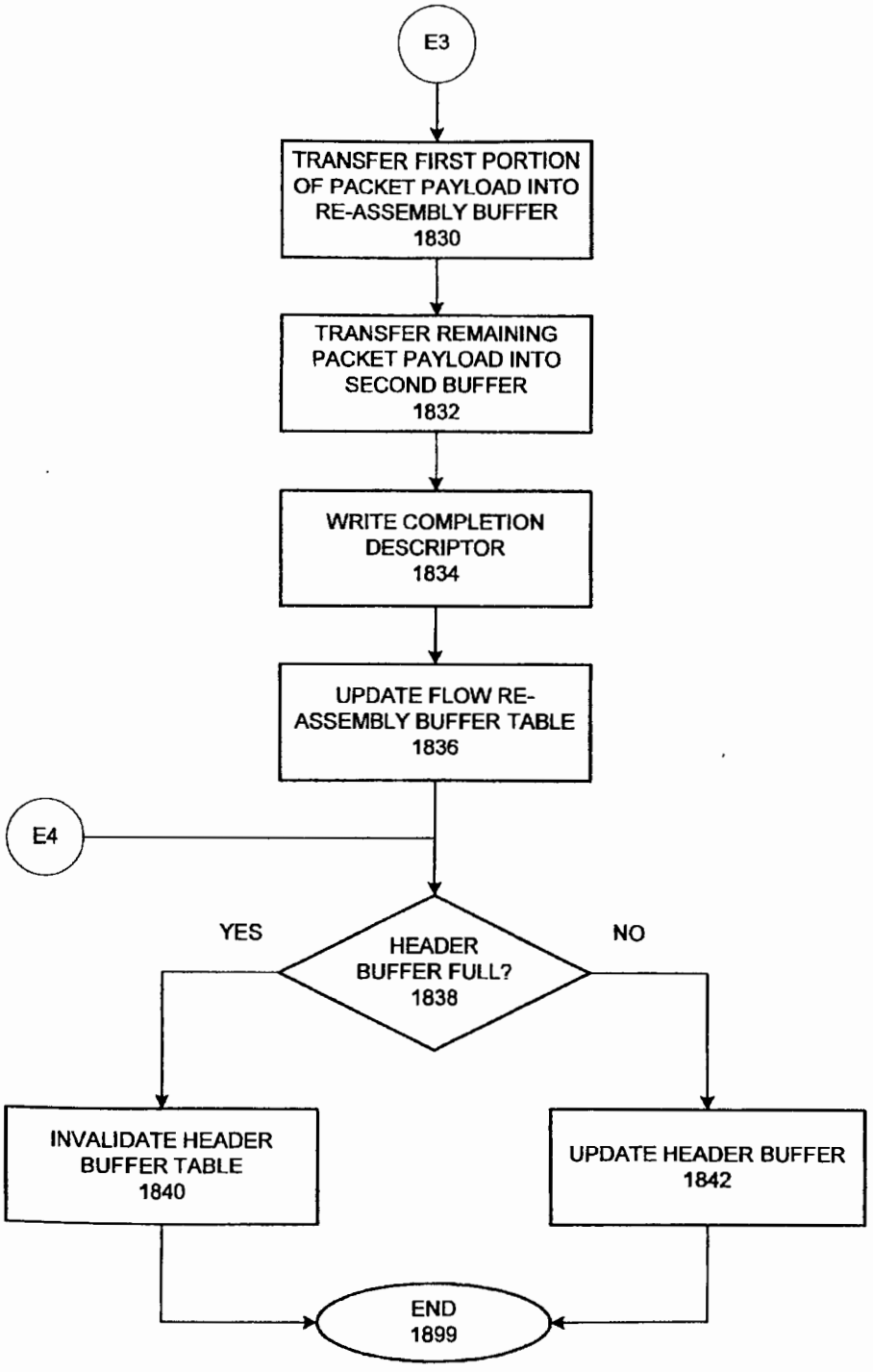

E3
TRANSFER FIRST PORTION OF PACKET PAYLOAD INTO RE-ASSEMBLY BUFFER 1830
TRANSFER REMAINING PACKET PAYLOAD INTO SECOND BUFFER 1832
WRITE COMPLETION DESCRIPTOR 1834
UPDATE FLOW RE-ASSEMBLY BUFFER TABLE 1836
E4
HEADER BUFFER FULL? 1838
YES
NO
INVALIDATE HEADER BUFFER TABLE 1840
UPDATE HEADER BUFFER 1842
END 1899

FIG. 18D

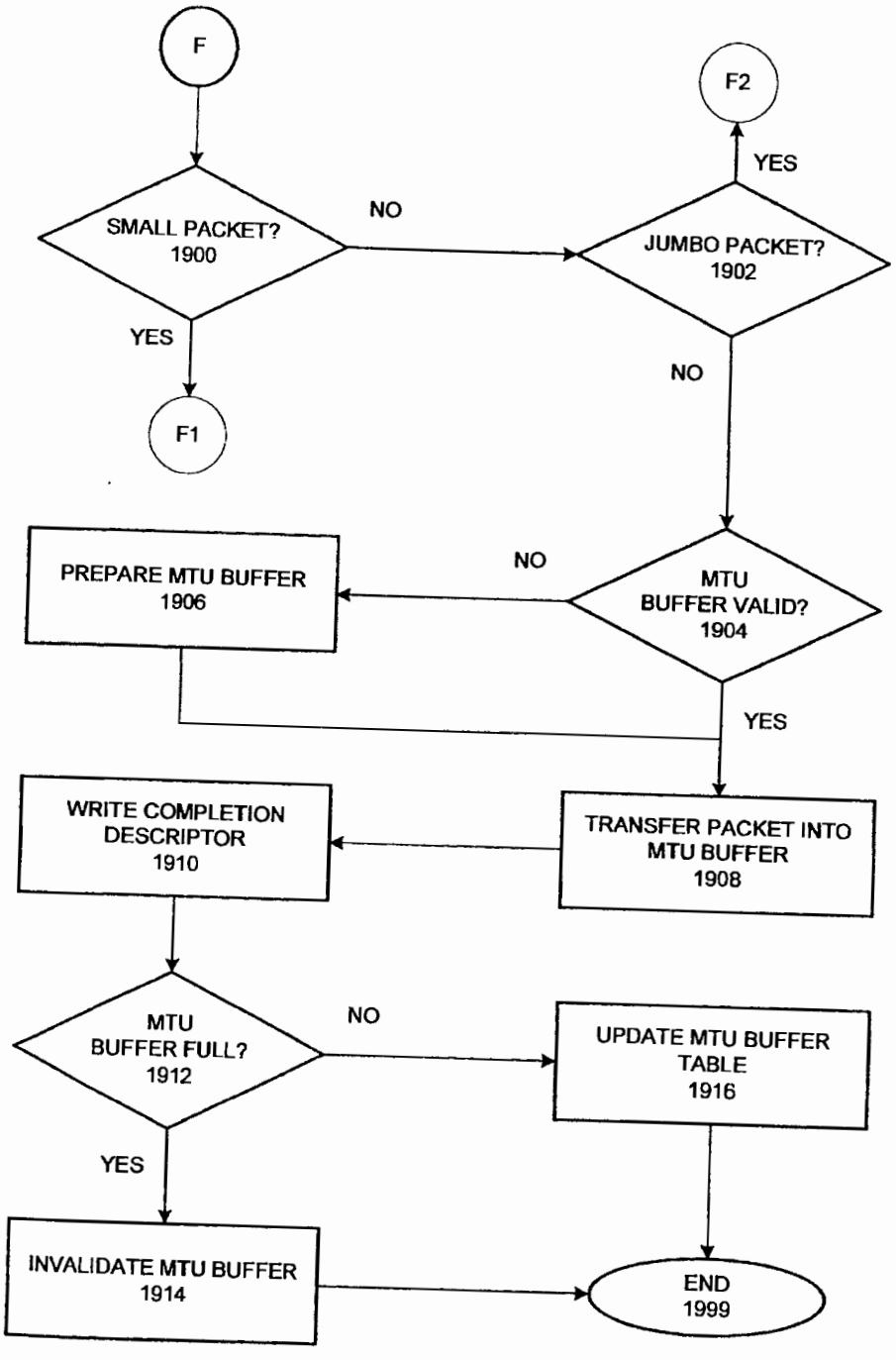

F
F2
YES
SMALL PACKET?
1900
NO
JUMBO PACKET?
1902
YES
NO
F1
PREPARE MTU BUFFER
1906
NO
MTU
BUFFER VALID?
1904
YES
WRITE COMPLETION
DESCRIPTOR
1910
TRANSFER PACKET INTO
MTU BUFFER
1908
MTU
BUFFER FULL?
1912
NO
UPDATE MTU BUFFER
TABLE
1916
YES
INVALIDATE MTU BUFFER
1914
END
1999

FIG. 19A

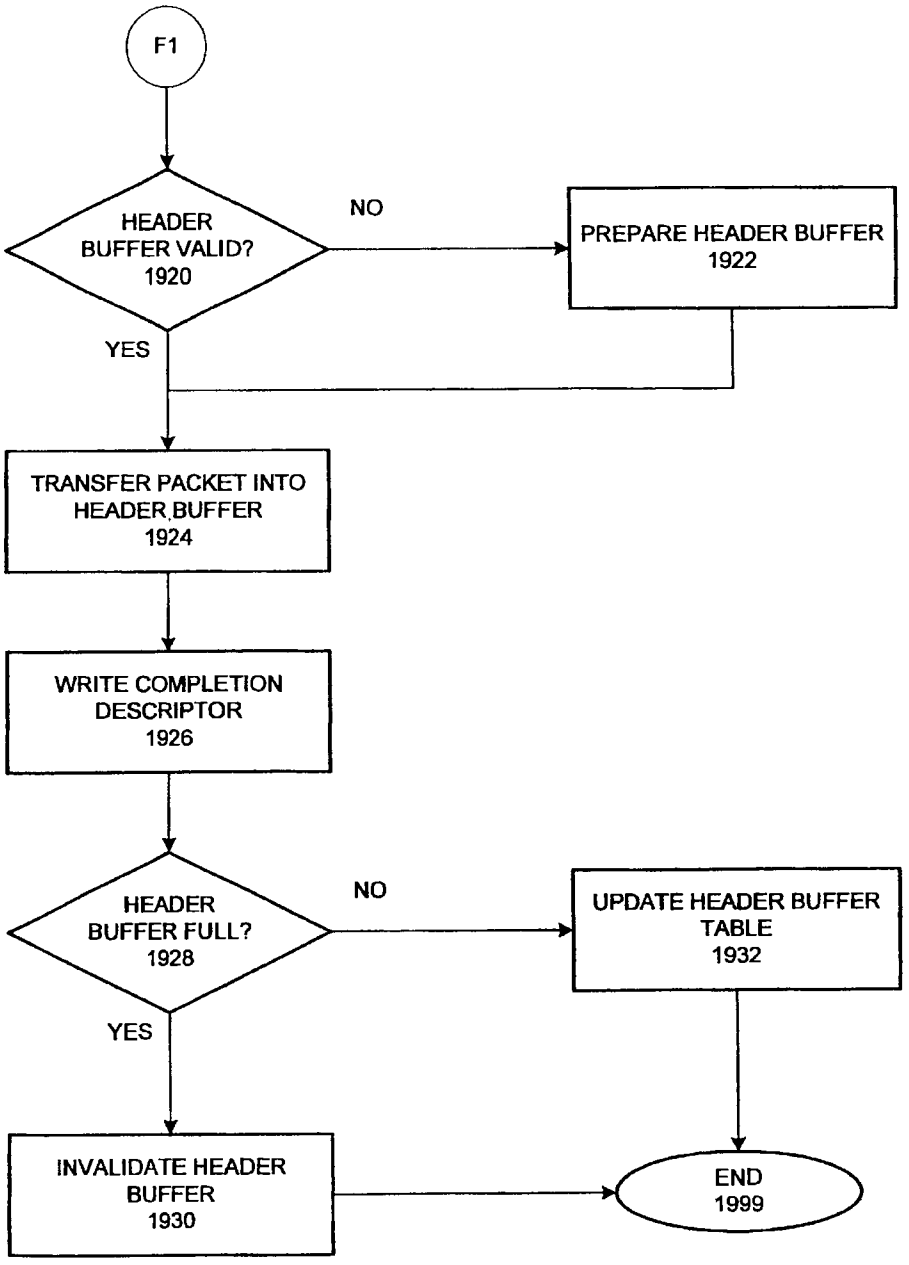

F1
HEADER BUFFER VALID? 1920
NO
PREPARE HEADER BUFFER 1922
YES
TRANSFER PACKET INTO HEADER BUFFER 1924
WRITE COMPLETION DESCRIPTOR 1926
HEADER BUFFER FULL? 1928
NO
UPDATE HEADER BUFFER TABLE 1932
YES
INVALIDATE HEADER BUFFER 1930
END 1999

FIG. 19B

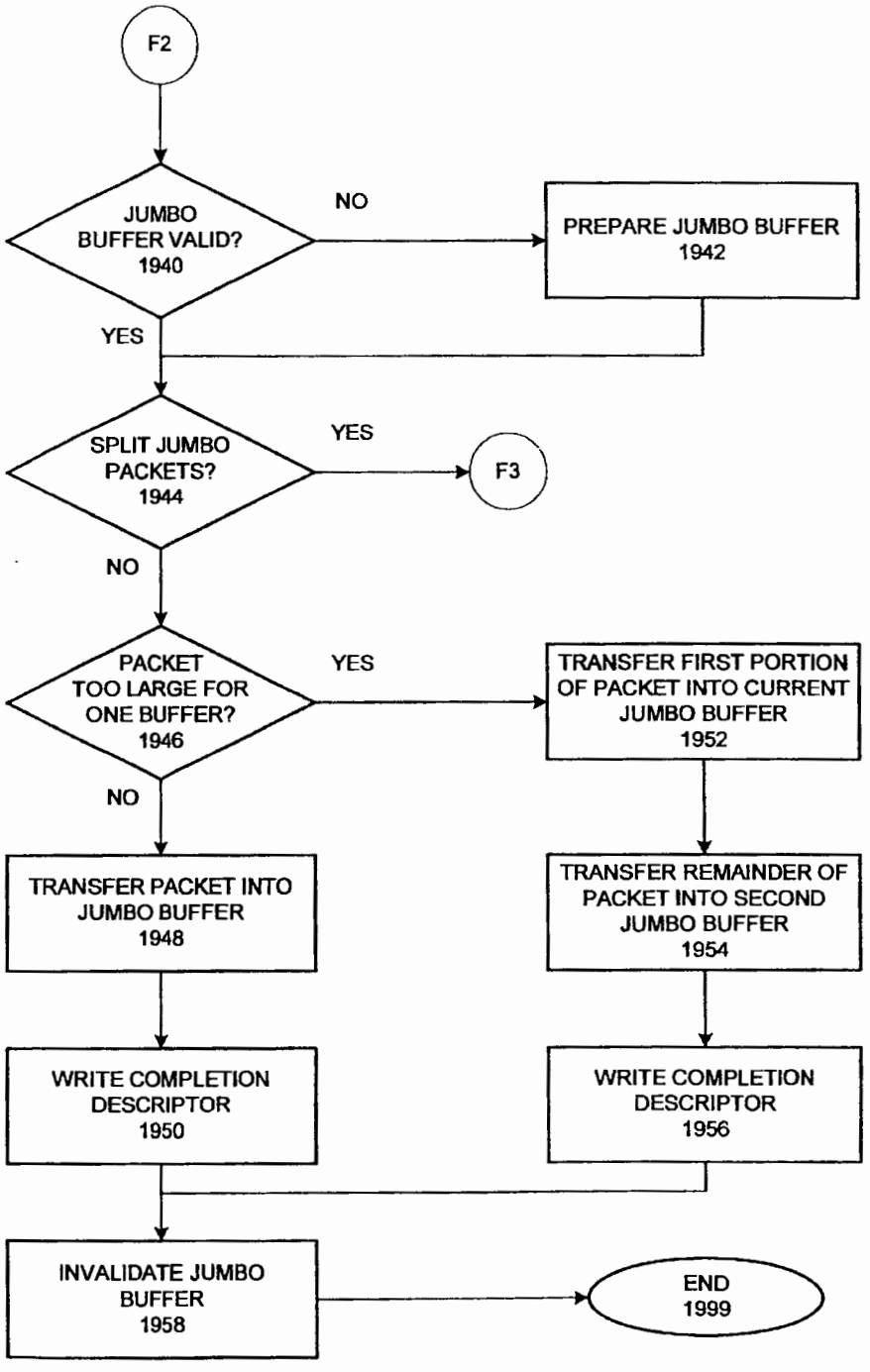

F2
JUMBO BUFFER VALID? 1940
NO
PREPARE JUMBO BUFFER 1942
YES
SPLIT JUMBO PACKETS? 1944
YES
F3
NO
PACKET TOO LARGE FOR ONE BUFFER? 1946
YES
TRANSFER FIRST PORTION OF PACKET INTO CURRENT JUMBO BUFFER 1952
NO
TRANSFER PACKET INTO JUMBO BUFFER 1948
TRANSFER REMAINDER OF PACKET INTO SECOND JUMBO BUFFER 1954
WRITE COMPLETION DESCRIPTOR 1950
WRITE COMPLETION DESCRIPTOR 1956
INVALIDATE JUMBO BUFFER 1958
END 1999

FIG. 19C

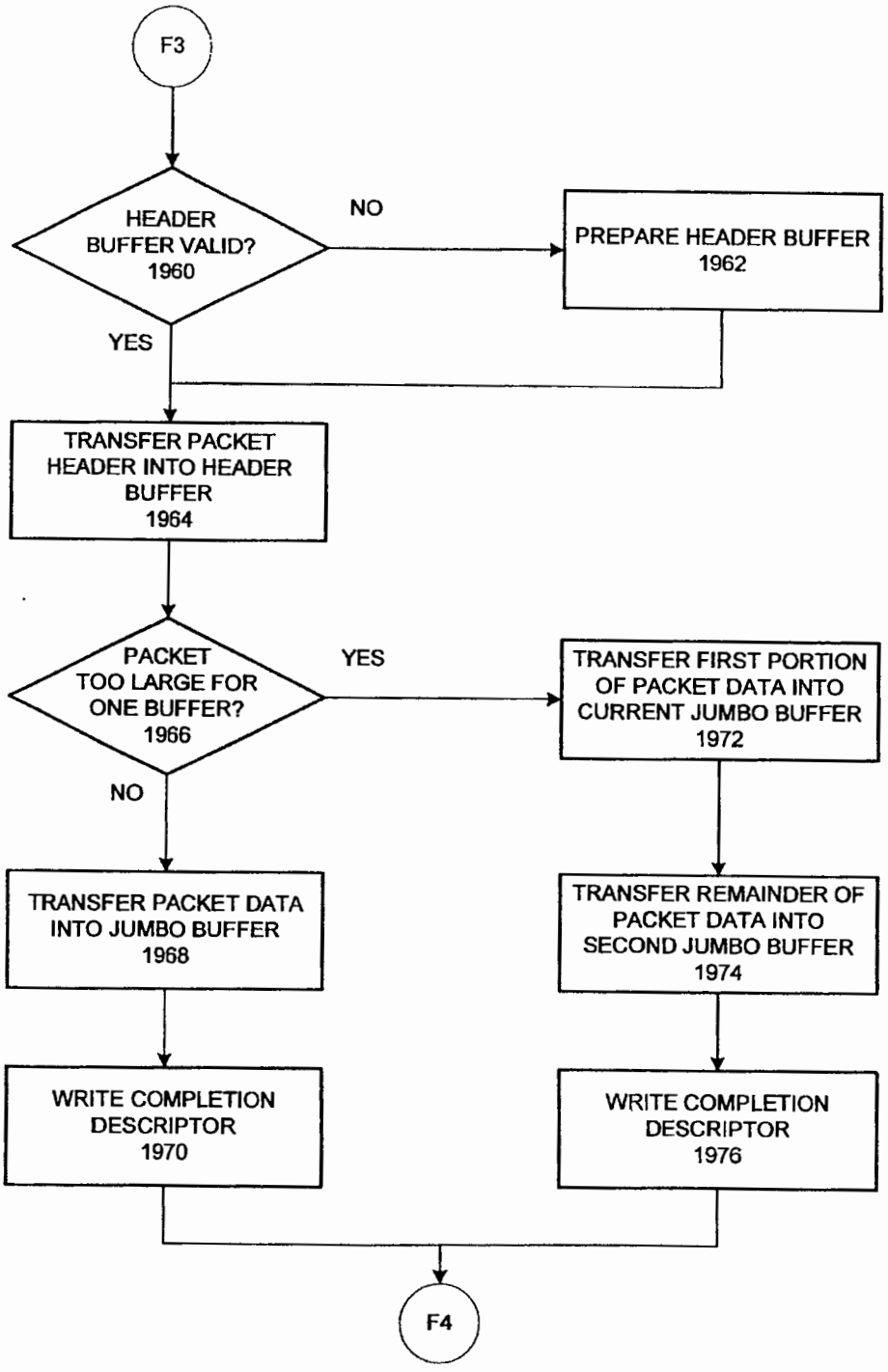

F3
HEADER BUFFER VALID? 1960
NO
PREPARE HEADER BUFFER 1962
YES
TRANSFER PACKET HEADER INTO HEADER BUFFER 1964
PACKET TOO LARGE FOR ONE BUFFER? 1966
YES
TRANSFER FIRST PORTION OF PACKET DATA INTO CURRENT JUMBO BUFFER 1972
NO
TRANSFER PACKET DATA INTO JUMBO BUFFER 1968
TRANSFER REMAINDER OF PACKET DATA INTO SECOND JUMBO BUFFER 1974
WRITE COMPLETION DESCRIPTOR 1970
WRITE COMPLETION DESCRIPTOR 1976
F4

FIG. 19D

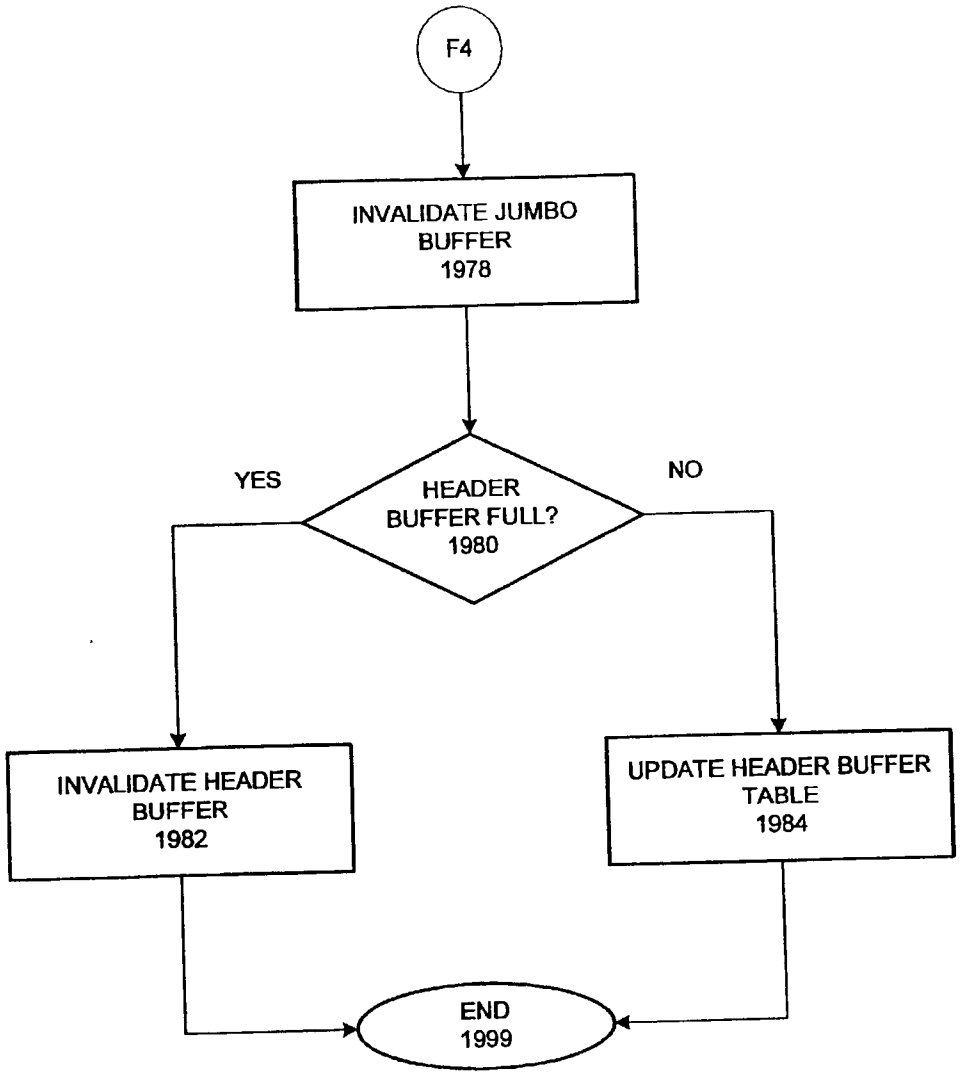

F4
INVALIDATE JUMBO BUFFER 1978
HEADER BUFFER FULL? 1980
YES
NO
INVALIDATE HEADER BUFFER 1982
UPDATE HEADER BUFFER TABLE 1984
END 1999

FIG. 19E

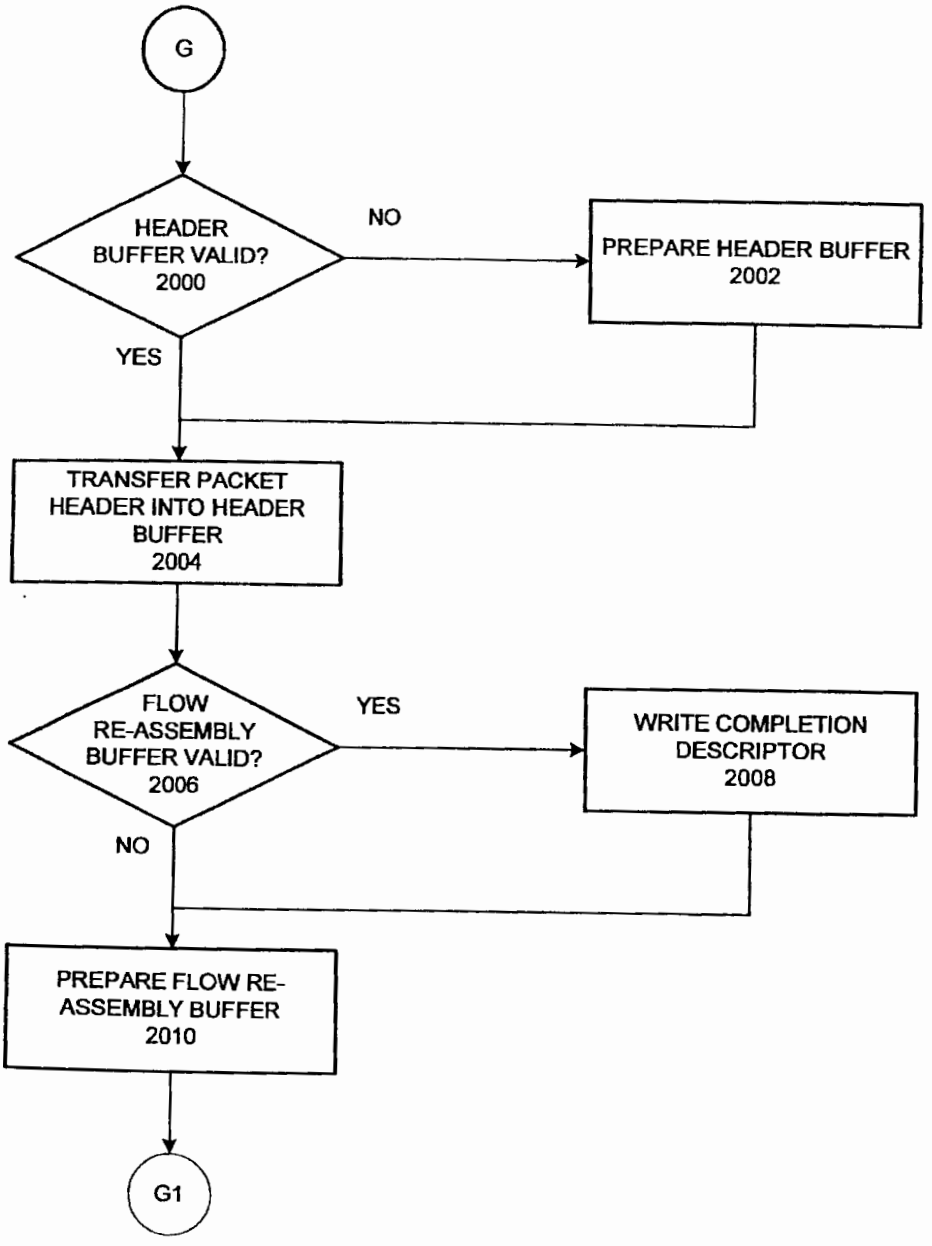

G
HEADER BUFFER VALID? 2000
NO
PREPARE HEADER BUFFER 2002
YES
TRANSFER PACKET HEADER INTO HEADER BUFFER 2004
FLOW RE-ASSEMBLY BUFFER VALID? 2006
YES
WRITE COMPLETION DESCRIPTOR 2008
NO
PREPARE FLOW RE-ASSEMBLY BUFFER 2010
G1

FIG. 20A

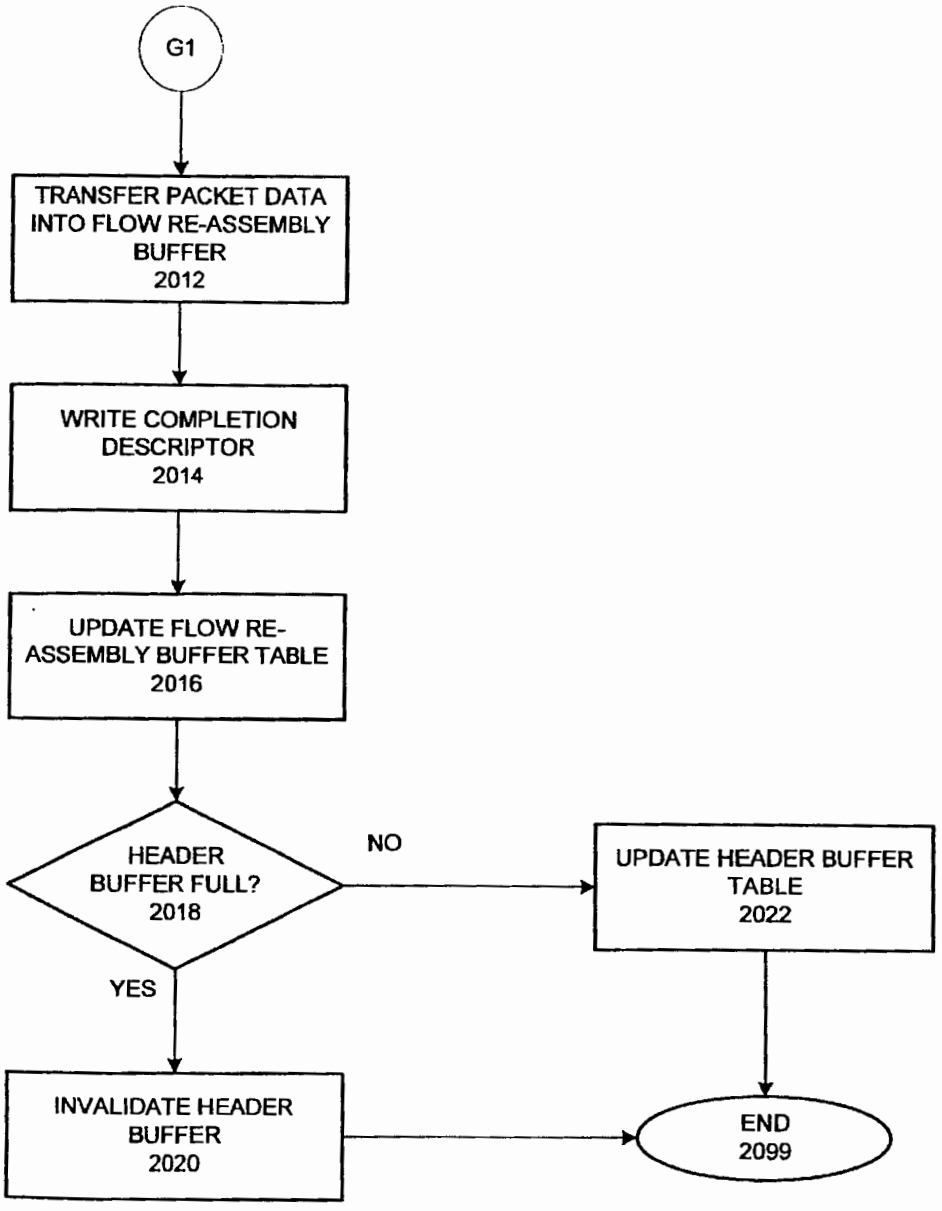

G1
TRANSFER PACKET DATA INTO FLOW RE-ASSEMBLY BUFFER 2012
WRITE COMPLETION DESCRIPTOR 2014
UPDATE FLOW RE-ASSEMBLY BUFFER TABLE 2016
HEADER BUFFER FULL? 2018
NO
UPDATE HEADER BUFFER TABLE 2022
YES
INVALIDATE HEADER BUFFER 2020
END 2099

FIG. 20B

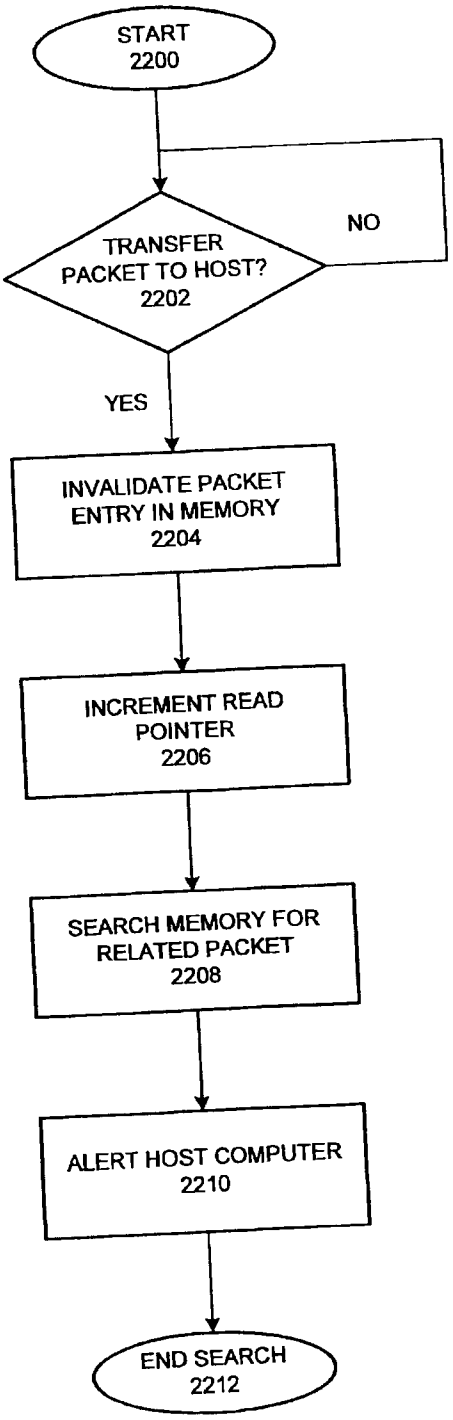

START
2200
TRANSFER PACKET TO HOST?
2202
NO
YES
INVALIDATE PACKET ENTRY IN MEMORY
2204
INCREMENT READ POINTER
2206
SEARCH MEMORY FOR RELATED PACKET
2208
ALERT HOST COMPUTER
2210
END SEARCH
2212

FIG. 22A

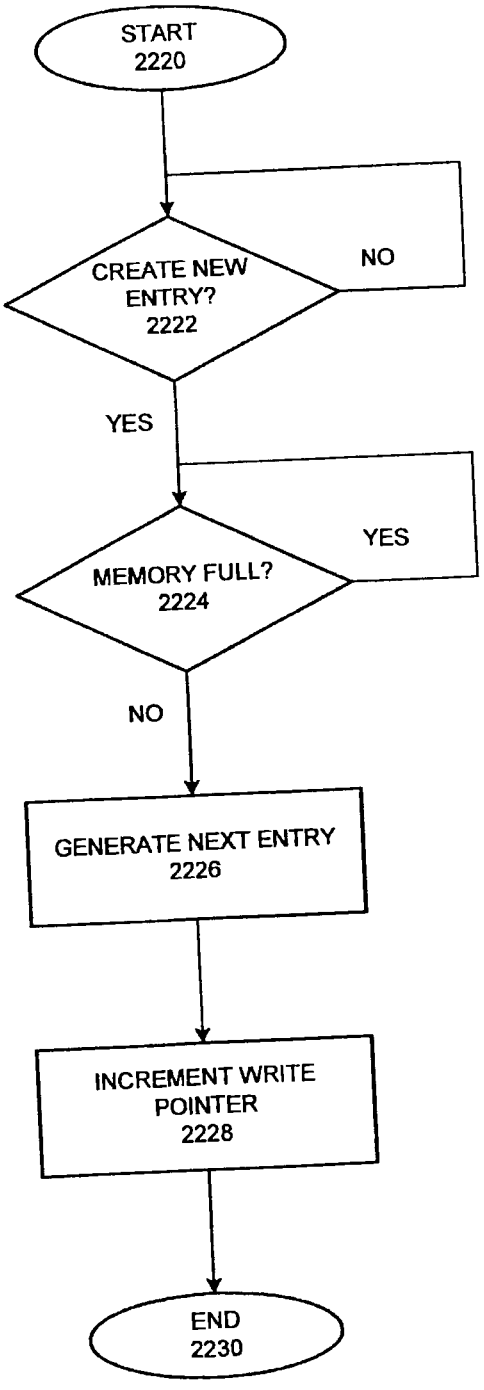

START
2220
CREATE NEW ENTRY?
2222
NO
YES
MEMORY FULL?
2224
YES
NO
GENERATE NEXT ENTRY
2226
INCREMENT WRITE POINTER
2228
END
2230

FIG. 22B

INSTR. NO. 2302	INSTR. NAME 2304	INSTRUCTION CONTENT 2306 (EXTRACTION MASK, COMPARE VALUE, OPERATOR, SUCCESS OFFSET, SUCCESS INSTRUCTION, FAILURE OFFSET, FAILURE INSTRUCTION, OUTPUT OPERATION, OPERATION ARGUMENT, OPERATION ENABLER, SHIFT, OUTPUT MASK)
0	WAIT	0xFFFF, 0x0000, NP, 6, VLAN, 0, WAIT, CLR_REG, 0x3FF, 1, 0, 0x0000
1	VLAN	0xFFFF, 0x8100, EQ, 1, CFI, 0, 802.3, IM_CTL, 0x00A, 3, 0, 0xFFFF
2	CFI	0x1000, 0x1000, EQ, 0, DONE, 1, 802.3, NONE, 0x000, 0, 0, 0x0000
3	802.3	0xFFFF, 0x0600, LT, 1, LLC_1, 0, IPV4_1, NONE, 0x000, 0, 0, 0x0000
4	LLC_1	0xFFFF, 0xAAAA, EQ, 1, LLC_2, 0, DONE, NONE, 0x000, 0, 0, 0x0000
5	LLC_2	0xFF00, 0x0300, EQ, 2, IPV4_1, 0, DONE, NONE, 0x000, 0, 0, 0x0000
6	IPV4_1	0xFFFF, 0x0800, EQ, 1, IPV4_2, 0, IPV6_1, LD_SAP, 0x100, 3, 0, 0xFFFF
7	IPV4_2	0xFF00, 0x4500, EQ, 3, IPV4_3, 0, DONE, LD_SUM, 0x00A, 1, 0, 0x0000
8	IPV4_3	0x3FFF, 0x0000, EQ, 1, IPV4_4, 0, DONE, LD_LEN, 0x03E, 1, 0, 0xFFFF
9	IPV4_4	0x00FF, 0x0006, EQ, 7, TCP_1, 0, DONE, LD_FID, 0x182, 1, 0, 0xFFFF
10	IPV6_1	0xFFFF, 0x86DD, EQ, 1, IPV6_2, 0, DONE, LD_SUM, 0x015, 1, 0x0000
11	IPV6_2	0xF000, 0x6000, EQ, 0, IPV6_3, 0, DONE, IM_R1, 0x114, 1, 0, 0xFFFF
12	IPV6_3	0x0000, 0x0000, EQ, 3, IPV6_4, 0, DONE, LD_FID, 0x484, 1, 0, 0xFFFF
13	IPV6_4	0xFF00, 0x0600, EQ, 18, TCP_1, 0, DONE, LD_LEN, 0x03F, 1, 0xFFFF
14	TCP_1	0x0000, 0x0000, EQ, 0, TCP_2, 4, TCP_2, LD_SEQ, 0x081, 3, 0, 0xFFFF
15	TCP_2	0x0000, 0x0000, EQ, 0, TCP_3, 0, TCP_3, ST_FLAG, 0x145, 3, 0, 0x002F
16	TCP_3	0x0000, 0x0000, EQ, 0, TCP_4, 0, TCP_4, LD_R1, 0x205, 3, 0xB, 0xF000
17	TCP_4	0x0000, 0x0000, EQ, 0, WAIT, 0, WAIT, LD_HDR, 0x0FF, 3, 0, 0xFFFF
18	DONE	0x0000, 0x0000, EQ, 0, WAIT, 0, WAIT, IM_CTL, 0x001, 3, 0x0000

PROGRAM 2300

FIG. 23

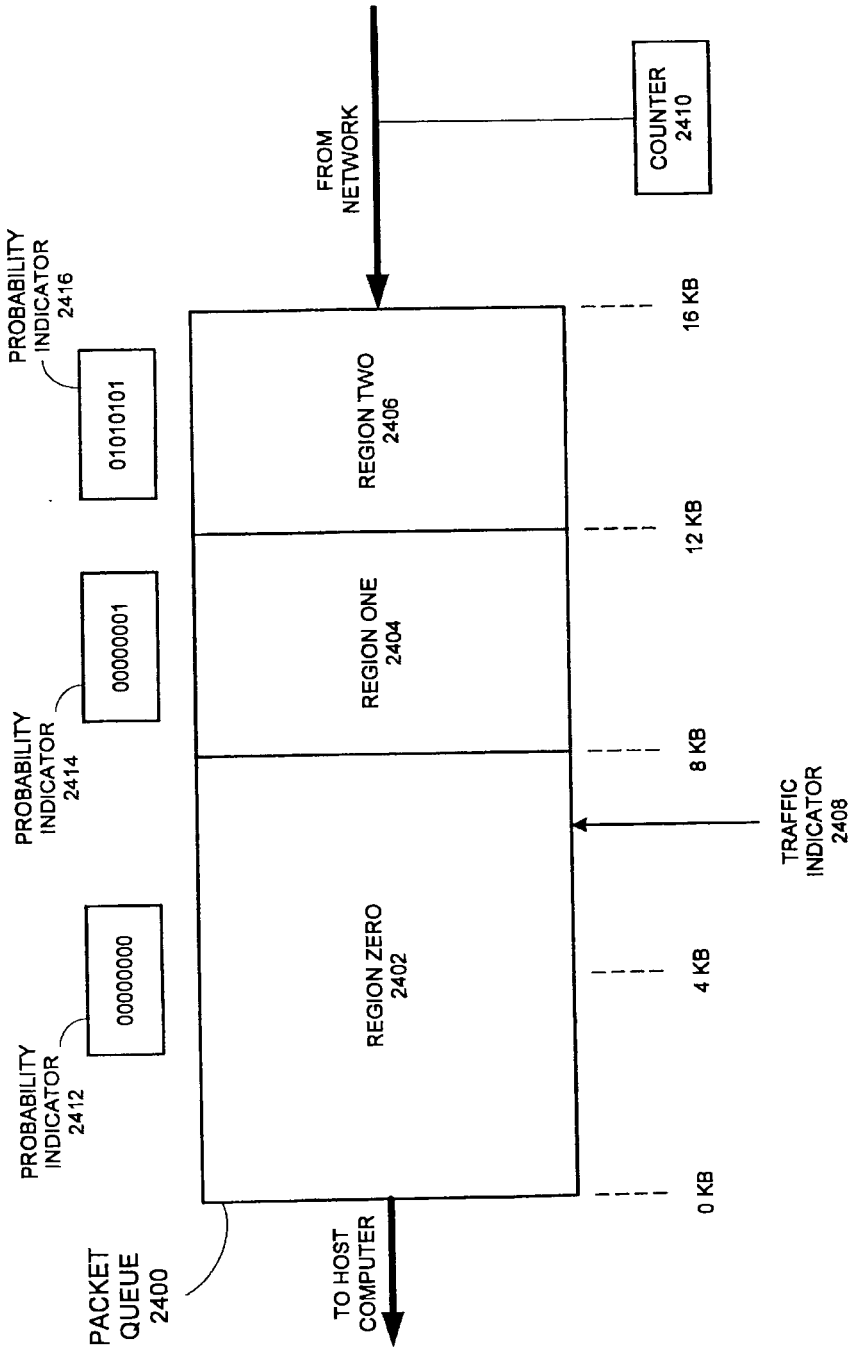

FROM NETWORK
COUNTER 2410
PROBABILITY INDICATOR 2416
01010101
REGION TWO 2406
16 KB
12 KB
PROBABILITY INDICATOR 2414
00000001
REGION ONE 2404
8 KB
TRAFFIC INDICATOR 2408
PROBABILITY INDICATOR 2412
00000000
REGION ZERO 2402
4 KB
0 KB
PACKET QUEUE 2400
TO HOST COMPUTER

FIG. 24

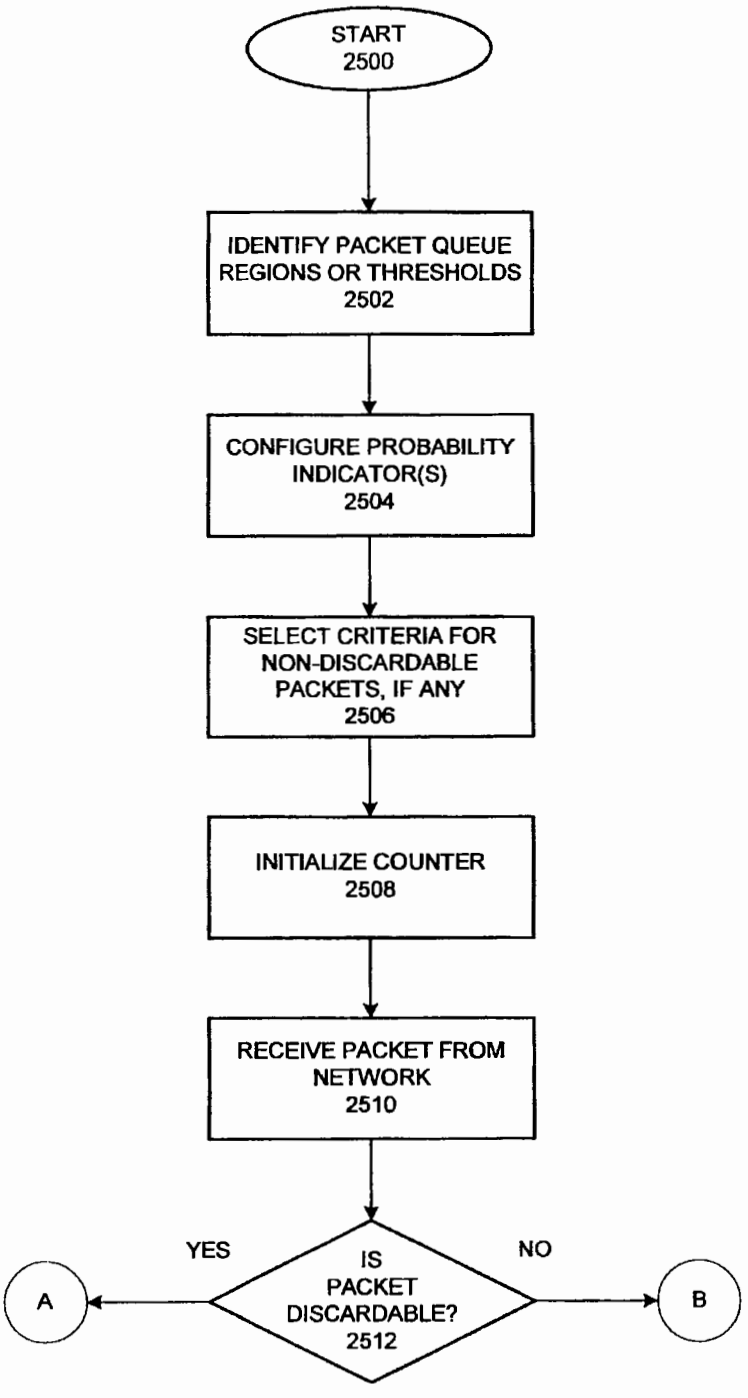

START
2500
IDENTIFY PACKET QUEUE REGIONS OR THRESHOLDS
2502
CONFIGURE PROBABILITY INDICATOR(S)
2504
SELECT CRITERIA FOR NON-DISCARDABLE PACKETS, IF ANY
2506
INITIALIZE COUNTER
2508
RECEIVE PACKET FROM NETWORK
2510
IS PACKET DISCARDABLE?
2512
YES
NO
A
B

FIG. 25A

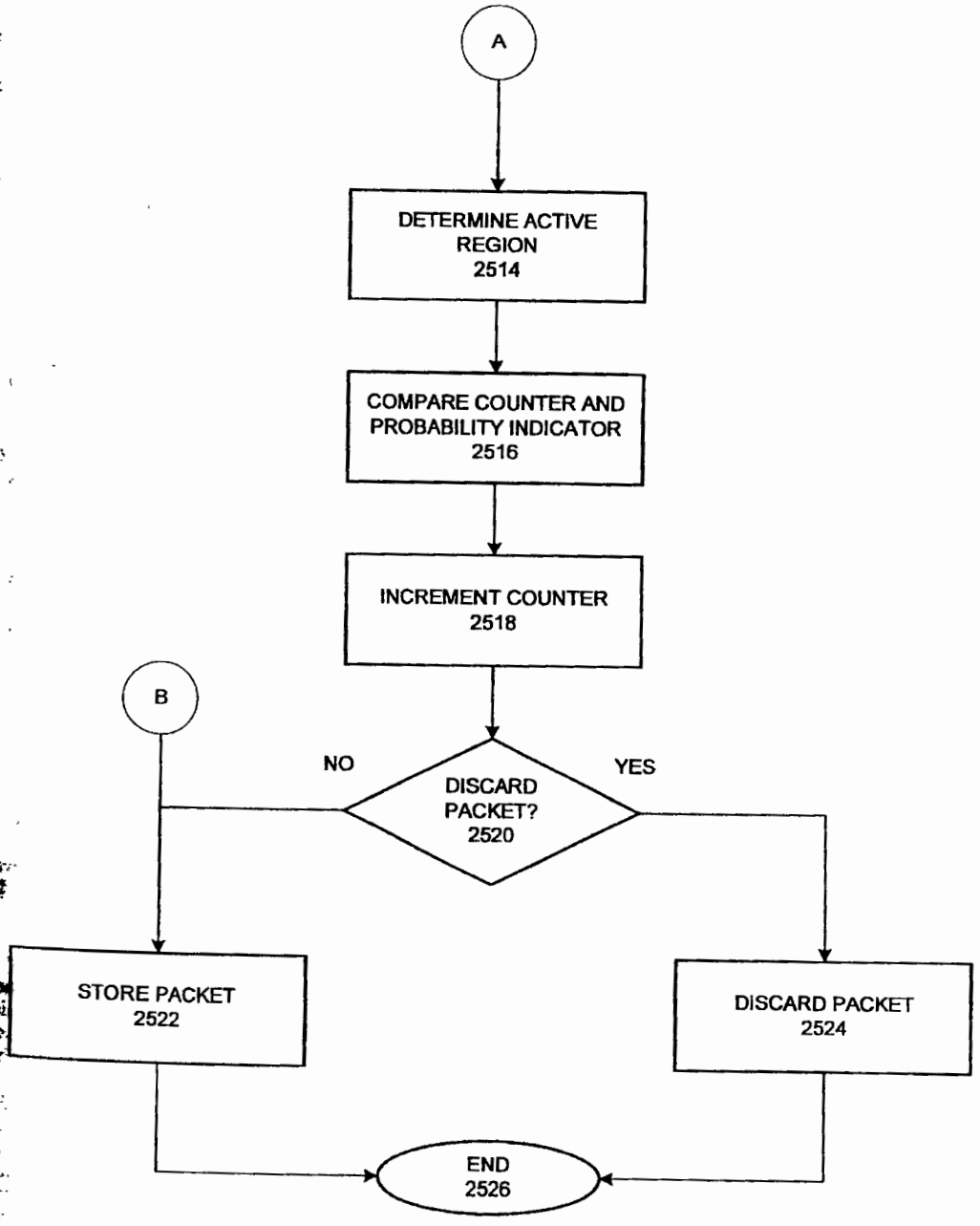

A
DETERMINE ACTIVE REGION 2514
COMPARE COUNTER AND PROBABILITY INDICATOR 2516
INCREMENT COUNTER 2518
B
NO
DISCARD PACKET? 2520
YES
STORE PACKET 2522
DISCARD PACKET 2524
END 2526

FIG. 25B

HIGH PERFORMANCE NETWORK INTERFACE

TABLE OF CONTENTS

BACKGROUND
SUMMARY
BRIEF DESCRIPTION OF THE FIGURES
DETAILED DESCRIPTION
Introduction
One Embodiment of a High Performance Network Interface Circuit
An Illustrative Packet
One Embodiment of a Header Parser
Dynamic Header Parsing Instructions in One Embodiment of the Invention
One Embodiment of a Flow Database
One Embodiment of a Flow Database Manager
One Embodiment of a Load Distributor
One Embodiment of a Packet Queue
One Embodiment of a Control Queue
One Embodiment of a DMA Engine
Methods of Transferring a Packet Into a Memory Buffer by a DMA Engine
A Method of Transferring a Packet with Operation Code 0
A Method of Transferring a Packet with Operation Code 1
A Method of Transferring a Packet with Operation Code 2
A Method of Transferring a Packet with Operation Code 3
A Method of Transferring a Packet with Operation Code 4
A Method of Transferring a Packet with Operation Code 5
A Method of Transferring a Packet with Operation Code 6 or 7
One Embodiment of a Dynamic Packet Batching Module
Early Random Packet Discard in One Embodiment of the Invention
CLAIMS

BACKGROUND

This invention relates to the fields of computer systems and computer networks. In particular, the present invention relates to a Network Interface Circuit (NIC) for processing communication packets exchanged between a computer network and a host computer system.

The interface between a computer and a network is often a bottleneck for communications passing between the computer and the network. While computer performance (e.g., processor speed) has increased exponentially over the years and computer network transmission speeds have undergone similar increases, inefficiencies in the way network interface circuits handle communications have become more and more evident. With each incremental increase in computer or network speed, it becomes ever more apparent that the interface between the computer and the network cannot keep pace. These inefficiencies involve several basic problems in the way communications between a network and a computer are handled.

Today's most popular forms of networks tend to be packet-based. These types of networks, including the Internet and many local area networks, transmit information in the form of packets. Each packet is separately created and transmitted by an originating endstation and is separately received and processed by a destination endstation. In addition, each packet may, in a bus topology network for example, be received and processed by numerous stations located between the originating and destination endstations.

One basic problem with packet networks is that each packet must be processed through multiple protocols or protocol levels (known collectively as a "protocol stack") on both the origination and destination endstations. When data transmitted between stations is longer than a certain minimal length, the data is divided into multiple portions, and each portion is carried by a separate packet. The amount of data that a packet can carry is generally limited by the network that conveys the packet and is often expressed as a maximum transfer unit (MTU). The original aggregation of data is sometimes known as a "datagram," and each packet carrying part of a single datagram is processed very similarly to the other packets of the datagram.

Communication packets are generally processed as follows. In the origination endstation, each separate data portion of a datagram is processed through a protocol stack. During this processing multiple protocol headers (e.g., TCP, IP, Ethernet) are added to the data portion to form a packet that can be transmitted across the network. The packet is received by a network interface circuit, which transfers the packet to the destination endstation or a host computer that serves the destination endstation. In the destination endstation, the packet is processed through the protocol stack in the opposite direction as in the origination endstation. During this processing the protocol headers are removed in the opposite order in which they were applied. The data portion is thus recovered and can be made available to a user, an application program, etc.

Several related packets (e.g., packets carrying data from one datagram) thus undergo substantially the same process in a serial manner (i.e., one packet at a time). The more data that must be transmitted, the more packets must be sent, with each one being separately handled and processed through the protocol stack in each direction. Naturally, the more packets that must be processed, the greater the demand placed upon an endstation's processor. The number of packets that must be processed is affected by factors other than just the amount of data being sent in a datagram. For example, as the amount of data that can be encapsulated in a packet increases, fewer packets need to be sent. As stated above, however, a packet may have a maximum allowable size, depending on the type of network in use (e.g., the maximum transfer unit for standard Ethernet traffic is approximately 1,500 bytes). The speed of the network also affects the number of packets that a NIC may handle in a given period of time. For example, a gigabit Ethernet network operating at peak capacity may require a NIC to receive approximately 1.48 million packets per second. Thus, the number of packets to be processed through a protocol stack may place a significant burden upon a computer's processor. The situation is exacerbated by the need to process each packet separately even though each one will be processed in a substantially similar manner.

A related problem to the disjoint processing of packets is the manner in which data is moved between "user space" (e.g., an application program's data storage) and "system space" (e.g., system memory) during data transmission and receipt. Presently, data is simply copied from one area of memory assigned to a user or application program into another area of memory dedicated to the processor's use. Because each portion of a datagram that is transmitted in a packet may be copied separately (e.g., one byte at a time),

there is a nontrivial amount of processor time required and frequent transfers can consume a large amount of the memory bus' bandwidth. Illustratively, each byte of data in a packet received from the network may be read from the system space and written to the user space in a separate copy operation, and vice versa for data transmitted over the network. Although system space generally provides a protected memory area (e.g., protected from manipulation by user programs), the copy operation does nothing of value when seen from the point of view of a network interface circuit. Instead, it risks over-burdening the host processor and retarding its ability to rapidly accept additional network traffic from the NIC. Copying each packet's data separately can therefore be very inefficient, particularly in a high-speed network environment.

In addition to the inefficient transfer of data (e.g., one packet's data at a time), the processing of headers from packets received from a network is also inefficient. Each packet carrying part of a single datagram generally has the same protocol headers (e.g., Ethernet, IP and TCP), although there may be some variation in the values within the packets' headers for a particular protocol. Each packet, however, is individually processed through the same protocol stack, thus requiring multiple repetitions of identical operations for related packets. Successively processing unrelated packets through different protocol stacks will likely be much less efficient than progressively processing a number of related packets through one protocol stack at a time.

Another basic problem concerning the interaction between present network interface circuits and host computer systems is that the combination often fails to capitalize on the increased processor resources that are available in multi-processor computer systems. In other words, present attempts to distribute the processing of network packets (e.g., through a protocol stack) among a number of protocols in an efficient manner are generally ineffective. In particular, the performance of present NICs does not come close to the expected or desired linear performance gains one may expect to realize from the availability of multiple processors. In some multi-processor systems, little improvement in the processing of network traffic is realized from the use of more than 4–6 processors, for example.

In addition, the rate at which packets are transferred from a network interface circuit to a host computer or other communication device may fail to keep pace with the rate of packet arrival at the network interface. One element or another of the host computer (e.g., a memory bus, a processor) may be over-burdened or otherwise unable to accept packets with sufficient alacrity. In this event one or more packets may be dropped or discarded. Dropping packets may cause a network entity to re-transmit some traffic and, if too many packets are dropped, a network connection may require re-initialization. Further, dropping one packet or type of packet instead of another may make a significant difference in overall network traffic. If, for example, a control packet is dropped, the corresponding network connection may be severely affected and may do little to alleviate the packet saturation of the network interface circuit because of the typically small size of a control packet. Therefore, unless the dropping of packets is performed in a manner that distributes the effect among many network connections or that makes allowance for certain types of packets, network traffic may be degraded more than necessary.

Thus, present NICs fail to provide adequate performance to interconnect today's high-end computer systems and high-speed networks. In addition, a network interface circuit that cannot make allowance for an over-burdened host computer may degrade the computer's performance.

SUMMARY

A high performance network interface is provided for receiving a packet from a network and transferring it to a host computer system. In various embodiments of the invention, the high performance network interface is configured to implement one or more enhanced operations in order to efficiently handle a range of packet arrival rates without unduly burdening the host computer system.

One such operation is the re-assembly of data from multiple packets in one communication flow, circuit or connection. In particular, data portions of such packets may be re-assembled by transferring or copying them into a single host memory area, or buffer, that is of a predetermined size (e.g., one memory page). The re-assembled data may then be provided to the destination entity in an efficient manner, such as a single copy or memory transfer.

Another operation for increasing the efficiency of handling network traffic in an embodiment of the invention is the batch processing of packet headers through an appropriate protocol stack. In this operation, a host computer system is alerted to the transfer, into host memory, of two or more packets from the same communication flow. When so alerted, the host computer may delay processing a first packet in the flow in order to await receipt of a second. The packets' headers may then be processed collectively, or in rapid sequence, rather than interspersing the processing of the packets with packets from other flows.

In yet another operation, the processing of packets or packet headers through their protocol stacks may be distributed among two or more processors in a multi-processor host computer system. In a load distribution operation in one embodiment of the invention, an identifier of the processor that is to process a packet is generated from a packet's flow key. In this embodiment, a flow key is assembled from identifiers of the packet's source and destination entities extracted from the packet's header portion. By using the packet's flow key, which uniquely identifies a particular communication flow all packets in the same flow will be sent to the same processor. One method of generating the processor identifier is to perform a hashing function on the flow key and then take the modulus of that result over the number of processors in the host computer system.

In one embodiment of the invention a high performance network interface includes a header parser module. When a packet is received from a network, the header parser module parses a header portion of the packet. The header parser module executes a series of parsing instructions configured in accordance with a set of selected communication protocols for conveying packets across the network. While parsing the packet, the header parser module compares a value extracted from a header field with an expected value in order to test the received packet for compatibility with the selected protocols. Instructions for operating the header parser module may be stored in a rewriteable memory so that the module may be reconfigured to parse packets conforming to virtually any communication protocol.

Besides parsing a packet to determine its compatibility with a set of protocols, a header parser module in one embodiment of the invention retrieves values from one or more fields in the packet's headers. The extracted values may be used to enable or assist one of the enhanced operations. In particular, in this embodiment a header parser module extracts identifiers of the packet's source and des-

tination entities. These identifiers may be combined to form a flow key for the purpose of identifying the communication flow, circuit or connection in which the packet was sent. In this embodiment, each separate datagram sent from a source entity to a destination entity may comprise a separate flow.

After a header parser module parses a packet received from a network, the header parser module passes the packet's flow key and, possibly, other information extracted from the packet, to a flow database manager. The flow database manager maintains a flow database to manage the communication flows received at the network interface. Within a flow database, a number of flow keys may be stored and indexed by flow numbers. The database is updated accordingly as flows are initiated and terminated and as flow packets are received.

From information received from a header parser module in this embodiment, the flow database manager assigns an operation code to the packet. Other modules of the network interface may use the operation code to determine the suitability of the packet for one or more of the enhanced operations described above or to identify a method of performing an operation. For example, the received packet's operation code may reveal whether the packet is compatible with the set of selected protocols, whether the packet contains data, whether the packet's data can be re-assembled with other flow packets, whether a flow is to be set up or torn down, etc.

In one embodiment of the invention, the high performance network interface includes a packet queue in which to store a packet received from a network prior to its transfer to a host computer system. The network interface may also include a control queue or other data structures (e.g., registers) in which to store data extracted from a packet and/or information concerning the extracted data, such as an operation code or flow number. Information stored in one or both of the packet and control queues may also include a checksum generated by a checksum module, a processor identifier generated by a load distributor module, offsets to specific portions of the packet, flags concerning statuses or conditions of the packet, etc.

In another embodiment of the invention, a DMA engine is provided for transferring a packet from a packet queue into a host memory area, such as a buffer, in the host computer system. The DMA engine may draw upon information in the packet queue or a control queue, such as an operation code, to determine which buffer or buffers to store a packet in. For example, a packet's header may be stored in a header buffer while its data portion is stored in a re-assembly buffer. Packets less than a specified size may also be stored in a header buffer. A packet that is not compatible with the selected protocols may be stored, intact, in a non-re-assembly buffer. In one embodiment, buffers are of a predetermined size that increases the efficiency of memory transfers or copies, such as one memory page.

In yet another embodiment of the invention, a high performance network interface includes a dynamic packet batching module for notifying a host computer when multiple packets in one communication are being transferred to the computer. In this embodiment, a packet batching module includes a memory for storing flow numbers or flow keys of multiple packets to be transferred to the host computer. When a packet is transferred or about to be transferred, the packet batching module searches its memory for other packets having the same flow number or flow key as the transferred packet. The host computer is notified accordingly and may delay processing one packet in a flow in order to process it in conjunction with another packet in the same flow.

The network interface may notify the host computer system of the arrival or transfer of a packet by configuring and releasing a descriptor that identifies where the packet is stored. In another embodiment, a high performance network interface issues an alert, such as an interrupt, to the host computer system. Interrupts issued by the network interface may be modulated, particularly as the rate of packets arriving from a network increases, so as to limit the number of interrupts or the frequency with which they are issued. In one method of modulating interrupts, after a first interrupt is issued further interrupts may be disabled until a specified number of packets have been received and/or a predetermined period of time elapses. In another method of modulating interrupts, interrupts may be disabled while software operating on the host computer polls the network interface to determine if a packet has been received or transferred. Packet and time counters may also be used in this method in order to allow interrupts to be generated in the event that the polling software is blocked or fails.

In one embodiment of the invention, if the rate at which a host computer accepts packets from a high-speed network interface does not keep pace with the rate at which packets are received at the network interface, a packet may be dropped. In this embodiment a method is provided for randomly selecting a packet to be discarded, before or after the packet is stored in a packet queue. A packet queue in this embodiment is logically separated into multiple regions or divisions, which may overlap. A probability indicator is associated with each region to indicate the probability of dropping a packet when the level of traffic stored in the queue is within the region. When the level of traffic is within a particular region, the probability indicator for that region is applied each time a discardable packet is to be stored in the packet queue. The region's probability indicator thus indicates whether to discard the packet or allow it to be stored in the queue. All packets may be considered discardable, or some packets (e.g., control packets, packets in a certain flow, packets adhering to a particular protocol) may be considered non-discardable. In one embodiment of the invention, the network interface includes a counter that is incremented through a limited range of values as discardable packets are received for storage in the queue. In this embodiment, a probability indicator consists of a set of numbers (e.g., a mask) to indicate, for each value in the range of counter values, whether or not to discard a packet.

DESCRIPTION OF THE FIGURES

FIG. **1A** is a block diagram depicting a network interface circuit (NIC) for receiving a packet from a network in accordance with an embodiment of the present invention.

FIG. **1B** is a flow chart demonstrating one method of operating the NIC of FIG. **1A** to transfer a packet received from a network to a host computer in accordance with an embodiment of the invention.

FIG. **2** is a diagram of a packet transmitted over a network and received at a network interface circuit in one embodiment of the invention.

FIG. **3** is a block diagram depicting a header parser of a network interface circuit for parsing a packet in accordance with an embodiment of the invention.

FIGS. **4A–4B** comprise a flow chart demonstrating one method of parsing a packet received from a network at a network interface circuit in accordance with an embodiment of the present invention.

FIG. **5** is a block diagram depicting a network interface circuit flow database in accordance with an embodiment of the invention.

FIGS. 6A–6E comprise a flowchart illustrating one method of managing a network interface circuit flow database in accordance with an embodiment of the invention.

FIG. 7 is a flow chart demonstrating one method of distributing the processing of network packets among multiple processors on a host computer in accordance with an embodiment of the invention.

FIG. 8 is a diagram of a packet queue for a network interface circuit in accordance with an embodiment of the invention.

FIG. 9 is a diagram of a control queue for a network interface circuit in accordance with an embodiment of the invention.

FIG. 10 is a block diagram of a DMA engine for transferring a packet received from a network to a host computer in accordance with an embodiment of the invention.

FIG. 11 includes diagrams of data structures for managing the storage of network packets in host memory buffers in accordance with an embodiment of the invention.

FIGS. 12A–12B are diagrams of a free descriptor, a completion descriptor and a free buffer array in accordance with an embodiment of the invention.

FIGS. 13–20 are flow charts demonstrating methods of transferring a packet received from a network to a buffer in a host computer memory in accordance with an embodiment of the invention.

FIG. 21 is a diagram of a dynamic packet batching module in accordance with an embodiment of the invention.

FIGS. 22A–22B comprise a flow chart demonstrating one method of dynamically searching a memory containing information concerning packets awaiting transfer to a host computer in order to locate a packet in the same communication flow as a packet being transferred, in accordance with an embodiment of the invention.

FIG. 23 depicts one set of dynamic instructions for parsing a packet in accordance with an embodiment of the invention.

FIG. 24 depicts a system for randomly discarding a packet from a network interface in accordance with an embodiment of the invention.

FIGS. 25A–25B comprise a flow chart demonstrating one method of discarding a packet from a network interface in accordance with an embodiment of the invention.

DETAILED DESCRIPTION

The following description is presented to enable any person skilled in the art to make and use the invention, and is provided in the context of particular applications of the invention and their requirements. Various modifications to the disclosed embodiments will be readily apparent to those skilled in the art and the general principles defined herein may be applied to other embodiments and applications without departing from the spirit and scope of the present invention. Thus, the present invention is not intended to be limited to the embodiments shown, but is to be accorded the widest scope consistent with the principles and features disclosed herein.

In particular, embodiments of the invention are described below in the form of a network interface circuit (NIC) receiving communication packets formatted in accordance with certain communication protocols compatible with the Internet. One skilled in the art will recognize, however, that the present invention is not limited to communication protocols compatible with the Internet and may be readily adapted for use with other protocols and in communication devices other than a NIC.

The program environment in which a present embodiment of the invention is executed illustratively incorporates a general-purpose computer or a special purpose device such a hand-held computer. Details of such devices (e.g., processor, memory, data storage, input/output ports and display) are well known and are omitted for the sake of clarity.

It should also be understood that the techniques of the present invention might be implemented using a variety of technologies. For example, the methods described herein may be implemented in software running on a programmable microprocessor, or implemented in hardware utilizing either a combination of microprocessors or other specially designed application specific integrated circuits, programmable logic devices, or various combinations thereof. In particular, the methods described herein may be implemented by a series of computer-executable instructions residing on a storage medium such as a carrier wave, disk drive, or other computer-readable medium.

Introduction

In one embodiment of the present invention, a network interface circuit (NIC) is configured to receive and process communication packets exchanged between a host computer system and a network such as the Internet. In particular, the NIC is configured to receive and manipulate packets formatted in accordance with a protocol stack (e.g., a combination of communication protocols) supported by a network coupled to the NIC.

A protocol stack may be described with reference to the seven-layer ISO-OSI (International Standards Organization-Open Systems Interconnection) model framework. Thus, one illustrative protocol stack includes the Transport Control Protocol (TCP) at layer four, Internet Protocol (IP) at layer three and Ethernet at layer two. For purposes of discussion, the term "Ethernet" may be used herein to refer collectively to the standardized IEEE (Institute of Electrical and Electronics Engineers) 802.3 specification as well as version two of the non-standardized form of the protocol. Where different forms of the protocol need to be distinguished, the standard form may be identified by including the "802.3" designation.

Other embodiments of the invention are configured to work with communications adhering to other protocols, both known (e.g., AppleTalk, IPX (Internetwork Packet Exchange), etc.) and unknown at the present time. One skilled in the art will recognize that the methods provided by this invention are easily adaptable for new communication protocols.

In addition, the processing of packets described below may be performed on communication devices other than a NIC. For example, a modem, switch, router or other communication port or device (e.g., serial, parallel, USB, SCSI) may be similarly configured and operated.

In embodiments of the invention described below, a NIC receives a packet from a network on behalf of a host computer system or other communication device. The NIC analyzes the packet (e.g., by retrieving certain fields from one or more of its protocol headers) and takes action to increase the efficiency with which the packet is transferred or provided to its destination entity. Equipment and methods discussed below for increasing the efficiency of processing or transferring packets received from a network may also be used for packets moving in the reverse direction (i.e., from the NIC to the network).

One technique that may be applied to incoming network traffic involves examining or parsing one or more headers of

NOAC Ex. 1015 Page 516

an incoming packet (e.g., headers for the layer two, three and four protocols) in order to identify the packet's source and destination entities and possibly retrieve certain other information. Using identifiers of the communicating entities as a key, data from multiple packets may be aggregated or re-assembled. Typically, a datagram sent to one destination entity from one source entity is transmitted via multiple packets. Aggregating data from multiple related packets (e.g., packets carrying data from the same datagram) thus allows a datagram to be re-assembled and collectively transferred to a host computer. The datagram may then be provided to the destination entity in a highly efficient manner. For example, rather than providing data from one packet at a time (and one byte at a time) in separate "copy" operations, a "page-flip" operation may be performed. In a page-flip, an entire memory page of data may be provided to the destination entity, possibly in exchange for an empty or unused page.

In another technique, packets received from a network are placed in a queue to await transfer to a host computer. While awaiting transfer, multiple related packets may be identified to the host computer. After being transferred, they may be processed as a group by a host processor rather than being processed serially (e.g., one at a time).

Yet another technique involves submitting a number of related packets to a single processor of a multi-processor host computer system. By distributing packets conveyed between different pairs of source and destination entities among different processors, the processing of packets through their respective protocol stacks can be distributed while still maintaining packets in their correct order.

The techniques discussed above for increasing the efficiency with which packets are processed may involve a combination of hardware and software modules located on a network interface and/or a host computer system. In one particular embodiment, a parsing module on a host computer's NIC parses header portions of packets. Illustratively, the parsing module comprises a microsequencer operating according to a set of replaceable instructions stored as micro-code. Using information extracted from the packets, multiple packets from one source entity to one destination entity may be identified. A hardware re-assembly module on the NIC may then gather the data from the multiple packets. Another hardware module on the NIC is configured to recognize related packets awaiting transfer to the host computer so that they may be processed through an appropriate protocol stack collectively, rather than serially. The re-assembled data and the packet's headers may then be provided to the host computer so that appropriate software (e.g., a device driver for the NIC) may process the headers and deliver the data to the destination entity.

Where the host computer includes multiple processors, a load distributor (which may also be implemented in hardware on the NIC) may select a processor to process the headers of the multiple packets through a protocol stack.

In another embodiment of the invention, a system is provided for randomly discarding a packet from a NIC when the NIC is saturated or nearly saturated with packets awaiting transfer to a host computer.

One Embodiment of a High Performance Network Interface Circuit

FIG. 1A depicts NIC **100** configured in accordance with an illustrative embodiment of the invention. A brief description of the operation and interaction of the various modules of NIC **100** in this embodiment follows. Descriptions incorporating much greater detail are provided in subsequent sections.

A communication packet may be received at NIC **100** from network **102** by a medium access control (MAC) module (not shown in FIG. 1A). The MAC module performs low-level processing of the packet such as reading the packet from the network, performing some error checking, detecting packet fragments, detecting over-sized packets, removing the layer one preamble, etc.

Input Port Processing (IPP) module **104** then receives the packet. The IPP module stores the entire packet in packet queue **116**, as received from the MAC module or network, and a portion of the packet is copied into header parser **106**. In one embodiment of the invention IPP module **104** may act as a coordinator of sorts to prepare the packet for transfer to a host computer system. In such a role, IPP module **104** may receive information concerning a packet from various modules of NIC **100** and dispatch such information to other modules.

Header parser **106** parses a header portion of the packet to retrieve various pieces of information that will be used to identify related packets (e.g., multiple packets from one same source entity for one destination entity) and that will affect subsequent processing of the packets. In the illustrated embodiment, header parser **106** communicates with flow database manager (FDBM) **108**, which manages flow database (FDB) **110**. In particular, header parser **106** submits a query to FDBM **108** to determine whether a valid communication flow (described below) exists between the source entity that sent a packet and the destination entity. The destination entity may comprise an application program, a communication module, or some other element of a host computer system that is to receive the packet.

In the illustrated embodiment of the invention, a communication flow comprises one or more datagram packets from one source entity to one destination entity. A flow may be identified by a flow key assembled from source and destination identifiers retrieved from the packet by header parser **106**. In one embodiment of the invention a flow key comprises address and/or port information for the source and destination entities from the packet's layer three (e.g., IP) and/or layer four (e.g., TCP) protocol headers.

For purposes of the illustrated embodiment of the invention, a communication flow is similar to a TCP end-to-end connection but is generally shorter in duration. In particular, in this embodiment the duration of a flow may be limited to the time needed to receive all of the packets associated with a single datagram passed from the source entity to the destination entity.

Thus, for purposes of flow management, header parser **106** passes the packet's flow key to flow database manager **108**. The header parser may also provide the flow database manager with other information concerning the packet that was retrieved from the packet (e.g., length of the packet).

Flow database manager **108** searches FDB **110** in response to a query received from header parser **106**. Illustratively, flow database **110** stores information concerning each valid communication flow involving a destination entity served by NIC **100**. Thus, FDBM **108** updates FDB **110** as necessary, depending upon the information received from header parser **106**. In addition, in this embodiment of the invention FDBM **108** associates an operation or action code with the received packet. An operation code may be used to identify whether a packet is part of a new or existing flow, whether the packet includes data or just control information, the amount of data within the packet, whether the packet data can be re-assembled with related data (e.g., other data in a datagram sent from the source entity to the destination entity), etc. FDBM **108** may use information

NOAC Ex. 1015 Page 517

retrieved from the packet and provided by header parser **106** to select an appropriate operation code. The packet's operation code is then passed back to the header parser, along with an index of the packet's flow within FDB **110**.

In one embodiment of the invention the combination of header parser **106**, FDBM **108** and FDB **110**, or a subset of these modules, may be known as a traffic classifier due to their role in classifying or identifying network traffic received at NIC **100**.

In the illustrated embodiment, header parser **106** also passes the packet's flow key to load distributor **112**. In a host computer system having multiple processors, load distributor **112** may determine which processor an incoming packet is to be routed to for processing through the appropriate protocol stack. For example, load distributor **112** may ensure that related packets are routed to a single processor. By sending all packets in one communication flow or end-to-end connection to a single processor, the correct ordering of packets can be enforced. Load distributor **112** may be omitted in one alternative embodiment of the invention. In another alternative embodiment, header parser **106** may also communicate directly with other modules of NIC **100** besides the load distributor and flow database manager.

Thus, after header parser **106** parses a packet FDBM **108** alters or updates FDB **110** and load distributor **112** identifies a processor in the host computer system to process the packet. After these actions, the header parser passes various information back to IPP module **104**. Illustratively, this information may include the packet's flow key, an index of the packet's flow within flow database **110**, an identifier of a processor in the host computer system, and various other data concerning the packet (e.g., its length, a length of a packet header).

Now the packet may be stored in packet queue **116**, which holds packets for manipulation by DMA (Direct Memory Access) engine **120** and transfer to a host computer. In addition to storing the packet in a packet queue, a corresponding entry for the packet is made in control queue **118** and information concerning the packet's flow may also be passed to dynamic packet batching module **122**. Control queue **118** contains related control information for each packet in packet queue **116**.

Packet batching module **122** draws upon information concerning packets in packet queue **116** to enable the batch (i.e., collective) processing of headers from multiple related packets. In one embodiment of the invention packet batching module **122** alerts the host computer to the availability of headers from related packets so that they may be processed together.

Although the processing of a packet's protocol headers is performed by a processor on a host computer system in one embodiment of the invention, in another embodiment the protocol headers may be processed by a processor located on NIC **100**. In the former embodiment, software on the host computer (e.g., a device driver for NIC **100**) can reap the advantages of additional memory and a replaceable or upgradeable processor (e.g., the memory may be supplemented and the processor may be replaced by a faster model).

During the storage of a packet in packet queue **116**, checksum generator **114** may perform a checksum operation. The checksum may be added to the packet queue as a trailer to the packet. Illustratively, checksum generator **114** generates a checksum from a portion of the packet received from network **102**. In one embodiment of the invention, a checksum is generated from the TCP portion of a packet (e.g., the TCP header and data). If a packet is not formatted according to TCP, a checksum may be generated on another portion of the packet and the result may be adjusted in later processing as necessary. For example, if the checksum calculated by checksum generator **114** was not calculated on the correct portion of the packet, the checksum may be adjusted to capture the correct portion. This adjustment may be made by software operating on a host computer system (e.g., a device driver). Checksum generator **114** may be omitted or merged into another module of NIC **100** in an alternative embodiment of the invention.

From the information obtained by header parser **106** and the flow information managed by flow database manager **108**, the host computer system served by NIC **100** in the illustrated embodiment is able to process network traffic very efficiently. For example, data portions of related packets may be re-assembled by DMA engine **120** to form aggregations that can be more efficiently manipulated. And, by assembling the data into buffers the size of a memory page, the data can be more efficiently transferred to a destination entity through "page-flipping," in which an entire memory page filled by DMA engine **120** is provided at once. One page-flip can thus take the place of multiple copy operations. Meanwhile, the header portions of the re-assembled packets may similarly be processed as a group through their appropriate protocol stack.

As already described, in another embodiment of the invention the processing of network traffic through appropriate protocol stacks may be efficiently distributed in a multi-processor host computer system. In this embodiment, load distributor **112** assigns or distributes related packets (e.g., packets in the same communication flow) to the same processor. In particular, packets having the same source and destination addresses in their layer three protocol (e.g., IP) headers and/or the same source and destination ports in their layer four protocol (e.g., TCP) headers may be sent to a single processor.

In the NIC illustrated in FIG. **1A**, the processing enhancements discussed above (e.g., re-assembling data, batch processing packet headers, distributing protocol stack processing) are possible for packets received from network **102** that are formatted according to one or more pre-selected protocol stacks. In this embodiment of the invention network **102** is the Internet and NIC **100** is therefore configured to process packets using one of several protocol stacks compatible with the Internet. Packets not configured according to the pre-selected protocols are also processed, but may not receive the benefits of the full suite of processing efficiencies provided to packets meeting the pre-selected protocols.

For example, packets not matching one of the pre-selected protocol stacks may be distributed for processing in a multi-processor system on the basis of the packets' layer two (e.g., medium access control) source and destination addresses rather than their layer three or layer four addresses. Using layer two identifiers provides less granularity to the load distribution procedure, thus possibly distributing the processing of packets less evenly than if layer three/four identifiers were used.

FIG. **1B** depicts one method of using NIC **100** of FIG. **1A** to receive one packet from network **102** and transfer it to a host computer. State **130** is a start state, possibly characterized by the initialization or resetting of NIC **100**.

In state **132**, a packet is received by NIC **100** from network **102**. As already described, the packet may be formatted according to a variety of communication protocols. The packet may be received and initially manipulated by a MAC module before being passed to an IPP module.

In state **134**, a portion of the packet is copied and passed to header parser **106**. Header parser **106** then parses the packet to extract values from one or more of its headers and/or its data. A flow key is generated from some of the retrieved information to identify the communication flow that includes the packet. The degree or extent to which the packet is parsed may depend upon its protocols, in that the header parser may be configured to parse headers of different protocols to different depths. In particular, header parser **106** may be optimized (e.g., its operating instructions configured) for a specific set of protocols or protocol stacks. If the packet conforms to one or more of the specified protocols it may be parsed more fully than a packet that does not adhere to any of the protocols.

In state **136**, information extracted from the packet's headers is forwarded to flow database manager **108** and/or load distributor **112**. The FDBM uses the information to set up a flow in flow database **110** if one does not already exist for this communication flow. If an entry already exists for the packet's flow, it may be updated to reflect the receipt of a new flow packet. Further, FDBM **108** generates an operation code to summarize one or more characteristics or conditions of the packet. The operation code may be used by other modules of NIC **100** to handle the packet in an appropriate manner, as described in subsequent sections. The operation code is returned to the header parser, along with an index (e.g., a flow number) of the packet's flow in the flow database.

In state **138**, load distributor **112** assigns a processor number to the packet, if the host computer includes multiple processors, and returns the processor number to the header processor. Illustratively, the processor number identifies which processor is to conduct the packet through its protocol stack on the host computer. State **138** may be omitted in an alternative embodiment of the invention, particularly if the host computer consists of only a single processor.

In state **140**, the packet is stored in packet queue **116**. As the contents of the packet are placed into the packet queue, checksum generator **114** may compute a checksum. The checksum generator may be informed by IPP module **104** as to which portion of the packet to compute the checksum on. The computed checksum is added to the packet queue as a trailer to the packet. In one embodiment of the invention, the packet is stored in the packet queue at substantially the same time that a copy of a header portion of the packet is provided to header parser **106**.

Also in state **140**, control information for the packet is stored in control queue **118** and information concerning the packet's flow (e.g., flow number, flow key) may be provided to dynamic packet batching module **122**.

In state **142**, NIC **100** determines whether the packet is ready to be transferred to host computer memory. Until it is ready to be transferred, the illustrated procedure waits.

When the packet is ready to be transferred (e.g., the packet is at the head of the packet queue or the host computer receives the packet ahead of this packet in the packet queue), in state **144** dynamic packet batching module **122** determines whether a related packet will soon be transferred. If so, then when the present packet is transferred to host memory the host computer is alerted that a related packet will soon follow. The host computer may then process the packets (e.g., through their protocol stack) as a group.

In state **146**, the packet is transferred (e.g., via a direct memory access operation) to host computer memory. And, in state **148**, the host computer is notified that the packet was transferred. The illustrated procedure then ends at state **150**.

One skilled in the art of computer systems and networking will recognize that the procedure described above is just one method of employing the modules of NIC **100** to receive a single packet from a network and transfer it to a host computer system. Other suitable methods are also contemplated within the scope of the invention.

An Illustrative Packet

FIG. **2** is a diagram of an illustrative packet received by NIC **100** from network **102**. Packet **200** comprises data portion **202** and header portion **204**, and may also contain trailer portion **206**. Depending upon the network environment traversed by packet **200**, its maximum size (e.g., its maximum transfer unit or MTU) may be limited.

In the illustrated embodiment, data portion **202** comprises data being provided to a destination or receiving entity within a computer system (e.g., user, application program, operating system) or a communication subsystem of the computer. Header portion **204** comprises one or more headers prefixed to the data portion by the source or originating entity or a computer system comprising the source entity. Each header normally corresponds to a different communication protocol.

In a typical network environment, such as the Internet, individual headers within header portion **204** are attached (e.g., prepended) as the packet is processed through different layers of a protocol stack (e.g., a set of protocols for communicating between entities) on the transmitting computer system. For example, FIG. **2** depicts protocol headers **210**, **212**, **214** and **216**, corresponding to layers one through four, respectively, of a suitable protocol stack. Each protocol header contains information to be used by the receiving computer system as the packet is received and processed through the protocol stack. Ultimately, each protocol header is removed and data portion **202** is retrieved.

As described in other sections, in one embodiment of the invention a system and method are provided for parsing packet **200** to retrieve various bits of information. In this embodiment, packet **200** is parsed in order to identify the beginning of data portion **202** and to retrieve one or more values for fields within header portion **204**. Illustratively, however, layer one protocol header or preamble **210** corresponds to a hardware-level specification related to the coding of individual bits. Layer one protocols are generally only needed for the physical process of sending or receiving the packet across a conductor. Thus, in this embodiment of the invention layer one preamble **210** is stripped from packet **200** shortly after being received by NIC **100** and is therefore not parsed.

The extent to which header portion **204** is parsed may depend upon how many, if any, of the protocols represented in the header portion match a set of pre-selected protocols. For example, the parsing procedure may be abbreviated or aborted once it is determined that one of the packet's headers corresponds to an unsupported protocol.

In particular, in one embodiment of the invention NIC **100** is configured primarily for Internet traffic. Thus, in this embodiment packet **200** is extensively parsed only when the layer two protocol is Ethernet (either traditional Ethernet or 802.3 Ethernet, with or without tagging for Virtual Local Area Networks), the layer three protocol is IP (Internet Protocol) and the layer four protocol is TCP (Transport Control Protocol). Packets adhering to other protocols may be parsed to some (e.g., lesser) extent. NIC **100** may, however, be configured to support and parse virtually any communication protocol's header. Illustratively, the protocol headers that are parsed, and the extent to which they are parsed, are determined by the configuration of a set of instructions for operating header parser **106**.

As described above, the protocols corresponding to headers **212**, **214** and **216** depend upon the network environment in which a packet is sent. The protocols also depend upon the communicating entities. For example, a packet received by a network interface may be a control packet exchanged between the medium access controllers for the source and destination computer systems. In this case, the packet would be likely to include minimal or no data, and may not include layer three protocol header **214** or layer four protocol header **216**. Control packets are typically used for various purposes related to the management of individual connections.

Another communication flow or connection could involve two application programs. In this case, a packet may include headers **212**, **214** and **216**, as shown in FIG. **2**, and may also include additional headers related to higher layers of a protocol stack (e.g., session, presentation and application layers in the ISO-OSI model). In addition, some applications may include headers or header-like information within data portion **202**. For example, for a Network File System (NFS) application, data portion **202** may include NFS headers related to individual NFS datagrams. A datagram may be defined as a collection of data sent from one entity to another, and may comprise data transmitted in multiple packets. In other words, the amount of data constituting a datagram may be greater than the amount of data that can be included in one packet.

One skilled in the art will appreciate that the methods for parsing a packet that are described in the following section are readily adaptable for packets formatted in accordance with virtually any communication protocol.

One Embodiment of a Header Parser

FIG. **3** depicts header parser **106** of FIG. **1A** in accordance with a present embodiment of the invention. Illustratively, header parser **106** comprises header memory **302** and parser **304**, and parser **304** comprises instruction memory **306**. Although depicted as distinct modules in FIG. **3**, in an alternative embodiment of the invention header memory **302** and instruction memory **306** are contiguous.

In the illustrated embodiment, parser **304** parses a header stored in header memory **302** according to instructions stored in instruction memory **306**. The instructions are designed for the parsing of particular protocols or a particular protocol stack, as discussed above. In one embodiment of the invention, instruction memory **306** is modifiable (e.g., the memory is implemented as RAM, EPROM, EEPROM or the like), so that new or modified parsing instructions may be downloaded or otherwise installed. Instructions for parsing a packet are further discussed in the following section.

In FIG. **3**, a header portion of a packet stored in IPP module **104** (shown in FIG. **1A**) is copied into header memory **302**. Illustratively, a specific number of bytes (e.g., **114**) at the beginning of the packet are copied. In an alternative embodiment of the invention, the portion of a packet that is copied may be of a different size. The particular amount of a packet copied into header memory **302** should be enough to capture one or more protocol headers, or at least enough information (e.g., whether included in a header or data portion of the packet) to retrieve the information described below. The header portion stored in header memory **302** may not include the layer one header, which may be removed prior to or in conjunction with the packet being processed by IPP module **104**.

After a header portion of the packet is stored in header memory **302**, parser **304** parses the header portion according to the instructions stored in instruction memory **306**. In the presently described embodiment, instructions for operating parser **304** apply the formats of selected protocols to step through the contents of header memory **302** and retrieve specific information. In particular, specifications of communication protocols are well known and widely available. Thus, a protocol header may be traversed byte by byte or some other fashion by referring to the protocol specifications. In a present embodiment of the invention the parsing algorithm is dynamic, with information retrieved from one field of a header often altering the manner in which another part is parsed.

For example, it is known that the Type field of a packet adhering to the traditional, form of Ethernet (e.g., version two) begins at the thirteenth byte of the (layer two) header. By comparison, the Type field of a packet following the IEEE 802.3 version of Ethernet begins at the twenty-first byte of the header. The Type field is in yet other locations if the packet forms part of a Virtual Local Area Network (VLAN) communication (which illustratively involves tagging or encapsulating an Ethernet header). Thus, in a present embodiment of the invention, the values in certain fields are retrieved and tested in order to ensure that the information needed from a header is drawn from the correct portion of the header. Details concerning the form of a VLAN packet may be found in specifications for the IEEE 802.3p and IEEE 802.3q forms of the Ethernet protocol.

The operation of header parser **106** also depends upon other differences between protocols, such as whether the packet uses version four or version six of the Internet Protocol, etc. Specifications for versions four and six of IP may be located in IETF (Internet Engineering Task Force) RFCs (Request for Comment) **791** and **2460**, respectively.

The more protocols that are "known" by parser **304**, the more protocols a packet may be tested for, and the more complicated the parsing of a packet's header portion may become. One skilled in the art will appreciate that the protocols that may be parsed by parser **304** are limited only by the instructions according to which it operates. Thus, by augmenting or replacing the parsing instructions stored in instruction memory **306**, virtually all known protocols may be handled by header parser **106** and virtually any information may be retrieved from a packet's headers.

If, of course, a packet header does not conform to an expected or suspected protocol, the parsing operation may be terminated. In this case, the packet may not be suitable for one more of the efficiency enhancements offered by NIC **100** (e.g., data re-assembly, packet batching, load distribution).

Illustratively, the information retrieved from a packet's headers is used by other portions of NIC **100** when processing that packet. For example, as a result of the packet parsing performed by parser **304** a flow key is generated to identify the communication flow or communication connection that comprises the packet. Illustratively, the flow key is assembled by concatenating one or more addresses corresponding to one or more of the communicating entities. In a present embodiment, a flow key is formed from a combination of the source and destination addresses drawn from the IP header and the source and destination ports taken from the TCP header. Other indicia of the communicating entities may be used, such as the Ethernet source and destination addresses (drawn from the layer two header), NFS file handles or source and destination identifiers for other application datagrams drawn from the data portion of the packet.

One skilled in the art will appreciate that the communicating entities may be identified with greater resolution by using indicia drawn from the higher layers of the protocol stack associated with a packet. Thus, a combination of IP and TCP indicia may identify the entities with greater particularity than layer two information.

Besides a flow key, parser **304** also generates a control or status indicator to summarize additional information concerning the packet. In one embodiment of the invention a control indicator includes a sequence number (e.g., TCP sequence number drawn from a TCP header) to ensure the correct ordering of packets when re-assembling their data. The control indicator may also reveal whether certain flags in the packet's headers are set or cleared, whether the packet contains any data, and, if the packet contains data, whether the data exceeds a certain size. Other data are also suitable for inclusion in the control indicator, limited only by the information that is available in the portion of the packet parsed by parser **304**.

In one embodiment of the invention, header parser **106** provides the flow key and all or a portion of the control indicator to flow database manager **108**. As discussed in a following section, FDBM **108** manages a database or other data structure containing information relevant to communication flows passing through NIC **100**.

In other embodiments of the invention, parser **304** produces additional information derived from the header of a packet for use by other modules of NIC **100**. For example, header parser **106** may report the offset, from the beginning of the packet or from some other point, of the data or payload portion of a packet received from a network. As described above, the data portion of a packet typically follows the header portion and may be followed by a trailer portion. Other data that header parser **106** may report include the location in the packet at which a checksum operation should begin, the location in the packet at which the layer three and/or layer four headers begin, diagnostic data, payload information, etc. The term "payload" is often used to refer to the data portion of a packet. In particular, in one embodiment of the invention header parser **106** provides a payload offset and payload size to control queue **118**.

In appropriate circumstances, header parser **106** may also report (e.g., to IPP module **104** and/or control queue **118**) that the packet is not formatted in accordance with the protocols that parser **304** is configured to manipulate. This report may take the form of a signal (e.g., the No_Assist signal described below), alert, flag or other indicator. The signal may be raised or issued whenever the packet is found to reflect a protocol other than the pre-selected protocols that are compatible with the processing enhancements described above (e.g., data re-assembly, batch processing of packet headers, load distribution). For example, in one embodiment of the invention parser **304** may be configured to parse and efficiently process packets using TCP at layer four, IP at layer three and Ethernet at layer two. In this embodiment, an IPX (Internetwork Packet Exchange) packet would not be considered compatible and IPX packets therefore would not be gathered for data re-assembly and batch processing.

At the conclusion of parsing in one embodiment of the invention, the various pieces of information described above are disseminated to appropriate modules of NIC **100**. After this (and as described in a following section), flow database manager **108** determines whether an active flow is associated with the flow key derived from the packet and sets an operation code to be used in subsequent processing. In addition, IPP module **104** transmits the packet to packet queue **116**. IPP module **104** may also receive some of the information extracted by header parser **106**, and pass it to another module of NIC **100**.

In the embodiment of the invention depicted in FIG. **3**, an entire header portion of a received packet to be parsed is copied and then parsed in one evolution, after which the header parser turns its attention to another packet. However, in an alternative embodiment multiple copy and/or parsing operations may be performed on a single packet. In particular, an initial header portion of the packet may be copied into and parsed by header parser **106** in a first evolution, after which another header portion may be copied into header parser **106** and parsed in a second evolution. A header portion in one evolution may partially or completely overlap the header portion of another evolution. In this manner, extensive headers may be parsed even if header memory **302** is of limited size. Similarly, it may require more than one operation to load a full set of instructions for parsing a packet into instruction memory **306**. Illustratively, a first portion of the instructions may be loaded and executed, after which other instructions are loaded.

With reference now to FIGS. **4A–4B**, a flow chart is presented to illustrate one method by which a header parser may parse a header portion of a packet received at a network interface circuit from a network. In this implementation, the header parser is configured, or optimized, for parsing packets conforming to a set of pre-selected protocols (or protocol stacks). For packets meeting these criteria, various information is retrieved from the header portion to assist in the re-assembly of the data portions of related packets (e.g., packets comprising data from a single datagram). Other enhanced features of the network interface circuit may also be enabled.

The information generated by the header parser includes, in particular, a flow key with which to identify the communication flow or communication connection that comprises the received packet. In one embodiment of the invention, data from packets having the same flow key may be identified and re-assembled to form a datagram. In addition, headers of packets having the same flow key may be processed collectively through their protocol stack (e.g., rather than serially).

In another embodiment of the invention, information retrieved by the header parser is also used to distribute the processing of network traffic received from a network. For example, multiple packets having the same flow key may be submitted to a single processor of a multi-processor host computer system.

In the method illustrated in FIGS. **4A–4B**, the set of pre-selected protocols corresponds to communication protocols frequently transmitted via the Internet. In particular, the set of protocols that may be extensively parsed in this method include the following. At layer two: Ethernet (traditional version), 802.3 Ethernet, Ethernet VLAN (Virtual Local Area Network) and 802.3 Ethernet VLAN. At layer three: IPv4 (with no options) and IPv6 (with no options). Finally, at layer four, only TCP protocol headers (with or without options) are parsed in the illustrated method. Header parsers in alternative embodiments of the invention parse packets formatted through other protocol stacks. In particular, a NIC may be configured in accordance with the most common protocol stacks in use on a given network, which may or may not include the protocols compatible with the header parser method illustrated in FIGS. **4A–4B**.

As described below, a received packet that does not correspond to the protocols parsed by a given method may be flagged and the parsing algorithm terminated for that packet. Because the protocols under which a packet has been formatted can only be determined, in the present method, by examining certain header field values, the determination that a packet does not conform to the selected set of protocols may be made at virtually any time during the procedure. Thus, the illustrated parsing method has as one goal the

identification of packets not meeting the formatting criteria for re-assembly of data.

Various protocol header fields appearing in headers for the selected protocols are discussed below. Communication protocols that may be compatible with an embodiment of the present invention (e.g., protocols that may be parsed by a header parser) are well known to persons skilled in the art and are described with great particularity in a number of references. They therefore need not be visited in minute detail herein. In addition, the illustrated method of parsing a header portion of a packet for the selected protocols is merely one method of gathering the information described below. Other parsing procedures capable of doing so are equally suitable.

In a present embodiment of the invention, the illustrated procedure is implemented as a combination of hardware and software. For example, updateable micro-code instructions for performing the procedure may be executed by a microsequencer. Alternatively, such instructions may be fixed (e.g., stored in read-only memory) or may be executed by a processor or microprocessor.

In FIGS. **4A–4B**, state **400** is a start state during which a packet is received by NIC **100** (shown in FIG. **1A**) and initial processing is performed. NIC **100** is coupled to the Internet for purposes of this procedure. Initial processing may include basic error checking and the removal of the layer one preamble. After initial processing, the packet is held by IPP module **104** (also shown in FIG. **1A**). In one embodiment of the invention, state **400** comprises a logical loop in which the header parser remains in an idle or wait state until a packet is received.

In state **402**, a header portion of the packet is copied into memory (e.g., header memory **302** of FIG. **3**). In a present embodiment of the invention a predetermined number of bytes at the beginning (e.g., 114 bytes) of the packet are copied. Packet portions of different sizes are copied in alternative embodiments of the invention, the sizes of which are guided by the goal of copying enough of the packet to capture and/or identify the necessary header information. Illustratively, the full packet is retained by IPP module **104** while the following parsing operations are performed, although the packet may, alternatively, be stored in packet queue **116** prior to the completion of parsing.

Also in state **402**, a pointer to be used in parsing the packet may be initialized. Because the layer one preamble was removed, the header portion copied to memory should begin with the layer two protocol header. Illustratively, therefore, the pointer is initially set to point to the twelfth byte of the layer two protocol header and the two-byte value at the pointer position is read. As one skilled in the art will recognize, these two bytes may be part of a number of different fields, depending upon which protocol constitutes layer two of the packet's protocol stack. For example, these two bytes may comprise the Type field of a traditional Ethernet header, the Length field of an 802.3 Ethernet header or the TPID (Tag Protocol IDentifier) field of a VLAN-tagged header.

In state **404**, a first examination is made of the layer two header to determine if it comprises a VLAN-tagged layer two protocol header. Illustratively, this determination depends upon whether the two bytes at the pointer position store the hexadecimal value 8100. If so, the pointer is probably located at the TPID field of a VLAN-tagged header. If not a VLAN header, the procedure proceeds to state **408**.

If, however, the layer two header is a VLAN-tagged header, in state **406** the CFI (Canonical Format Indicator) bit is examined. If the CFI bit is set (e.g., equal to one), the illustrated procedure jumps to state **430**, after which it exits. In this embodiment of the invention the CFI bit, when set, indicates that the format of the packet is not compatible with (i.e., does not comply with) the pre-selected protocols (e.g., the layer two protocol is not Ethernet or 802.3 Ethernet). If the CFI bit is clear (e.g., equal to zero), the pointer is incremented (e.g., by four bytes) to position it at the next field that must be examined.

In state **408**, the layer two header is further tested. Although it is now known whether this is or is not a VLAN-tagged header, depending upon whether state **408** was reached through state **406** or directly from state **404**, respectively, the header may reflect either the traditional Ethernet format or the 802.3 Ethernet format. At the beginning of state **408**, the pointer is either at the twelfth or sixteenth byte of the header, either of which may correspond to a Length field or a Type field. In particular, if the two-byte value at the position identified by the pointer is less than 0600 (hexadecimal), then the packet corresponds to 802.3 Ethernet and the pointer is understood to identify a Length field. Otherwise, the packet is a traditional (e.g., version two) Ethernet packet and the pointer identifies a Type field.

If the layer two protocol is 802.3 Ethernet, the procedure continues at state **410**. If the layer two protocol is traditional Ethernet, the Type field is tested for the hexadecimal values of 0800 and 08DD. If the tested field has one of these values, then it has also been determined that the packet's layer three protocol is the Internet Protocol. In this case the illustrated procedure continues at state **412**. Lastly, if the field is a Type field having a value other than 0800 or 86DD (hexadecimal), then the packet's layer three protocol does not match the pre-selected protocols according to which the header parser was configured. Therefore, the procedure continues at state **430** and then ends.

In one embodiment of the invention the packet is examined in state **408** to determine if it is a jumbo Ethernet frame. This determination would likely be made prior to deciding whether the layer two header conforms to Ethernet or 802.3 Ethernet. Illustratively, the jumbo frame determination may be made based on the size of the packet, which may be reported by IPP module **104** or a MAC module. If the packet is a jumbo frame, the procedure may continue at state **410**; otherwise, it may resume at state **412**.

In state **410**, the procedure verifies that the layer two protocol is 802.3 Ethernet with LLC SNAP encapsulation. In particular, the pointer is advanced (e.g., by two bytes) and the six-byte value following the Length field in the layer two header is retrieved and examined. If the header is an 802.3 Ethernet header, the field is the LLC_SNAP field and should have a value of AAAA03000000 (hexadecimal). The original specification for an LLC SNAP header may be found in the specification for IEEE 802.2. If the value in the packet's LLC_SNAP field matches the expected value the pointer is incremented another six bytes, the two-byte 802.3 Ethernet Type field is read and the procedure continues at state **412**. If the values do not match, then the packet does not conform to the specified protocols and the procedure enters state **430** and then ends.

In state **412**, the pointer is advanced (e.g., another two bytes) to locate the beginning of the layer three protocol header. This pointer position may be saved for later use in quickly identifying the beginning of this header. The packet is now known to conform to an accepted layer two protocol (e.g., traditional Ethernet, Ethernet with VLAN tagging, or 802.3 Ethernet with LLC SNAP) and is now checked to ensure that the packet's layer three protocol is IP. As

NOAC EX. 1015 Page 522

discussed above, in the illustrated embodiment only packets conforming to the IP protocol are extensively processed by the header parser.

Illustratively, if the value of the Type field in the layer two header (retrieved in state **402** or state **410**) is 0800 (hexadecimal), the layer three protocol is expected to be IP, version four. If the value is 86DD (hexadecimal), the layer three protocol is expected to be IP, version six. Thus, the Type field is tested in state **412** and the procedure continues at state **414** or state **418**, depending upon whether the hexadecimal value is 0800 or 86DD, respectively.

In state **414**, the layer three header's conformity with version four of IP is verified. In one embodiment of the invention the Version field of the layer three header is tested to ensure that it contains the hexadecimal value **4**, corresponding to version four of IP. If in state **414** the layer three header is confirmed to be IP version four, the procedure continues at state **416**; otherwise, the procedure proceeds to state **430** and then ends at state **432**.

In state **416**, various pieces of information from the IP header are saved. This information may include the IHL (IP Header Length), Total Length, Protocol and/or Fragment Offset fields. The IP source address and the IP destination addresses may also be stored. The source and destination address values are each four bytes long in version four of IP. These addresses are used, as described above, to generate a flow key that identifies the communication flow in which this packet was sent. The Total Length field stores the size of the IP segment of this packet, which illustratively comprises the IP header, the TCP header and the packet's data portion. The TCP segment size of the packet (e.g., the size of the TCP header plus the size of the data portion of the packet) may be calculated by subtracting twenty bytes (the size of the IP version four header) from the Total Length value. After state **416**, the illustrated procedure advances to state **422**.

In state **418**, the layer three header's conformity with version six of IP is verified by testing the Version field for the hexadecimal value **6**. If the Version field does not contain this value, the illustrated procedure proceeds to state **430**.

In state **420**, the values of the Payload Length (e.g., the size of the TCP segment) and Next Header field are saved, plus the IP source and destination addresses. Source and destination addresses are each sixteen bytes long in version six of IP.

In state **422** of the illustrated procedure, it is determined whether the IP header (either version four or version six) indicates that the layer four header is TCP. Illustratively, the Protocol field of a version four IP header is tested while the Next Header field of a version six header is tested. In either case, the value should be 6 (hexadecimal). The pointer is then incremented as necessary (e.g., twenty bytes for IP version four, forty bytes for IP version six) to reach the beginning of the TCP header. If it is determined in state **422** that the layer four header is not TCP, the procedure advances to state **430** and ends at end state **432**.

In one embodiment of the invention, other fields of a version four IP header may be tested in state **422** to ensure that the packet meets the criteria for enhanced processing by NIC **100**. For example, an IHL field value other than 5 (hexadecimal) indicates that IP options are set for this packet, in which case the parsing operation is aborted. A fragmentation field value other than zero indicates that the IP segment of the packet is a fragment, in which case parsing is also aborted. In either case, the procedure jumps to state **430** and then ends at end state **432**.

In state **424**, the packet's TCP header is parsed and various data are collected from it. In particular, the TCP source port and destination port values are saved. The TCP sequence number, which is used to ensure the correct re-assembly of data from multiple packets, is also saved. Further, the values of several components of the Flags field—illustratively, the URG (urgent), PSH (push), RST (reset), SYN (synch) and FIN (finish) bits—are saved. As will be seen in a later section, in one embodiment of the invention these flags signal various actions to be performed or statuses to be considered in the handling of the packet.

Other signals or statuses may be generated in state **424** to reflect information retrieved from the TCP header. For example, the point from which a checksum operation is to begin may be saved (illustratively, the beginning of the TCP header); the ending point of a checksum operation may also be saved (illustratively, the end of the data portion of the packet). An offset to the data portion of the packet may be identified by multiplying the value of the Header Length field of the TCP header by four. The size of the data portion may then be calculated by subtracting the offset to the data portion from the size of the entire TCP segment.

In state **426**, a flow key is assembled by concatenating the IP source and destination addresses and the TCP source and destination ports. As already described, the flow key may be used to identify a communication flow or communication connection, and may be used by other modules of NIC **100** to process network traffic more efficiently. Although the sizes of the source and destination addresses differ between IP versions four and six (e.g., four bytes each versus sixteen bytes each, respectively), in the presently described embodiment of the invention all flow keys are of uniform size. In particular, in this embodiment they are thirty-six bytes long, including the two-byte TCP source port and two-byte TCP destination port. Flow keys generated from IP, version four, packet headers are padded as necessary (e.g., with twenty-four clear bytes) to fill the flow key's allocated space.

In state **428**, a control or status indicator is assembled to provide various information to one or more modules of NIC **100**. In one embodiment of the invention a control indicator includes the packet's TCP sequence number, a flag or identifier (e.g., one or more bits) indicating whether the packet contains data (e.g., whether the TCP payload size is greater than zero), a flag indicating whether the data portion of the packet exceeds a pre-determined size, and a flag indicating whether certain entries in the TCP Flags field are equivalent to pre-determined values. The latter flag may, for example, be used to inform another module of NIC **100** that components of the Flags field do or do not have a particular configuration. After state **428**, the illustrated procedure ends with state **432**.

State **430** may be entered at several different points of the illustrated procedure. This state is entered, for example, when it is determined that a header portion that is being parsed by a header parser does not conform to the pre-selected protocol stacks identified above. As a result, much of the information described above is not retrieved. A practical consequence of the inability to retrieve this information is that it then cannot be provided to other modules of NIC **100** and the enhanced processing described above and in following sections may not be performed for this packet. In particular, and as discussed previously, in a present embodiment of the invention one or more enhanced operations may be performed on parsed packets to increase the efficiency with which they are processed. Illustrative operations that may be applied include the re-assembly of data from related packets (e.g., packets containing data from a single datagram), batch processing of packet headers through a protocol stack, load distribution or load sharing of

NOAC Ex. 1015 Page 523

protocol stack processing, efficient transfer of packet data to a destination entity, etc.

In the illustrated procedure, in state **430** a flag or signal (illustratively termed No_Assist) is set or cleared to indicate that the packet presently held by IPP module **104** (e.g., which was just processed by the header parser) does not conform to any of the pre-selected protocol stacks. This flag or signal may be relied upon by another module of NIC **100** when deciding whether to perform one of the enhanced operations.

Another flag or signal may be set or cleared in state **430** to initialize a checksum parameter indicating that a checksum operation, if performed, should start at the beginning of the packet (e.g., with no offset into the packet). Illustratively, incompatible packets cannot be parsed to determine a more appropriate point from which to begin the checksum operation. After state **430**, the procedure ends with end state **432**.

After parsing a packet, the header parser may distribute information generated from the packet to one or more modules of NIC **100**. For example, in one embodiment of the invention the flow key is provided to flow database manager **108**, load distributor **112** and one or both of control queue **118** and packet queue **116**. Illustratively, the control indicator is provided to flow database manager **108**. This and other control information, such as TCP payload size, TCP payload offset and the No_Assist signal may be returned to IPP module **104** and provided to control queue **118**. Yet additional control and/or diagnostic information, such as offsets to the layer three and/or layer four headers, may be provided to IPP module **104**, packet queue **116** and/or control queue **118**. Checksum information (e.g., a starting point and either an ending point or other means of identifying a portion of the packet from which to compute a checksum) may be provided to checksum generator **114**.

As discussed in a following section, although a received packet is parsed on NIC **100** (e.g., by header parser **106**), the packets are still processed (e.g., through their respective protocol stacks) on the host computer system in the illustrated embodiment of the invention. However, after parsing a packet in an alternative embodiment of the invention, NIC **100** also performs one or more subsequent processing steps. For example, NIC **100** may include one or more protocol processors for processing one or more of the packet's protocol headers.

Dynamic Header Parsing Instructions in One Embodiment of the Invention

In one embodiment of the present invention, header parser **106** parses a packet received from a network according to a dynamic sequence of instructions. The instructions may be stored in the header parser's instruction memory (e.g., RAM, SRAM, DRAM, flash) that is re-programmable or that can otherwise be updated with new or additional instructions. In one embodiment of the invention software executing on a host computer (e.g., a device driver) may download a set of parsing instructions for storage in the header parser memory.

The number and format of instructions stored in a header parser's instruction memory may be tailored to one or more specific protocols or protocol stacks. An instruction set configured for one collection of protocols, or a program constructed from that instruction set, may therefore be updated or replaced by a different instruction set or program. Packets received at the network interface that are formed in accordance with the selected protocols (e.g., "compatible" packets), as determined by analyzing or parsing packets, various enhancements in the handling of network traffic become possible as described in the following sections. In particular, packets from one datagram that are configured according to a selected protocol may be re-assembled for efficient transfer in a host computer. In addition, header portions of such packets may be processed collectively rather than serially. And, the processing of packets from different datagrams by a multi-processor host computer may be shared or distributed among the processors. Therefore, one objective of a dynamic header parsing operation is to identify a protocol according to which a received packet has been formatted or determine whether a packet header conforms to a particular protocol.

FIG. **23**, discussed in detail shortly, presents an illustrative series of instructions for parsing the layer two, three and four headers of a packet to determine if they are Ethernet, IP and TCP, respectively. The illustrated instructions comprise one possible program or microcode for performing a parsing operation. As one skilled in the art will recognize, after a particular set of parsing instructions is loaded into a parser memory, a number of different programs may be assembled. FIG. **23** thus presents merely one of a number of programs that may be generated from the stored instructions. The instructions presented in FIG. **23** may be performed or executed by a microsequencer, a processor, a microprocessor or other similar module located within a network interface circuit.

In particular, other instruction sets and other programs may be derived for different communication protocols, and may be expanded to other layers of a protocol stack. For example, a set of instructions could be generated for parsing NFS (Network File System) packets. Illustratively, these instructions would be configured to parse layer five and six headers to determine if they are Remote Procedure Call (RPC) and External Data Representation (XDR), respectively. Other instructions could be configured to parse a portion of the packet's data (which may be considered layer seven). An NFS header may be considered a part of a packet's layer six protocol header or part of the packet's data.

One type of instruction executed by a microsequencer may be designed to locate a particular field of a packet (e.g., at a specific offset within the packet) and compare the value stored at that offset to a value associated with that field in a particular communication protocol. For example, one instruction may require the microsequencer to examine a value in a packet header at an offset that would correspond to a Type field of an Ethernet header. By comparing the value actually stored in the packet with the value expected for the protocol, the microsequencer can determine if the packet appears to conform to the Ethernet protocol. Illustratively, the next instruction applied in the parsing program depends upon whether the previous comparison was successful. Thus, the particular instructions applied by the microsequencer, and the sequence in which applied, depend upon which protocols are represented by the packet's headers.

The microsequencer may test one or more field values within each header included in a packet. The more fields that are tested and that are found to comport with the format of a known protocol, the greater the certainty that the packet conforms to that protocol. As one skilled in the art will appreciate, one communication protocol may be quite different than another protocol, thus requiring examination of different parts of packet headers for different protocols. Illustratively, the parsing of one packet may end in the event of an error or because NIC was determined that the packet being parsed does or does not conform to the protocol(s) the instructions are designed for.

NOAC Ex. 1015 Page 524

Each instruction in FIG. **23** may be identified by a number and/or a name. A particular instruction may perform a variety of tasks other than comparing a header field to an expected value. An instruction may, for example, call another instruction to examine another portion of a packet header, initialize, load or configure a register or other data structure, prepare for the arrival and parsing of another packet, etc. In particular, a register or other storage structure may be configured in anticipation of an operation that is performed in the network interface after the packet is parsed. For example, a program instruction in FIG. **23** may identify an output operation that may or may not be performed, depending upon the success or failure of the comparison of a value extracted from a packet with an expected value. An output operation may store a value in a register, configure a register (e.g., load an argument or operator) for a post-parsing operation, clear a register to await a new packet, etc.

A pointer may be employed to identify an offset into a packet being parsed. In one embodiment, such a pointer is initially located at the beginning of the layer two protocol header. In another embodiment, however, the pointer is situated at a specific location within a particular header (e.g., immediately following the layer two destination and/or source addresses) when parsing commences. Illustratively, the pointer is incremented through the packet as the parsing procedure executes. In one alternative embodiment, however, offsets to areas of interest in the packet may be computed from one or more known or computed locations. In the parsing program depicted in FIG. **23**, a header is navigated (e.g., the pointer is advanced) in increments of two bytes (e.g., sixteen-bit words). In addition, where a particular field of a header is compared to a known or expected value, up to two bytes are extracted at a time from the field. Further, when a value or header field is copied for storage in a register or other data structure, the amount of data that may be copied in one operation may be expressed in multiples of two-byte units or in other units altogether (e.g., individual bytes). This unit of measurement (e.g., two bytes) may be increased or decreased in an alternative embodiment of the invention. Altering the unit of measurement may alter the precision with which a header can be parsed or a header value can be extracted.

In the embodiment of the invention illustrated in FIG. **23**, a set of instructions loaded into the header parser's instruction memory comprises a number of possible operations to be performed while testing a packet for compatibility with selected protocols. Program **2300** is generated from the instruction set. Program **2300** is thus merely one possible program, microcode or sequence of instructions that can be created from the available instruction set.

In this embodiment, the loaded instruction set enables the following sixteen operations that may be performed on a packet that is being parsed. Specific implementations of these operations in program **2300** are discussed in additional detail below. These instructions will be understood to be illustrative in nature and do not limit the composition of instruction sets in other embodiments of the invention. In addition, any subset of these operations may be employed in a particular parsing program or microcode. Further, multiple operations may employ the same operation and have different effects.

A CLR_REG operation allows the selective initialization of registers or other data structures used in program **2300** (possibly, data structures used in functions performed after a packet is parsed. Initialization may comprise storing a value of zero. A number of illustrative registers that may be cleared by a CLR_REG operation are identified in the following operations.

A LD_FID operation copies a variable amount of data from a particular offset within the packet into a register configured to store a packet's flow key or other flow identifier. This register may be termed a FLOWID register. The effect of an LD_FID operation is cumulative. In other words, each time it is invoked for one packet the generated data is appended to the flow key data stored previously.

A LD_SEQ operation copies a variable amount of data from a particular offset within the packet into a register configured to store a packet's sequence number (e.g., a TCP sequence number). This register may be assigned the label SEQNO. This operation is also cumulative—the second and subsequent invocations of this operation for the packet cause the identified data to be appended to data stored previously.

A LD_CTL operation loads a value from a specified offset in the packet into a CONTROL register. The CONTROL register may comprise a control indicator discussed in a previous section for identifying whether a packet is suitable for data re-assembly, packet batching, load distribution or other enhanced functions of NIC **100**. In particular, a control indicator may indicate whether a No_Assist flag should be raised for the packet, whether the packet includes any data, whether the amount of packet data is larger than a predetermined threshold, etc. Thus, the value loaded into a CONTROL register in a LD_CTL operation may affect the post-parsing handling of the packet.

A LD_SAP operation loads a value into the CONTROL register from a variable offset within the packet. The loaded value may comprise the packet's ethertype. In one option that may be associated with a LD_SAP operation, the offset of the packet's layer three header may also be stored in the CONTROL register or elsewhere. As one skilled in the art will recognize, a packet's layer three header may immediately follow its layer two ethertype field if the packet conforms to the Ethernet and IP protocols.

A LD_R1 operation may be used to load a value into a temporary register (e.g., named R1) from a variable offset within the packet. A temporary register may be used for a variety of tasks, such as accumulating values to determine the length of a header or other portion of the packet. A LD_R1 operation may also cause a value from another variable offset to be stored in a second temporary register (e.g., named R2). The values stored in the R1 and/or R2 registers during the parsing of a packet may or may not be cumulative.

A LD_L3 operation may load a value from the packet into a register configured to store the location of the packet's layer three header. This register may be named L3OFFSET. In one optional method of invoking this operation, it may be used to load a fixed value into the L3OFFSET register. As another option, the LD_L3 operation may add a value stored in a temporary register (e.g., R1) to the value being stored in the L3OFFSET register.

A LD_SUM operation stores the starting point within the packet from which a checksum should be calculated. The register in which this value is stored may be named a CSUMSTART register. In one alternative invocation of this operation, a fixed or predetermined value is stored in the register. As another option, the LD_SUM operation may add a value stored in a temporary register (e.g., R1) to the value being stored in the CSUMSTART register.

A LD_HDR operation loads a value into a register configured to store the location within the packet at which the header portion may be split. The value that is stored may, for example, be used during the transfer of the packet to the host computer to store a data portion of the packet in a separate location than the header portion. The loaded value

may thus identify the beginning of the packet data or the beginning of a particular header. In one invocation of a LD_HDR operation, the stored value may be computed from a present position of a parsing pointer described above. In another invocation, a fixed or predetermined value may be store. As yet another alternative, a value stored in a temporary register (e.g., R1) and/or a constant may be added to the loaded value.

A LD_LEN operation stores the length of the packet's payload into a register (e.g., a PAYLOADLEN register).

An IM_FID operation appends or adds a fixed or predetermined value to the existing contents of the FLOWID register described above.

An IM_SEQ operation appends or adds a fixed or predetermined value to the contents of the SEQNO register described above.

An IM_SAP operation loads or stores a fixed or predetermined value in the CSUMSTART register described above.

An IM_R1 operation may add or load a predetermined value in one or more temporary registers (e.g., R1, R2).

An IM_CTL operation loads or stores a fixed or predetermined value in the CONTROL register described above.

A ST_FLAG operation loads a value from a specified offset in the packet into a FLAGS register. The loaded value may comprise one or more fields or flags from a packet header.

One skilled in the art will recognize that the labels assigned to the operations and registers described above and elsewhere in this section are merely illustrative in nature and in no way limit the operations and parsing instructions that may be employed in other embodiments of the invention.

Instructions in program **2300** comprise instruction number field **2302**, which contains a number of an instruction within the program, and instruction name field **2304**, which contains a name of an instruction. In an alternative embodiment of the invention instruction number and instruction name fields may be merged or one of them may be omitted.

Instruction content field **2306** includes multiple portions for executing an instruction. An "extraction mask" portion of an instruction is a two-byte mask in hexadecimal notation. An extraction mask identifies a portion of a packet header to be copied or extracted, starting from the current packet offset (e.g., the current position of the parsing pointer). Illustratively, each bit in the packet's header that corresponds to a one in the hexadecimal value is copied for comparison to a comparison or test value. For example, a value of 0xFF00 in the extraction mask portion of an instruction signifies that the entire first byte at the current packet offset is to be copied and that the contents of the second byte are irrelevant. Similarly, an extraction mask of 0x3FFF signifies that all but the two most significant bits of the first byte are to be copied. A two-byte value is constructed from the extracted contents, using whatever was copied from the packet. Illustratively, the remainder of the value is padded with zeros. One skilled in the art will appreciate that the format of an extraction mask (or an output mask, described below) may be adjusted as necessary to reflect little endian or big endian representation.

One or more instructions in a parsing program may not require any data extracted from the packet at the pointer location to be able to perform its output operation. These instructions may have an extraction mask value of 0x0000 to indicate that although a two-byte value is still retrieved from the pointer position, every bit of the value is masked off. Such an extraction mask thus yields a definite value of zero. This type of instruction may be used when, for example, an output operation needs to be performed before another substantive portion of header data is extracted with an extraction mask other than 0x0000.

A "compare value" portion of an instruction is a two-byte hexadecimal value with which the extracted packet contents are to be compared. The compare value may be a value known to be stored in a particular field of a specific protocol header. The compare value may comprise a value that the extracted portion of the header should match or have a specified relationship to in order for the packet to be considered compatible with the pre-selected protocols.

An "operator" portion of an instruction identifies an operator signifying how the extracted and compare values are to be compared. Illustratively, EQ signifies that they are tested for equality, NE signifies that they are tested for inequality, LT signifies that the extracted value must be less than the compare value for the comparison to succeed, GE signifies that the extracted value must be greater than or equal to the compare value, etc. An instruction that awaits arrival of a new packet to be parsed may employ an operation of NP. Other operators for other functions may be added and the existing operators may be assigned other monikers.

A "success offset" portion of an instruction indicates the number of two-byte units that the pointer is to advance if the comparison between the extracted and test values succeeds. A "success instruction" portion of an instruction identifies the next instruction in program **2300** to execute if the comparison is successful.

Similarly, "failure offset" and "failure instruction" portions indicate the number of two-byte units to advance the pointer and the next instruction to execute, respectively, if the comparison fails. Although offsets are expressed in units of two bytes (e.g., sixteen-bit words) in this embodiment of the invention, in an alternative embodiment of the invention they may be smaller or larger units. Further, as mentioned above an instruction may be identified by number or name.

Not all of the instructions in a program are necessarily used for each packet that is parsed. For example, a program may include instructions to test for more than one type or version of a protocol at a particular layer. In particular, program **2300** tests for either version four or six of the IP protocol at layer three. The instructions that are actually executed for a given packet will thus depend upon the format of the packet. Once a packet has been parsed as much as possible with a given program or it has been determined that the packet does or does not conform to a selected protocol, the parsing may cease or an instruction for halting the parsing procedure may be executed. Illustratively, a next instruction portion of an instruction (e.g., "success instruction" or "failure instruction") with the value "DONE" indicates the completion of parsing of a packet. A DONE, or similar, instruction may be a dummy instruction. In other words, "DONE" may simply signify that parsing to be terminated for the present packet. Or, like instruction eighteen of program **2300**, a DONE instruction may take some action to await a new packet (e.g., by initializing a register).

The remaining portions of instruction content field **2306** are used to specify and complete an output or other data storage operation. In particular, in this embodiment an "output operation" portion of an instruction corresponds to the operations included in the loaded instruction set. Thus, for program **2300**, the output operation portion of an instruction identifies one of the sixteen operations described above. The output operations employed in program **2300** are further described below in conjunction with individual instructions.

An "operation argument" portion of an instruction comprises one or more arguments or fields to be stored, loaded

therwise used in conjunction with the instruction's ut operation. Illustratively, the operation argument takes the form of a multi-bit hexadecimal value. For ram **2300**, operation arguments are eleven bits in size. rgument or portion of an argument may have various nings, depending upon the output operation. For nple, an operation argument may comprise one or more erical values to be stored in a register or to be used to e or delimit a portion of a header. Or, an argument bit comprise a flag to signal an action or status. In cular, one argument bit may specify that a particular ter is to be reset; a set of argument bits may comprise ffset into a packet header to a value to be stored in a ter, etc. Illustratively, the offset specified by an operaargument is applied to the location of the parsing pointer tion before the pointer is advanced as specified by the icable success offset or failure offset. The operation ments used in program **2300** are explained in further il below.

n "operation enabler" portion of an instruction content specifies whether or when an instruction's output ation is to be performed. In particular, in the illustrated odiment of the invention an instruction's output operamay or may not be performed, depending on the result e comparison between a value extracted from a header the compare value. For example, an output enabler may et to a first value (e.g., zero) if the output operation is r to be performed. It may take different values if it is to erformed only when the comparison does or does not fy the operator (e.g., one or two, respectively). An ation enabler may take yet another value (e.g., three) if always to be performed.

"shift" portion of an instruction comprises a value cating how an output value is to be shifted. A shift may ecessary because different protocols sometime require es to be formatted differently. In addition, a value cating a length or location of a header or header field require shifting in order to reflect the appropriate nitude represented by the value. For example, because ram **2300** is designed to use two-byte units, a value may l to be shifted if it is to reflect other units (e.g., bytes). ift value in a present embodiment indicates the number positions (e.g., bits) to right-shift an output value. In her embodiment of the invention a shift value may esent a different shift type or direction.

inally, an "output mask" specifies how a value being ed in a register or other data structure is to be formatted. stated above, an output operation may require an acted, computed or assembled value to be stored. Similar he extraction mask, the output mask is a two-byte adecimal value. For every position in the output mask contains a one, in this embodiment of the invention the esponding bit in the two-byte value identified by the ut operation and/or operation argument is to be stored. example, a value of 0xFFFF indicates that the specified -byte value is to be stored as is. Illustratively, for every ition in the output mask that contains a zero, a zero is ed. Thus, a value of 0xF000 indicates that the most ificant four bits of the first byte are to be stored, but the of the stored value is irrelevant, and may be padded with s.

An output operation of "NONE" may be used to indicate there is no output operation to be performed or stored, which case other instruction portions pertaining to output be ignored or may comprise specified values (e.g., all s). In the program depicted in FIG. **23**, however, a R_REG output operation, which allows the selective re-initialization of registers, may be used with an operation argument of zero to effectively perform no output. In particular, an operation argument of zero for the CLR_REG operation indicates that no registers are to be reset. In an alternative embodiment of the invention the operation enabler portion of an instruction could be set to a value (e.g., zero) indicating that the output operation is never to be performed.

The format and sequence of instructions in FIG. **23** will be understood to represent just one method of parsing a packet to determine whether it conforms to a particular communication protocol. In particular, the instructions are designed to examine one or more portions of one or more packet headers for comparison to known or expected values and to configure or load a register or other storage location as necessary. As one skilled in the art will appreciate, instructions for parsing a packet may take any of a number of forms and be performed in a variety of sequences without exceeding the scope of the invention.

With reference now to FIG. **23**, instructions in program **2300** may be described in detail. Prior to execution of the program depicted in FIG. **23**, a parsing pointer is situated at the beginning of a packet's layer two header. The position of the parsing pointer may be stored in a register for easy reference and update during the parsing procedure. In particular, the position of the parsing pointer as an offset (e.g., from the beginning of the layer two header) may be used in computing the position of a particular position within a header.

Program **2300** begins with a WAIT instruction (e.g., instruction zero) that waits for a new packet (e.g., indicated by operator NP) and, when one is received, sets a parsing pointer to the twelfth byte of the layer two header. This offset to the twelfth byte is indicated by the success offset portion of the instruction. Until a packet is received, the WAIT instruction loops on itself. In addition, a CLR_REG operation is conducted, but the operation enabler setting indicates that it is only conducted when the comparison succeeds (e.g., when a new packet is received).

The specified CLR_REG operation operates according to the WAIT instruction's operation argument (i.e., 0x3FF). In this embodiment, each bit of the argument corresponds to a register or other data structure. The registers initialized in this operation may include the following: ADDR (e.g., to store the parsing pointer's address or location), FLOWID (e.g., to store the packet's flow key), SEQNO (e.g., to store a TCP sequence number), SAP (e.g., the packet's ethertype) and PAYLOADLEN (e.g., payload length). The following registers configured to store certain offsets may also be reset: FLOWOFF (e.g., offset within FLOWID register), SEQOFF (e.g., offset within SEQNO register), L3OFFSET (e.g., offset of the packet's layer three header), HDRSPLIT (e.g., location to split packet) and CSUMSTART (e.g., starting location for computing a checksum). Also, one or more status or control indicators (e.g. CONTROL or FLAGS register) for reporting the status of one or more flags of a packet header may be reset. In addition, one or more temporary registers (e.g., R1, R2) or other data structures may also be initialized. These registers are merely illustrative of the data structures that may be employed in one embodiment of the invention. Other data structures may be employed in other embodiments for the same or different output operations.

Temporary registers such as R1 and/or R2 may be used in program **2300** to track various headers and header fields. One skilled in the art will recognize the number of possible combinations of communication protocols and the effect of

those various combinations on the structure and format of a packet's headers. More information may need to be examined or gathered from a packet conforming to one protocol or set of protocols than from a packet conforming to another protocol or set of protocols. For example, if extension headers are used with an Internet Protocol header, values from those extension headers and/or their lengths may need to be stored, which values are not needed if extension headers are not used. When calculating a particular offset, such as an offset to the beginning of a packet's data portion for example, multiple registers may need to be maintained and their values combined or added. In this example, one register or temporary register may track the size or format of an extension header, while another register tracks the base IP header.

Instruction VLAN (e.g., instruction one) examines the two-byte field at the parsing pointer position (possibly a Type, Length or TPID field) for a value indicating a VLAN-tagged header (e.g., 8100 in hexadecimal). If the header is VLAN-tagged, the pointer is incremented a couple of bytes (e.g., one two-byte unit) and execution continues with instruction CFI; otherwise, execution continues with instruction 802.3. In either event, the instruction's operation enabler indicates that an IM_CTL operation is always to be performed.

As described above, an IM_CTL operation causes a control register or other data structure to be populated with one or more flags to report the status or condition of a packet. As described in the previous section, a control indicator may indicate whether a packet is suitable for enhanced processing (e.g., whether a No_Assist signal should be generated for the packet), whether a packet includes any data and, if so, whether the size of the data portion exceeds a specified threshold. The operation argument 0x00A for instruction VLAN comprises the value to be stored in the control register, with individual bits of the argument corresponding to particular flags. Illustratively, flags associated with the conditions just described may be set to one, or true, in this IM_CTL operation.

Instruction CFI (e.g., instruction two) examines the CFI bit or flag in a layer two header. If the CFI bit is set, then the packet is not suitable for the processing enhancements described in other sections and the parsing procedure ends by calling instruction DONE (e.g., instruction eighteen). If the CFI bit is not set, then the pointer is incremented another couple of bytes and execution continues with instruction 802.3. As explained above, a null output operation (e.g., "NONE") indicates that no output operation is performed. In addition, the output enabler value (e.g., zero) further ensures that no output operation is performed.

In instruction 802.3 (e.g., instruction three), a Type or Length field (depending on the location of the pointer and format of the packet) is examined to determine if the packet's layer two format is traditional Ethernet or 802.3 Ethernet. If the value in the header field appears to indicate 802.3 Ethernet (e.g., contains a hexadecimal value less than 0600), the pointer is incremented two bytes (to what should be an LLC SNAP field) and execution continues with instruction LLC_1. Otherwise, the layer two protocol may be considered traditional Ethernet and execution continues with instruction IPV4_1. Instruction 802.3 in this embodiment of the invention does not include an output operation. In instructions LLC_1 and LLC_2 (e.g., instructions four and five), a suspected layer two LLC SNAP field is examined to ensure that the packet conforms to the 802.3 Ethernet protocol. In instruction LLC_1, a first part of the field is tested and, if successful, the pointer is incremented two bytes and a second part is tested in instruction LLC_2. If instruction LLC_2 succeeds, the parsing pointer is advanced four bytes to reach what should be a Type field and execution continues with instruction IPV4_1. If either test fails, however, the parsing procedure exits. In the illustrated embodiment of the invention, no output operation is performed while testing the LLC SNAP field.

In instruction IPV4_1 (e.g., instruction six), the parsing pointer should be at an Ethernet Type field. This field is examined to determine if the layer three protocol appears to correspond to version four of the Internet Protocol. If this test is successful (e.g., the Type field contains a hexadecimal value of 0800), the pointer is advanced two bytes to the beginning of the layer three header and execution of program **2300** continues with instruction IPV4_2. If the test is unsuccessful, then execution continues with instruction IPV6_1. Regardless of the test results, the operation enabler value (e.g., three) indicates that the specified LD_SAP output operation is always performed.

As described previously, in a LD_SAP operation a packet's ethertype (or Service Access Point) is stored in a register. Part of the operation argument of 0x100, in particular the right-most six bits (e.g., zero) constitute an offset to a two-byte value comprising the ethertype. The offset in this example is zero because, in the present context, the parsing pointer is already at the Type field that contains the ethertype. In the presently described embodiment, the remainder of the operation argument constitutes a flag specifying that the starting position of the layer three header (e.g., an offset from the beginning of the packet) is also to be saved (e.g., in the L3OFFSET register). In particular, the beginning of the layer three header is known to be located immediately after the two-byte Type field.

Instruction IPV4_2 (e.g., instruction seven) tests a suspected layer three version field to ensure that the layer three protocol is version four of IP. In particular, a specification for version four of IP specifies that the first four bits of the layer three header contain a value of 0x4. If the test fails, the parsing procedure ends with instruction DONE. If the test succeeds, the pointer advances six bytes and instruction IPV4_3 is called.

The specified LD_SUM operation, which is only performed if the comparison in instruction IPV4_2 succeeds, indicates that an offset to the beginning of a point from which a checksum may be calculated should be stored. In particular, in the presently described embodiment of the invention a checksum should be calculated from the beginning of the TCP header (assuming that the layer four header is TCP). The value of the operation argument (e.g., 0x00A) indicates that the checksum is located twenty bytes (e.g., ten two-byte increments) from the current pointer. Thus, a value of twenty bytes is added to the parsing pointer position and the result is stored in a register or other data structure (e.g., the CSUMSTART register).

Instruction IPV4_3 (e.g., instruction eight) is designed to determine whether the packet's IP header indicates IP fragmentation. If the value extracted from the header in accordance with the extraction mask does not equal the comparison value, then the packet indicates fragmentation. If fragmentation is detected, the packet is considered unsuitable for the processing enhancements described in other sections and the procedure exits (e.g., through instruction DONE). Otherwise, the pointer is incremented two bytes and instruction IPV4_4 is called after performing a LD_LEN operation.

In accordance with the LD_LEN operation, the length of the IP segment is saved. The illustrated operation argument

(e.g., 0x03E) comprises an offset to the Total Length field where this value is located. In particular, the least-significant six bits constitute the offset. Because the pointer has already been advanced past this field, the operation argument comprises a negative value. One skilled in the art will recognize that this binary value (e.g., 111110) may be used to represent the decimal value of negative two. Thus, the present offset of the pointer, minus four bytes (e.g., two two-byte units), is saved in a register or other data structure (e.g., the PAYLOADLEN register). Any other suitable method of representing a negative offset may be used. Or, the IP segment length may be saved while the pointer is at a location preceding the Total Length field (e.g., during a previous instruction).

In instruction IPV4_4 (e.g., instruction nine), a one-byte Protocol field is examined to determine whether the layer four protocol appears to be TCP. If so, the pointer is advanced fourteen bytes and execution continues with instruction TCP_1; otherwise the procedure ends.

The specified LD_FID operation, which is only performed when the comparison in instruction IPV4_4 succeeds, involves retrieving the packet's flow key and storing it in a register or other location (e.g., the FLOWID register). One skilled in the art will appreciate that in order for the comparison in instruction IPV4_4 to be successful, the packet's layer three and four headers must conform to IP (version four) and TCP, respectively. If so, then the entire flow key (e.g., IP source and destination addresses plus TCP source and destination port numbers) is stored contiguously in the packet's header portion. In particular, the flow key comprises the last portion of the IP header and the initial portion of the TCP header and may be extracted in one operation. The operation argument (e.g., 0x182) thus comprises two values needed to locate and delimit the flow key. Illustratively, the right-most six bits of the argument (e.g., 0x02) identify an offset from the pointer position, in two-byte units, to the beginning of the flow key. The other five bits of the argument (e.g., 0x06) identify the size of the flow key, in two-byte units, to be stored.

In instruction IPV6_1 (e.g., instruction ten), which follows the failure of the comparison performed by instruction IPV4_1, the parsing pointer should be at a layer two Type field. If this test is successful (e.g., the Type field holds a hexadecimal value of 86DD), instruction IPV6_2 is executed after a LD_SUM operation is performed and the pointer is incremented two bytes to the beginning of the layer three protocol. If the test is unsuccessful, the procedure ends.

The indicated LD_SUM operation in instruction IPV6_1 is similar to the operation conducted in instruction IPV4_2 and utilizes a different argument. Again, the checksum is to be calculated from the beginning of the TCP header assuming the layer four header is TCP). The specified operation argument (e.g., 0x015) thus comprises an offset to the beginning of the TCP header—twenty—one two-byte unit ahead. The indicated offset is added to the present pointer position and saved in a register or other data structure (e.g., the CSUMSTART register).

Instruction IPV6_2 (e.g., instruction eleven) tests a suspected layer three version field to further ensure that the layer three protocol is version six of IP. If the comparison succeeds, the parsing procedure ends with the invocation of instruction DONE. If it succeeds, instruction IPV6_3 is executed. Operation IM_R1, which is performed only when the comparison succeeds in this embodiment, saves the Length field of the IP header from a Payload Length field. As one skilled in the art will appreciate, the Total Length field (e.g. IP segment size) of an IP, version four, header includes the size of the version four header. However, the Payload Length field (e.g., IP segment size) of an IP, version six, header does not include the size of the version six header. Thus, the size of the version six header, which is identified by the right-most eight bits of the output argument (e.g., 0x14, indicating twenty two-byte units) is saved. Illustratively, the remainder of the argument identifies the data structure in which to store the header length (e.g., temporary register R1). Because of the variation in size of layer three headers between protocols, in one embodiment of the invention the header size is indicated in different units to allow greater precision. In particular, in one embodiment of the invention the size of the header is specified in bytes in instruction IPV6_2, in which case the output argument could be 0x128.

Instruction IPV6_3 (e.g., instruction twelve) in this embodiment does not examine a header value. In this embodiment, the combination of an extraction mask of 0x0000 with a comparison value of 0x0000 indicates that an output operation is desired before the next examination of a portion of a header. After the LD_FID operation is performed, the parsing pointer is advanced six bytes to a Next Header field of the version six IP header. Because the extraction mask and comparison values are both 0x0000, the comparison should never fail and the failure branch of instruction should never be invoked.

As described previously, a LD_FID operation stores a flow key in an appropriate register or other data structure (e.g., the FLOWID register). Illustratively, the operation argument of 0x484 comprises two values for identifying and delimiting the flow key. In particular, the right-most six bits (e.g., 0x04) indicates that the flow key portion is located at an offset of eight bytes (e.g., four two-byte increments) from the current pointer position. The remainder of the operation argument (e.g., 0x12) indicates that thirty-six bytes (e.g., the decimal equivalent of 0x12 two-byte units) are to be copied from the computed offset. In the illustrated embodiment of the invention the entire flow key is copied intact, including the layer three source and destination addresses and layer four source and destination ports.

In instruction IPV6_4 (e.g., instruction thirteen), a suspected Next Header field is examined to determine whether the layer four protocol of the packet's protocol stack appears to be TCP. If so, the procedure advances thirty-six bytes (e.g., eighteen two-byte units) and instruction TCP_L is called; otherwise the procedure exits (e.g., through instruction DONE). Operation LD_LEN is performed if the value in the Next Header field is 0x06. As described above, this operation stores the IP segment size. Once again the argument (e.g., 0x03F) comprises a negative offset, in this case negative one. This offset indicates that the desired Payload Length field is located two bytes before the pointer's present position. Thus, the negative offset is added to the present pointer offset and the result saved in an appropriate register or other data structure (e.g., the PAYLOADLEN register).

In instructions TCP_1, TCP_2, TCP_3 and TCP_4 (e.g., instructions fourteen through seventeen), no header values—other than certain flags specified in the instruction's output operations—are examined, but various data from the packet's TCP header are saved. In the illustrated embodiment, the data that is saved includes a TCP sequence number, a TCP header length and one or more flags. For each instruction, the specified operation is performed and the next instruction is called. As described above, a comparison between the comparison value of 0x0000 and a null extraction value, as used in each of these instructions, will never

fail. After instruction TCP_4, the parsing procedure returns to instruction WAIT to await a new packet.

For operation LD_SEQ in instruction TCP_1, the operation argument (e.g., 0x081) comprises two values to identify and extract a TCP sequence number. The right-most six bits (e.g., 0x01) indicate that the sequence number is located two bytes from the pointer's current position. The rest of the argument (e.g., 0x2) indicates the number of two-byte units that must be copied from that position in order to capture the sequence number. Illustratively, the sequence number is stored in the SEQNO register.

For operation ST_FLAG in instruction TCP_2, the operation argument (e.g., 0x145) is used to configure a register (e.g., the FLAGS register) with flags to be used in a post-parsing task. The right-most six bits (e.g., 0x05) constitute an offset, in two-byte units, to a two-byte portion of the TCP header that contains flags that may affect whether the packet is suitable for post-parsing enhancements described in other sections. For example, URG, PSH, RST, SYN and FIN flags may be located at the offset position and be used to configure the register. The output mask (e.g., 0x002F) indicates that only particular portions (e.g., bits) of the TCP header's Flags field are stored.

Operation LD_R1 of instruction TCP_3 is similar to the operation conducted in instruction IPV6_2. Here, an operation argument of 0x205 includes a value (e.g., the least-significant six bits) identifying an offset of five two-byte units from the current pointer position. That location should include a Header Length field to be stored in a data structure identified by the remainder of the argument (e.g., temporary register R1). The output mask (e.g., 0xF000) indicates that only the first four bits are saved (e.g., the Header Length field is only four bits in size).

As one skilled in the art may recognize, the value extracted from the Header Length field may need to be adjusted in order to reflect the use of two-byte units (e.g., sixteen bit words) in the illustrated embodiment. Therefore, in accordance with the shift portion of instruction TCP_3, the value extracted from the field and configured by the output mask (e.g., 0xF000) is shifted to the right eleven positions when stored in order to simplify calculations.

Operation LD_HDR of instruction TCP_4 causes the loading of an offset to the first byte of packet data following the TCP header. As described in a later section, packets that are compatible with a pre-selected protocol stack may be separated at some point into header and data portions. Saving an offset to the data portion now makes it easier to split the packet later. Illustratively, the right-most seven bits of the 0x0FF operation argument comprise a first element of the offset to the data. One skilled in the art will recognize the bit pattern (e.g., 1111111) as equating to negative one. Thus, the offset value equal to the current parsing pointer (e.g., the value in the ADDR register) minus two bytes—which indicates the beginning of the TCP header—is saved. The remainder of the argument signifies that the value of a temporary data structure (e.g., temporary register R1) is to be added to this offset. In this particular context, the value saved in the previous instruction (e.g., the length of the TCP header) is added. These two values combine to form an offset to the beginning of the packet data, which is stored in the appropriate register or other data structure (e.g., the PRSPLIT register).

Finally, and as mentioned above, instruction DONE (e.g., instruction eighteen) indicates the end of parsing of a packet when it is determined that the packet does not conform to one or more of the protocols associated with the illustrated sections. This may be considered a "clean-up" instruction. In particular, output operation LD_CTL, with an operation argument of 0x001 indicates that a No_Assist flag is to be set (e.g., to one) in the control register described above in conjunction with instruction VLAN. The No_Assist flag, as described elsewhere, may be used to inform other modules of the network interface that the present packet, is unsuitable for one or more processing enhancements described elsewhere.

It will be recognized by one skilled in the art that the illustrated program or microcode merely provides one method of parsing a packet. Other programs, comprising the same instructions in a different sequence or different instructions altogether, with similar or dissimilar formats, may be employed to examine and store portions of headers and to configure registers and other data structures.

The efficiency gains to be realized from the application of the enhanced processing described in following sections more than offset the time required to parse a packet with the illustrated program. Further, even though a header parser parses a packet on a NIC in a current embodiment of the invention, the packet may still need to be processed through its protocol stack (e.g., to remove the protocol headers) by a processor on a host computer. Doing so avoids burdening the communication device (e.g., network interface) with such a task.

One Embodiment of a Flow Database

FIG. **5** depicts flow database (FDB) **110** according to one embodiment of the invention. Illustratively FDB **110** is implemented as a CAM (Content Addressable Memory) using a re-writeable memory component (e.g., RAM, SRAM, DRAM). In this embodiment, FDB **110** comprises associative portion **502** and associated portion **504**, and may be indexed by flow number **506**.

The scope of the invention does not limit the form or structure of flow database **110**. In alternative embodiments of the invention virtually any form of data structure may be employed (e.g., database, table, queue, list, array), either monolithic or segmented, and may be implemented in hardware or software. The illustrated form of FDB **110** is merely one manner of maintaining useful information concerning communication flows through NIC **100**. As one skilled in the art will recognize, the structure of a CAM allows highly efficient and fast associative searching.

In the illustrated embodiment of the invention, the information stored in FDB **110** and the operation of flow database manager (FDBM) **108** (described below) permit functions such as data re-assembly, batch processing of packet headers, and other enhancements. These functions are discussed in detail in other sections but may be briefly described as follows.

One form of data re-assembly involves the re-assembly or combination of data from multiple related packets (e.g., packets from a single communication flow or a single datagram). One method for the batch processing of packet headers entails processing protocol headers from multiple related packets through a protocol stack collectively rather than one packet at a time. Another illustrative function of NIC **100** involves the distribution or sharing of such protocol stack processing (and/or other functions) among processors in a multi-processor host computer system. Yet another possible function of NIC **100** is to enable the transfer of re-assembled data to a destination entity (e.g., an application program) in an efficient aggregation (e.g., a memory page), thereby avoiding piecemeal and highly inefficient transfers of one packet's data at a time. Thus, in this embodiment of the invention, one purpose of FDB **110** and FDBM **108** is to generate information for the use of NIC **100** and/or a host

NOAC Ex. 1015 Page 530

computer system in enabling, disabling or performing one or more of these functions.

Associative portion **502** of FDB **110** in FIG. **5** stores the flow key of each valid flow destined for an entity served by NIC **100**. Thus, in one embodiment of the invention associative portion **502** includes IP source address **510**, IP destination address **512**, TCP source port **514** and TCP destination port **516**. As described in a previous section these fields may be extracted from a packet and provided to FDBM **108** by header parser **106**.

Although each destination entity served by NIC **100** may participate in multiple communication flows or end-to-end TCP connections, only one flow at a time will exist between a particular source entity and a particular destination entity. Therefore, each flow key in associative portion **502** that corresponds to a valid flow should be unique from all other valid flows. In alternative embodiments of the invention, associative portion **502** is composed of different fields, reflecting alternative flow key forms, which may be determined by the protocols parsed by the header parser and the information used to identify communication flows.

Associated portion **504** in the illustrated embodiment comprises flow validity indicator **520**, flow sequence number **522** and flow activity indicator **524**. These fields provide information concerning the flow identified by the flow key stored in the corresponding entry in associative portion **502**. The fields of associated portion **504** may be retrieved and/or updated by FDBM **108** as described in the following section.

Flow validity indicator **520** in this embodiment indicates whether the associated flow is valid or invalid. Illustratively, the flow validity indicator is set to indicate a valid flow when the first packet of data in a flow is received, and may be reset to reassert a flow's validity every time a portion of a flow's datagram (e.g., a packet) is correctly received.

Flow validity indicator **520** may be marked invalid after the last packet of data in a flow is received. The flow validity indicator may also be set to indicate an invalid flow whenever a flow is to be torn down (e.g., terminated or aborted) for some reason other than the receipt of a final data packet. For example, a packet may be received out of order from other packets of a datagram, a control packet indicating that a data transfer or flow is being aborted may be received, an attempt may be made to re-establish or re-synchronize a flow (in which case the original flow is terminated), etc. In one embodiment of the invention flow validity indicator **520** is a single bit, flag or value.

Flow sequence number **522** in the illustrated embodiment comprises a sequence number of the next portion of data that is expected in the associated flow. Because the datagram being sent in a flow is typically received via multiple packets, the flow sequence number provides a mechanism to ensure that the packets are received in the correct order. For example, in one embodiment of the invention NIC **100** re-assembles data from multiple packets of a datagram. To perform this re-assembly in the most efficient manner, the packets need to be received in order. Thus, flow sequence number **522** stores an identifier to identify the next packet or portion of data that should be received.

In one embodiment of the invention, flow sequence number **522** corresponds to the TCP sequence number field found in TCP protocol headers. As one skilled in the art will recognize, a packet's TCP sequence number identifies the position of the packet's data relative to other data being sent in a datagram. For packets and flows involving protocols other than TCP, an alternative method of verifying or ensuring the receipt of data in the correct order may be employed.

Flow activity indicator **524** in the illustrated embodiment reflects the recency of activity of a flow or, in other words, the age of a flow. In this embodiment of the invention flow activity indicator **524** is associated with a counter, such as a flow activity counter (not depicted in FIG. **5**). The flow activity counter is updated (e.g., incremented) each time a packet is received as part of a flow that is already stored in flow database **110**. The updated counter value is then stored in the flow activity indicator field of the packet's flow. The flow activity counter may also be incremented each time a first packet of a new flow that is being added to the database is received. In an alternative embodiment, a flow activity counter is only updated for packets containing data (e.g., it is not updated for control packets). In yet another alternative embodiment, multiple counters are used for updating flow activity indicators of different flows.

Because it can not always be determined when a communication flow has ended (e.g., the final packet may have been lost), the flow activity indicator may be used to identify flows that are obsolete or that should be torn down for some other reason. For example, if flow database **110** appears to be fully populated. (e.g., flow validity indicator **520** is set for each flow number) when the first packet of a new flow is received, the flow having the lowest flow activity indicator may be replaced by the new flow.

In the illustrated embodiment of the invention, the size of fields in FDB **110** may differ from one entry to another. For example, IP source and destination addresses are four bytes large in version four of the protocol, but are sixteen bytes large in version six. In one alternative embodiment of the invention, entries for a particular field may be uniform in size, with smaller entries being padded as necessary.

In another alternative embodiment of the invention, fields within FDB **110** may be merged. In particular, a flow's flow key may be stored as a single entity or field instead of being stored as a number of separate fields as shown in FIG. **5**. Similarly, flow validity indicator **520**, flow sequence number **522** and flow activity indicator **524** are depicted as separate entries in FIG. **5**. However, in an alternative embodiment of the invention one or more of these entries may be combined. In particular, in one alternative embodiment flow validity indicator **520** and flow activity indicator **524** comprise a single entry having a first value (e.g., zero) when the entry's associated flow is invalid. As long as the flow is valid, however, the combined entry is incremented as packets are received, and is reset to the first value upon termination of the flow.

In one embodiment of the invention FDB **110** contains a maximum of sixty-four entries, indexed by flow number **506**, thus allowing the database to track sixty-four valid flows at a time. In alternative embodiments of the invention, more or fewer entries may be permitted, depending upon the size of memory allocated for flow database **110**. In addition to flow number **506**, a flow may be identifiable by its flow key (stored in associative portion **502**).

In the illustrated embodiment of the invention, flow database **110** is empty (e.g., all fields are filled with zeros) when NIC **100** is initialized. When the first packet of a flow is received header parser **106** parses a header portion of the packet. As described in a previous section, the header parser assembles a flow key to identify the flow and extracts other information concerning the packet and/or the flow. The flow key, and other information, is passed to flow database manager **108**. FDBM **108** then searches FDB **110** for an active flow associated with the flow key. Because the database is empty, there is no match.

In this example, the flow key is therefore stored (e.g., as flow number zero) by copying the IP source address, IP

destination address, TCP source port and TCP destination port into the corresponding fields. Flow validity indicator **520** is then set to indicate a valid flow, flow sequence number **522** is derived from the TCP sequence number (illustratively provided by the header parser), and flow activity indicator **524** is set to an initial value (e.g., one), which may be derived from a counter. One method of generating an appropriate flow sequence number, which may be used to verify that the next portion of data received for the flow is received in order, is to add the TCP sequence number and the size of the packet's data. Depending upon the configuration of the packet (e.g., whether the SYN bit in a Flags field of the packet's TCP header is set), however, the sum may need to be adjusted (e.g., by adding one) to correctly identify the next expected portion of data.

As described above, one method of generating an appropriate initial value for a flow activity indicator is to copy a counter value that is incremented for each packet received as part of a flow. For example, for the first packet received after NIC **100** is initialized, a flow activity counter may be incremented to the value of one. This value may then be stored in flow activity indicator **524** for the associated flow. The next packet received as part of the same (or a new) flow causes the counter to be incremented to two, which value is stored in the flow activity indicator for the associated flow. In this example, no two flows should have the same flow activity indicator except at initialization, when they may all equal zero or some other predetermined value.

Upon receipt and parsing of a later packet received at NIC **100**, the flow database is searched for a valid flow matching that packet's flow key. Illustratively, only the flow keys of active flows (e.g., those flows for which flow validity indicator **520** is set) are searched. Alternatively, all flow keys (e.g., all entries in associative portion **502**) may be searched but a match is only reported if its flow validity indicator indicates a valid flow. With a CAM such as FDB **110** in FIG. 5, flow keys and flow validity indicators may be searched in parallel.

If a later packet contains the next portion of data for a previous flow (e.g., flow number zero), that flow is updated appropriately. In one embodiment of the invention this entails updating flow sequence number **522** and incrementing flow activity indicator **524** to reflect its recent activity. Flow validity indicator **520** may also be set to indicate the validity of the flow, although it should already indicate that the flow is valid.

As new flows are identified, they are added to FDB **110** in a similar manner to the first flow. When a flow is terminated or torn down, the associated entry in FDB **110** is invalidated. In one embodiment of the invention, flow validity indicator **520** is merely cleared (e.g., set to zero) for the terminated flow. In another embodiment, one or more fields of a terminated flow are cleared or set to an arbitrary predetermined value. Because of the bursty nature of network packet traffic, all or most of the data from a datagram is generally received in a short amount of time. Thus, each valid flow in FDB **110** normally only needs to be maintained for a short period of time, and its entry can then be used to store a different flow.

Due to the limited amount of memory available for flow database **110** in one embodiment of the invention, the size of each field may be limited. In this embodiment, sixteen bytes are allocated for IP source address **510** and sixteen bytes are allocated for IP destination address **512**. For IP addresses greater than sixteen bytes in length, the extra space may be padded with zeros. Further, TCP source port **514** and TCP destination port **516** are each allocated two bytes. Also in this embodiment, flow validity indicator **520** comprises one bit, flow sequence number **522** is allocated four bytes and flow activity indicator **524** is also allocated four bytes.

As one skilled in the art will recognize from the embodiments described above, a flow is similar, but not identical, to an end-to-end TCP connection. A TCP connection may exist for a relatively extended period of time, sufficient to transfer multiple datagrams from a source entity to a destination entity. A flow, however, may exist only for one datagram. Thus, during one end-to-end TCP connection, multiple flows may be set up and torn down (e.g., once for each datagram). As described above, a flow may be set up (e.g., added to FDB **110** and marked valid) when NIC **100** detects the first portion of data in a datagram and may be torn down (e.g., marked invalid in FDB **110**) when the last portion of data is received. Illustratively, each flow set up during a single end-to-end TCP connection will have the same flow key because the layer three and layer four address and port identifiers used to form the flow key will remain the same.

In the illustrated embodiment, the size of flow database **110** (e.g., the number of flow entries) determines the maximum number of flows that may be interleaved (e.g., simultaneously active) at one time while enabling the functions of data re-assembly and batch processing of protocol headers. In other words, in the embodiment depicted in FIG. 5, NIC **100** can set up sixty-four flows and receive packets from up to sixty-four different datagrams (i.e., sixty-four flows may be active) without tearing down a flow. If a maximum number of flows through NIC **100** were known, flow database **110** could be limited to the corresponding number of entries.

The flow database may be kept small because a flow only lasts for one datagram in the presently described embodiment and, because of the bursty nature of packet traffic, a datagram's packets are generally received in a short period of time. The short duration of a flow compensates for a limited number of entries in the flow database. In one embodiment of the invention, if FDB **110** is filled with active flows and a new flow is commenced (i.e., a first portion of data in a new datagram), the oldest (e.g., the least recently active) flow is replaced by the new one.

In an alternative embodiment of the invention, flows may be kept active for any number of datagrams (or other measure of network traffic) or for a specified length or range of time. For example, when one datagram ends its flow in FDB **110** may be kept "open" (i.e., not torn down) if the database is not full (e.g., the flow's entry is not needed for a different flow). This scheme may further enhance the efficient operation of NIC **100** if another datagram having the same flow key is received. In particular, the overhead involved in setting up another flow is avoided and more data re-assembly and packet batching (as described below) may be performed. Advantageously, a flow may be kept open in flow database **110** until the end-to-end TCP connection that encompasses the flow ends.

One Embodiment of a Flow Database Manager

FIGS. **6A–6E** depict one method of operating a flow database manager (FDBM), such as flow database manager **108** of FIG. **1A**, for managing flow database (FDB) **110**. Illustratively, FDBM **108** stores and updates flow information stored in flow database **110** and generates an operation code for a packet received by NIC **100**. FDBM **108** also tears down a flow (e.g., replaces, removes or otherwise invalidates an entry in FDB **10**) when the flow is terminated or aborted.

In one embodiment of the invention a packet's operation code reflects the packet's compatibility with pre determined

NOAC Ex. 1015 Page 532

criteria for performing one or more functions of NIC **100** (e.g., data re-assembly, batch processing of packet headers, load distribution). In other words, depending upon a packet's operation code, other modules of NIC **100** may or may not perform one of these functions, as described in following sections.

In another embodiment of the invention, an operation code indicates a packet status. For example, an operation code may indicate that a packet: contains no data, is a control packet, contains more than a specified amount of data, is the first packet of a new flow, is the last packet of an existing flow, is out of order, contains a certain flag (e.g., in a protocol header) that does not have an expected value (thus possibly indicating an exceptional circumstance), etc.

The operation of flow database manager **108** depends upon packet information provided by header parser **106** and data drawn from flow database **110**. After FDBM **108** processes the packet information and/or data, control information (e.g., the packet's operation code) is stored in control queue **118** and FDB **110** may be altered (e.g., a new flow may be entered or an existing one updated or torn down).

With reference now to FIGS. **6A–6E**, state **600** is a start state in which FDBM **108** awaits information drawn from a packet received by NIC **100** from network **102**. In state **602**, header parser **106** or another module of NIC **100** notifies FDBM **108** of a new packet by providing the packet's flow key and some control information. Receipt of this data may be interpreted as a request to search FDB **110** to determine whether a flow having this flow key already exists.

In one embodiment of the invention the control information passed to FDBM **108** includes a sequence number (e.g., a TCP sequence number) drawn from a packet header. The control information may also indicate the status of certain flags in the packet's headers, whether the packet includes data and, if so, whether the amount of data exceeds a certain size. In this embodiment, FDBM **108** also receives a No_Assist signal for a packet if the header parser determines that the packet is not formatted according to one of the pre-selected protocol stacks (i.e., the packet is not "compatible"), as discussed in a previous section. Illustratively, the No_Assist signal indicates that one or more functions of NIC **100** (e.g., data re-assembly, batch processing, load-balancing) may not be provided for the packet.

In state **604**, FDBM **108** determines whether a No_Assist signal was asserted for the packet. If so, the procedure proceeds to state **668** (FIG. **6E**). Otherwise, FDBM **108** searches FDB **110** for the packet's flow key in state **606**. In one embodiment of the invention only valid flow entries in the flow database are searched. As discussed above, a flow's validity may be reflected by a validity indicator such as flow validity indicator **520** (shown in FIG. **5**). If, in state **608**, it is determined that the packet's flow key was not found in the database, or that a match was found but the associated flow is not valid, the procedure advances to state **646** (FIG. **6D**). If a valid match is found in the flow database, in state **610** the flow number (e.g., the flow database index for the matching entry) of the matching flow is noted and flow information stored in FDB **110** is read. Illustratively, this information includes flow validity indicator **520**, flow sequence number **522** and flow activity indicator **524** (shown in FIG. **5**).

In state **612**, FDBM **108** determines from information received from header parser **106** whether the packet contains payload data. If not, the illustrated procedure proceeds to state **638** (FIG. **6C**); otherwise the procedure continues to state **614**.

In state **614**, the flow database manager determines whether the packet constitutes an attempt to reset a communication connection or flow. Illustratively, this may be determined by examining the state of a SYN bit in one of the packet's protocol headers (e.g., a TCP header). In one embodiment of the invention the value of one or more control or flag bits (such as the SYN bit) are provided to the FDBM by the header parser. As one skilled in the art will recognize, one TCP entity may attempt to reset a communication flow or connection with another entity (e.g., because of a problem on one of the entity's host computers) and send a first portion of data along with the re-connection request. This is the situation the flow database manager attempts to discern in state **614**. If the packet is part of an attempt to re-connect or reset a flow or connection, the procedure continues at state **630** (FIG. **6C**).

In state **616**, flow database manager **108** compares a sequence number (e.g., a TCP sequence number) extracted from a packet header with a sequence number (e.g., flow sequence number **522** of FIG. **5**) of the next expected portion of data for this flow. As discussed in a previous section, these sequence numbers should correlate if the packet contains the flow's next portion of data. If the sequence numbers do not match, the procedure continues at state **628**.

In state **618**, FDBM **108** determines whether certain flags extracted from one or more of the packet's protocol headers match expected values. For example, in one embodiment of the invention the URG, PSH, RST and FIN flags from the packet's TCP header are expected to be clear (i.e., equal to zero). If any of these flags are set (e.g., equal to one) an exceptional condition may exist, thus making it possible that one or more of the functions (e.g., data re-assembly, batch processing, load distribution) offered by NIC **100** should not be performed for this packet. As long as the flags are clear, the procedure continues at state **620**; otherwise the procedure continues at state **626**.

In state **620**, the flow database manager determines whether more data is expected during this flow. As discussed above, a flow may be limited in duration to a single datagram. Therefore, in state **620** the FDBM determines if this packet appears to be the final portion of data for this flow's datagram. Illustratively, this determination is made on the basis of the amount of data included with the present packet. As one skilled in the art will appreciate, a datagram comprising more data than can be carried in one packet is sent via multiple packets. The typical manner of disseminating a datagram among multiple packets is to put as much data as possible into each packet. Thus, each packet except the last is usually equal or nearly equal in size to the maximum transfer unit (MTU) allowed for the network over which the packets are sent. The last packet will hold the remainder, usually causing it to be smaller than the MTU.

Therefore, one manner of identifying the final portion of data in a flow's datagram is to examine the size of each packet and compare it to a figure (e.g., MTU) that a packet is expected to exceed except when carrying the last data portion. It was described above that control information is received by FDBM **108** from header parser **106**. An indication of the size of the data carried by a packet may be included in this information. In particular, header parser **106** in one embodiment of the invention is configured to compare the size of each packet's data portion to a pre-selected value. In one embodiment of the invention this value is programmable. This value is set, in the illustrated embodiment of the invention, to the maximum amount of data a packet can carry without exceeding MTU. In one alternative embodiment, the value is set to an amount somewhat less than the maximum amount of data that can be carried.

Thus, in state **620**, flow database manager **108** determines whether the received packet appears to carry the final portion of data for the flow's datagram. If not, the procedure continues to state **626**.

In state **622**, it has been ascertained that the packet is compatible with pre-selected protocols and is suitable for one or more functions offered by NIC **100**. In particular, the packet has been formatted appropriately for one or more of the functions discussed above. FDBM **108** has determined that the received packet is part of an existing flow, is compatible with the pre-selected protocols and contains the next portion of data for the flow (but not the final portion). Further, the packet is not part of an attempt to re-set a flow/connection, and important flags have their expected values. Thus, flow database **110** can be updated as follows.

The activity indicator (e.g., flow activity indicator **524** of FIG. **5**) for this flow is modified to reflect the recent flow activity. In one embodiment of the invention flow activity indicator **524** is implemented as a counter, or is associated with a counter, that is incremented each time data is received for a flow. In another embodiment of the invention, an activity indicator or counter is updated every time a packet having a flow key matching a valid flow (e.g., whether or not the packet includes data) is received.

In the illustrated embodiment, after a flow activity indicator or counter is incremented it is examined to determine if it "rolled over" to zero (i.e., whether it was incremented past its maximum value). If so, the counter and/or the flow activity indicators for each entry in flow database **110** are set to zero and the current flow's activity indicator is once again incremented. Thus, in one embodiment of the invention the rolling over of a flow activity counter or indicator causes the re-initialization of the flow activity mechanism for flow database **110**. Thereafter, the counter is incremented and the flow activity indicators are again updated as described previously. One skilled in the art will recognize that there are many other suitable methods that may be applied in an embodiment of the present invention to indicate that one flow was active more recently than another was.

Also in state **622**, flow sequence number **522** is updated. Illustratively, the new flow sequence number is determined by adding the size of the newly received data to the existing flow sequence number. Depending upon the configuration of the packet (e.g., values in its headers), this sum may need to be adjusted. For example, this sum may indicate simply the total amount of data received thus far for the flow's datagram. Therefore, a value may need to be added (e.g., one byte) in order to indicate a sequence number of the next byte of data for the datagram. As one skilled in the art will recognize, other suitable methods of ensuring that data is received in order may be used in place of the scheme described here.

Finally, in state **622** in one embodiment of the invention, flow validity indicator **520** is set or reset to indicate the flow's validity.

Then, in state **624**, an operation code is associated with the packet. In the illustrated embodiment of the invention, operation codes comprise codes generated by flow database manager **108** and stored in control queue **118**. In this embodiment, an operation code is three bits in size, thus allowing for eight operation codes. Operation codes may take a variety of other forms and ranges in alternative embodiments. For the illustrated embodiment of the invention, TABLE 1 describes each operation code in terms of the criteria that lead to each code's selection and the qualifications of that selection. For purposes of TABLE 1, setting up a flow.comprises inserting a flow into flow database **110**. Tearing down a flow comprises removing or invalidating a flow in flow database **110**. The re-assembly of data is discussed in a following section describing DMA engine **120**.

In the illustrated embodiment of the invention, operation code **4** is selected in state **624** for packets in the present context of the procedure (e.g., compatible packets carrying the next, but not last, data portion of a flow). Thus, the existing flow is not torn down and there is no need to set up a new flow. As described above, a compatible packet in this embodiment is a packet conforming to one or more of the pre-selected protocols. By changing or augmenting the pre-selected protocols, virtually any packet may be compatible in an alternative embodiment of the invention.

Returning now to FIGS. **6A–6E**, after state **624** the illustrated procedure ends at state **670**.

In state **626** (reached from state **618** or state **620**), operation code **3** is selected for the packet. Illustratively, operation code **3** indicates that the packet is compatible and matches a valid flow (e.g., the packet's flow key matches the flow key of a valid flow in FDB **110**). Operation code **3** may also signify that the packet contains data, does not constitute an attempt to re-synchronize or reset a communication flow/connection and the packet's sequence number matches the expected sequence number (from flow database **110**). But, either an important flag (e.g., one of the TCP flags URG, PSH, RST or FIN) is set (determined in state **618**) or the packet's data is less than the threshold value described above (in state **620**), thus indicating that no more data is likely to follow this packet in this flow. Therefore, the existing flow is torn down but no new flow is created. Illustratively, the flow may be torn down by clearing the flow's validity indicator (e.g., setting it to zero). After state **626**, the illustrated procedure ends at state **670**.

In state **628** (reached from state **616**), operation code **2** is selected for the packet. In the present context, operation code **2** may indicate that the packet is compatible, matches a valid flow (e.g., the packet's flow key matches the flow key of a valid flow in FDB **110**), contains data and does not constitute an attempt to re-synchronize or reset a communication flow/connection. However, the sequence number extracted from the packet (in state **616**) does not match the expected sequence number from flow database **110**. This may occur, for example, when a packet is received out of order. Thus, the existing flow is torn down but no new flow is established. Illustratively, the flow may be torn down by clearing the flow's validity indicator (e.g., setting it to zero). After state **628**, the illustrated procedure ends at state **670**.

State **630** is entered from state **614** when it is determined that the received packet constitutes an attempt to reset a communication flow or connection (e.g., the TCP SYN bit is set). In state **630**, flow database manager **108** determines whether more data is expected to follow. As explained in conjunction with state **620**, this determination may be made on the basis of control information received by the flow database manager from the header parser. If more data is expected (e.g., the amount of data in the packet equals or exceeds a threshold value), the procedure continues at state **634**.

In state **632**, operation code **2** is selected for the packet. Operation code **2** was also selected in state **628** in a different context. In the present context, operation code **2** may indicate that the packet is compatible, matches a valid flow and contains data. Operation code **2** may also signify in this context that the packet constitutes an attempt to re-synchronize or reset a communication flow or connection, but that no more data is expected once the flow/connection

is reset. Therefore, the existing flow is torn down and no new flow is established. Illustratively, the flow may be torn down by clearing the flow's validity indicator (e.g., setting it to zero). After state **632**, the illustrated procedure ends at state **670**.

In state **634**, flow database manager **108** responds to an attempt to reset or re-synchronize a communication flow/connection whereby additional data is expected. Thus, the existing flow is torn down and replaced as follows. The existing flow may be identified by the flow number retrieved in state **610** or by the packet's flow key. The flow's sequence number (e.g., flow sequence number **522** in FIG. **5**) is set to the next expected value. Illustratively, this value depends upon the sequence number (e.g., TCP sequence number) retrieved from the packet (e.g., by header parser **106**) and the amount of data included in the packet. In one embodiment of the invention these two values are added to determine a new flow sequence number. As discussed previously, this sum may need to be adjusted (e.g., by adding one). Also in state **634**, the flow activity indicator is updated (e.g., incremented). As explained in conjunction with state **622**, if the flow activity indicator rolls over, the activity indicators for all flows in the database are set to zero and the present flow is again incremented. Finally, the flow validity indicator is set to indicate that the flow is valid.

In state **636**, operation code **7** is selected for the packet. In the present context, operation code **7** indicates that the packet is compatible, matches a valid flow and contains data. Operation code **7** may further signify, in this context, that the packet constitutes an attempt to re-synchronize or reset a communication flow/connection and that additional data is expected once the flow/connection is reset. In effect, therefore, the existing flow is torn down and a new one (with the same flow key) is stored in its place. After state **636**, the illustrated procedure ends at end state **670**.

State **638** is entered after state **612** when it is determined that the received packet contains no data. This often indicates that the packet is a control packet. In state **638**, flow database manager **108** determines whether one or more flags extracted from the packet by the header parser match expected or desired values. For example, in one embodiment of the invention the TCP flags URG, PSH, RST and FIN must be clear in order for DMA engine **120** to re-assemble data from multiple related packets (e.g., packets having an identical flow key). As discussed above, the TCP SYN bit may also be examined. In the present context (e.g., a packet with no data), the SYN bit is also expected to be clear (e.g., to store a value of zero). If the flags (and SYN bit) have their expected values the procedure continues at state **642**. If, however, any of these flags are set, an exceptional condition may exist, thus making it possible that one or more functions offered by NIC **100** (e.g., data re-assembly, batch processing, load distribution) are unsuitable for this packet, in which case the procedure proceeds to state **640**.

In state **640**, operation code **1** is selected for the packet. Illustratively, operation code **1** indicates that the packet is compatible and matches a valid flow, but does not contain any data and one or more important flags or bits in the packet's header(s) are set. Thus, the existing flow is torn down and no new flow is established. Illustratively, the flow may be torn down by clearing the flow's validity indicator (e.g., setting it to zero). After state **640**, the illustrated procedure ends at end state **670**.

In state **642**, the flow's activity indicator is updated (e.g., incremented) even though the packet contains no data. As described above in conjunction with state **622**, if the activity indicator rolls over, in a present embodiment of the invention all flow activity indicators in the database are set to zero and the current flow is again incremented. The flow's validity indicator may also be reset, as well as the flow's sequence number.

In state **644**, operation code **0** is selected for the packet. Illustratively, operation code **0** indicates that the packet is compatible, matches a valid flow, and that the packet does not contain any data. The packet may, for example, be a control packet. Operation code **0** further indicates that none of the flags checked by header parser **106** and described above (e.g., URG, PSH, RST and FIN) are set. Thus, the existing flow is not torn down and no new flow is established. After state **644**, the illustrated procedure ends at end state **670**.

State **646** is entered from state **608** if the packet's flow key does not match any of the flow keys of valid flows in the flow database. In state **646**, FDBM **108** determines whether flow database **110** is full and may save some indication of whether the database is full. In one embodiment of the invention the flow database is considered full when the validity indicator (e.g., flow validity indicator **520** of FIG. **5**) is set for every flow number (e.g., for every flow in the database). If the database is full, the procedure continues at state **650**, otherwise it continues at state **648**.

In state **648**, the lowest flow number of an invalid flow (e.g., a flow for which the associated flow validity indicator is equal to zero) is determined. Illustratively, this flow number is where a new flow will be stored if the received packet warrants the creation of a new flow. After state **648**, the procedure continues at state **652**.

In state **650**, the flow number of the least recently active flow is determined. As discussed above, in the illustrated embodiment of the invention a flow's activity indicator (e.g., flow activity indicator **524** of FIG. **5**) is updated (e.g., incremented) each time data is received for a flow. Therefore, in this embodiment the least recently active flow can be identified as the flow having the least recently updated (e.g., lowest) flow activity indicator. Illustratively, if multiple flows have flow activity indicators set to a common value (e.g., zero), one flow number may be chosen from them at random or by some other criteria. After state **650**, the procedure continues at state **652**.

In state **652**, flow database manager **108** determines whether the packet contains data. Illustratively, the control information provided to FDBM **108** by the header parser indicates whether the packet has data. If the packet does not include data (e.g., the packet is a control packet), the illustrated procedure continues at state **668**.

In state **654**, flow database manager **108** determines whether the data received with the present packet appears to contain the final portion of data for the associated datagram/flow. As described in conjunction with state **620**, this determination may be made on the basis of the amount of data included with the packet. If the amount of data is less than a threshold value (a programmable value in the illustrated embodiment), then no more data is expected and this is likely to be the only data for this flow. In this case the procedure continues at state **668**. If, however, the data meets or exceeds the threshold value, in which case more data may be expected, the procedure proceeds to state **656**.

In state **656**, the values of certain flags are examined. These flags may include, for example, the URG, PSH, RST, FIN bits of a TCP header. If any of the examined flags do not have their expected or desired values (e.g., if any of the flags are set), an exceptional condition may exist making one or more of the functions of NIC **100** (e.g., data re-assembly, batch processing, load distribution) unsuitable for this

packet. In this case the procedure continues at state **668**; otherwise the procedure proceeds to state **658**.

In state **658**, the flow database manager retrieves the information stored in state **646** concerning whether flow database **110** is full. If the database is full, the procedure continues at state **664**; otherwise the procedure continues at state **660**.

In state **660**, a new flow is added to flow database **110** for the present packet. Illustratively, the new flow is stored at the flow number identified or retrieved in state **648**. The addition of a new flow may involve setting a sequence number (e.g., flow sequence number **522** from FIG. **5**). Flow sequence number **522** may be generated by adding a sequence number (e.g., TCP sequence number) retrieved from the packet and the amount of data included in the packet. As discussed above, this sum may need to be adjusted (e.g., by adding one).

Storing a new flow may also include initializing an activity indicator (e.g., flow activity indicator **524** of FIG. **5**). In one embodiment of the invention this initialization involves storing a value retrieved from a counter that is incremented each time data is received for a flow. Illustratively, if the counter or a flow activity indicator is incremented past its maximum storable value, the counter and all flow activity indicators are cleared or reset. Also in state **660**, a validity indicator (e.g., flow validity indicator **520** of FIG. **5**) is set to indicate that the flow is valid. Finally, the packet's flow key is also stored in the flow database, in the entry corresponding to the assigned flow number.

In state **662**, operation code **6** is selected for the packet. Illustratively, operation code **6** indicates that the packet is compatible, did not match any valid flows and contains the first portion of data for a new flow. Further, the packet's flags have their expected or necessary values, additional data is expected in the flow and the flow database is not full. Thus, operation code **6** indicates that there is no existing flow to tear down and that a new flow has been stored in the flow database. After state **662**, the illustrated procedure ends at state **670**.

In state **664**, an existing entry in the flow database is replaced so that a new flow, initiated by the present packet, can be stored. Therefore, the flow number of the least recently active flow, identified in state **650**, is retrieved. This flow may be replaced as follows. The sequence number of the existing flow (e.g., flow sequence number **522** of FIG. **5**) is replaced with a value derived by combining a sequence number extracted from the packet (e.g., TCP sequence number) with the size of the data portion of the packet. This sum may need to be adjusted (e.g., by adding one). Then the existing flow's activity indicator (e.g., flow activity indicator **524**) is replaced. For example, the value of a flow activity counter may be copied into the flow activity indicator, as discussed above. The flow's validity indicator (e.g., flow validity indicator **520** of FIG. **5**) is then set to indicate that the flow is valid. Finally, the flow key of the new flow is stored.

In state **666**, operation code **7** is selected for the packet. Operation code **7** was also selected in state **636**. In the present context, operation code **7** may indicate that the packet is compatible, did not match the flow key of any valid flows and contains the first portion of data for a new flow. However, the packet's flags have compatible values and additional data is expected in the flow. Lastly, however, in this context operation code **7** indicates that the flow database is full, so an existing entry was torn down and the new one stored in its place. After state **666**, the illustrated procedure ends at end state **670**.

In state **668**, operation code **5** is selected for the packet. State **668** is entered from various states and operation code **5** thus represents a variety of possible conditions or situations. For example, operation code **5** may be selected when a No_Assist signal is detected (in state **604**) for a packet. As discussed above, the No_Assist signal may indicate that the corresponding packet is not compatible with a set of pre-selected protocols. In this embodiment of the invention, incompatible packets are ineligible for one or more of the various functions of NIC **100** (e.g., data re-assembly, batch processing, load distribution).

State **668** may also be entered, and operation code **5** selected, from state **652**, in which case the code may indicate that the received packet does not match any valid flow keys and, further, contains no data (e.g., it may be a control packet).

State **668** may also be entered from state **654**. In this context operation code **5** may indicate that the packet does not match any valid flow keys. It may further indicate that the packet contains data, but that the size of the data portion is less than the threshold discussed in conjunction with state **654**. In this context, it appears that the packet's data is complete (e.g., comprises all of the data for a datagram), meaning. that there is no other data to re-assemble with this packet's data and therefore there is no reason to make a new entry in the database for this one-packet flow.

Finally, state **668** may also be entered from state **656**. In this context, operation code **5** may indicate that the packet does not match any valid flow keys, contains data, and more data is expected, but at least one flag in one or more of the packet's protocol headers does not have its expected value. For example, in one embodiment of the invention the TCP flags URG, PSH, RST and FIN are expected to be clear. If any of these flags are set an exceptional condition may exist, thus making it possible that one of the functions offered by NIC **100** is unsuitable for this packet.

As TABLE 1 reflects, there is no flow to tear down and no new flow is established when operation code **5** is selected. Following state **668**, the illustrated procedure ends at state **670**.

One skilled in the art will appreciate that the procedure illustrated in FIGS. **6A–6E** and discussed above is but one suitable procedure for maintaining and updating a flow database and for determining a packet's suitability for certain processing functions. In particular, different operation codes may be utilized or may be implemented in a different manner, a goal being to produce information for later processing of the packet through NIC **100**.

Although operation codes are assigned for all packets by a flow database manager in the illustrated procedure, in an alternative procedure an operation code assigned by the FDBM may be replaced or changed by another module of NIC **100**. This may be done to ensure a particular method of treating certain types of packets. For example, in one embodiment of the invention IPP module **104** assigns a predetermined operation code (e.g., operation code **2** of TABLE 1) to jumbo packets (e.g., packets greater in size than MTU) so that DMA engine **120** will not re-assemble them. In particular, the IPP module may independently determine that the packet is a jumbo packet (e.g., from information provided by a MAC module) and therefore assign the predetermined code. Illustratively, header parser **106** and FDBM **108** perform their normal functions for a jumbo packet and IPP module **104** receives a first operation code assigned by the FDBM. However, the IPP module replaces that code before storing the jumbo packet and information concerning the packet. In one alternative

embodiment header parser **106** and/or flow database manager **108** may be configured to recognize a particular type of packet (e.g., jumbo) and assign a predetermined operation code.

The operation codes applied in the embodiment of the invention illustrated in FIGS. **6A–6E** are presented and explained in the following TABLE 1. TABLE 1 includes illustrative criteria used to select each operation code and illustrative results or effects of each code.

TABLE 1

Op. Code	Criteria for Selection	Result of Operation Code
0	Compatible control packet with clear flags; a flow was previously established for this flow key.	Do not set up a new flow; Do not tear down existing flow; Do not re-assemble data (packet contains no data).
1	Compatible control packet with at least one flag or SYN bit set; a flow was previously established.	Do not set up a new flow; Tear down existing flow; Do no re-assemble data packet contains no data).
2	Compatible packet whose sequence number does not match sequence number in flow database, or SYN bit is set (indicating attempt to re-establish a connection) but there is no more data to come; a flow was previously established. Or Jumbo packet.	Do not set up a new flow; Tear down existing flow; Do not re-assemble packet data.
3	A compatible packet carrying a final portion of flow data, or a flag is set (but packet is in sequence, unlike operation code 2); a flow was previously established.	Do not set up a new flow; Tear down existing flow; Re-assemble data with previous packets.
4	Receipt of next compatible packet in sequence; a flow was previously established.	Do not set up a new flow; Do not tear down existing flow; Re-assemble data with other packets.
5	Packet cannot be re-assembled because: incompatible, a flag is set, packet contains no data or there is no more data to come. No flow was previously established.	Do not set up a flow; There is no flow to tear down; Do not re-assemble.
6	First compatible packet of a new flow; no flow was previously established.	Set up a new flow; There is no flow to tear down; Re-assemble data with packets to follow.
7	First compatible packet of a new flow, but flow database is full; no flow was previously established. Or Compatible packet, SYN bit is set and additional data will follow; a flow was previously established.	Replace existing flow; Re-assemble data with packets to follow.

he Embodiment of a Load Distributor

In one embodiment of the invention, load distributor **112** ables the processing of packets through their protocol cks to be distributed among a number of processors. ustratively, load distributor **112** generates an identifier g., a processor number) of a processor to which a packet o be submitted. The multiple processors may be located thin a host computer system that is served by NIC **100**. In e alternative embodiment, one or more processors for nipulating packets through a protocol stack are located on C **100**.

Without an effective method of sharing or distributing the cessing burden, one processor could become overloaded it were required to process all or most network traffic eived at NIC **100**, particularly in a high-speed network vironment. The resulting delay in processing network ffic could deteriorate operations on the host computer system as well as other computer systems communicating with the host system via the network.

As one skilled in the art will appreciate, simply distributing packets among processors in a set of processors (e.g., such as in a round-robin scheme) may not be an efficient plan. Such a plan could easily result in packets being processed out of order. For example, if two packets from one communication flow or connection that are received at a network interface in the correct order were submitted to two different processors, the second packet may be processed before the first. This could occur, for example, if the processor that received the first packet could not immediately process the packet because it was busy with another task. When packets are processed out of order a recovery scheme must generally be initiated, thus introducing even more inefficiency and more delay.

Therefore, in a present embodiment of the invention packets are distributed among multiple processors based upon their flow identities. As described above, a header parser may generate a flow key from layer three (e.g., IP) and layer four (e.g., TCP) source and destination identifiers retrieved from a packet's headers. The flow key may be used to identify the communication flow to which the packet belongs. Thus, in this embodiment of the invention all packets having an identical flow key are submitted to a single processor. As long as the packets are received in order by NIC **100**, they should be provided to the host computer and processed in order by their assigned processor.

Illustratively, multiple packets sent from one source entity to one destination entity will have the same flow key even if the packets are part of separate datagrams, as long as their layer three and layer four identifiers remain the same. As discussed above, separate flows are set up and torn down for each datagram within one TCP end-to-end connection. Therefore, just as all packets within one flow are sent to one processor, all packets within a TCP end-to-end connection will also be sent to the same processor. This helps ensure the correct ordering of packets for the entire connection, even between datagrams.

Depending upon the network environment in which NIC **100** operates (e.g., the protocols supported by network **102**), the flow key may be too large to use as an identifier of a processor. In one embodiment of the invention described above, for example, a flow key measures 288 bits. Meanwhile, the number of processors participating in the load-balancing scheme may be much smaller. For example, in the embodiment of the invention described below in conjunction with FIG. **7**, a maximum of sixty-four processors is supported. Thus, in this embodiment only a six-bit number is needed to identify the selected processor. The larger flow key-may therefore be mapped or hashed into a smaller range of values.

FIG. **7** depicts one method of generating an identifier (e.g., a processor number) to specify a processor to process a packet received by NIC **100**, based on the packet's flow key. In this embodiment of the invention, network **102** is the Internet and a received packet is formatted according to a compatible protocol stack (e.g., Ethernet at layer two, IP at layer three and TCP at layer four).

State **700** is a start state. In state **702** a packet is received by NIC **100** and a header portion of the packet is parsed by header parser **106** (a method of parsing a packet is described in a previous section). In state **704**, load distributor **112** receives the packet's flow key that was generated by header parser **106**.

Because a packet's flow key is 288 bits wide in this embodiment, in state **706** a hashing function is performed to

generate a value that is smaller in magnitude. The hash operation may, for example, comprise a thirty-two bit CRC (cyclic redundancy check) function such as ATM (Asynchronous Transfer Mode) Adaptation Layer 5 (AAL5). AAL5 generates thirty-two bit numbers that are fairly evenly distributed among the 2^{32} possible values. Another suitable method of hashing is the standard Ethernet CRC-32 function. Other hash functions that are capable of generating relatively small numbers from relatively large flow keys, where the numbers generated are well distributed among a range of values, are also suitable.

With the resulting hash value, in state **708** a modulus operation is performed over the number of processors available for distributing or sharing the processing. Illustratively, software executing on the host computer (e.g., a device driver for NIC **100**) programs or stores the number of processors such that it may be read or retrieved by load distributor **112** (e.g., in a register). The number of processors available for load balancing may be all or a subset of the number of processors installed on the host computer system. In the illustrated embodiment, the number of processors available in a host computer system is programmable, with a maximum value of sixty-four. The result of the modulus operation in this embodiment, therefore, is the number of the processor (e.g., from zero to sixty-three) to which the packet is to be submitted for processing. In this embodiment of the invention, load distributor **112** is implemented in hardware, thus allowing rapid execution of the hashing and modulus functions. In an alternative embodiment of the invention, virtually any number of processors may be accommodated.

In state **710**, the number of the processor that will process the packet through its protocol stack is stored in the host computer's memory. Illustratively, state **710** is performed in parallel with the storage of the packet in a host memory buffer. As described in a following section, in one embodiment of the invention a descriptor ring in the host computer's memory is constructed to hold the processor number and possibly other information concerning the packet (e.g., a pointer to the packet, its size, its TCP checksum).

A descriptor ring in this embodiment is a data structure comprising a number of entries, or "descriptors," for storing information to be used by a network interface circuit's host computer system. In the illustrated embodiment, a descriptor temporarily stores packet information after the packet has been received by NIC **100**, but before the packet is processed by the host computer system. The information stored in a descriptor may be used, for example, by the device driver for NIC **100** or for processing the packet through its protocol stack.

In state **712**, an interrupt or other alert is issued to the host computer to inform it that a new packet has been delivered from NIC **100**. In an embodiment of the invention in which NIC **100** is coupled to the host computer by a PCI (Peripheral Component Interconnect) bus, the INTA signal may be asserted across the bus. A PCI controller in the host receives the signal and the host operating system is alerted (e.g., via an interrupt).

In state **714**, software operating on the host computer (e.g., a device driver for NIC **100**) is invoked (e.g., by the host computer's operating system interrupt handler) to act upon a newly received packet. The software gathers information from one or more descriptors in the descriptor ring and places information needed to complete the processing of each new packet into a queue for the specified processor (i.e., according to the processor number stored in the packet's descriptor). Illustratively, each descriptor corresponds to a separate packet. The information stored in the. processor queue for each packet may include a pointer to a buffer containing the packet, the packet's TCP checksum, offsets of one or more protocol headers, etc. In addition, each processor participating in the load distribution scheme may have an associated queue for processing network packets. In an alternative embodiment of the invention, multiple queues may be used (e.g., for multiple priority levels or for different protocol stacks).

Illustratively, one processor on the host computer system is configured to receive all alerts and/or interrupts associated with the receipt of network packets from NIC **100** and to alert the appropriate software routine or device driver. This initial processing may, alternatively, be distributed among multiple processors. In addition, in one embodiment of the invention a portion of the retrieval and manipulation of descriptor contents is performed as part of the handling of the interrupt that is generated when a new packet is stored in the descriptor ring. The processor selected to process the packet will perform the remainder of the retrieval/manipulation procedure.

In state **716**, the processor designated to process a new packet is alerted or woken. In an embodiment of the invention operating on a Solaris™ workstation, individual processes executed by the processor are configured as "threads." A thread is a process running in a normal mode (e.g., not at an interrupt level) so as to have minimal impact on other processes executing on the workstation. A normal mode process may, however, execute at a high priority. Alternatively, a thread may run at a relatively low interrupt level.

A thread responsible for processing an incoming packet may block itself when it has no packets to process, and awaken when it has work to do. A "condition variable" may be used to indicate whether the thread has a packet to process. Illustratively, the condition variable is set to a first value when the thread is to process a packet (e.g., when a packet is received for processing by the processor) and is set to a second value when there are no more packets to process. In the illustrated embodiment of the invention, one condition variable may be associated with each processor's queue.

In an alternative embodiment, the indicated processor is alerted in state **716** by a "cross-processor call." A cross-processor call is one way of communicating among processors whereby one processor is interrupted remotely by another processor. Other methods by which one processor alerts, or dispatches a process to, another processor may be used in place of threads and cross-processor calls.

In state **718**, a thread or other process on the selected processor begins processing the packet that was stored in the processor's queue. Methods of processing a packet through its protocol stack are well known to those skilled in the art and need not be described in detail. The illustrated procedure then ends with end state **720**.

In one alternative embodiment of the invention, a high-speed network interface is configured to receive and process ATM (Asynchronous Transfer Mode) traffic. In this embodiment, a load distributor is implemented as a set of instructions (e.g., as software) rather than as a hardware module. As one skilled in the art is aware, ATM traffic is connection-oriented and may be identified by a virtual connection identifier (VCI), which corresponds to a virtual circuit established between the packet's source and destination entities. Each packet that is part of a virtual circuit includes the VCI in its header.

Advantageously, a VCI is relatively small in size (e.g., sixteen bits). In this alternative embodiment, therefore, a packet's VCI may be used in place of a flow key for the

purpose of distributing or sharing the burden of processing packets through their protocol stacks. Illustratively, traffic from different VCIs is sent to different processors, but, to ensure correct ordering of packets, all packets having the same VCI are sent to the same processor. When an ATM packet is received at a network interface, the VCI is retrieved from its header and provided to the load distributor. The modulus of the VCI over the number of processors that are available for load distribution is then computed. Similar to the illustrated embodiment, the packet and its associated processor number are then provided to the host computer.

As described above, load distribution in a present embodiment of the invention is performed on the basis of a packet's layer three and/or layer four source and destination entity identifiers. In an alternative embodiment of the invention, however, load distribution may be performed on the basis of layer two addresses. In this alternative embodiment, packets having the same Ethernet source and destination addresses, for example, are sent to a single processor.

As one of skill in the art will recognize, however, this may result in a processor receiving many more packets than it would if layer three and/or layer four identifiers were used. For example, if a large amount of traffic is received through a router situated near (in a logical sense) to the host computer, the source Ethernet address for all of the traffic may be the router's address even though the traffic is from a multitude of different end users and/or computers. In contrast, if the host computer is on the same Ethernet segment as all of the end users/computers, the layer two source addresses will show greater variety and allow more effective load sharing.

Other methods of distributing the processing of packets received from a network may differ from the embodiment illustrated in FIG. **7** without exceeding the scope of the invention. In particular, one skilled in the art will appreciate that many alternative procedures for assigning a flow's packets to a processor and delivering those packets to the processor may be employed.

One Embodiment of a Packet Queue

As described above, packet queue **1 16** stores packets received from IPP module **104** prior to their re-assembly by DMA engine **120** and their transfer to the host computer system. FIG. **8** depicts packet queue **116** according to one embodiment of the invention.

In the illustrated embodiment, packet queue **116** is implemented as a FIFO (First-In First-Out) queue containing up to 256 entries. Each packet queue entry in this embodiment stores one packet plus various information concerning the packet. For example, entry **800** includes packet portion **802** plus a packet status portion. Because packets of various sizes are stored in packet queue **116**, packet portion **802** may include filler **802** a to supplement the packet so that the packet portion ends at an appropriate boundary (e.g., byte, word, double word).

Filler **802***a* may comprise random data or data having a specified pattern. Filler **802***a* may be distinguished from the stored packet by the pattern of the filler data or by a tag field.

Illustratively, packet status information includes TCP checksum value **804** and packet length **806** (e.g., length of the packet stored in packet portion **802**). Storing the packet length may allow the packet to be easily identified and retrieved from packet portion **802**. Packet status information may also include diagnostic/status information **808**. Diagnostic/status information **808** may include a flag indicating that the packet is bad (e.g., incomplete, received with an error), an indicator that a checksum was or was not computed for the packet, an indicator that the checksum has a certain value, an offset to the portion of the packet on which the checksum was computed, etc. Other flags or indicators may also be included for diagnostics, filtering, or other purposes. In one embodiment of the invention, the packet's flow key (described above and used to identify the flow comprising the packet) and/or flow number (e.g., the corresponding index of the packet's flow in flow database **110**) are included in diagnostic/status information **808**. In another embodiment, a tag field to identify or delimit filler **802** a is included in diagnostic/status information **808**.

In one alternative embodiment of the invention, any or all of the packet status information described above is stored in control queue **118** rather than packet queue **116**.

In the illustrated embodiment of the invention packet queue **116** is implemented in hardware (e.g., as random access memory). In this embodiment, checksum value **804** is sixteen bits in size and may be stored by checksum generator **114**. Packet length **806** is fourteen bits large and may be stored by header parser **106**. Finally, portions of diagnostic/status information **808** may be stored by one or more of IPP module **104**, header parser **106**, flow database manager **108**, load distributor **112** and checksum generator **114**.

Packet queue **116** in FIG. **8** is indexed with two pointers. Read pointer **810** identifies the next entry to be read from the queue, while write pointer **812** identifies the entry in which the next received packet and related information is to be stored. As explained in a subsequent section, the packet stored in packet portion **802** of an entry is extracted from packet queue **116** when its data is to be-reassembled by DMA engine **120** and/or transferred to the host computer system.

One Embodiment of a Control Queue

In one embodiment of the invention, control queue **118** stores control and status information concerning a packet received by NIC **100**. In this embodiment, control queue **118** retains information used to enable the batch processing of protocol headers and/or the re-assembly of data from multiple related packets. Control queue **118** may also store information to be used by the host computer or a series of instructions operating on a host computer (e.g., a device driver for NIC **100**). The information stored in control queue **118** may supplement or duplicate information stored in packet queue **116**.

FIG. **9** depicts control queue **118** in one embodiment of the invention. The illustrated control queue contains one entry for each packet stored in packet queue **116** (e.g., up to 256 entries). In one embodiment of the invention each entry in control queue **118** corresponds to the entry (e.g., packet) in packet queue **116** having the same number. FIG. **9** depicts entry **900** having various fields, such as CPU number **902**, No_Assist signal **904**, operation code **906**, payload offset **908**, payload size **910** and other status information **912**. An entry may also include other status or control information (not shown in FIG. **9**). Entries in control queue **118** in alternative embodiments of the invention may comprise different information.

CPU (or processor) number **902**, discussed in a previous section, indicates which one of multiple processors on the host computer system should process the packet's protocol headers. Illustratively, CPU number **902** is six bits in size. No_Assist signal **904**, also described in a preceding section, indicates whether the packet is compatible with (e.g., is formatted according to) any of a set of pre-selected protocols that may be parsed by header parser **106**. No_Assist signal **904** may comprise a single flag (e.g. one bit). In one embodiment of the invention the state or value of No_Assist signal **904** may be used by flow database manager **108** to

determine whether a packet's data is re-assembleable and/or whether its headers may be processed with those of related packets. In particular, the FDBM may use the No_Assist signal in determining which operation code to assign to the packet.

Operation code **906** provides information to DMA engine **120** to assist in the re-assembly of the packet's data. As described in a previous section, an operation code may indicate whether a packet includes data or whether a packet's data is suitable for re-assembly. Illustratively, operation code **906** is three bits in size. Payload offset **908** and payload size **910** correspond to the offset and size of the packet's TCP payload (e.g., TCP data), respectively. These fields may be seven and fourteen bits large, respectively.

In the illustrated embodiment, other status information **912** includes diagnostic and/or status information concerning the packet. Status information **912** may include a starting position for a checksum calculation (which may be seven bits in size), an offset of the layer three (e.g., IP) protocol header (which may also be seven bits in size), etc. Status information **912** may also include an indicator as to whether the size of the packet exceeds a first threshold (e.g., whether the packet is greater than 1522 bytes) or falls under a second threshold (e.g., whether the packet is 256 bytes or less). This information may be useful in re-assembling packet data. Illustratively, these indicators comprise single-bit flags.

In one alternative embodiment of the invention, status information **912** includes a packet's flow key and/or flow number (e.g., the index of the packet's flow in flow database **110**). The flow key or flow number may, for example, be used for debugging or other diagnostic purposes. In one embodiment of the invention, the packet's flow number may be stored in status information **912** so that multiple packets in a single flow may be identified. Such related packet may then be collectively transferred to and/or processed by a host computer.

FIG. **9** depicts a read pointer and a write pointer for indexing control queue **118**. Read pointer **914** indicates an entry to be read by DMA engine **120**. Write pointer **916** indicates the entry in which to store information concerning the next packet stored in packet queue **116**.

In an alternative embodiment of the invention, a second read pointer (not shown in FIG. **9**) may be used for indexing control queue **118**. As described in a later section, when a packet is to be transferred to the host computer, information drawn from entries in the control queue is searched to determine whether a related packet (e.g., a packet in the same flow as the packet to be transferred) is also going to be transferred. If so, the host computer is alerted so that protocol headers from the related packets may be processed collectively. In this alternative embodiment of the invention, related packets are identified by matching their flow numbers (or flow keys) in status information **912**. The second read pointer may be used to look ahead in the control queue for packets with matching flow numbers.

In one embodiment of the invention CPU number **902** may be stored in the control queue by load distributor **112** and No_Assist signal **904** may be stored by header parser **106**. Operation code **906** may be stored by flow database manager **108**, and payload offset **908** and payload size **910** may be stored by header parser **106**. Portions of other status information may be written by the preceding modules and/or others, such as IPP module **104** and checksum generator **114**. In one particular embodiment of the invention, however, many of these items of information are stored by IPP module **104** or some other module acting in somewhat of a coordinator role.

One Embodiment of a DMA Engine

FIG. **10** is a block diagram of DMA (Direct Memory Access) engine **120** in one embodiment. of the invention. One purpose of DMA engine **120** in this embodiment is to transfer packets from packet queue **116** into buffers in host computer memory. Because related packets (e.g., packets that are part of one flow) can be identified by their flow numbers or flow keys, data from the related packets may be transferred together (e.g., in the same buffer). By using one buffer for data from one flow, the data can be provided to an application program or other destination in a highly efficient manner. For example, after the host computer receives the data, a page-flip operation may be performed to transfer the data to an application's memory space rather than performing numerous copy. operations.

With reference back to FIGS. **1A–B**, a packet that is to be transferred into host memory by DMA engine **120** is stored in packet queue **116** after being received from network **102**. Header parser **106** parses a header portion of the packet and generates a flow key, and flow database manager **108** assigns an operation code to the packet. In addition, the communication flow that includes the packet is registered in flow database **110**. The packet's flow may be identified by its flow key or flow number (e.g., the index of the flow in flow database **110**). Finally, information concerning the packet (e.g., operation code, a packet size indicator, flow number) is stored in control queue **118** and, possibly, other portions or modules of NIC **100**, and the packet is transferred to the host computer by DMA engine **120**. During the transfer process, the DMA engine may draw upon information stored in the control queue to copy the packet into an appropriate buffer, as described below. Dynamic packet batching module **122** may also use information stored in the control queue, as discussed in detail in a following section.

With reference now to FIG. **10**, one embodiment of a DMA engine is presented. In this embodiment, DMA manager **1002** manages the transfer of a packet, from packet queue **116**, into one or more buffers in host computer memory. Free ring manager **1012** identifies or receives empty buffers from host memory and completion ring manager **1014** releases the buffers to the host computer, as described below. The free ring manager and completion ring managers may be controlled with logic contained in DMA manager **1002**. In the illustrated embodiment, flow re-assembly table **1004**, header table **1006**, MTU table **1008** and jumbo table **1010** store information concerning buffers used to store different types of packets (as described below). Information stored in one of these tables may include a reference to, or some other means of identifying, a buffer. In FIG. **10**, DMA engine **120** is partially or fully implemented in hardware.

Empty buffers into which packets may be stored are identified via a free descriptor ring that is maintained in host memory. As one skilled in the art is aware, a descriptor ring is a data structure that is logically arranged as a circular queue. A descriptor ring contains descriptors for storing information (e.g., data, flag, pointer, address). In one embodiment of the invention, each descriptor stores its index within the free descriptor ring and an identifier (e.g., memory address, pointer) of a free buffer that may be used to store packets. In this embodiment a buffer is identified in a descriptor by its address in memory, although other means of identifying a memory buffer are also suitable. In one embodiment of the invention a descriptor index is thirteen bits large, allowing for a maximum of 8,192 descriptors in the ring, and a buffer address is sixty-four bits in size.

In the embodiment of FIG. **10**, software that executes on a host computer, such as a device driver for NIC **100**,

maintains a free buffer array or other data structure (e.g., list, table) for storing references to (e.g., addresses of) the buffers identified in free descriptors. As descriptors are retrieved from the ring their buffer identifiers are placed in the array. Thus, when a buffer is needed for the storage of a packet, it may be identified by its index (e.g., cell, element) in the free buffer array. Then, when the buffer is no longer needed, it may be released to the host computer by placing its array index or reference in a completion descriptor. A packet stored in the buffer can then be retrieved by accessing the buffer identified in the specified element of the array. Thus, in this embodiment of the invention the size of a descriptor index (e.g., thirteen bits) may. not limit the number of buffers that may be assigned by free ring manager **1012**. In particular, virtually any number of buffers or descriptors could be managed by the software. For example, in one. alternative embodiment of the invention buffer identifiers may be stored in one or more linked lists after being retrieved from descriptors in a free descriptor ring. When the buffer is released to the host computer, a reference to the head of the buffer's linked list may be provided. The list could then be navigated to locate the particular buffer (e.g., by its address).

As one skilled in the art will appreciate, the inclusion of a limited number of descriptors in the free descriptor ring (e.g., 8,192 in this embodiment) means that they may be re-used in a round-robin fashion. In the presently described embodiment, a descriptor is just needed long enough to retrieve its buffer identifier (e.g., address) and place it in the free buffer array, after which it may be re-used relatively quickly. In other embodiments of the invention free descriptor rings having different numbers of free descriptors may be used, thus allowing some control over the rate at which free descriptors must be re-used.

In one alternative embodiment of the invention, instead of using a separate data structure to identify a buffer for storing a packet, a buffer may be identified within DMA engine **120** by the index of the free descriptor within the free descriptor ring that referenced the buffer. One drawback to this scheme when the ring contains a limited number of descriptors, however, is that a particular buffer's descriptor may need to be re-used before its buffer has been released to the host computer. Thus, either a method of avoiding or skipping the re-use of such a descriptor must be implemented or the buffer referenced by the descriptor must be released before the descriptor is needed again. Or, in another alternative, a free descriptor ring may be of such a large size that a lengthy or even virtually infinite period of time may pass from the time a free descriptor is first used until it needs to be re-used.

Thus, in the illustrated embodiment of the invention free ring manager **1012** retrieves a descriptor from the free descriptor ring, stores its buffer identifier (e.g., memory address) in a free buffer array, and provides the array index and/or buffer identifier to flow re-assembly table **1004**, header table **1006**, MTU table **1008** or jumbo table **1010**.

Free ring manager **1012** attempts to ensure that a buffer is always available for a packet. Thus, in one embodiment of the invention free ring manager **1012** includes descriptor cache **1012** a configured to store a number of descriptors (e.g., up to eight) at a time. Whenever there are less than a threshold number of entries in the cache (e.g., five), additional descriptors may be retrieved from the free descriptor ring. Advantageously, the descriptors are of such a size (e.g., sixteen bytes) that some multiple (e.g., four) of them can be efficiently retrieved in a sixty-four byte cache line transfer from the host computer.

Returning now to the illustrated embodiment of the invention, each buffer in host memory is one memory page in size. However, buffers and the packets stored in the buffers may be divided into multiple categories based on packet size and whether a packet's data is being re-assembled. Re-assembly refers to the accumulation of data from multiple packets of a single flow into one buffer for efficient transfer from kernel space to user or application space within host memory. In particular, re-assembleable packets may be defined as packets that conform to a pre-selected protocol (e.g., a protocol that is parseable by header parser **106**). By filling a memory page with data for one destination, page-flipping may be performed to provide a page in kernel space to the application or user space. A packet's category (e.g., whether re-assembleable or non-re-assembleable) may be determined from information retrieved from the control queue or flow database manager. In particular, and as described previously, an operation code may be used to determine whether a packet contains a re-assembleable portion of data.

In the illustrated embodiment of the invention, data portions of related, re-assembleable, packets are placed into a first category of buffers—which may be termed re-assembly buffers. A second category of buffers, which may be called header buffers, stores the headers of those packets whose data portions are being re-assembled and may also store small packets (e.g., those less than or equal to 256 bytes in size). A third category of buffers, MTU buffers, stores non-re-assembleable packets that are larger than 256 bytes, but no larger than MTU size (e.g., 1522 bytes). Finally, a fourth category of buffers, jumbo buffers, stores jumbo packets (e.g., large packets that are greater than 1522 bytes in size) that are not being re-assembled. Illustratively, a jumbo packet may be stored intact (e.g., its headers and data portions kept together in one buffer) or its headers may be stored in a header buffer while its data portion is stored in an appropriate (e.g., jumbo) non-re-assembly buffer.

In one alternative embodiment of the invention, no distinction is made between MTU and jumbo packets. Thus, in this alternative embodiment, just three types of buffers are used: re-assembly and header buffers, as described above, plus non-re-assembly buffers. Illustratively, all non-small packets (e.g., larger than 256 bytes) that are not re-assembled are placed in a non-re-assembly buffer.

In another alternative embodiment, jumbo packets may be re-assembled in jumbo buffers. In particular, in this embodiment data portions of packets smaller than a predetermined size (e.g., MTU) are re-assembled in normal re-assembly buffers while data portions of jumbo packets (e.g., packets greater in size than MTU) are re-assembled in jumbo buffers. Re-assembly of jumbo packets may be particularly effective for a communication flow that comprises jumbo frames of a size such that multiple frames can fit in one buffer. Header portions of both types of packets may be stored in one type of header buffer or, alternatively, different header buffers may be used for the headers of the different types of re-assembleable packets.

In yet another alternative embodiment of the invention buffers may be of varying sizes and may be identified in different descriptor rings or other data structures. For example, a first descriptor ring or other mechanism may be used to identify buffers of a first size for storing large or jumbo packets. A second ring may store descriptors referencing buffers for MTU-sized packets, and another ring may contain descriptors for identifying page-sized buffers (e.g., for data re-assembly).

A buffer used to store portions of more than one type of packet—such as a header buffer used to store headers and small packets, or a non-re-assembly buffer used to store MTU and jumbo packets—may be termed a "hybrid" buffer.

Illustratively, each time a packet or a portion of a packet is stored in a buffer, completion ring manager **1014** populates a descriptor in a completion descriptor ring with information concerning the packet. Included in the information stored in a completion descriptor in this embodiment is a number or reference identifying the free buffer array cell or element in which an identifier (e.g., memory address) of a buffer in which a portion of the packet is stored. The information may also include an offset into the buffer (e.g., to the beginning of the packet portion), the identity of another free buffer array entry that stores a buffer identifier for a buffer containing another portion of the packet, a size of the packet, etc. A packet may be stored in multiple buffers, for example, if the packet data and header are stored separately (e.g., the packet's data is being re-assembled in a re-assembly buffer while the packet's header is placed in a header buffer). In addition, data portions of a jumbo packet or a re-assembly packet may span two or more buffers, depending on the size of the data portion.

A distinction should be kept in mind between a buffer identifier (e.g., the memory address of a buffer) and the entry in the free buffer array in which the buffer identifier is stored. In particular, it has been described above that when a memory buffer is released to a host computer it is identified to the host computer by its position within a free buffer array (or other suitable data structure) rather than by its buffer identifier. The host computer retrieves the buffer identifier from the specified array element and accesses the specified buffer to locate a packet stored in the buffer. As one skilled in the art will appreciate, identifying memory buffers in completion descriptors by the buffers' positions in a free buffer array can be more efficient than identifying them by their memory addresses. In particular, in FIG. **10** buffer identifiers are sixty-four bits in size while an index in a free buffer array or similar data structure will likely be far smaller. Using array positions thus saves space compared to using buffer identifiers. Nonetheless, buffer identifiers may be used to directly identify buffers in an alternative embodiment of the invention, rather than filtering access to them through a free buffer array. However, completion descriptors would have to be correspondingly larger in order to accommodate them.

A completion descriptor may also include one or more flags indicating the type or size of a packet, whether the packet data should be re-assembled, whether the packet is the last of a datagram, whether the host computer should delay processing the packet to await a related packet, etc. As described in a following section, in one embodiment of the invention dynamic packet batching module **122** determines, at the time a packet is transferred to the host computer, whether a related packet will be sent shortly. If so, the host computer may be advised to delay processing the transferred packet and await the related packet in order to allow more efficient processing.

A packet's completion descriptor may be marked appropriately when the buffer identified by its buffer identifier is to be released to the host computer. For example, a flag may be set in the descriptor to indicate that the packet's buffer is being released from DMA engine **120** to the host computer or software operating on the host computer (e.g., a driver associated with NIC **100**). In one embodiment of the invention, completion ring manager **1014** includes completion descriptor cache **1014***a*. Completion descriptor cache **1014***a* may store one or more completion descriptors for collective transfer from DMA engine **120** to the host computer.

Thus, empty buffers are retrieved from a free ring and used buffers are released to the host computer through a completion ring. One reason that a separate ring is employed to release used buffers to the host computer is that buffers may not be released in the order in which they were taken. In one embodiment of the invention, a buffer (especially a flow re-assembly buffer) may not be released until it is full. Alternatively, a buffer may be released at virtually any time, such as when the end of a communication flow is detected. Free descriptors and completion descriptors are further described below in conjunction with FIG. **12**.

Another reason that separate rings are used for free and completion descriptors is that the number of completion descriptors that are required in an embodiment of the invention may exceed the number of free descriptors provided in a free descriptor ring. For example, a buffer provided by a free descriptor may be used to store multiple headers and/or small packets. Each time a header or small packet is stored in the header buffer, however, a separate completion descriptor is generated. In an embodiment of the invention in which a header buffer is eight kilobytes in size, a header buffer may store up to thirty-two small packets. For each packet stored in the header buffer, another completion descriptor is generated.

FIG. **11** includes diagrams of illustrative embodiments of flow re-assembly table **1004**, header table **1006**, MTU table **1008** and jumbo table **1010**. One alternative embodiment of the invention includes a non-re-assembly table in place of MTU table **1008** and jumbo table **1010**, corresponding to a single type of non-re-assembly buffer for both MTU and jumbo packets. Jumbo table **1010** may also be omitted in another alternative embodiment of the invention in which jumbo buffers are retrieved or identified only when needed. Because a jumbo buffer is used only once in this alternative embodiment, there is no need to maintain a table to track its use.

Flow re-assembly table **1004** in the illustrated embodiment stores information concerning the re-assembly of packets in one or more communication flows. For each flow that is active through DMA engine **120**, separate flow re-assembly buffers may be used to store the flow's data. More than one buffer may be used for a particular flow, but each flow has one entry in flow re-assembly table **1004** with which to track the use of a buffer. As described in a previous section, one embodiment of the invention supports the interleaving of up to sixty-four flows. Thus, flow re-assembly buffer table **1004** in this embodiment maintains up to sixty-four entries. A flow's entry in the flow re-assembly table may match its flow number (e.g., the index of the flow's flow key in flow database **110**) or, in an alternative embodiment, an entry may be used for any flow.

In FIG. **11**, an entry in flow re-assembly table **1004** includes flow re-assembly buffer index **1102**, next address **1104** and validity indicator **1106**. Flow re-assembly buffer index **1102** comprises the index, or position, within a free buffer array or other data structure for storing buffer identifiers identified in free descriptors, of a buffer for storing data from the associated flow. Illustratively, this value is written into each completion descriptor associated with a packet whose data portion is stored in the buffer. This value may be used by software operating on the host computer to access the buffer and process the data. Next address **1104** identifies the location within the buffer (e.g., a memory address) at which to store the next portion of data. Illustratively, this field is updated each time data is added to the buffer. Validity indicator **1106** indicates whether the entry is valid. Illustratively, each entry is set to a valid state (e.g., stores a first value) when a first portion of data is stored in the flow's re-assembly buffer and is invalidated (e.g.,

stores a second value) when the buffer is full. When an entry is invalidated, the buffer may be released or returned to the host computer (e.g., because it is full).

Header table **1006** in the illustrated embodiment stores information concerning one or more header buffers in which packet headers and small packets are stored. In the illustrated embodiment of the invention, only one header buffer is active at a time. That is, headers and small packets are stored in one buffer until it is released, at which time a new buffer is used. In this embodiment, header table **1006** includes header buffer index **1112**, next address **1114** and validity indicator **1116**. Similar to flow re-assembly table **1004**, header buffer index **1112** identifies the cell or element in the free buffer array that contains a buffer identifier for a header buffer. Next address **1114** identifies the location within the header buffer at which to store the next header or small packet. This identifier, which may be a counter, may be updated each time a header or small packet is stored in the header buffer. Validity indicator **1116** indicates whether the header buffer table and/or the header buffer is valid. This indicator may be set to valid when a first packet or header is stored in a header buffer and may be invalidated when it is released to the host computer.

MTU table **1008** stores information concerning one or more MTU buffers for storing MTU packets (e.g., packets larger than 256 bytes but less than 1523 bytes) that are not being re-assembled. MTU buffer index **1122** identifies the free buffer array element that contains a buffer identifier (e.g., address) of a buffer for storing MTU packets. Next address **1124** identifies the location in the current MTU buffer at which to store the next packet. Validity indicator **1126** indicates the validity of the table entry. The validity indicator may be set to a valid state when a first packet is stored in the MTU buffer and an invalid state when the buffer is to be released to the host computer.

Jumbo table **1010** stores information concerning one or more jumbo buffers for storing jumbo packets (e.g., packets larger than 1522 bytes) that are not being re-assembled. Jumbo buffer index **1132** identifies the element within the free buffer array that stores a buffer identifier corresponding to a jumbo buffer. Next address **1134** identifies the location in the jumbo buffer at which to store the next packet. Validity indicator **1136** indicates the validity of the table entry. Illustratively, the validity indicator is set to a valid state when a first packet is stored in the jumbo buffer and is set to an invalid state when the buffer is to be released to the host computer.

In the embodiment of the invention depicted in FIG. **11**, a packet larger than a specified size (e.g., 256 bytes) is not re-assembled if it is incompatible with the pre-selected protocols for NIC **100** (e.g., TCP, IP, Ethernet) or if the packet is too large (e.g., greater than 1522 bytes). Although two types of buffers (e.g., MTU and jumbo) are used for non-re-assembleable packets in this embodiment, in an alternative embodiment of the invention any number may be used, including one. Packets less than the specified size are generally not re-assembled. Instead, as described above, they are stored intact in a header buffer.

In the embodiment of the invention depicted in FIG. **11**, next address fields may store a memory address, offset, pointer, counter or other means of identifying a position within a buffer. Advantageously, the next address field of a table or table entry is initially set to the address of the buffer assigned to store packets of the type associated with the table (e.g., for re-assembly table **1004**, the particular flow). As the buffer is populated, the address is updated to identify the location in the buffer at which to store the next packet or portion of a packet.

Illustratively, each validity indicator stores a first value (e.g., one) to indicate validity, and a second value (e.g., zero) to indicate invalidity. In the illustrated embodiment of the invention, each index field is thirteen bits, each address field is sixty-four bits and the validity indicators are each one bit in size.

Tables **1004**, **1006**, **1008** and **1010** may take other forms and remain within the scope of the invention as contemplated. For example, these data structures may take the form of arrays, lists, databases, etc., and may be implemented in hardware or software. In the illustrated embodiment of the invention, header table **1006**, MTU table **1008** and jumbo table **1010** each contain only one entry at a time. Thus, only one header buffer, MTU buffer and jumbo buffer are active (e.g., valid) at a time in this embodiment. In an alternative embodiment of the invention, multiple header buffers, MTU buffers and/or jumbo buffers may be used (e.g., valid) at once.

In one embodiment of the invention, certain categories of buffers (e.g., header, non-re-assembly) may store a predetermined number of packets or packet portions. For example, where the memory page size of a host computer processor is eight kilobytes, a header buffer may store a maximum of thirty-two entries, each of which is 256 bytes. Illustratively, even when one packet or header is less than 256 bytes, the next entry in the buffer is stored at the next 256-byte boundary. A counter may be associated with the buffer and decremented (or incremented) each time a new entry is stored in the buffer. After thirty-two entries have been made, the buffer may be released.

In one embodiment of the invention, buffers other than header buffers may be divided into fixed-size regions. For example, in an eight-kilobyte MTU buffer, each MTU packet may be allocated two kilobytes. Any space remaining in a packet's area after the packet is stored may be left unused or may be padded.

In one alternative embodiment of the invention, entries in a header buffer and/or non-re-assembly buffer (e.g., MTU, jumbo) are aligned for more efficient transfer. In particular, two bytes of padding (e.g., random bytes) are stored at the beginning of each entry in such a buffer. Because a packet's layer two Ethernet header is fourteen bytes long, by adding two pad bytes each packet's layer three protocol header (e.g., IP) will be aligned with a sixteen-byte boundary. Sixteen-byte alignment, as one skilled in the art will appreciate, allows efficient copying of packet contents (such as the layer three header). The addition of two bytes may, however, decrease the size of the maximum packet that may be stored in a header buffer (e.g., to 254 bytes).

As explained above, counters and/or padding may also be used with non-re-assembly buffers. Some non-re-assembleable packets (e.g., jumbo packets) may, however, be split into separate header and data portions, with each portion being stored in a separate buffer—similar to the re-assembly of flow packets. In one embodiment of the invention padding is only used with header portions of split packets. Thus, when a non-re-assembled (e.g., jumbo) packet is split, padding may be applied to the header/small buffer in which the packet's header portion is stored but not to the non-re-assembly buffer in which the packet's data portion is stored. When, however, a non-re-assembly packet is stored with its header and data together in a non-re-assembly buffer, then padding may be applied to that buffer.

In another alternative embodiment of the invention, a second level of padding may be added to each entry in a buffer that stores non-re-assembled packets that are larger than 256 bytes (e.g., MTU packets and jumbo packets that

are not split). In this alternative embodiment, a cache line of storage (e.g., sixty-four bytes for a Solaris™ workstation) is skipped in the buffer before storing each packet. The extra padding area may be used by software that processes the packets and/or their completion descriptors. The software may use the extra padding area for routing or as temporary storage for information needed in a secondary or later phase of processing.

For example, before actually processing the packet, the software may store some data that promotes efficient multi-tasking in the padding area. The information is then available when the packet is finally extracted from the buffer. In particular, in one embodiment of the invention a network interface may generate one or more data values to identify multicast or alternate addresses that correspond to a layer two address of a packet received from a network. The multicast or alternate addresses may be stored in a network interface memory by software operating on a host computer (e.g., a device driver). By storing the data value(s) in the padding, enhanced routing functions can be performed when the host computer processes the packet.

Reserving sixty-four bytes at the beginning of a buffer also allows header information to be modified or prepended if necessary. For example, a regular Ethernet header of a packet may, because of routing requirements, need to be replaced with a much larger FDDI (Fiber Distributed Data Interface) header. One skilled in the art will recognize the size disparity between these headers. Advantageously, the reserved padding area may be used for the FDDI header rather than allocating another block of memory.

In a present embodiment of the invention DMA engine **120** may determine which category a packet belongs in, and which type of buffer to store the packet in, by examining the packet's operation code. As described in a previous section, an operation code may be stored in control queue **118** for each packet stored in packet queue **116**. Thus, when DMA engine **120** detects a packet in packet queue **116**, it may fetch the corresponding information in the control queue and act appropriately.

An operation code may indicate whether a packet is compatible with the protocols pre-selected for NIC **100**. In an illustrative embodiment of the invention, only compatible packets are eligible for data re-assembly and/or other enhanced operations offered by NIC **100** (e.g., packet batching or load distribution). An operation code may also reflect the size of a packet (e.g., less than or greater than a predetermined size), whether a packet contains data or is a control packet, and whether a packet initiates, continues or ends a flow. In this embodiment of the invention, eight different operation codes are used. In alternative embodiments of the invention more or less than eight codes may be used. TABLE 1 lists operation codes that may be used in one embodiment of the invention.

FIGS. **12A–12B** illustrate descriptors from a free descriptor ring and a completion descriptor ring in one embodiment of the invention. FIG. **12A** also depicts a free buffer array for storing buffer identifiers retrieved from free descriptors.

Free descriptor ring **1200** is maintained in host memory and is populated with descriptors such as free descriptor **1202**. Illustratively, free descriptor **1202** comprises ring index **1204**, the index of descriptor **1202** in free ring **1200**, and buffer identifier **1206**. A buffer identifier in this embodiment is a memory address, but may, alternatively, comprise a pointer or any other suitable means of identifying a buffer in host memory.

In the illustrated embodiment, free buffer array **1210** is constructed by software operating on a host computer (e.g., a device driver). An entry in free buffer array **1210** in this embodiment includes array index field **1212**, which may be used to identify the entry, and buffer identifier field **1214**. Each entry's buffer identifier field thus stores a buffer identifier retrieved from a free descriptor in free descriptor ring **1200**.

In one embodiment of the invention, free ring manager **1012** of DMA engine **120** retrieves descriptor **1202** from the ring and stores buffer identifier **1206** in free buffer array **1210**. The free ring manager also passes the buffer identifier to flow re-assembly table **1004**, header table **1006**, MTU table **1008** or jumbo table **1010** as needed. In another embodiment the free ring manager extracts descriptors from the free descriptor ring and stores them in a descriptor cache until a buffer is needed, at which time the buffer's buffer identifier is stored in the free buffer array. In yet another embodiment, a descriptor may be used (e.g., the buffer that it references may be used to store a packet) while still in the cache.

In one embodiment of the invention descriptor **1202** is sixteen bytes in length. In this embodiment, ring index **1204** is thirteen bits in size, buffer identifier **1206** (and buffer identifier field **1214** in free buffer array **1210**) is sixty-four bits, and the remaining space may store other information or may not be used. The size of array index field **1212** depends upon the dimensions of array **1210**; in one embodiment the field is thirteen bits in size.

Completion descriptor ring **1220** is also maintained in host memory. Descriptors in completion ring **1220** are written or configured when a packet is transferred to the host computer by DMA engine **120**. The information written to a descriptor, such as descriptor **1222**, is used by software operating on the host computer (e.g., a driver associated with NIC **100**) to process the packet. Illustratively, an ownership indicator (described below) in the descriptor indicates whether DMA engine **120** has finished using the descriptor. For example, this field may be set to a particular value (e.g., zero) when the DMA engine finishes using the descriptor and a different value (e.g., one) when it is available for use by the DMA engine. However, in another embodiment of the invention, DMA engine **120** issues an interrupt to the host computer when it releases a completion descriptor. Yet another means of alerting the host computer may be employed in an alternative embodiment. Descriptor **1222**, in one embodiment of the invention, is thirty-two bytes in length.

In the illustrated embodiment of the invention, information stored in descriptor **1222** concerns a transferred packet and/or the buffer it was stored in, and includes the following fields. Data size **1230** reports the amount of data in the packet (e.g., in bytes). The data size field may contain a zero if there is no data portion in the packet or no data buffer (e.g., flow re-assembly buffer, non-re-assembly buffer, jumbo buffer, MTU buffer) was used. Data buffer index **1232** is the index, within free buffer array **1210**, of the buffer identifier for the flow re-assembly buffer, non-re-assembly buffer, jumbo buffer or MTU buffer in which the packet's data was stored. When the descriptor corresponds to a small packet fully stored in a header buffer, this field may store a zero or remain unused. Data offset **1234** is the offset of the packet's data within the flow re-assembly buffer, non-re-assembly buffer, jumbo buffer or MTU buffer (e.g., the location of the first byte of data within the data buffer).

In FIG. **12B**, flags field **1236** includes one or more flags concerning a buffer or packet. For example, if a header buffer or data is being released (e.g., because it is full), a release header or release data flag, respectively, is set. A

NOAC Ex. 1015 Page 544

release flow flag may be used to indicate whether a flow has, at least temporarily, ended. In other words, if a release flow flag is set (e.g., stores a value of one), this indicates that there are no other packets waiting in the packet queue that are in the same flow as the packet associated with descriptor **1222**. Otherwise, if this flag is not set (e.g., stores a value of zero), software operating on the host computer may queue this packet to await one or more additional flow packets so that they may be processed collectively. A split flag may be included in flags field **1236** to identify whether a packet's contents (e.g., data) spans multiple buffers. Illustratively, if the split flag is set, there will be an entry in next data buffer index **1240**, described below.

Descriptor type **1238**, in the presently described embodiment of the invention, may take any of three values. A first value (e.g., one) indicates that DMA engine **120** is releasing a flow buffer for a flow that is stale (e.g., no packet has been received in the flow for some period of time). A second value (e.g., two) may indicate that a non-re-assembleable packet was stored in a buffer. A third value (e.g., three) may be used to indicate that a flow packet (e.g., a packet that is part of a flow through NIC **100**) was stored in a buffer.

Next buffer index **1240** stores an index, in free buffer array **1210**, of an entry containing a buffer identifier corresponding to a buffer storing a subsequent portion of a packet if the entire packet, or its data, could not fit into the first assigned buffer. The offset in the next buffer may be assumed to be zero. Header size **1242** reports the length of the header (e.g., in bytes). The header size may be set to zero if the header buffer was not used for this packet (e.g., the packet is not being re-assembled and is not a small packet). Header buffer index **1244** is the index, in free buffer array **1210**, of the buffer identifier for the header buffer used to store this packet's header. Header offset **1246** is the offset of the packet's header within the buffer (e.g., header buffer) in which the header was stored. The header offset may take the form of a number of bytes into the buffer at which the header may be found. Alternatively, the offset may be an index value, reporting the index position of the header. For example, in one embodiment of the invention mentioned above, entries in a header buffer are stored in 256-byte units. Thus, each entry begins at a 256-byte boundary regardless of the actual size of the entries. The 256-byte entries may be numbered or indexed within the buffer.

In the illustrated embodiment, flow number **1250** is the packet's flow number (e.g., the index in flow database **110** to the packet's flow key). Flow number **1250** may be used to identify packets in the same flow. Operation code **1252** is a code generated by flow database manager **108**, as described in a previous section, and used by DMA engine **120** to process the packet and transfer it into an appropriate buffer. Methods of transferring a packet depending upon its operation code are described in detail in the following section. No_Assist signal **1254**, also described in a previous section, may be set or raised when the packet is not compatible with the protocols pre-selected for NIC **100**. One type of incompatibility is that header parser **106** may not properly parse the packet, in which case the packet will not receive the subsequent benefits. Processor identifier **1256**, which may be generated by load distributor **112**, identifies a host computer system processor for processing the packet. As described in a previous section, load distributor attempts to share or distribute the load of processing packets among multiple processors by having all packets within one flow processed by the same processor. Layer three header offset **1258** reports an offset within the packet to the first byte of the packet's layer three protocol (e.g., IP) header. With this value, software operating on the host computer may easily strip off one or more headers or header portions.

Checksum value **1260** is a checksum computed for this packet by checksum generator **114**. Packet length **1262** is the length (e.g., in bytes) of the entire packet.

Ownership indicator **1264** is used in the presently described embodiment of the invention to indicate whether NIC **100** or software operating on the host computer "owns" completion descriptor **1222**. In particular, a first value (e.g., zero) is placed in the ownership indicator field when NIC **100** (e.g., DMA engine **120**) has completed configuring the descriptor. Illustratively, this first value is understood to indicate that the software may now process the descriptor. When finished processing the descriptor, the software may store a second value (e.g., one) in the ownership indicator to indicate that NIC **100** may now use the descriptor for another packet.

One skilled in the art will recognize that there are numerous methods that may be used to inform host software that a descriptor has been used by, or returned to, DMA engine **120**. In one embodiment of the invention, for example, one or more registers, pointers or other data structures are maintained to indicate which completion descriptors in a completion descriptor ring have or have not been used. In particular, a head register may be used to identify a first of a series of descriptors that are owned by host software, while a tail register identifies the last descriptor in the series. DMA engine **120** may update these registers as it configures and releases descriptors. Thus, by examining these registers the host software and the DMA engine can determine how many descriptors have or have not been used.

Finally, other information, flags and indicators may be stored in other field **1266**. Other information that may be stored in one embodiment of the invention includes the length and/or offset of a TCP payload, flags indicating a small packet (e.g., less than 257 bytes) or a jumbo packet (e.g., more than 1522 bytes), a flag indicating a bad packet (e.g., CRC error), a checksum starting position, etc.

In alternative embodiments of the invention only information and flags needed by the host computer (e.g., driver software) are included in descriptor **1222**. Thus, in one alternative embodiment one or more fields other than the following may be omitted: data size **1230**, data buffer index **1232**, data offset **1234**, a split flag, next data buffer index **1240**, header size **1242**, header buffer index **1244**, header offset **1246** and ownership indicator **1264**.

In addition, a completion descriptor may be organized in virtually any form; the order of the fields of descriptor **1222** in FIG. **12** is merely one possible configuration. It is advantageous, however, to locate ownership indicator **1264** towards the end of a completion descriptor since this indicator may be used to inform host software when the DMA engine has finished populating the descriptor. If the ownership indicator were placed in the beginning of the descriptor, the software may read it and attempt to use the descriptor before the DMA engine has finished writing to it.

One skilled in the art will recognize that other systems and methods than those described in this section may be implemented to identify storage areas in which to place packets being transferred from a network to a host computer without exceeding the scope of the invention.

Methods of Transferring a Packet into a Memory Buffer by a DMA Engine

FIGS. **13–20** are flow charts describing procedures for transferring a packet into a host memory buffer in these procedures, a packet's operation code helps determine

NOAC Ex. 1015 Page 545

which buffer or buffers the packet is stored in. An illustrative selection of operation codes that may be used in this procedure are listed and explained in TABLE 1.

The illustrated embodiments of the invention employ four categories of host memory buffers, the sizes of which are programmable. The buffer sizes are programmable in order to accommodate various host platforms, but are programmed to be one memory page in size in present embodiments in order to enhance the efficiency of handling and processing network traffic. For example, the embodiments discussed in this section are directed to the use of a host computer system employing a SPARC™ processor, and so each buffer is eight kilobytes in size. These embodiments are easily adjusted, however, for host computer systems employing memory pages having other dimensions.

One type of buffer is for re-assembling data from a flow, another type is for headers of packets being re-assembled and for small packets (e.g., those less than or equal to 256 bytes in size) that are not re-assembled. A third type of buffer stores packets up to MTU size (e.g., 1522 bytes) that are not re-assembled, and a fourth type stores jumbo packets that are greater than MTU size and which are not re-assembled. These buffers are called flow re-assembly, header, MTU and jumbo buffers, respectively.

The procedures described in this section make use of free descriptors and completion descriptors as depicted in FIG. **12**. In particular, in these procedures free descriptors retrieved from a free descriptor ring store buffer identifiers (e.g., memory addresses, pointers) for identifying buffers in which to store a portion of a packet. A used buffer may be returned to a host computer by identifying the location within a free buffer array or other data structure used to store the buffer's buffer identifier. One skilled in the art will recognize that these procedures may be readily adapted to work with alternative methods of obtaining and returning buffers for storing packets.

FIG. **13** is a top-level view of the logic controlling DMA engine **120** in this embodiment of the invention. State **1300** is a start state.

In state **1302**, a packet is stored in packet queue **116** and associated information is stored in control queue **118**. One embodiment of a packet queue is depicted in FIG. **8** and one embodiment of a control queue is depicted in FIG. **9**. DMA engine **120** may detect the existence of a packet in packet queue **116** by comparing the queue's read and write pointers. As long as they do not reference the same entry, then it is understood that a packet is stored in the queue. Alternatively, DMA engine **120** may examine control queue **118** to determine whether an entry exists there, which would indicate that a packet is stored in packet queue **116**. As long as the control queue's read and write pointers do not reference the same entry, then an entry is stored in the control queue and a packet must be stored in the packet queue.

In state **1304**, the packet's associated entry in the control queue is read. Illustratively, the control queue entry includes the packet's operation code, the status of the packet's No_Assist signal (e.g., indicating whether or not the packet is compatible with a pre-selected protocol), one or more indicators concerning the size of the packet (and/or its data portion), etc.

In state **1306**, DMA engine **120** retrieves the packet's flow number. As described previously, a packet's flow number is the index of the packet's flow in flow database **110**. A packet's flow number may, as described in a following section, be provided to and used by dynamic packet batching module **122** to enable the collective processing of headers from related packets. In one embodiment of the invention, a packet's flow number may be provided to any of a number of NIC modules (e.g., IPP module **104**, packet batching module **122**, DMA engine **120**, control queue **118**) after being generated by flow database manager **108**. The flow number may also be stored in a separate data structure (e.g., a register) until needed by dynamic packet batching module **122** and/or DMA engine **120**. In one embodiment of the invention DMA engine **120** retrieves a packet's flow number from dynamic packet batching module **122**. In an alternative embodiment of the invention, the flow number may be retrieved from a different location or module.

Then, in states **1308–1318**, DMA engine **120** determines the appropriate manner of processing the packet by examining the packet's operation code. The operation code may, for example, indicate which buffer the engine should transfer the packet into and whether a flow is to be set up or torn down in flow re-assembly buffer table **1004**.

The illustrated procedure continues at state **1400** (FIG. **14**) if the operation code is **0**, state **1500** (FIG. **15**) for operation code **1**, state **1600** (FIG. **16**) for operation code **2**, state **1700** (FIG. **17**) for operation code **3**, state **1800** (FIG. **18**) for operation code **4**, state **1900** (FIG. **19**) for operation code **5** and state **2000** (FIG. **20**) for operation codes **6** and **7**.

A Method of Transferring a Packet with Operation **0**

FIG. **14** depicts an illustrative procedure in which DMA engine **120** transfers a packet associated with operation code **0** to a host memory buffer. As reflected in TABLE 1, operation code **0** indicates in this embodiment that the packet is compatible with the protocols that may be parsed by NIC **100**. As explained above, compatible packets are eligible for re-assembly, such that data from multiple packets of one flow may be stored in one buffer that can then be efficiently provided (e.g., via a page-flip) to a user or program's memory space. Packets having operation code **0**, however, are small and contain no flow data for re-assembly. They are thus likely to be control packets. Therefore, no new flow is set up, no existing flow is torn down and the entire packet may be placed in a header buffer.

In state **1400**, DMA engine **120** (e.g., DMA manager **1002**) determines whether there is a valid (e.g., active) header buffer. Illustratively, this determination is made by examining validity indicator **1116** of header buffer table **1006**, which manages the active header buffer. If the validity indicator is set (e.g., equal to one), then there is a header buffer ready to receive this packet and the procedure continues at state **1404**.

Otherwise, in state **1402** a header buffer is prepared or initialized for storing small packets (e.g., packets less than 257 bytes in size) and headers of re-assembled packets (and, possibly, headers of other packets—such as jumbo packets). In the illustrated embodiment, this initialization process involves obtaining a free ring descriptor and retrieving its buffer identifier (e.g., its reference to an available host memory buffer). The buffer identifier may then be stored in a data structure such as free buffer array **1210** (shown in FIG. **12A**). As described above, in one embodiment of the invention free ring manager **1012** maintains a cache of descriptors referencing empty buffers. Thus, a descriptor may be retrieved from this cache and its buffer allocated to header buffer table **1006**. If the cache is empty, new descriptors may be retrieved from a free descriptor ring in host memory to replenish the cache.

When a new buffer identifier is retrieved from the cache or from the free descriptor ring, the buffer identifier's position in the free buffer array is placed in header buffer index **1112** of header buffer table **1006**. Further, an initial

storage location in the buffer identifier (e.g., its starting address) is stored in next address field **1114** and validity indicator **1116** is set to a valid state.

In state **1404**, the packet is copied or transferred (e.g., via a DMA operation) into the header buffer at the address or location specified in the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes are inserted before the packet in order to align the beginning of the packet's layer three (e.g., IP) header with a sixteen-byte boundary. In addition, a header buffer may be logically partitioned into cells of predetermined size (e.g., 256 bytes), in which case the packet or padding may begin at a cell boundary.

In state **1406**, a completion descriptor is written or configured to provide information to the host computer (e.g., a software driver) for processing the packet. In particular, the header buffer index (e.g. the index within the free buffer array of the buffer identifier that references the header buffer) and the packet's offset in the header buffer are placed in the descriptor. Illustratively, the offset may identify the location of the cell in which the header is stored, or it may identify the first byte of the packet. The size of the packet is also stored in the descriptor, illustratively within a header size field. A data size field within the descriptor is set to zero to indicate that the entire packet was placed in the header buffer (e.g., there was no data portion to store in a separate data buffer). A release header flag is set in the descriptor if the header buffer is full. However, the header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release header flag may be set (or cleared) at that time.

As described in a later section, in one embodiment of the invention a release flow flag may also be set, depending upon dynamic packet batching module **122**. For example, if the packet batching module determines that another packet in the same flow will soon be transferred to the host computer, the release flow flag will be cleared (e.g., a zero will be stored). This indicates that the host computer should wait the next flow packet before processing this one. Then, by collectively processing multiple packets from a single flow, the packets can be processed more efficiently while requiring less processor time.

In the descriptor type field, a value is stored to indicate that a flow packet was transferred to host memory. Also, a predetermined value (e.g., zero) is stored in the ownership indicator field to indicate that DMA engine **120** is done using the descriptor and/or is releasing a packet to the host computer. Illustratively, the host computer will detect the change in the ownership indicator (e.g., from one to zero) and use the stored information to process the packet. In one alternative embodiment of the invention, DMA engine **120** sends an interrupt or other signal to alert the host computer that a descriptor is being released. In another alternative embodiment, the host computer polls the NIC to determine whether a packet has been received and/or transferred. In yet another alternative embodiment, the descriptor type field is used to inform the host computer that the DMA engine is releasing a descriptor. In this alternative embodiment, when a non-zero value is placed in the descriptor type field the host computer may understand that the DMA engine is releasing the descriptor.

In a present embodiment of the invention, the ownership indicator field is not changed until DMA engine **120** is finished with any other processing involving this packet or header, such as making all entries in the descriptor. For example, as described below a header buffer or other buffer may be determined to be full at some time. after state **1406**. By delaying the setting of the ownership indicator, a release header flag can be set before the descriptor is reclaimed by the host computer, thus avoiding the use of another descriptor.

In state **1408**, it is determined whether the header buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, up to thirty-two entries may be stored in a header buffer. Thus, a counter may be used to keep track of entries placed in each new header buffer and the buffer can be considered full when thirty-two entries are stored. Other methods of determining whether a buffer is full are also suitable. For example, after a packet is stored in the header buffer a new next address field may be calculated and the difference between the new next address field and the initial address of the buffer may be compared to the size of the buffer (e.g., eight kilobytes). If less than a predetermined number of bytes (e.g., 256) are unused, the buffer may be considered full.

If the buffer is full, in state **1410** the header buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer via a descriptor. In this embodiment of the invention a release header flag in the descriptor is set. If the descriptor that was written in state **1406** was already released (e.g., its ownership indicator field changed), another descriptor may be used in this state. If another descriptor is used simply to report a full header buffer, the descriptor's header size and data size fields may be set to zero to indicate that no new packet was transferred with this descriptor.

If the header buffer is not full, then in state **1412** the next address field of header buffer table **1006** is updated to indicate the address at which to store the next header or small packet. The processing associated with a packet having operation code **0** then ends with end state **1499**. In one embodiment of the invention, the ownership indicator field of a descriptor that is written in state **1406** is not changed, or an interrupt is not issued, until end state **1499**. Delaying the notification of the host computer allows the descriptor to be updated or modified for as long as possible before turning it over to the host.

A Method of Transferring a Packet with Operation Code **1**

FIG. **15** depicts an illustrative procedure in which DMA engine **120** transfers a packet associated with operation code **1** to a host memory buffer. As reflected in TABLE 1, in this embodiment operation code **1** indicates that the packet is compatible with the protocols that may be parsed by NIC **100**. A packet having operation code **1**, however, may be a control packet having a particular flag set. No new flow is set up, but a flow should already exist and is to be torn down; there is no data to re-assemble and the entire packet may be stored in a header buffer.

In state **1500**, DMA engine **120** (e.g., DMA manager **1002**) determines whether there is a valid (e.g., active) header buffer. Illustratively, this determination is made by examining validity indicator **1116** of header buffer table **1006**, which manages the active header buffer. If the validity indicator is set, then there is a header buffer ready to receive this packet and the procedure continues at state **1504**.

Otherwise, in state **1502** a new header buffer is prepared or initialized for storing small packets and headers of re-assembled packets. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

NOAC Ex. 1015 Page 547

When a new descriptor is obtained from the cache or from the free descriptor ring, its buffer identifier (e.g., pointer, address, index) is stored in free buffer array **1210** and its initial storage location (e.g., address or cell location) is stored in next address field **1114** of header buffer table **1006**. The index or position of the buffer identifier within the free buffer array is stored in header buffer index **1112**. Finally, validity indicator **1116** is set to a valid state.

In state **1504** the packet is copied into the header buffer at the address or location specified in the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes are inserted before the packet in order to align the beginning of the packet's layer three (e.g., IP) header with a sixteen-byte boundary. And, the packet (with or without padding) may be placed into a pre-defined area or cell of the buffer.

In the illustrated embodiment, operation code **1** indicates that the packet's existing flow is to be torn down. Thus, in state **1506** it is determined whether a flow re-assembly buffer is valid (e.g., active) for this flow by examining the flow's validity indicator in flow re-assembly buffer table **1004**. If, for example, the indicator is valid, then there is an active buffer storing data from one or more packets in this flow. Illustratively, the flow is torn down by invalidating the flow re-assembly buffer and releasing it to the host computer. If there is no valid flow re-assembly buffer for this flow, the illustrated procedure continues at state **1512**. Otherwise, the procedure proceeds to state **1508**.

In state **1508**, a completion descriptor is configured to release the flow's re-assembly buffer and to provide information to the host computer for processing the current packet. In particular, the header buffer index and the offset the first byte of the packet (or location of the packet's cell) within the header buffer are placed in the descriptor. The index within the free buffer array of the entry containing the re-assembly buffer's buffer identifier is stored in a data index field of the descriptor. The size of the packet is stored in a buffer size field and a data size field is set to zero to indicate that no separate buffer was used for storing this packet's data. A release header flag is set in the descriptor if the header buffer is full and a release data flag is set to indicate that no more data will be placed in this flow's present re-assembly buffer (e.g., it is being released). In addition, a tear down flow flag is set to indicate that DMA engine **120** is tearing down the packet's flow. The header buffer may not be full to see if it is full until a later state of this procedure. Thus, in an embodiment of the invention, the release header flag may be set at that time.

In state **1510**, the flow's entry in flow re-assembly buffer table **1004** is invalidated. After state **1510**, the procedure proceeds at state **1514**.

In state **1512**, a completion descriptor is configured with information somewhat different than that of state **1508**. In particular, the header buffer index, the offset to this packet in the header buffer and the packet size are placed in the same descriptor fields as above. The data size field is zero, as above, but no data index needs to be stored because a release data flag is set (e.g., because there is no flow re-assembly buffer to release). A release header flag is still set in the descriptor if the header buffer is full and a release flag is again set to indicate that DMA engine **120** is tearing down the packet's flow. Also, the descriptor type is configured to a value indicating that DMA engine **120** is transferring a flow packet into host memory.

In state **1514**, it is determined whether the header buffer are no larger than 256 bytes, a counter is used to keep track of entries placed into each new header buffer. The buffer is considered full when thirty-two entries are stored.

If the buffer is full, in state **1516** the header buffer is invalidated. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer via the descriptor configured in state **1508** or state **1512**. In this embodiment of the invention a release header flag in the descriptor is set to indicate that the header buffer is full.

If the header buffer is not full, then in state **1518** the next address field of header buffer table **1006** is updated to indicate the address at which to store the next header or small packet.

The processing associated with a packet having operation code **1** then ends with end state **1599**. In this end state, the descriptor used for this packet is turned over to the host computer by changing its ownership indicator field (e.g., from one to zero), issuing an interrupt, or some other mechanism.

One skilled in the art will appreciate that in an alternative embodiment of the invention a change in the descriptor type field to any value other than the value (e.g., zero) it had when DMA engine **120** was using it, may constitute a surrender of "ownership" of the descriptor to the host computer or software operating on the host computer. The host computer will detect the change in the descriptor type field and subsequently use the stored information to process the packet.

A Method of Transferring a Packet with Operation Code **2**

FIGS. **16A–16F** illustrate a procedure in which DMA engine **120** transfers a packet associated with operation code **2** to a host memory buffer. As reflected in TABLE 1, operation code **2** may indicate that the packet is compatible with the protocols that may be parsed by NIC **100**, but that it is out of sequence with another packet in the same flow. It may also indicate an attempt to re-establish a flow, but that no more data is likely to be received after this packet. For operation code **2**, no new flow is set up and any existing flow with the packet's flow number is to be torn down. The packet's data is not to be re-assembled with data from other packets in the same flow.

Because an existing flow is to be torn down (e.g., the flow's re-assembly buffer is to be invalidated and released to the host computer), in state **1600** it is determined whether a flow re-assembly buffer is valid (e.g., active) for the flow having the flow number that was read in state **1306**. This determination may be made by examining the validity indicator in the flow's entry in flow re-assembly buffer table **1004**. Illustratively, if the indicator is valid then there is an active buffer storing data from one or more packets in the flow. If there is a valid flow re-assembly buffer for this flow, the illustrated procedure continues at state **1602**. Otherwise, the procedure proceeds to state **1606**.

In state **1602**, a completion descriptor is written or configured to release the existing flow re-assembly buffer. In particular, the flow re-assembly buffer's index (e.g., the location within the free buffer array that contains the buffer identifier corresponding to the flow re-assembly buffer) is written to the descriptor. In this embodiment of the invention, no offset needs to be stored in the descriptor's data offset field and the data size field may be set to zero because no new data was stored in the re-assembly buffer. Similarly, the header buffer is not yet being released, therefore the header index and header offset fields of the descrip-

Illustratively, the descriptor's release header flag is cleared (e.g., a zero is stored in the flag) because the header buffer is not to be released. The release data flag is set (e.g., a one is stored in the flag), however, because no more data will be placed in the released flow re-assembly buffer. Further, a release flow flag in the descriptor is also set, to indicate that the flow associated with the released flow re-assembly buffer is being torn down.

The descriptor type field may be changed to a value indicating that DMA engine **120** is releasing a stale flow buffer (e.g., a flow re-assembly buffer that has not been used for some time). Finally, the descriptor is turned over to the host computer by changing its ownership indicator field or by issuing an interrupt or using some other mechanism. In one embodiment of the invention, however, the descriptor is not released to the host computer until end state **1699**.

Then, in state **1604**, the flow re-assembly buffer is invalidated by modifying validity indicator **1106** in the flow's entry in flow re-assembly buffer table **1004** appropriately.

In state **1606**, it is determined whether the present packet is a small packet (e.g., less than or equal to 256 bytes in size), suitable for storage in a header buffer. If so, the illustrated procedure proceeds to state **1610**. Information stored in packet queue **116** and/or control queue **118** may be used to make this determination.

In state **1608**, it is determined whether the present packet is a jumbo packet (e.g., greater than 1522 bytes in size), such that it should be stored in a jumbo buffer. If so, the illustrated procedure proceeds to state **1650**. If not, the procedure continues at state **1630**.

In state **1610** (reached from state **1606**), it has been determined that the present packet is a small packet suitable for storage in a header buffer. Therefore, DMA engine **120** (e.g., DMA manager **1002**) determines whether there is a valid (e.g., active) header buffer. Illustratively, this determination is made by examining validity indicator **1116** of header buffer table **1006**, which manages the active header buffer. If the validity indicator is set, then there should be a header buffer ready to receive this packet and the procedure continues at state **1614**.

Otherwise, in state **1612** a new header buffer is prepared or initialized for storing small packets and headers of re-assembled packets. This initialization process may involve obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or some other indicator of the first storage location in the buffer is placed in next address field **1114** of header buffer table **1006**. The buffer identifier's position or index within the free buffer array is stored in header buffer index **1112**, and validity indicator **1116** is set to a valid state.

In state **1614** the packet is copied or transferred (e.g., via DMA operation) into the header buffer at the address or location specified in the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes are inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In addition, the packet may be positioned within a cell of predetermined size (e.g., 256 bytes) within the header buffer.

In state **1616**, a completion descriptor is written or configured to provide necessary information to the host computer (e.g., a software driver) for processing the packet. In particular, the header buffer index (e.g. the position within the free buffer array of the header buffer's buffer identifier) and the packet's offset within the header buffer are placed in the descriptor. Illustratively, this offset may serve to identify the first byte of the packet, the first pad byte before the packet or the beginning of the packet's cell within the buffer. The size of the packet is also stored in the descriptor in a header size field. A data size field within the descriptor may be set to zero to indicate that the entire packet was placed in the header buffer (e.g., no separate data portion was stored). A release header flag is set in the descriptor if the header buffer is full. However, the header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release header flag may be set (or cleared) at that time. A release data flag is cleared (e.g., set to a value of zero), because there is no separate data portion being conveyed to the host computer.

Also, the descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. And, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention the ownership indicator field is not changed until end state **1699** below. In one alternative embodiment of the invention, DMA engine **120** issues an interrupt or other signal to alert the host computer that a descriptor is being released.

In state **1618**, it is determined whether the header buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, a counter may be used to keep track of entries placed into each new header buffer. The buffer is considered full when thirty-two entries are stored.

If the buffer is full, in state **1620** the header buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to an invalid state and communicating this status to the host computer. In this embodiment of the invention, a release header flag in the descriptor is set. The illustrated procedure then ends with end state **1699**.

If the header buffer is not full, then in state **1622** the next address field of header buffer table **1006** is updated to indicate the address or cell boundary at which to store the next header or small packet. The illustrated procedure then ends with end state **1699**.

In state **1630** (reached from state **1608**), it has been determined that the packet is not a small packet or a jumbo packet. The packet may, therefore, be stored in a non-re-assembly buffer (e.g., an MTU buffer) used to store packets that are up to MTU in size (e.g., 1522 bytes). Thus, in state **1630** DMA engine **120** determines whether a valid (e.g., active) MTU buffer exists. Illustratively, this determination is made by examining validity indicator **1126** of MTU buffer table **1008**, which manages an active MTU buffer. If the validity indicator is set, then there is an MTU buffer ready to receive this packet and the procedure continues at state **1634**.

Otherwise, in state **1632** a new MTU buffer is prepared or initialized for storing non-re-assembleable packets up to 1522 bytes in size. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer (e.g., a buffer identifier). If the

cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in the free buffer array. The buffer's initial address or some other indication of the first storage location in the buffer is placed in next address field **1124** of MTU buffer table **1008**. Further, the position of the buffer identifier within the free buffer array is stored in MTU buffer index **1122** and validity indicator **1126** is set to a valid state.

In state **1634** the packet is copied or transferred (e.g., via a DMA operation) into the MTU buffer at the address or location specified in the next address field. As described above, in one embodiment of the invention pad bytes may be inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In another embodiment of the invention packets may be aligned in an MTU buffer in cells of predefined size (e.g., two kilobytes), similar to entries in a header buffer.

In state **1636**, a completion descriptor is written or configured to provide necessary information to the host computer (e.g., a software driver) for processing the packet. In particular, the MTU buffer index (e.g. the free buffer array element that contains the buffer identifier for the MTU buffer) and offset (e.g., the offset of the first byte of this packet within the MTU buffer) are placed in the descriptor in data index and data offset fields, respectively. The size of the packet is also stored in the descriptor, illustratively within a data size field. A header size field within the descriptor is set to zero to indicate that the entire packet was placed in the MTU buffer (e.g., no separate header portion was stored in a header buffer). A release data flag is set in the descriptor if the MTU buffer is full. However, the MTU buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release data flag may be set (or cleared) at that time. A release header flag is cleared (e.g., set to zero), because there is no separate header portion being conveyed to the host computer.

Further, the descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. Also, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In a present embodiment of the invention the ownership field is not set until end state **1699** below. In one alternative embodiment of the invention, DMA engine **120** issues an interrupt or other signal to alert the host computer that a descriptor is being released, or communicates this event to the host computer through the descriptor type field.

In state **1638**, it is determined whether the MTU buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the MTU buffer are allocated two kilobytes, a counter may be used to keep track of entries placed into each new header buffer. The buffer may be considered full when a predetermined number of entries (e.g., four) are stored. In an alternative embodiment of the invention DMA engine **120** determines how much storage space within the buffer has yet to be used. If no space remains, or if less than a predetermined amount of space is available, the buffer may be considered full.

If the MTU buffer is full, in state **1640** it is invalidated to ensure that it is not used again. Illustratively, this involves setting the MTU buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention, a release data flag in the descriptor is set. The illustrated procedure then ends with end state **1699**.

If the MTU buffer is not full, then in state **1642** the next address field of MTU buffer table **1008** is updated to indicate the address or location (e.g., cell boundary) at which to store the next packet. The illustrated procedure then ends with end state **1699**.

In state **1650** (reached from state **1608**), it has been determined that the packet is a jumbo packet (e.g., that it is greater than 1522 bytes in size). In this embodiment of the invention jumbo packets are stored in jumbo buffers and, if splitting of jumbo packets is enabled (e.g., as determined in state **1654** below), headers of jumbo packets are stored in a header buffer. DMA engine **120** determines whether a valid (e.g., active) jumbo buffer exists. Illustratively, this a determination is made by examining validity indicator **1136** of jumbo buffer table **1010**, which manages the active jumbo buffer. If the validity indicator is set, then there is a jumbo buffer ready to receive this packet and the procedure continues at state **1654**. As explained above, a jumbo buffer table may not be used in an. embodiment of the invention in which a jumbo buffer is used only once (e.g., to store just one, or just part of one, jumbo packet).

Otherwise, in state **1652** a new jumbo buffer is prepared or initialized for storing a non-re-assembleable packet that is larger than 1522 bytes. This initialization process may involve obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer (e.g., a buffer identifier). If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, its buffer identifier (e.g., pointer, address, index) is stored in a free buffer array (or other data structure). The buffer's initial address or other indication of the first storage location in the buffer is placed in next address field **1134** of jumbo buffer table **1010**. Also, the location of the buffer identifier within the free buffer array is stored in jumbo buffer index **1132** and validity indicator **1136** is set to a valid state.

Then, in state **1654** DMA engine **120** determines whether splitting of jumbo buffers is enabled. If enabled, the header of a jumbo packet is stored in a header buffer while the packet's data is stored in one or more jumbo buffers. If not enabled, the entire packet will be stored in one or more jumbo buffers. Illustratively, splitting of jumbo packets is enabled or disabled according to the configuration of a programmable indicator (e.g., flag, bit, register) that may be set by software operating on the host computer (e.g., a device driver). If splitting is enabled, the illustrated procedure continues at state **1670**. Otherwise, the procedure continues with state **1656**.

In state **1656**, DMA engine **120** determines whether the packet will fit into one jumbo buffer. For example, in an embodiment of the invention using eight kilobyte pages, if the packet is larger than eight kilobytes a second jumbo buffer will be needed to store the additional contents. If the packet is too large, the illustrated procedure continues at state **1662**.

In state **1658**, the packet is copied or transferred (e.g., via a DMA operation) into the current jumbo buffer, at the location specified in the next address field **1134** of jumbo buffer table **1010**. When the packet is transferred intact like this, padding may be added to align a header portion of the

ket with a sixteen-byte boundary. One skilled in the art ll appreciate that the next address field may not need to be dated to account for this new packet because the jumbo fer will be released. In other words, in one embodiment he invention a jumbo buffer may be used just once (e.g., tore one packet or a portion of one packet).

n state **1660**, a completion descriptor is written or con- red to release the jumbo buffer and to provide informa- to the host computer for processing the packet. The bo buffer index (e.g., the position within the free buffer y of the buffer identifier for the jumbo buffer) and the t of the packet within the jumbo buffer are placed in the riptor. Illustratively, these values are stored in data x and data offset fields, respectively. The size of the et (e.g., the packet length) may be stored in a data size

header size field is cleared (e.g., a zero is stored) to ate that the header buffer was not used (e.g., the header ot stored separately from the packet's data). Because is no separate packet header, header index and header fields are not used or are set to zero (e.g., the values in their fields do not matter). A release header flag is d and a release data flag is set to indicate that no more ill be placed in this jumbo buffer (e.g., because it is eleased).

o, the descriptor type field is changed to a value ting that DMA engine **120** transferred a non-re- bleable packet into host memory. And, a predeter- value (e.g., zero) is stored in the descriptor's own- indicator field to indicate that DMA engine **120** is g a packet to the host computer and turning over hip of the descriptor. In an alternative embodiment, riptor may be released by issuing an interrupt or lert. In yet another embodiment, changing the tor type field (e.g., to a non-zero value) may signal ase of the descriptor. In one embodiment of the n the ownership indicator is not set until end state low. After state **1660**, the illustrated procedure at state **1668**.

1662, a first portion of the packet is stored in the g., valid) jumbo buffer, at the location identified in s next address field **1134**. Then, because the full ll not fit into this buffer, in state **1664** a new jumbo repared and the remainder of the packet is stored ffer.

1666, a completion descriptor is written or con- e contents are similar to those described in state is descriptor must reflect that two jumbo buffers to store the packet.

jumbo buffer index (e.g., the index, within the array, of the buffer identifier that identifies the r) and the offset of the packet within the first r are placed in the descriptor, as above. The size et (e.g., the packet length) is stored in a data size

size field is cleared (e.g., a zero is stored) to t the header buffer was not used (e.g., the header red separately from the packet's data). Because eparate packet header, header index and header re not used (e.g., the values stored in their fields r).

header flag is cleared and a release data flag is te that no more data will be placed in these rs (e.g., because they are being released). t packet flag is set to reflect the use of a second and the index (within the free buffer array) of tifier for the second buffer is stored in a next

Further, the descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-reassembleable packet into host memory. Finally, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field, or some other mechanism is employed, to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention, the descriptor is not released to the host computer until end state **1699** below.

In state **1668**, the jumbo buffer entry or entries in jumbo buffer table **1010** are invalidated (e.g., validity indicator **1136** is set to invalid) to ensure that they are not used again. In the procedure described above a jumbo packet was stored in, at most, two jumbo buffers. In an alternative embodiment of the invention a jumbo buffer may be stored across any number of buffers. The descriptor(s) configured to report the transfer of such a packet is/are constructed accordingly, as will be obvious to one skilled in the art.

After state **1668**, the illustrated procedure ends with end state **1699**.

In state **1670** (reached from state **1654**), it has been determined that the present jumbo packet will be split to store the packet header in a header buffer and the packet data in one or more jumbo buffers. Therefore, DMA engine **120** (e.g., DMA manager **1002**) determines whether there is a valid (e.g., active) header buffer. Illustratively, this determination is made by examining validity indicator **1116** of header buffer table **1006**, which manages the active header buffer. If the validity indicator is set, then there is a header buffer ready to receive this packet and the procedure continues at state **1674**.

Otherwise, in state **1672** a new header buffer is prepared or initialized for storing small packets and headers of other packets. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or some other indication of the first storage location or cell in the buffer is placed in next address field **1114** of header buffer table **1006**. Also, the index of the buffer identifier within the free buffer array is stored in header buffer index **1112** and validity indicator **1116** is set to a valid state.

In state **1674** the packet's header is copied or transferred (e.g., via a DMA operation) into the header buffer at the, address or location specified in the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes are inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In addition, the packet's header may be positioned within a cell of predetermined size (e.g., 256 bytes) within the buffer.

In state **1676**, DMA engine **120** determines whether the packet's data (e.g., the TCP payload) will fit into one jumbo buffer. If the packet is too large, the illustrated procedure continues at state **1682**.

In state **1678**, the packet's data is copied or transferred (e.g., via a DMA operation) into the current jumbo buffer, at the location specified in the next address field **1134** of jumbo buffer table **1010**. One skilled in the art will appreciate that

NOAC Ex. 1015 Page 551

the next address field may not need to be updated to account for this new packet because the jumbo buffer will be released. In other words, in one embodiment of the invention a jumbo buffer may be used just once (e.g., to store one packet or a portion of one packet).

In state **1680**, a completion descriptor is written or configured to release the jumbo buffer and to provide information to the host computer for processing the packet. The header buffer index (e.g. the index of the header buffer's buffer identifier within the free buffer array) and offset of the packet's header within the buffer are placed in the descriptor in header index and header offset fields, respectively. Illustratively, this offset may serve to identify the first byte of the header, the first pad byte before the header or the location of the cell in which the header is stored. The jumbo buffer index (e.g., the position or index within the free buffer array of the buffer identifier that identifies the jumbo buffer) and the offset of the first byte of the packet's data within the jumbo buffer are placed in data index and data offset fields, respectively. Header size and data size fields are used to store the size of the packet's header (e.g., the offset of the payload within the packet) and data (e.g., payload size), respectively.

A release header flag is set in the descriptor if the header buffer is full. However, the header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release header flag may be set (or cleared) at that time. A release data flag is also set, because no more data will be placed in the jumbo buffer (e.g., it is being released to the host computer).

The descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. Also, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention the ownership indicator is not changed until end state **1699** below. In an alternative embodiment, the descriptor may be released by issuing an interrupt or other alert. In yet another alternative embodiment, changing the descriptor type value may signal release of the descriptor.

After state **1680**, the illustrated procedure proceeds to state **1688**.

In state **1682**, a first portion of the packet's data is stored in the present (e.g., valid) jumbo buffer, at the location identified in the buffer's next address field **1134**.

Because all of the packet's data will not fit into this buffer, in state **1684** a new jumbo buffer is prepared and the remainder of the packet is stored in that buffer.

In state **1686**, a completion descriptor is written or configured. The contents are similar to those described in states above but this descriptor must reflect that two jumbo buffers were used to store the packet. The header buffer index (e.g. the index of the free buffer array element containing the header buffer's buffer identifier) and offset (e.g., the location of the packet's header within the header buffer) are placed in the descriptor in header index and header offset fields, respectively. The jumbo buffer index (e.g., the index, within the free buffer array, of the buffer identifier that references the jumbo buffer) and the offset of the first byte of the packet's data within the jumbo buffer are placed in data index and data offset fields, respectively. Header size and data size fields are used to store the size of the packet's header (e.g., as measured by the offset of the packet's header from the start of the packet) and data (e.g., payload size), respectively.

A release header flag is set in the descriptor if the header buffer is full. However, the header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release header flag may be set (or cleared) at that time. A release data flag is also set, because no more data will be placed in the jumbo buffer (e.g., it is being released to the host computer). Further, a split packet flag is set to indicate that a second jumbo buffer was used, and the location (within the free buffer array or other data structure) of the second buffer's buffer identifier is stored in a next index field

The descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. Finally, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention the ownership indicator is not changed until end state **1699** below.

In state **1688**, the jumbo buffer's entry in jumbo buffer table **1010** is invalidated (e.g., validity indicator **1136** is set to invalid) to ensure that it is not used again. In the procedure described above, a jumbo packet was stored in, at most, two jumbo buffers. In an alternative embodiment of the invention a jumbo packet may be stored across any number of buffers. The descriptor that is configured to report the transfer of such a packet is constructed accordingly, as will be obvious to one skilled in the art.

In state **1690**, it is determined whether the header buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, a counter may be used to keep track of entries placed into each new header buffer. The buffer may be considered full when thirty-two entries are stored.

If the buffer is full, in state **1692** the header buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention, a release header flag in the descriptor is set. The illustrated procedure then ends with end state **1699**.

If the header buffer is not full, then in state **1694** the next address field of header buffer table **1006** is updated to indicate the address at which to store the next header or small packet. The illustrated procedure then ends with end state **1699**.

In end state **1699**, a descriptor may be turned over to the host computer by changing a value in the descriptor's descriptor type field (e.g., from one to zero), as described above. Illustratively, the host computer (or software operating on the host computer) detects the change and understands that DMA engine **120** is returning ownership of the descriptor to the host computer.

A Mehtod of Transferring a Packet with Operation Code 3

FIGS. **17A–17** C illustrate one procedure in which DMA engine **120** transfers a packet associated with operation code **3** to a host memory buffer. As reflected in TABLE 1, operation code **3** may indicate that the packet is compatible with a protocol that can be parsed by NIC **100** and that it carries a final portion of data for its flow. No new flow is set up, but a flow should already exist and is to be torn down. The packet's data is to be re-assembled with data from previous flow packets. Because the packet is to be re-assembled, the packet's header should be stored in a header buffer and its data in the flow's re-assembly buffer. The flow's active re-assembly buffer may be identified by the flow's entry in flow re-assembly buffer table **1004**.

In state **1700**, DMA engine **120** (e.g., DMA manager 002) determines whether there is a valid (e.g., active) eader buffer. Illustratively, this determination is made by xamining validity indicator **1116** of header buffer table 006, which manages the active header buffer. If the validity dicator is set (e.g., equal to one), then it is assumed that ere is a header buffer ready to receive this packet and the ocedure continues at state **1704**.

Otherwise, in state **1702** a new header buffer is prepared initialized for storing small packets and headers of assembled packets. This initialization process may volve obtaining a free ring descriptor from a cache main- ed by free ring manager **1012** and retrieving its buffer ntifier (e.g., a reference to an available memory buffer). he cache is empty, new descriptors may be retrieved from free descriptor ring in host memory to replenish the he.

llustratively, when a new descriptor is obtained from the e or from the free descriptor ring, the buffer identifier pointer, address, index) contained in the descriptor is d in a free buffer array. The buffer's initial address or e other indication of the first storage location or cell in uffer is placed in next address field **1114** of header r table **1006**. Further, the index of the buffer identifier n the free buffer array is stored in header buffer index and validity indicator **1116** is set to a valid state.

state **1704** the packet's header is copied or transferred e header buffer at the address or location specified in xt address field of header buffer table **1006**. As ed above, in one embodiment of the invention pad may be inserted before the header in order to align the ing of the packet's layer three protocol (e.g., IP) with a sixteen-byte boundary. In addition, the header e positioned within a cell of predetermined size (e.g., tes) within the header buffer.

he illustrated embodiment, operation code 3 indicates , existing flow is to be torn down (e.g., the flow mbly buffer is to be invalidated and released to the mputer). Thus, in state **1706** it is determined whether e-assembly buffer is valid (e.g., active) for this flow mining the validity indicator in the flow's entry in assembly buffer table **1004**. Illustratively, if the r is valid then there should be an active buffer ata from one or more packets in this flow. If there flow re-assembly buffer for this flow, the illustrated e continues at state **1712**. Otherwise, the procedure to state **1708**.

1708, a new flow re-assembly buffer is prepared his packet's data. Illustratively, a free ring descrip- ined from a cache maintained by free ring manager its reference to an empty buffer is retrieved. If the mpty, new descriptors may be retrieved from the iptor ring in host memory to replenish the cache. new descriptor is obtained from the cache or from descriptor ring, the buffer identifier (e.g., pointer, ndex) contained in the descriptor is stored in a free y. The buffer's initial address or other indication storage location is placed in next address field he flow's entry in flow re-assembly buffer table flow's entry in the re-assembly buffer table may ized by its flow number. The location within the r array of the buffer identifier is stored in buffer index **1102**, and validity indicator **1106** is id state.

1710, the packet's data is copied or transferred DMA operation) into the address or location the next address field of the flow's entry in flow buffer table **1004**.

In state **1712**, a completion descriptor is written or configured to release the flow's re-assembly buffer and to provide information to the host computer for processing the packet. In particular, the header buffer index (e.g., the index, within the free buffer array, of the header buffer's identifier) and the offset of the packet's header within the header buffer are placed in the descriptor. Illustratively, this offset serves to identify the first byte of the header, the first pad byte preceding the header or the cell in which the header is stored. The flow re-assembly buffer index (e.g., the index, within the free buffer array, of the flow re-assembly buffer's identifier) and the offset of the packet's data within that buffer are also stored in the descriptor.

The size of the packet's data (e.g., the size of the packet's TCP payload) and header (e.g., the offset of the TCP payload within the packet) portions are stored in data size and header size fields, respectively. The descriptor type field is given a value that indicates that a flow packet has been transferred to host memory. A release header flag may be set if the header buffer is full and a release data flag may be set to indicate that no more data will be placed in this flow re-assembly buffer (e.g., because it is being released). In addition, a release flow flag is set to indicate that DMA engine **120** is tearing down the packet's flow. The header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment, the release header flag may be set (or cleared) at that time.

Then, in state **1714**, the flow re-assembly buffer is invalidated by modifying validity indicator **1106** in the flow's entry in flow re-assembly buffer table **1004** appropriately. After state **1714**, the procedure continues at state **1730**.

In state **1716**, DMA engine **120** determines whether the packet's TCP payload (e.g., the packet's data portion) will fit into the valid flow re-assembly buffer. If not, the illustrated procedure continues at state **1722**.

In state **1718**, the packet data is copied or transferred (e.g., via a DMA operation) into the flow's re-assembly buffer, at the location specified in the next address field **1104** of the flow's entry in flow re-assembly table **1004**. One skilled in the art will appreciate that the next address field may or may not be updated to account for this new packet because the re-assembly buffer is being released.

In state **1720**, a completion descriptor is written or configured to release the flow's re-assembly buffer and to provide information to the host computer for processing the packet. The header buffer index (e.g., the location or index, within the free buffer array, of the header buffer's identifier) and the offset of the packet's header within the header buffer are placed in the descriptor. The flow re-assembly buffer index (e.g., the location or index within the free buffer array of the flow re-assembly buffer's identifier) and the offset of the packet's data within that buffer are also stored in the descriptor.

The size of the packet's data (e.g., the size of the packet's TCP payload) and header (e.g., the offset of the TCP payload within the packet) are stored in data size and header size fields, respectively. The descriptor type field is given a value that indicates that a flow packet has been transferred to host memory. A release header flag is set if the header buffer is full and a release data flag is set to indicate that no more data will be placed in this flow re-assembly buffer (e.g., because it is being released). As explained above, the header buffer may not be tested to see if it is full until a later state of this procedure, at which time the release header flag may be set. Finally, a release flow flag is set to indicate that DMA engine **120** is tearing down the packet's flow. After state **1720**, the illustrated procedure resumes at state **1728**.

NetApp Ex. 1015 Page 553

In state **1722**, a first portion of the packet's payload (e.g., data) is stored in the flow's present (e.g., valid) re-assembly buffer, at the location identified in the buffer's next address field **1104**.

Because the full payload will not fit into this buffer, in state **1724** a new flow re-assembly buffer is prepared and the remainder of the payload is stored in that buffer. In one embodiment of the invention information concerning the first buffer is stored in a completion descriptor. This information may include the position within the free buffer array of the first buffer's buffer identifier and the offset of the first portion of data within the buffer. The flow's entry in flow re-assembly buffer table **1004** may then be updated for the second buffer (e.g., store a first address in next address field **1104** and the location of buffer's identifier in the free buffer array in re-assembly buffer index **1102**).

In state **1726**, a completion descriptor is written or configured. The contents are similar to those described for states **1712** and **1720** but this descriptor must reflect that two re-assembly buffers were used.

Thus, the header buffer index (e.g., the position within the free buffer array of the buffer identifier corresponding to the header buffer) and the offset of the packet's header within the header buffer are placed in the descriptor, as above. The first flow re-assembly buffer index (e.g., the position, within the free buffer array, of the buffer identifier corresponding to the first flow re-assembly buffer used to store this packet's payload) and the offset of the packet's first portion of data within that buffer are also stored in the descriptor.

The size of the packet's data (e.g., the size of the packet's TCP payload) and header (e.g., the offset of the TCP payload within the packet) are stored in data size and header size fields, respectively. The descriptor type field is given a value that indicates that a flow packet has been transferred to host memory. A release header flag is set if the header buffer is full and a release data flag is set to indicate that no more data will be placed in this flow re-assembly buffer. A release flow flag is set to indicate that DMA engine **120** is tearing down the packet's flow.

Because two re-assembly buffers were used, a split packet flag is set and the index, within the free buffer array, of the re-assembly buffer's buffer identifier is stored in a next index field. Additionally, because the packet contains the final portion of data for the flow, a release next data buffer flag will also be set to indicate that the second flow re-assembly buffer is being released.

In state **1728**, the flow's entry in flow re-assembly buffer table **1004** is invalidated to ensure that it is not used again.

In state **1730**, it is determined whether the header buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, a counter is used to keep track of entries placed into each new header buffer. The buffer is considered full when thirty-two entries are stored.

If the buffer is full, in state **1732** the header buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention a release header flag in the descriptor is set.

If the header buffer is not full, then in state **1734** the next address field of header buffer table **1006** is updated to indicate the address at which to store the next header or packet.

Processing associated with a packet having operation code 3 then ends with end state **1799**. In this end state, the buffer used for this packet is turned over to the host computer by changing its ownership indicator field (e.g., from one to zero). Alternatively, some other mechanism may be used, such as issuing an interrupt or changing the descriptor' descriptor type field. Illustratively, the descriptor type field would be changed to a value indicating that DMA engine **120** transferred a flow packet into host memory.

In one alternative embodiment of the invention an optimization may be performed when processing a packet with operation code **3**. This optimization takes advantage of the knowledge that the packet contains the last portion of data for its flow. In particular, instead of loading a descriptor into flow re-assembly buffer table **1004** the descriptor may be used where it is—in a descriptor cache maintained by free ring manager **1012**.

For example, instead of retrieving a buffer identifier from a descriptor and storing it in an array in state **1708** above, only to store one packet's data in the identified buffer before releasing it, it may be more efficient to use the descriptor without removing it from the cache. In this embodiment, when a completion descriptor is written the values stored in its data index and data offset fields are retrieved from a descriptor in the descriptor cache. Similarly, when the first portion of a code **3** packet's data fits into the flow's active buffer but a new one is needed just for the remaining data, a descriptor in the descriptor cache may again be used without first loading it into a free buffer array and the flow re-assembly buffer table. In this situation, the completion descriptor's next index field is retrieved from the descriptor in the descriptor cache.

A Method of Transferring a Packet with Operation Code **4**

FIGS. **8A–18D** depict an illustrative procedure in which DMA engine **120** transfers a packet associated with operation code **4** to a host memory buffer. As reflected in TABLE **1**, operation code **4** in this embodiment indicates that the packet is compatible with the protocols that may be parsed by NIC **100** and continues a flow that is already established. No new flow is set up, the existing flow is not to be torn down, and the packet's data is to be re-assembled with data from other flow packets. Because the packet is to be re-assembled, the packet's header should be stored in a header buffer and its data in the flow's re-assembly buffer.

In state **1800**, DMA engine **120** determines whether there is a valid (e.g., active) header buffer. Illustratively, this determination is made by examining validity indicator **1116** of header buffer table **1006**, which manages the active header buffer. If the validity indicator is set, then there should be a header buffer ready to receive this packet and the procedure continues at state **1804**.

Otherwise, in state **1802** a new header buffer is prepared or initialized for storing small packets and headers of re-assembled packets. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or some other indication of the first storage location in the buffer is place in next address field **1114** of header buffer table **1006**. Also, the position or index of the buffer identifier within the free buffer array is stored in header buffer index **1112** and validity indicator **1116** is set to a valid state.

In state **1804** the packet's header is copied or transferred into the header buffer at the address or location specified in

NOAC EX. 1015 Page 554

the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes are inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In addition, the packet's header may be positioned within a cell of predetermined size (e.g., 256 bytes) within the buffer.

In the illustrated embodiment, operation code **4** indicates that an existing flow is to be continued. Thus, in state **1806** it is determined whether a flow re-assembly buffer is valid (e.g., active) for this flow by examining the validity indicator in the flow's entry in flow re-assembly buffer table **1004**. Illustratively, if the indicator is valid then there is an active buffer storing data from one or more packets in this flow. If there is a valid flow re-assembly buffer for this flow, the illustrated procedure continues at state **1808**. Otherwise, the procedure proceeds to state **1810**.

In state **1808**, it is determined whether the packet's data (e.g., its TCP payload) portion is too large for the current flow re-assembly buffer. If the data portion is too large, two flow re-assembly buffers will be used and the illustrated procedure proceeds to state **1830**. Otherwise, the procedure continues at state **1820**.

In state **1810**, because it was found (in state **1806**) that there was no valid flow re-assembly buffer for this packet, a new flow re-assembly buffer is prepared. Illustratively, a free ring descriptor is obtained from a cache maintained by free ring manager **1012** and its reference to an empty buffer is retrieved. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or other indicator of its first storage location is placed in next address field **1104** of the flow's entry in flow re-assembly buffer table **1004**. The flow's entry in the table may be recognized by its flow number. The location of the buffer identifier in the free buffer array is stored in re-assembly buffer index **1102**, and validity indicator **1106** is set to a valid state.

In state **1812**, the packet's data is copied or transferred (e.g., via a DMA operation) into the address or location specified in the next address field of the flow's entry in flow re-assembly buffer table **1004**.

In state **1814**, a completion descriptor is written or configured to provide information to the host computer for processing the packet. In particular, the header buffer index (e.g., the index within the free buffer array of the buffer identifier that identifies the header buffer) and the offset of the packet's header within the header buffer are placed in the descriptor. Illustratively, this offset may serve to identify the first byte of the header, the first pad byte preceding the header or the header's cell within the header buffer. The flow re-assembly buffer index (e.g., the index within the free buffer array of the buffer identifier that identifies the flow re-assembly buffer) and the offset of the packet's data within that buffer are also stored in the descriptor.

The size of the packet's data (e.g., the size of the packet's TCP payload) and header (e.g., the offset of the TCP payload within the packet) are stored in data size and header size fields, respectively. The descriptor type field is given a value indicating that a flow packet has been transferred to host memory. A release header flag is set if the header buffer is full but a release data flag is not set, because more data will be placed in this flow re-assembly buffer. The header buffer may be tested to see if it is full until a later state of this procedure. In such an embodiment, the release header flag may be set (or cleared) at that time.

In one embodiment of the invention a release flow flag may also be set, depending upon dynamic packet batching module **122**. For example, if the packet batching module determines that another packet in the same flow will soon be transferred to the host computer, the release flow flag will be cleared (e.g., a zero will be stored). This indicates that the host computer should await the next flow packet before processing this one. By collectively processing multiple packets from a single flow, the packets can be processed more efficiently and less processor time is required. If, however, no other packets in the same flow are identified, the release flow flag may be set (e.g., a one is stored) to indicate that the host computer should process the flow packets it has received so far, without waiting for more.

In state **1816**, the flow's entry in flow re-assembly buffer table **1004** is updated. In particular, next address field **1104** is updated to identify the location in the re-assembly buffer at which the next flow packet's data should be stored. After state **1816**, the illustrated procedure continues at state **1838**.

In state **1820** (reached from state **1808**), it is known that the packet's data, or TCP payload, will fit within the flow's current re-assembly buffer. Thus, the packet data is copied or transferred into the buffer at the location identified in next address field **1104** of the flow's entry in flow re-assembly buffer table **1004**.

In state **1822**, a completion descriptor is written or configured to provide information to the host computer for processing the packet. In particular, the header buffer index (e.g., the index within the free buffer array of the buffer identifier that identifies the header buffer) and the offset of the packet's header within the header buff&r are placed in the descriptor. The flow re-assembly buffer index (e.g., the index within the free buffer array of the buffer identifier that identifies the flow re-assembly buffer) and the offset of the packet's data within that buffer are also stored in the descriptor.

The size of the packet's data (e.g., the size of the packet's TCP payload) and header (e.g., the offset of the TCP payload within the packet) are stored in data size and header size fields, respectively. The descriptor type field is given a value indicating that a flow packet has been transferred to host memory. A release header flag is set if the header buffer is full but a release data flag is set only if the flow re-assembly buffer is now full. The header and flow re-assembly buffers may not be tested to see if they are full until a later state of this procedure. In such an embodiment, the flags may be set (or cleared) at that time.

In one embodiment of the invention a release flow flag may also be set, depending upon dynamic packet batching module **122**. For example, if the packet batching module determines that another packet in the same flow will soon be transferred to the host computer, the release flow flag will be cleared. This indicates that the host computer should await the next flow packet before processing this one. By collectively processing multiple packets from a single flow, the packets can be processed more efficiently and less processor time is required. If, however, no other packets in the same flow are identified, the release flow flag may be set to indicate that the host computer should process the flow packets received so far, without waiting for more.

In state **1824**, the flow re-assembly buffer is examined to determine if it is full. In the presently described embodiment of the invention this test is conducted by first determining how much data (e.g., how many bytes) has been stored in the buffer. Illustratively, the flow's next address field and the

amount of data stored from this packet are summed. Then, the initial buffer address (e.g., before any data was stored in it) is subtracted from this sum. This value, representing how much data is now stored in the buffer, is then compared to the size of the buffer (e.g., eight kilobytes).

If the amount of data currently stored in the buffer equals the size of the buffer, then it is full. In the presently described embodiment of the invention it is desirable to completely fill flow re-assembly buffers. Thus, a flow re-assembly buffer is not considered full until its storage space is completely populated with flow data. This scheme enables the efficient processing of network packets.

If the flow re-assembly buffer is full, in state **1826** the buffer is invalidated to ensure it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention, a release data flag in the descriptor is set. After state **1826**, the procedure continues at state **1838**.

If the flow re-assembly buffer is not full, then in state **1828** next address field **1104** in the flow's entry in flow re-assembly buffer table **1004** is updated to indicate the address at which to store the next portion of flow data. After state **1828**, the procedure continues at state **1838**.

In state **1830** (reached from state **1808**), it is known that the packet's data will not fit into the flow's current re-assembly buffer. Therefore, some of the data is stored in the current buffer and the remainder in a new buffer. In particular, in state **1830** a first portion of data (e.g., an amount sufficient to fill the buffer) is copied or transferred into the current flow re-assembly buffer.

In state **1832**, a new descriptor is loaded from a descriptor cache maintained by free ring manager **1012**. Its identifier of a new buffer is retrieved and the remaining data from the packet is stored in the new buffer. In one embodiment of the invention, after the first portion of data is stored information from the flow's entry in flow re-assembly table **1004** is stored in a completion descriptor. Illustratively, this information includes re-assembly buffer index **1102** and the offset of the first portion of data within the full buffer. Then the new descriptor can be loaded—its index is stored in re-assembly buffer index **1102** and an initial address is stored in next address **1104**.

In state **1834**, a completion descriptor is written or configured to provide information to the host computer for processing the packet. In particular, the header buffer index (e.g., the location of the header buffer's buffer identifier within the free buffer array) and the offset of the packet's header within the header buffer are placed in the descriptor. The flow re-assembly buffer index (e.g., the location of the flow re-assembly buffer's buffer identifier within the free buffer array) and the offset of the packet's data within that buffer are also stored in the descriptor.

The size of the packet's data (e.g., the size of the packet's TCP payload) and header (e.g., the offset of the TCP payload within the packet) are stored in data size and header size fields, respectively. The descriptor type field is given a value indicating that a flow packet has been transferred to host memory. A release header flag is set if the header buffer is full and a release data flag is set because the first flow re-assembly buffer is being released. The header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment, the release header flag may be set (or cleared) at that time.

Because two re-assembly buffers were used, a split packet flag in the descriptor is set and the index, within the free descriptor ring, of the descriptor that references the second re-assembly buffer is stored in a next index field.

In one embodiment of the invention a release flow flag may also be set, depending upon dynamic packet batching module **122**. For example, if the packet batching module determines that another packet in the same flow will soon be transferred to the host computer, the release flow flag will be cleared. This indicates that the host computer should await the next flow packet before processing this one. By collectively processing multiple packets from a single flow, the packets can be processed more efficiently and less processor time is required. If, however, no other packets in the same flow are identified, the release flow flag may be set to indicate that the host computer should process the flow packets received so far, without waiting for more.

In state **1836**, next address field **1104** in the flow's entry in flow re-assembly buffer table **1004** is updated to indicate the address in the new buffer at which to store the next portion of flow data.

In state **1838**, it is determined whether the header buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, a counter may be used to keep track of entries placed into each new header buffer. The buffer is considered full when thirty-two entries are stored.

If the buffer is full, in state **1840** the header buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention, a release header flag in the descriptor is set.

If the header buffer is not full, then in state **1842** the next address field of header buffer table **1006** is updated to indicate the address at which to store the next header or small packet.

The processing associated with a packet having operation code **4** then ends with end state **1899**. In this end state, the descriptor used for this packet is turned over to the host computer by changing its ownership indicator field (e.g., from one to zero). In one alternative embodiment of the invention, DMA engine **120** issues an interrupt or uses other means to alert the host computer that a descriptor is being released.

In one alternative embodiment of the invention the optimization described above for packets associated with operation code **3** may be performed when processing a packet with operation code **4**. This optimization is useful, for example, when a code **4** packet's data is too large to fit in the current flow re-assembly buffer. Instead of loading a new descriptor for the second portion of data, the descriptor may be used where it is—in a descriptor cache maintained by free ring manager **1012**. This allows DMA engine **120** to finish transferring the packet and turn over the completion descriptor before adjusting flow re-assembly buffer table **1004** to reflect a new buffer.

In particular, instead of loading information from a new descriptor in state **1832** above, it may be more efficient to use the descriptor without removing it from the cache. In this embodiment a new buffer for storing a remainder of the packet's data is accessed by retrieving its buffer identifier from a descriptor in the free ring manager's descriptor cache. The data is stored in the buffer and, after the packet's completion descriptor is configured and released, the necessary information is loaded into the flow re-assembly table as described above.

Illustratively, re-assembly buffer index **1102** stores the buffer identifier's index within the free buffer array, and an initial memory address of the buffer, taking into account the newly stored data, is placed in next address **1104**.

A Method of Transferring a Packet with Operation Code 5

FIGS. 19A–19E depict a procedure in which DMA engine **120** transfers a packet associated with operation code **5** to a host memory buffer. As reflected in TABLE 1, operation code **5** in one embodiment of the invention may indicate that a packet is incompatible with the protocols that may be parsed by NIC **100**. It may also indicate that a packet contains all of the data for a new flow (e.g., no more data will be received for the packet's flow). Therefore, for operation code **5**, no new flow is set up and there should not be any flow to tear down. The packet's data, if there is any, is not to be re-assembled.

In state **1900**, it is determined whether the present packet is a small packet (e.g., less than or equal to 256 bytes in size) suitable for storage in a header buffer. If so, the illustrated procedure proceeds to state **1920**.

Otherwise, in state **1902** it is determined whether the present packet is a jumbo packet (e.g., greater than 1522 bytes in size), such that it should be stored in a jumbo buffer. If so, the illustrated procedure proceeds to state **1940**. If not, the procedure continues at state **1904**.

In state **1904**, it has been determined that the packet is not a small packet or a jumbo packet. The packet may, therefore, be stored in a non-re-assembly buffer used to store packets that are no greater in size than MTU (Maximum Transfer Unit) in size, which is 1522 bytes in a present embodiment. This buffer may be called an MTU buffer. Therefore, DMA engine **120** determines whether a valid (e.g., active) MTU buffer exists. Illustratively, this determination is made by examining validity indicator **1126** of MTU buffer table **1008**, which manages the active MTU buffer. If the validity indicator is set, then there should be a MTU buffer ready to receive this packet and the procedure continues at state **1908**.

Otherwise, in state **1906** a new MTU buffer is prepared or initialized for storing non-re-assembleable packets up to 1522 bytes in size. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its buffer identifier (e.g., a reference to an empty host memory buffer). If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or some other indication of the first storage location in the buffer is placed in next address field **1124** of MTU buffer table **1008**. The buffer identifier's index or position within the free buffer array is stored in MTU buffer index **1122**, and validity indicator **1126** is set to a valid state.

In state **1908** the packet is copied or transferred (e.g., via a DMA operation) into the MTU buffer at the address or location specified in the next address field of MTU buffer table **1008**. As described above, in one embodiment of the invention pad bytes may be inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In addition, the packet may be positioned within a cell of predetermined size (e.g., two kilobytes) within the MTU buffer.

In state **1910**, a completion descriptor is written or configured to provide necessary information to the host computer for processing the packet. In particular, the MTU buffer index (e.g. the location within the free buffer array of the buffer identifier for the MTU buffer) and offset (e.g., the offset to the packet or the packet's cell within the buffer) are placed in the descriptor in data index and data offset fields, respectively. The size of the packet is stored in a data size field. A header size field within the descriptor may be set to zero to indicate that the entire packet was placed in the MTU buffer (e.g., no separate header portion was stored in a header buffer). A release data flag is set in the descriptor if the MTU buffer is full. The MTU buffer may not, however, be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release data flag may be set (or cleared) at that time. A release header flag may be cleared (e.g., not set), because there is no separate header portion being conveyed to the host computer.

Further, the descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. Also, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention, the ownership indicator is not set until end state **1999** below. In an alternative embodiment of the invention, the descriptor may be released by issuing an interrupt or other alert. In yet another alternative embodiment, changing the descriptor's descriptor type field may signal the descriptor's release.

In state **1912**, DMA engine **120** determines whether the MTU buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size, each entry in the MTU buffer may be allotted two kilobytes of space and a counter may be used to keep track of entries placed into an MTU buffer. The buffer may be considered full when a predetermined number of entries (e.g., four) are stored. In an alternative embodiment of the invention entries in an MTU buffer may or may not be allocated a certain amount of space, in which case DMA engine **120** may calculate how much storage space within the buffer has yet to be used. If no space remains, or if less than a predetermined amount of space is still available, the buffer may be considered full.

If the MTU buffer is full, in state **1914** the buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the MTU buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention a release data flag in the descriptor is set. The illustrated procedure then ends with end state **1999**.

If the MTU buffer is not full, then in state **1916** the next address field of MTU buffer table **1008** is updated to indicate the address at which to store the next packet. The illustrated procedure then ends with end state **1999**.

In state **1920** (reached from state **1900**), it has been determined that the present packet is a small packet suitable for storage in a header buffer. Therefore, DMA engine **120** (e.g., DMA manager **1002**) determines whether there is a valid (e.g., active) header buffer. Illustratively, this determination is made by examining validity indicator **1116** of header buffer table **1006**, which manages the active header buffer. If the validity indicator is set, then there is a header buffer ready to receive this packet and the procedure continues at state **1924**.

Otherwise, in state **1922** a new header buffer is prepared or initialized for storing small packets and headers of re-assembled packets. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or some other indicator of the first storage location or cell in the buffer is placed in next address field **1114** of header buffer table **1006**. Further, the buffer identifier's position within the free buffer array is stored in header buffer index **1112** and validity indicator **1116** is set to a valid state.

In state **1924** the packet is copied or transferred (e.g., via a DMA operation) into the header buffer at the address or location specified in the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes may be inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In addition, the packet may be positioned within a cell of predetermined size (e.g., 256 bytes) within the buffer.

In state **1926**, a completion descriptor is written or configured to provide necessary information to the host computer (e.g., a software driver) for processing the packet. In particular, the header buffer index (e.g. the index of the free buffer array element that contains the header buffer's identifier) and offset are placed in the descriptor, in header index and header offset fields, respectively. Illustratively, this offset serves to identify the first byte of the packet, the first pad byte preceding the packet or the location of the packet's cell within the buffer. The size of the packet is also stored in the descriptor, illustratively within a header size field. A data size field within the descriptor may be set to zero to indicate that the entire packet was placed in the header buffer (e.g., no separate data portion was stored in another buffer). A release header flag may be set in the descriptor if the header buffer is full. However, the header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release header flag may be set (or cleared) at that time. A release data flag may be cleared (e.g., not set), because there is no separate data portion being conveyed to the host computer.

The descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. Also, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention the ownership indicator is not set until end state **1999** below.

In state **1928** it is determined whether the header buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, a counter is used to keep track of entries placed into each new header buffer. The buffer is considered full when thirty-two entries are stored.

If the buffer is full, in state **1930** the header buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention a release header flag in the descriptor is set. The illustrated procedure then ends with end state **1999**.

If the header buffer is not full, then in state **1932** the next address field of header buffer table **1006** is updated to indicate the address at which to store the next header or small packet. The illustrated procedure then ends with end state **1999**.

In state **1940** (reached from state **1902**), it has been determined that the packet is a jumbo packet (e.g., that it is greater than 1522 bytes in size). In this embodiment of the invention a jumbo packet's data portion is stored in a jumbo buffer. Its header is also stored in the jumbo buffer unless splitting of jumbo packets is enabled, in which case its header is stored in a header buffer. DMA engine **120** thus determines whether a valid (e.g., active) jumbo buffer exists. Illustratively, this determination is made by examining validity indicator **1136** of jumbo buffer table **1010**, which manages an active jumbo buffer. If the validity indicator is set, then there is a jumbo buffer ready to receive this packet and the procedure continues at state **1944**.

Otherwise, in state **1942** a new jumbo buffer is prepared or initialized for storing a non-re-assembleable packet that is larger than 1522 bytes. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or other indication of the first storage location within the buffer is placed in next address field **1134** of jumbo buffer table **1010**. The position of the buffer identifier within the free buffer array is stored in jumbo buffer index **1132**, and validity indicator **1136** is set to a valid state.

Then, in state **1944**, DMA engine **120** determines whether splitting of jumbo buffers is enabled. If enabled, the header of a jumbo packet is stored in a header buffer while the packet's data is stored in one or more jumbo buffers. If not enabled, the entire packet will be stored in one or more jumbo buffers. Illustratively, splitting of jumbo packets is enabled or disabled according to the configuration of a programmable indicator (e.g., flag, bit, register) that is set by software operating on the host computer (e.g., a device driver). If splitting is enabled, the illustrated procedure continues at state **1960**. Otherwise, the procedure proceeds to state **1946**.

In state **1946**, DMA engine **120** determines whether the packet will fit into one jumbo buffer. For example, in an embodiment of the invention using eight kilobyte pages, if the packet is larger than eight kilobytes a second jumbo buffer will be needed to store the additional contents. If the packet is too large, the illustrated procedure continues at state **1952**.

Otherwise, in state **1948** the packet is copied or transferred (e.g., via a DMA operation) into the current jumbo buffer, at the location specified in the next address field **1134** of jumbo buffer table **1010**. When the packet is transferred intact like this, padding may be added to align a header portion of the packet with a sixteen-byte boundary. One skilled in the art will appreciate that the next address field may not need to be updated to account for this new packet because the jumbo buffer will be released. In other words, in one embodiment of the invention a jumbo buffer is only used once (e.g., to store one packet or a portion of one packet). In an alternative embodiment of the invention a jumbo buffer may store portions of two or more packets, in which case next address field **1134** may need to be updated.

In state **1950**, a completion descriptor is written or configured to release the jumbo buffer and to provide information to the host computer for processing the packet. The jumbo buffer index (e.g., the index, within the free buffer array, of the buffer identifier that corresponds to the jumbo buffer) and the offset of the first byte of the packet within the

jumbo buffer are placed in the descriptor, in data index and data size fields, respectively. The size of the packet (e.g., the packet length) is stored in a data size field.

A header size field may be cleared (e.g., a zero is stored) to indicate that the header buffer was not used (e.g., the header was not stored separately from the packet's data). Because the packet was stored intact, header index and header offset fields may or may not be used (e.g., the values stored in their fields do not matter). A release header flag is cleared and a release data flag is set to indicate that no more data will be placed in this jumbo buffer (e.g., because it is being released).

The descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. Finally, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention, the ownership indicator is not changed until end state **1999** below. After state **1950**, the illustrated procedure resumes at state **1958**. In one alternative embodiment of the invention, DMA engine **120** issues an interrupt or uses some other means, possibly not until end state **1999**, to alert the host computer that a descriptor is being released.

In state **1952**, a first portion of the packet is stored in the present (e.g., valid) jumbo buffer, at the location identified in the buffer's next address field **1134**. Because the whole packet will not fit into this buffer, in state **1954** a new jumbo buffer is prepared and the remainder of the packet is stored in that buffer.

In state **1956**, a completion descriptor is written or configured. The contents are similar to those described in state **1950** but this descriptor must reflect that two jumbo buffers were used to store the packet. Thus, the jumbo buffer index (e.g., the index, within the free buffer array, of the array element containing the header buffer's buffer identifier) and the offset of the first byte of the packet within the first jumbo buffer are placed in the descriptor, as above. The size of the packet (e.g., the packet length) is stored in a data size field.

A header size field may be cleared (e.g., a zero is stored) to indicate that the header buffer was not used (e.g., the header was not stored separately from the packet's data). Because there is no separate packet header, header index and header offset fields may or may not be used (e.g., the values stored in their fields do not matter).

A release header flag is cleared and a release data flag is set to indicate that no more data will be placed in these jumbo buffers (e.g., because they are being released). Further, a split packet flag is set to indicate that a second jumbo buffer was used, and the index (within the free buffer array) of the buffer identifier for the second buffer is stored in a next index field.

The descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. And, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention the ownership indicator is not changed until end state **1999** below.

In state **1958**, the jumbo buffer's entry in jumbo buffer table **1010** is invalidated (e.g., validity indicator **1136** is set to invalid) to ensure that it is not used again. In the procedure described above, a jumbo packet was stored in, at most, two jumbo buffers. In an alternative embodiment of the invention, a jumbo buffer may be stored across any number of buffers. The descriptor that is configured to report the transfer of such a packet is constructed accordingly, as will be obvious to one skilled in the art.

After state **1958**, the illustrated procedure ends at end state **1999**.

In state **1960** (reached from state **1944**), it has been determined that the present jumbo packet will be split to store the packet header in a header buffer and the packet data in one or more jumbo buffers. Therefore, DMA engine **120** (e.g., DMA manager **1002**) first determines whether there is a valid (e.g., active) header buffer. Illustratively, this determination is made by examining validity indicator **1116** of header buffer table **1006**, which manages the active header buffer. If the validity indicator is set, then there is a header buffer ready to receive this packet and the procedure continues at state **1964**.

Otherwise, in state **1962** a new header buffer is prepared or initialized for storing small packets and headers of other packets. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or some other indication of the first storage location or cell in the buffer is placed in next address field **1114** of header buffer table **1006**. The index or position of the buffer identifier within the free buffer array is stored in header buffer index **1112**, and validity indicator **1116** is set to a valid state.

In state **1964** the packet's header is copied or transferred (e.g., via a DMA operation) into the header buffer at the address or location specified in the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes may be inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In addition, the header may be positioned within a cell of predetermined size (e.g., 256 bytes) in the buffer.

In state **1966**, DMA engine **120** determines whether the packet's data (e.g., the TCP payload) will fit into one jumbo buffer. If the packet is too large to fit into one (e.g., the current jumbo buffer), the illustrated procedure continues at state **1972**.

In state **1968**, the packet's data is copied or transferred (e.g., via a DMA operation) into the current jumbo buffer, at the location specified in the next address field **1134** of jumbo buffer table **1010**. One skilled in the art will appreciate that the next address field may not need to be updated to account for this new packet because the jumbo buffer will be released. In other words, in one embodiment of the invention a jumbo buffer is only used once (e.g., to store one packet or a portion of one packet).

In state **1970**, a completion descriptor is written or configured to release the jumbo buffer and to provide information to the host computer for processing the packet. The header buffer index (e.g. the free buffer array position of the buffer identifier corresponding to the header buffer) and offset of the packet's header are placed in the descriptor in header index and header offset fields, respectively. Illustratively, this offset serves to identify the first byte of the header, the first pad byte preceding the header or the cell in which the header is stored. The jumbo buffer index (e.g., the

index within the free buffer array of the buffer identifier that references the jumbo buffer) and the offset of the first byte of the packet's data within the jumbo buffer are placed in data index and data offset fields, respectively. Header size and data size fields are used to store the size of the packet's header (e.g., the offset of the payload within the packet) and data (e.g., payload size), respectively.

A release header flag may be set in the descriptor if the header buffer is full. However, the header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release header flag may be set (or cleared) at that time. A release data flag is also set, because no more data will be placed in the jumbo buffer (e.g., it is being released to the host computer).

The descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. Also, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention the ownership indicator is not set until end state **1999** below.

After state **1970**, the illustrated procedure proceeds to state **1978**.

In state **1972**, a first portion of the packet's data is stored in the present (e.g., valid) jumbo buffer, at the location identified in the buffer's next address field **1134**. Because all of the packet's data will not fit into this buffer, in state **1974** a new jumbo buffer is prepared and the remainder of the packet is stored in that buffer.

In state **1976**, a completion descriptor is written or configured. The contents are similar to those described in states **1970** but this descriptor must reflect that two jumbo buffers were used to store the packet. The header buffer index (e.g. the free buffer array element that contains the header buffer's identifier) and offset of the header are placed in the descriptor in header index and header offset fields, respectively. The jumbo buffer index (e.g., the free buffer array element containing the jumbo buffer's buffer identifier) and the offset of the first byte of the packet's data within the jumbo buffer are placed in data index and data offset fields, respectively. Header size and data size fields are used to store the size of the packet's header (e.g., the offset of the payload within the packet) and data (e.g., payload size), respectively.

A release header flag is set in the descriptor if the header buffer is full. However, the header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release header flag may be set (or cleared) at that time. A release data flag is also set, because no more data will be placed in the jumbo buffer (e.g., it is being released to the host computer). Further, a split packet flag is set to indicate that a second jumbo buffer was used, and the position or index within the free buffer array of the second buffer's buffer identifier is stored in a next index field.

The descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. Finally, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention the ownership indicator is not set until end state **1999** below. In an alternative embodiment of the invention DMA engine **120** issues an interrupt or uses some other signal to alert the host computer that a descriptor is being released.

In state **1978**, the jumbo buffer's entry in jumbo buffer table **1010** is invalidated (e.g., validity indicator **1136** is set to invalid) to ensure that it is not used again. In the procedure described above, a jumbo packet was stored in, at most, two jumbo buffers. In an alternative embodiment of the invention a jumbo buffer may be stored across any number of buffers. The descriptor that is configured to report the transfer of such a packet is constructed accordingly, as will be obvious to one skilled in the art.

In state **1980**, it is determined whether the header buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, a counter may be used to keep track of entries placed into each new header buffer. The buffer is considered full when thirty-two entries are stored.

If the buffer is full, in state **1982** the header buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention a release header flag in the descriptor is set. The illustrated procedure then ends with end state **1999**.

If the header buffer is not full, then in state **1984** the next address field of header buffer table **1006** is updated to indicate the address at which to store the next header or small packet. The illustrated procedure then ends with end state **1999**.

In end state **1999**, a descriptor may be turned over to the host computer by storing a particular value (e.g., zero) in the descriptor's ownership indicator field as described above. Illustratively, the host computer (or software operating on the host computer) detects the change and understands that DMA engine **120** is returning ownership of the descriptor to the host computer.

A Method of Transferring a Packet with Operation Code **6** or Operation Code **7**

FIGS. **20A–20B** depict an illustrative procedure in which DMA engine **120** transfers a packet associated with operation code **6** or **7** to a host memory buffer. As reflected in TABLE 1, operation codes **6** and **7** may indicate that a packet is compatible with the protocols pre-selected for NIC **100** and is the first packet of a new flow. The difference between these operation codes in this embodiment of the invention is that operation code **7** is used when an existing flow is to be replaced (e.g., in flow database **110** and/or flow re-assembly buffer table **1004**) by the new flow. With operation code **6**, in contrast, no flow needs to be torn down. For both codes, however, a new flow is set up and the associated packet's data may be re-assembled with data from other packets in the newly established flow. Because the packet data is to be re-assembled, the packet's header should be stored in a header buffer and its data in a new flow re-assembly buffer.

As described in a previous section, the flow that is torn down to make room for a new flow (in the case of operation code **7**) may be the least recently used flow. Because flow database **110** and flow re-assembly buffer table **1004** contain only a limited number of entries in the presently described embodiment of the invention, when they are full and a new flow arrives an old one must be torn down. Choosing the least recently active flow for replacement is likely to have the least impact on network traffic through NIC **100**. In one embodiment of the invention DMA engine **120** tears down the flow in flow re-assembly buffer table **1004** that has the same flow number as the flow that has been replaced in flow database **110**.

In state **2000**, DMA engine **120** determines whether there is a valid (e.g., active) header buffer. Illustratively, this

NOAC Ex. 1015 Page 560

determination is made by examining validity indicator **1116** of header buffer table **1006**, which manages the active header buffer. If the validity indicator is set, then there is a header buffer ready to receive this packet and the procedure continues at state **2004**.

Otherwise, in state **2002** a new header buffer is prepared or initialized for storing small packets and headers of re-assembled packets. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or some other indication of the first storage location or cell in the buffer is placed in next address field **1114** of header buffer table **1006**. The position or index of the buffer identifier within the free buffer array is stored in header buffer index **1112**, and validity indicator **1116** is set to a valid state.

In state **2004** the packet's header is copied or transferred into the header buffer at the address or location specified in the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes may be inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In addition, the packet's header may be positioned in a cell of predetermined size (e.g., 256 bytes) within the buffer.

As discussed above, operation code **7** indicates that an old flow is to be torn down in flow re-assembly buffer table **1004** to make room for a new flow. This requires the release of any flow re-assembly buffer that may be associated with the flow being torn down.

Thus, in state **2006** it is determined whether a flow re-assembly buffer is valid (e.g., active) for a flow having the flow number that was read from control queue **118** for this packet. As explained in a previous section, for operation code **7** the flow number represents the entry in flow database **110** (and flow re-assembly buffer table **1004**) that is being replaced with the new flow. DMA engine **120** thus examines the validity indicator in the flow's entry in flow re-assembly buffer table **1004**. Illustratively, if the indicator is valid then there is an active buffer storing data from one or more packets in the flow that is being replaced. If there is a valid flow re-assembly buffer for this flow, the illustrated procedure continues at state **2008**. Otherwise, the procedure proceeds to state **2010**. It will be understood that the illustrated procedure will normally proceed to state **2008** for operation code **7** and state **2010** for operation code **6**.

In state **2008**, a completion descriptor is written or configured to release the replaced flow's re-assembly buffer. In particular, the flow re-assembly buffer index (e.g., the index within the free buffer array of the flow re-assembly buffer's buffer identifier) is written to the descriptor. In this embodiment of the invention, no offset needs to be stored in the descriptor's data offset field and the data size field is set to zero because no new data was stored in the buffer that is being released. Similarly, the header buffer is not yet being released, and therefore the header index and header offset fields of the descriptor need not be used and a zero may be stored in the header size field.

The descriptor's release header flag is cleared (e.g., a zero is stored in the flag) because the header buffer is not being released. The release data flag is set (e.g., a one is stored in the flag), however, because no more data will be placed in the released flow re-assembly buffer. Further, a release flow flag in the descriptor is set to indicate that the flow associated with the released flow re-assembly buffer is being torn down.

The descriptor type field is changed to a value indicating that DMA engine **120** is releasing a stale flow buffer (e.g., a flow re-assembly buffer that has not been used for some time). Finally, the descriptor used to release the replaced flow's re-assembly buffer and terminate the associated flow is turned over to the host computer by changing its ownership indicator field (e.g., from one to zero). In one alternative embodiment of the invention, DMA engine **120** issues an interrupt or employs some other means of alerting the host computer that a descriptor is being released.

In state **2010**, a new flow re-assembly buffer is prepared for the flow that is being set up. Illustratively, a free ring descriptor is obtained from a cache maintained by free ring manager **1012** and its buffer identifier (e.g., a reference to an empty memory buffer) is retrieved. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or other indication of the first storage location in the buffer is placed in next address field **1104** of the flow's entry in flow re-assembly buffer table **1004**. The flow's entry in the table may be recognized by its flow number. The position or index of the buffer identifier within the free buffer array is stored in re-assembly buffer index **1102**, and validity indicator **1106** is set to a valid state.

In state **2012**, the packet's data is copied or transferred (e.g., via a DMA operation) into the address or location specified in the next address field of the flow's entry in flow re-assembly buffer table **1004**.

In state **2014**, a completion descriptor is written or configured to provide information to the host computer for processing the packet. In particular, the header buffer index (e.g., the location or position within the free buffer array of the buffer identifier that references the header buffer) and the offset of the packet's header within the header buffer are placed in the descriptor. Illustratively, the offset identifies the first byte of the header, the first pad byte preceding the header or the location of the header's cell in the header buffer.

The flow re-assembly buffer index (e.g., the location or position, within the free buffer array, of the buffer identifier that references the flow re-assembly buffer) and the offset of the packet's data within that buffer are also stored in the descriptor. It will be recognized, however, that the offset reported for this packet's data may be zero, because the packet data is stored at the very beginning of the new flow re-assembly buffer.

The size of the packet's data (e.g., the size of the packet's TCP payload) and header (e.g., the offset of the TCP payload within the packet) are stored in data size and header size fields, respectively. The descriptor type field is changed to a value indicating that DMA engine **120** transferred a flow packet into host memory. A release header flag is set if the header buffer is full but a release data flag is not set, because more data will be placed in this flow re-assembly buffer. The header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment, the release header flag may be set (or cleared) at that time.

In one embodiment of the invention a release flow flag may also be set, depending upon dynamic packet batching

module **122**. For example, if the packet batching module determines that another packet in the same flow will soon be transferred to the host computer, the release flow flag will be cleared (e.g., a zero will be stored). This indicates that the host computer should await the next flow packet before processing this one. By collectively processing multiple packets from a single flow, the packets can be processed more efficiently and less processor time will be required for network traffic. If, however, no other packets in the same flow are identified, the release flow flag may be set to indicate that the host computer should process the flow packets received so far, without waiting for more.

In state **2016**, the flow's entry in flow re-assembly buffer table **1004** is updated. In particular, next address field **1104** is updated to identify the location in the re-assembly buffer which the next flow packet's data should be stored.

In state **2018**, it is determined whether the header buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, a counter may be used to keep track of entries placed into each new header buffer. The buffer is considered fall when thirty-two entries are stored.

If the buffer is full, in state **2020** the header buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention, a release header flag in the descriptor is set.

If the header buffer is not full, then in state **2022** the next address field of header buffer table **1006** is updated to indicate the address at which to store the next header or full packet.

The processing associated with a packet having operation codes **6** and **7** then ends with end state **2099**. In this end state, a descriptor used for this packet (e.g., the descriptor that was configured in state **2014**) is turned over to the host computer by changing its ownership indicator field (e.g., from one to zero). In one alternative embodiment of the invention, DMA engine **120** issues an interrupt or employs other means (e.g., such as the descriptor's descriptor type field) to alert the host computer that a descriptor is being released.

Embodiment of a Packet Batching Module

FIG. **21** is a diagram of dynamic packet batching module **122** in one embodiment of the invention. In this embodiment, packet batching module **122** alerts a host computer to the transfer, or impending transfer, of multiple packets from one communication flow. The related packets may then be processed through an appropriate protocol stack collectively, rather than processing one at a time. As one skilled in the art will recognize, this increases the efficiency with which network traffic may be handled by the host computer.

In the illustrated embodiment, a packet is transferred from **1200** to the host computer by DMA engine **120** (e.g., by copying its payload into an appropriate buffer). When a packet is transferred, packet batching module **122** determines whether a related packet (e.g., a packet in the same flow) will soon be transferred as well. In particular, packet batching module **122** examines packets that are to be transferred after the present packet. One skilled in the art will appreciate that the higher the rate of packet arrival at NIC **1200**, more packets that are likely to await transfer to a host computer at a given time. The more packets that await transfer, the more packets that may be examined by the packet batching module and the greater the benefit it may provide. In particular, as the number of packets awaiting transfer increases, packet batching module **122** may identify a greater number of related packets for collective processing. As the number of packets processed together increases, the amount of host processor time required to process each packet decreases.

Thus, if a related packet is found the packet batching module alerts the host computer so that the packets may be processed as a group. As described in a previous section, in one embodiment of the invention dynamic packet batching module **122** alerts the host computer to the availability of a related packet by clearing a release flow flag in a completion descriptor associated with a transferred packet. The flag may, for example, be cleared by DMA engine **120** in response to a signal or alert from dynamic packet batching module **122**.

In contrast, in an alternative embodiment of the invention dynamic packet batching module **122** or DMA engine **120** may alert the host computer when no related packets are found or when, for some other reason, the host processor should not delay processing a transferred packet. In particular, a release flow flag may be set when the host computer is not expected to receive a packet related to a transferred packet in the near future (e.g., thus indicating that the associated flow is being released or torn down). For example, it may be determined that the transferred packet is the last packet in its flow or that a particular packet does't even belong to a flow (e.g., this may be reflected in the packet's associated operation code).

With reference now to FIG. **21**, packet batching module **122** in one embodiment of the invention includes memory **2102** and controller **2104**. Illustratively, each entry in memory **2102**, such as entry **2106**, comprises two fields: flow number **2108** and validity indicator **2110**. In alternative embodiments of the invention, other information may be stored in memory **2102**. Read pointer **2112** and write pointer **2114** serve as indices into memory **2102**.

In the illustrated embodiment, memory **2102** is an associative memory (e.g., a CAM) configured to store up to 256 entries. Each entry corresponds to and represents a packet stored in packet queue **116**. As described in a previous section, packet queue **116** may also contain up to 256 packets in one embodiment of the invention. When a packet is, or is about to be transferred, by DMA engine **120** from packet queue **116** to the host computer, memory **2102** may be searched for an entry having a flow number that matches the flow number of the transferred packet. Because memory **2102** is a CAM in this embodiment, all entries in the memory may be searched simultaneously or nearly simultaneously. In this embodiment, memory **2102** is implemented in hardware, with the entries logically arranged as a ring. In alternative embodiments, memory **2102** may be virtually any type of data structure (e.g., array, table, list, queue) implemented in hardware or software. In one particular alternative embodiment, memory **2102** is implemented as a RAM, in which case the entries may be examined in a serial manner.

The maximum of 256 entries in the illustrated embodiment matches the maximum number of packets that may be stored in a packet queue. Because the depth of memory **2102** matches the depth of the packet queue, when a packet is stored in the packet queue its flow number may be automatically stored in memory **2102**. Although the same number of entries are provided for in this embodiment, in an alternative embodiment of the invention memory **2102** may be configured to hold a smaller or greater number of entries than the packet queue. And, as discussed in a previous

NOAC Ex. 1015 Page 562

section, for each packet stored in the packet queue, related information may also be stored in the control queue.

In the illustrated embodiment of the invention, flow number **2108** is the index into flow database **110** of the flow comprising the corresponding packet. As described above, in one embodiment of the invention a flow includes packets carrying data from one datagram sent from a source entity to a destination entity. Illustratively, each related packet has the same flow key and the same flow number. Flow number **2108** may comprise the index of the packet's flow key in flow database **110**.

Validity indicator **2110** indicates whether the information stored in the entry is valid or current. In this embodiment, validity indicator **2110** may store a first value (e.g., one) when the entry contains valid data, and a second value (e.g., zero) when the data is invalid. For example, validity indicator **2110** in entry **2106** may be set to a valid state when the corresponding entry in packet queue **116** contains a packet awaiting transfer to the host computer and belongs to a flow (e.g., which may be indicated by the packet's operation code). Similarly, validity indicator **2110** may be set to an invalid state when the entry is no longer needed (e.g., when the corresponding packet is transferred to the host computer).

Flow validity indicator **2110** may also be set to an invalid state when a corresponding packet's operation code indicates that the packet does not belong to a flow. It may also be set to an invalid state when the corresponding packet is a control packet (e.g., contains no data) or is otherwise non-re-assembleable (e.g., because it is out of sequence, incompatible with a pre-selected protocol, has an unexpected control flag set). Validity indicator **2110** may be managed by controller **2104** during operation of the packet batching module.

In the illustrated embodiment of the invention, an entry's flow number is received from a register in which it was placed for temporary storage. A packet's flow number may be temporarily stored in a register, or other data structure, in order to facilitate its timely delivery to packet batching module **122**. Temporary storage of the flow number also allows the flow database manager to turn its attention to a later packet. A flow number may, for example, be provided to dynamic packet batching module **122** at nearly the same time that the associated packet is stored in packet queue **116**. Illustratively, the flow number may be stored in the register by flow database manager **108** or by IPP module **104**. In an alternative embodiment, the flow number is received from control queue **118** or some other module of NIC **100**.

In the illustrated embodiment of the invention, memory **2102** contains an entry corresponding to each packet in packet queue **116**. When a packet in the packet queue is transferred to a host computer (e.g., when it is written to a reassembly buffer), controller **2104** invalidates the memory entry that corresponds to that packet. Memory **2102** is then searched for another entry having the same flow number as the transferred packet. Afterwards, when a new packet is stored in packet queue **116**, perhaps in place of the transferred packet, a new entry is stored in memory **2102**.

In an alternative embodiment of the invention, memory **2102** may be configured to hold entries for only a subset of the maximum number of packets stored in packet queue **116** (e.g., just re-assembleable packets). Entries in memory **2102** would still be populated when a packet is stored in the packet queue. However, if memory **2102** is full when a new packet is received, then creation of an entry for the new packet must wait until a packet is transferred and its entry in memory is invalidated. Therefore, in this alternative embodiment entries in memory **2102** may be created by extracting information from entries in control queue **118** rather than packet queue **116**. Controller **2104** would therefore continually attempt to copy information from entries in control queue **118** into memory **2102**. The function of populating memory **2102** may be performed independently or semi-independently of the function of actually comparing the flow numbers of memory entries to the flow number of a packet being transferred to the host computer.

In this alternative embodiment a second read pointer may be used to index control queue **118** to assist in the population of memory **2102**. In particular, the second read pointer may be used by packet batching module **122** to find and fetch entries for memory **2102**. Illustratively, if the second, or "lookahead" read pointer references the same entry as the control queue's write pointer, then it could be determined that no new entries were added to control queue **118** since the last check by controller **2104**. Otherwise, as long as there is an empty (e.g., invalid) entry in memory **2102**, the necessary information (e.g., flow number) may be copied into memory **2102** for the packet corresponding to the entry referenced by the lookahead read pointer. The lookahead read pointer would then be incremented.

Returning now to FIG. **21**, read pointer **2112** of dynamic packet batching module **122** identifies the current entry in memory **2102** (e.g., the entry corresponding to the packet at the front of the packet queue or the next packet to be transferred). Illustratively, this pointer is incremented each time a packet is transferred to the host computer. Write pointer **2114** identifies the position at which the next entry in memory **2102** is to be stored. Illustratively, the write pointer is incremented each time an entry is added to memory **2102**. One manner of collectively processing headers from related packets is to form them into one "super-" header. In this method, the packets' data portions are stored separately (e.g., in a separate memory page or buffer) from the super-header.

Illustratively, a super-header comprises one combined header for each layer of the packet' associated protocol stack (e.g., one TCP header and one IP header). To form each layer's portion of a super-header, the packet's individual headers may be merged to make a regular-sized header whose fields accurately reflect the assembled data and combined headers. For example, merged header fields relating to payload or header length would indicate the size of the aggregated data or aggregated headers, the sequence number of a merged TCP header would be set appropriately, etc. The super-header portion may then be processed through its protocol stack similar to the manner in which a single packet's header is processed.

This method of collectively processing related packet' headers (e.g., with "super-" headers) may require modification of the instructions for processing packets (e.g., a device driver). For example, because multiple headers are merged for each layer of the protocol stack, the software may require modification to recognize and handle the super-headers. In one embodiment of the invention the number of headers folded or merged into a super-header may be limited. In an alternative embodiment of the invention the headers of all the aggregated packets, regardless of number, may be combined.

In another method of collectively processing related packet' header portions, packet data and headers may also be stored separately (e.g., in separate memory pages). But, instead of combining the packet' headers for each layer of the appropriate protocol stack to form a super-header, they may be submitted for individual processing in quick suc-

NOAC Ex. 1015 Page 563

cession. For example, all of the packet' layer two headers may be processed in a rapid sequence—one after the other—then all of the layer three headers, etc. In this manner, packet processing instructions need not be modified, but headers are still processed more efficiently. In particular, a set of instructions (e.g., for each protocol layer) may be loaded once for all related packets rather than being separately loaded and executed for each packet.

As discussed in a previous section, data portions of related packets may be transferred into storage areas of predetermined size (e.g., memory pages) for efficient transfer from the host computer's kernel space into application or user space. Where the transferred data is of memory page size, the data may be transferred using highly efficient "page-flipping," wherein a full page of data is provided to application or user memory space.

FIGS. 22A–22B present one method of dynamic packet batching with packet batching module **122**. In the illustrated method, memory **2102** is populated with flow numbers of packets stored in packet queue **116**. In particular, a packet's flow number and operation code are retrieved from control queue **118**, IPP module **104**, flow database manager **108** or other module(s) of NIC **100**. The packet's flow number is stored in the flow number portion of an entry in memory **2102**, and validity indicator **2110** is set in accordance with the operation code. For example, if the packet is not reassembleable (e.g., codes **2** and **5** in TABLE 1), the validity indicator may be set to zero; otherwise it may be set to one.

The illustrated method may operate in parallel to the operation of DMA engine **120**. In other words, dynamic packet batching module **122** may search for packets related to a packet in the process of being transferred to a host memory buffer. Alternatively, a search may be conducted shortly after or before the packet is transferred. Because memory **2102** may be associative in nature, the search operation may be conducted quickly, thus introducing little, if any, delay into the transfer process.

FIG. **22**A may be considered a method of searching for a related packet, while FIG. 22B may be considered a method of populating the dynamic packet batching module's memory.

FIGS. 22A–22B each reflect one "cycle" of a dynamic packet batching operation (e.g., one search and creation of a new memory entry). Illustratively, however, the operation of packet batching module **122** runs continuously. That is, the end of one cycle of operation another cycle immediately begins. In this manner, controller **2104** strives to have memory **2102** is populated with entries for packets that are stored in packet queue **116**. If memory **2102** is large enough to store an entry for each packet in packet queue **116**, then controller **2104** attempts to keep the memory as full as possible and to quickly replace an invalidated entry with a new one.

State **2200** is a start state for a memory search cycle. In state **2202**, it is determined whether a packet (e.g., the packet at front of the packet queue) is being transferred to the host computer. This determination may, for example, be made by the operation of DMA engine **120** or the status of a packet, in packet queue **116** or control queue **118**. Alternatively, state **2202** is initiated by DMA engine **120** as a packet is copied into a buffer in the host computer. One purpose of state **2202** is simply to determine whether memory **2102** should be searched for a packet related to one that will be, or is being transferred. Until a packet is transferred, or about to be transferred, the illustrated procedure continues in state **2202**.

When, however, it is time for a search to be conducted (e.g., a packet is being transferred), the method continues at state **2204**. In state **2204**, the entry in memory **2102** corresponding to the packet being transferred is invalidated. Illustratively, this consists of storing a predetermined value (e.g., zero) in validity indicator **2110** for the packet's entry. In a present embodiment of the invention read pointer **2112** identifies the entry corresponding to the packet to be transferred. As one skilled in the art will recognize, one reason for invalidating a transferred packet's entry is so that when memory **2102** is searched for an entry associated with a packet related to the transferred packet, the transferred packet's own entry will not be identified.

In one embodiment of the invention the transferred packet's flow number is copied into a register (e.g., a hardware register) when dynamic packet batching module **122** is to search for a related packet. This may be particularly helpful (e.g., to assist in comparing the flow number to flow numbers of other packets) if memory **2102** is implemented as a RAM instead of a CAM.

In state **2206**, read pointer **2112** is incremented to point to the next entry in memory **2102**. If read pointer is incremented to the same entry that is referenced by write pointer **2114**, and that entry is also invalid (as indicated by validity indicator **2110**), it may be determined that memory **2102** is now empty.

Then, in state **2208**, memory **2102** is searched for a packet related to the packet being transferred (e.g., the memory is searched for an entry having the same flow number). As described above, entries in memory **2102** are searched associatively in one embodiment of the invention. Thus, the result of the search operation may be a single signal indicating whether or not a match was found.

In the illustrated embodiment of the invention, only valid entries (e.g., those having a value of one in their validity indicators) are searched. As explained above, an entry may be marked invalid (e.g., its validity indicator stores a value of zero) if the associated packet is considered incompatible. Entries for incompatible packets may be disregarded because their data is not ordinarily re-assembled and their headers are not normally batched. In an alternative embodiment of the invention, all entries may be searched but a match is reported only if a matching entry is valid.

In state **2210**, the host computer is alerted to the availability or non-availability of a related packet. In this embodiment of the invention, the host computer is alerted by storing a predetermined value in a specific field of the transferred packet's completion descriptor (described in a previous section). As discussed in the previous section, when a packet is transferred a descriptor in a descriptor ring in host memory is populated with information concerning the packet (e.g., an identifier of its location in host memory, its size, an identifier of a processor to process the packet's headers). In particular, a release flow flag or indicator is set to a first value (e.g., zero) if a related packet is found, and a second value if no related packet is found. Illustratively, DMA engine **120** issues the alert or stores the necessary information to indicate the existence of a related packet in response to notification from dynamic packet batching module **122**. Other methods of notifying the host computer of the presence of a related packet are also suitable (e.g., an indicator, flag, key), as will be appreciated by one skilled in the art.

In FIG. **22**B, state **2220** is a start state for a memory population cycle.

In state **2222**, it is determined whether a new packet has been received at the network interface. Illustratively, a new

entry is made in the packet batching module's memory for each packet received from the network. The receipt of a new packet may be signaled by IPP module **104**. For example, the receipt of a new packet may be indicated by the storage of the packet's flow number, by IPP module **104**, in a temporary location (e.g., a register). Until a new packet is received, the illustrated procedure waits. When a packet is received, the procedure continues at state **2224**.

In state **2224**, if memory **2102** is configured to store fewer entries than packet queue **116** (and, possibly, control queue **118**), memory **2102** is examined to determine if it is full.

In one embodiment of the invention memory **2102** may be considered full if the validity indicator is set (e.g., equal to one) for each entry or for the entry referenced by write pointer **2114**. If the memory is full, the illustrated procedure waits until the memory is not full. As one skilled in the art will recognize, memory **2102** and other data structures in NIC **100** may be tested for saturation (e.g., whether they are filled) by comparing their read and write pointers.

In state **2226**, a new packet is represented in memory **2102** by storing its flow number in the entry identified by write pointer **2114** and storing an appropriate value in the entry's validity indicator field. If, for example, the packet is not re-assembleable (e.g., as indicated by its operation code), the entry's validity indicator may be set to an invalid state. For purposes of the operation of dynamic packet batching module **122**, a TCP control packet may or may not be considered re-assembleable. Thus, depending upon the implementation of a particular embodiment the validity indicator for a packet that is a TCP control packet may be set to a valid or invalid state.

In an alternative embodiment of the invention an entry in memory **2102** is populated with information from the control queue entry identified by the second read pointer described above. This pointer may then be incremented to the next entry in control queue **118**.

In state **2228**, write pointer **2114** is incremented to the next entry of memory **2102**, after which the illustrated method ends at end state **2230**. If write pointer **2114** references the same entry as read pointer **2112**, it may be determined that memory **2102** is full. One skilled in the art will recognize that many other suitable methods of managing pointers for memory **2102** may be employed.

As mentioned above, in one embodiment of the invention one or both of the memory search and memory population operations run continuously. Thus, end state **2230** may be removed from the procedure illustrated in FIG. **22B**, in which case the procedure would return to state **2222** after state **2228**.

Advantageously, in the illustrated embodiment of the invention the benefits provided to the host computer by dynamic packet batching module **122** increase as the host computer becomes increasingly busy. In particular, the greater the load placed on a host processor, the more delay that will be incurred until a packet received from NIC **100** may be processed. As a result, packets may queue up in packet queue **116** and, the more packets in the packet queue, the more entries that can be maintained in memory **2102**.

The more entries that are stored in memory **2102**, the further ahead dynamic packet batching module can look for a related packet. The further ahead it scans, the more likely it is that a related packet will be found. As more related packets are found and identified to the host computer for collective processing, the amount of processor time spent on network traffic decreases and overall processor utilization increases.

One skilled in the art will appreciate that other systems and methods may be employed to identify multiple packets from a single communication flow or connection without exceeding the scope of the present invention.

Early Random Packet Discard in One Embodiment of the Invention

Packets may arrive at a network interface from a network at a rate faster than they can be transferred to a host computer. When such a situation exists, the network interface must often drop, or discard, one or more packets. Therefore, in one embodiment of the present invention a system and method for randomly discarding a packet are provided. Systems and methods discussed in this section may be applicable to other communication devices as well, such as gateways, routers, bridges, modems, etc.

As one skilled in the art will recognize, one reason that a packet may be dropped is that a network interface is already storing the maximum number of packets that it can store for transfer to a host computer. In particular, a queue that holds packets to be transferred to a host computer, such as packet queue **116** (shown in FIG. **1A**), may be fully populated when another packet is received from a network. Either the new packet or a packet already stored in the queue may be dropped.

Partly because of the bursty nature of much network traffic, multiple packets may often be dropped when a network interface is congested. And, in some network interfaces, if successive packets are dropped one particular network connection or flow (e.g., a connection or flow that includes all of the dropped packets) may be penalized even if it is not responsible for the high rate of packet arrival. If a network connection or flow is penalized too heavily, the network entity generating the traffic in that connection or flow may tear it down in the belief that a "broken pipe" has been encountered. As one skilled in the art will recognize, a broken pipe occurs when a network entity interprets a communication problem as indicating that a connection has been severed.

For certain network traffic (e.g., TCP traffic), the dropping of a packet may initiate a method of flow control in which a network entity's window (e.g., number of packets it transmits before waiting for an acknowledgement) shrinks or is reset to a very low number. Thus, every time a packet from a TCP communicant is dropped by a network interface at a receiving entity, the communicant must re-synchronize its connection with the receiving entity. If one or a subset of communicants are responsible for a large percentage of network traffic received at the entity, then it seems fair that those communicants should be penalized in proportion to the amount of traffic that it is responsible for.

In addition, it may be wise to prevent certain packets or types of packets from being discarded. For example, discarding a small control packet may do very little to alleviate congestion in a network interface and yet have a drastic and negative effect upon a network connection or flow. Further, if a network interface is optimized for packets adhering to a particular protocol, it may be more efficient to avoid dropping such packets. Even further, particular connections, flows or applications may be prioritized, in which case higher priority traffic should not be dropped.

Thus, in one embodiment of a network interface according to the present invention, a method is provided for randomly discarding a packet when a communication device's packet queue is full or is filled to some threshold level. Intelligence may be added to such a method by selecting certain types of packets for discard (e.g., packets from a particular flow, connection or application) or excepting certain types of packets from being discarded (e.g., control packets, packets conforming to a particular protocol or set of protocols).

A provided method is random in that discarded packets are selected randomly from those packets that are considered discardable. Applying a random discard policy may be sufficient to avoid broken pipes by distributing the impact of dropped packets among multiple connections or flows. In addition, if a small number of transmitting entities are responsible for a majority of the traffic received at a network interface, dropping packets randomly may ensure that the offending entities are penalized proportionately. Different embodiments of the invention that are discussed below provide various combinations of randomness and intelligence, and one of these attributes may be omitted in one or more embodiments.

FIG. **24** depicts a system and method for randomly discarding packets in a present embodiment of the invention. In this embodiment, packet queue **2400** is a hardware FIFO (e.g., first-in first-out) queue that is 16 KB in size. In other embodiments of the invention the packet queue may be smaller or larger or may comprise another type of data structure (e.g., list, array, table, heap) implemented in hardware or software.

Similar to packet queue **116** discussed in a previous section, packet queue **2400** receives packets from a network and holds them for transfer to a host computer. Packets arriving from a network may arrive from the network at a high rate and may be processed or examined by one or more modules (e.g., header parser **106**, flow database manager **108**) prior to being stored in packet queue **2400**. For example, where the network is capable of transmitting one gigabit of traffic per second, packets conforming to one set of protocols (e.g., Ethernet, IP and TCP) may be received at a rate of approximately 1.48 million packets per second. After being stored in packet queue **2400**, packets are transferred to a host computer at a rate partially dependent upon events and conditions internal to the host computer. Thus, the network interface may not be able to control the rate of packet transmittal to the host computer.

In the illustrated embodiment, packet queue **2400** is divided into a plurality of zones or regions, any of which may overlap or share a common boundary. Packet queue **2400** may be divided into any number of regions, and the invention is not limited to the three regions depicted in FIG. **24**. Illustratively, region zero (represented by the numeral **2402**) encompasses the portion of packet queue **2400** from 0 KB (e.g., no packets are stored in the queue) to 8 KB (e.g., half full). Region one (represented by the numeral **2404**) encompasses the portion of the packet queue from 8 KB to 12 KB. Region two (represented by the numeral **2406**) encompasses the remaining portion of the packet queue, from 12 KB to 16 KB. In an alternative embodiment, regions may only be defined for a portion of packet queue **2400**. For example, only the upper half (e.g., above 8 KB) may be divided into one or more regions.

The number and size of the different regions and the location of boundaries between the regions may vary according to several factors. Among the factors are the type of packets received at the network interface (e.g., the protocols according to which the packets are configured), the size of the packets, the rate of packet arrival (e.g., expected rate, average rate, peak rate), the rate of packet transfer to the host computer, the size of the packet queue, etc. For example, in another embodiment of the invention, packet queue **2400** is divided into five regions. A first region extends from 0 KB to 8 KB; a second region ranges from 8 KB to 10 KB; a third from 10 KB to 12 KB; a fourth from 12 KB to 14 KB; and a final region extends from 14 KB to 16 KB.

During operation of a network interface according to a present embodiment, traffic indicator **2408** indicates how full packet queue **2400** is. Traffic indicator **2408**, in one embodiment of the invention, comprises read pointer **810** and/or write pointer **812** (shown in FIG. **8**). In the presently discussed embodiment in which packet queue **2400** is fully partitioned, traffic indicator **2408** will generally be located in one of the regions into which the packet queue was divided or at a dividing boundary. Thus, during operation of a network interface appropriate action may be taken, as described below, depending upon how full the packet queue is (e.g., depending upon which region is identified by traffic indicator **2408**).

In FIG. **24**, counter **2410** is incremented as packets arrive at packet queue **2400**. In the illustrated embodiment, counter **2410** continuously cycles through a limited range of values, such as zero through seven. In one embodiment of the invention, each time a new packet is received the counter is incremented by one. In an alternative embodiment, counter **2410** may not be incremented when certain "non-discardable" packets are received. Various illustrative criteria for identifying non-discardable packets are presented below.

For one or more regions of packet queue **2400**, an associated programmable probability indicator indicates the probability that a packet will be dropped when traffic indicator **2408** indicates that the level of traffic in the packet queue has reached the associated region. Therefore, in the illustrated embodiment probability indicator **2412** indicates the probability that a packet will be dropped while the packet queue is less than half full (e.g., when traffic indicator **2408** is located in region zero). Similarly, probability indicators **2414** and **2416** specify the probability that a new packet will be dropped when traffic indicator **2408** identifies regions one and two, respectively.

In the illustrated embodiment, probability indicators **2412**, **2414** and **2416** each comprise a set, or mask, of sub-indicators such as bits or flags. Illustratively, the number of sub-indicators in a probability indicator matches the range of counter values—in this case, eight. In one embodiment of the invention, each sub-indicator may have one of two values (e.g., zero or one) indicating whether a packet is dropped. Thus, the sub-elements of a probability indicator may be numbered from zero to seven (illustratively, from right to left) to correspond to the eight possible values of counter **2410**. For each position in a probability indicator that stores a first value (e.g., one), when the value of counter **2410** matches the number of that bit, the next discardable packet received for packet queue **2400** will be dropped. As discussed above, certain types of packets (e.g., control packets) may not be dropped. Illustratively, counter **2410** is only incremented for discardable packets.

In FIG. **24**, probability indicator **2412** (e.g., 00000000) indicates that no packets are to be dropped as long as the packet queue is less than half full (e.g., as long as traffic indicator **2408** is in region zero). Probability indicator **2414** (e.g., 00000001) indicates that every eighth packet is to be dropped when there is at least 8 KB stored in the packet queue. In other words, when traffic indicator **2408** is located in region one, there is a 12.5% probability that a discardable packet will be dropped. In particular, when counter **2410** equals zero the next discardable packet, or a packet already stored in the packet queue, is discarded. Probability indicator **2416** (e.g., 01010101) specifies that every other discardable packet is to be dropped. There is thus a 50% probability that a discardable packet will be dropped when the queue is more than three-quarters full. Illustratively, when a packet is dropped, counter **2410** is still incremented.

NOAC Ex. 1015 Page 566

As another example, in the alternative embodiment described above in which the packet queue is divided into five regions, suitable probability indicators may include the following. For regions zero and one, 00000000; for region two, 00000001; for region three, 00000101; and for region four, 01111111. Thus, in this alternative embodiment, region one is treated as an extension to region zero. Further, the probability of dropping a packet has a wider range, from 0% to 87.5%.

In one alternative embodiment described above, only a portion of a packet queue is partitioned into regions. In this alternative embodiment, a default probability or null probability (e.g., 00000000) of dropping a packet may be associated with the un-partitioned portion. Illustratively, this ensures that no packets are dropped before the level of traffic stored in the queue reaches a first threshold. Even in an embodiment where the entire queue is partitioned, a default or null probability may be associated with a region that encompasses or borders a 0 KB threshold.

Just as a packet queue may be divided into any number of regions for purposes of the present invention, probability indicators may comprise bit masks of any size or magnitude, and need not be of equal size or magnitude. Further, probability indicators are programmable in a present embodiment, thus allowing them to be altered even during the operation of a network interface.

One skilled in the art will recognize that discarding packets on the basis of a probability indicator injects randomness into the discard process. A random early discard policy may be sufficient to avoid the problem of broken pipes discussed above. In particular, in one embodiment of the invention, all packets are considered discardable, such that all packets are counted by counter **2410** and all are candidates for being dropped. As already discussed, however, in another embodiment of the invention intelligence is added in the process of excluding certain types of packets from being discarded.

It will be understood that probability indicators and a counter simply constitute one system for enabling the random discard of packets in a network interface. Other mechanisms are also suitable. In one alternative embodiment, a random number generator may be employed in place of a counter and/or probability indicators to enable a random discard policy. For example, when a random number is generated, such as M, the Mth packet (or every Mth packet) after the number is generated may be dropped. Or, the random number may specify a probability of dropping a packet. The random number may thus be limited to (e.g., hashed into) a certain range of values or probabilities. As another alternative, a random number generator may be used in tandem with multiple regions or thresholds within a packet queue. In this alternative embodiment a programmable value, represented here as N, may be associated with a region or queue threshold. Then, when a traffic indicator reaches that threshold or region, the Nth packet (or every Nth packet) may be dropped until another threshold or boundary is reached.

In yet another alternative embodiment of the invention, the probability of dropping a packet is expressed as a binary fraction. As one skilled in the art will recognize, a binary fraction consists of a series of bits in which each bit represents one half of the magnitude of its more significant neighbor. For example, a binary fraction may use four digits in one embodiment of the invention. From left to right, the bits may represent 0.5, 0.25, 0.125 and 0.0625, respectively. Thus, a binary fraction of 1010 would be interpreted as indicating a 62.5% probability of dropping a packet (e.g., 50% plus 12.5%). The more positions (e.g., bits) used in a binary fraction, the greater precision that may be attained.

In one implementation of this alternative embodiment a separate packet counter is associated with each digit. The counter for the leftmost bit increments at twice the rate of the next counter, which increments twice as fast as the next counter, etc. In other words, when the counter for the most significant (e.g., left) bit increments from 0 to 1 the other counters do not change. When the most significant counter increments again, from 1 back to 0, then the next counter increments from 0 to 1. Likewise, the counter for the third bit does not increment from 0 to 1 until the second counter returns to 0. In summary, the counter for the most significant bit changes (i.e., increments) each time a packet is received. The counter for the next most significant bit maintains each value (i.e., 0 or 1) for two packets before incrementing. Similarly, the counter for the third most significant bit maintains each counter value for four packets before incrementing and the counter for the least significant bit maintains its values for eight packets before incrementing.

Each time a packet is received or a counter is incremented the counters are compared to the probability indicator (e.g., the specified binary fraction). In one embodiment the determination of whether a packet is dropped depends upon which of the fraction's bits are equal to one. Illustratively, for each fraction bit equal to one a random packet is dropped if the corresponding counter is equal to one and the counters for any bits of higher significance are equal to zero. Thus for the example fraction 1010, whenever the most significant bit's counter is equal to one a random packet is dropped. In addition, a random packet is also dropped whenever the counter for the third bit is equal to one and the counters for the first two bits are equal to zero.

A person skilled in the art may also derive other suitable mechanisms for specifying and enforcing a probability of dropping a packet received at a network interface without exceeding the scope of the present invention.

As already mentioned, intelligence may be imparted to a random discard policy in order to avoid discarding certain types of packets. In a previous section, methods of parsing a packet received from a network were described. In particular, in a present embodiment of the invention a packet received from a network is parsed before it is placed into a packet queue such as packet queue **2400**. During the parsing procedure various information concerning the packet may be gleaned. This information may be used to inject intelligence into a random discard policy. In particular, one or more fields of a packet header may be copied, an originating or destination entity of the packet may be identified, a protocol may be identified, etc.

Thus, in various embodiments of the invention, certain packets or types of packets may be immune from being discarded. In the embodiment illustrated in FIG. **24**, for example, control packets are immune. As one skilled in the art will appreciate, control packets often contain information essential to the establishment, re-establishment or maintenance of a communication connection. Dropping a control packet may thus have a more serious and damaging effect than dropping a packet that is not a control packet. In addition, because control packets generally do not contain data, dropping a control packet may save very little space in the packet queue.

Many other criteria for immunizing packets are possible. For example, when a packet is parsed according to a procedure described in a previous section, a No_Assist flag or signal may be associated with the packet to indicate whether the packet is compatible with a set of pre-selected

communication protocols. Illustratively, if the flag is set to a first value (e.g., one) or the signal is raised, the packet is considered incompatible and is therefore ineligible for certain processing enhancements (e.g., re-assembly of packet data, batch processing of packet headers, load-balancing). Because a packet for which a No_Assist flag is set to the first value may be a packet conforming to an unexpected protocol or unique format, it may be better not to drop such packets. For example, a network manager may want to ensure receipt of all such packets in order to determine whether a parsing procedure should be augmented with the ability to parse additional protocols.

Another reason for immunizing a No_Assist packet (e.g., packets that are incompatible with a set of selected protocols) from being discarded concerns the reaction to dropping the packet. Because the packet's protocols were not identified, it may not be known how the packet's protocols respond to the loss of a packet. In particular, if the sender of the packet does not lower its transmission rate in response to the dropped packet (e.g., as a form of congestion control), then there is no benefit to dropping it.

A packet's flow number may be used to immunize certain packets in another alternative embodiment of the invention. As discussed in a previous section, a network interface may include a flow database and flow database manager to maintain a record of multiple communication flows received by the network interface. It may be efficacious to prevent packets from one or more certain flows from being discarded. Immunized flows may include a flow involving a high-priority network entity, a flow involving a particular application, etc. For example, it may be considered relatively less damaging to discard packets from an animated or streaming graphics application in which a packet, or a few packets, may be lost without seriously affecting the destination entity and the packets may not even need to be retransmitted. In contrast, the consequences may be more severe if a few packets are dropped from a file transfer connection. The packets will likely need to be retransmitted, and the transmitting entity's window may be shrunk as a result—thus decreasing the rate of file transfer.

In yet another alternative embodiment of the invention, a probability indicator may comprise a bit mask in which each bit corresponds to a separate, specific flow through the network interface. In particular, the bits may correspond to the flows maintained in the flow database described in a previous section.

Although embodiments of the invention discussed thus far in this section involve discarding packets as they arrive at a packet queue, in an alternative embodiment packets may be discarded from within the packet queue. In particular, as the packet queue is filled (e.g., as a traffic indicator reaches pre-defined regions or thresholds), packets already stored in the queue may be discarded at random according to one or more probability indicators. In the embodiment illustrated in FIG. **24**, for example, when traffic indicator **2408** reaches a certain threshold, such as the boundary between regions one and two or the end of the queue, packets may be deleted in one or more regions according to related probability indicators. Such probability indicators would likely have different values than those indicated in FIG. **24**.

In a present embodiment of the invention, probability indicators and/or the specifications (e.g., boundaries) into which a packet queue is partitioned are programmable and may be adjusted by software operating on a host computer (e.g., a device driver). Criteria for immunizing packets may also be programmable. Methods of discarding packets in a network interface or other communication device may thus be altered in accordance with the embodiments described in this section, even during continued operation of such a device. Various other embodiments and criteria for randomly discarding packets and/or applying criteria for the intelligent discard of packets will be apparent to those skilled in the art.

FIGS. **25A–25B** comprise a flow chart demonstrating one method of implementing a policy for randomly discarding packets in a network interface according to the embodiment of the invention substantially similar to the embodiment illustrated in FIG. **24**. In this embodiment, a packet is received while packet queue **2400** is not yet full. As one skilled in the will appreciate, this embodiment provides a method of determining whether to discard the packet. Once packet queue **2400** is full, when another packet is received the network interface generally must drop a packet—either the one just received or one already stored in the queue—in which case the only decision is which packet to drop.

In FIG. **25A**, state **2500** is a start state. State **2500** may reflect the initialization of the network interface (and packet queue **2400**) or may reflect a point in the operation of the network interface at which one or more parameters or aspects concerning the packet queue and the random discard policy are to be modified.

In state **2502**, one or more regions are identified in packet queue **2400**, perhaps by specifying boundaries such as the 8 KB and 12 KB boundaries depicted in FIG. **24**. Although the regions depicted in FIG. **24** fully encompass packet queue **2400** when viewed in unison, regions in an alternative embodiment of the invention may encompass less than the entire queue.

In state **2504**, one or more probability indicators are assigned and configured. In the illustrated embodiment, one probability indicator is associated with each region. Alternatively, multiple regions may be associated with one probability indicator. Even further, one or more regions may not be explicitly associated with a probability indicator, in which case a default or null probability indicator may be assumed. As described above, a probability indicator may take the form of a multi-bit mask, whereby the number of bits in the mask reflect the range of possible values maintained by a packet counter. In another embodiment of the invention, a probability indicator may take the form of a random number or a threshold value against which a randomly generated number is compared when a decision must be whether to discard a packet.

In state **2506**, if certain types of packets are to be prevented from being discarded, criteria are expressed to identify the exempt packets. Some packets that may be exempted are control packets, packets conforming to unknown or certain known protocols, packets belonging to a particular network connection or flow, etc. In one embodiment of the invention, no packets are exempt from being discarded.

In state **2508**, a packet or traffic counter is initialized. As described above, the counter may be incremented, possibly through a limited range of values, when a discardable packet is received for storage in packet queue **2400**. The limited range of counter values may correspond to the number of bits in a mask form of a probability indicator. Alternatively, the counter may be configured to increment through a greater range, in which case a counter value may be filtered through a modulus or hash function prior to being compared to a probability indicator as described below.

In state **2510**, a packet is received from a network and may be processed through one or more modules (e.g., a header parser, an IPP module) prior to its arrival at packet

queue **2400**. Thus, in state **2510** the packet is ready to be stored in the packet queue. One or more packets may already be stored in the packet queue and a traffic indicator (e.g., a pointer or index) identifies the level of traffic stored in the queue (e.g., by a storage location and/or region in the queue).

In state **2512**, it may be determined whether the received packet is discardable. For example, if the random discard policy that is in effect allows for the exemption of some packets from being discarded, in state **2512** it is determined whether the received packet meets any of the exemption criteria. If so, the illustrated procedure continues at state **2522**. Otherwise, the procedure continues at state **2514**.

In state **2514**, an active region of packet queue **2400** is identified. In particular, the region of the packet queue to which the queue is presently populated with traffic is determined. The level of traffic stored in the queue depends upon the number and size of packets that have been stored in the queue to await transfer to a host computer. The slower the transfer process, the higher the level of traffic may reach in the queue. Although the level of traffic stored in the queue rises and falls as packets are stored and transferred, the level may be identified at a given time by examining the traffic indicator. The traffic indicator may comprise a pointer identifying the position of the last or next packet to be stored in the queue. Such a pointer may be compared to another pointer that identifies the next packet to be transferred to the host computer in order to reveal how much traffic is stored in the queue.

In state **2516**, the counter value (e.g., a value between zero and seven in the embodiment of FIG. **24**) is compared to the probability indicator associated with the active region. As previously described, the counter is incremented as discardable packets are received at the queue. This comparison is conducted so as to determine whether the received packet should be discarded. As explained above, in the embodiment of FIG. **24** the setting of the probability indicator bit corresponding to the counter value is examined. For example, if the counter has a value of N, then bit number N of the probability indicator mask is examined. If the bit is set to a first state (e.g., one) the packet is to be discarded; otherwise it is not to be discarded.

In state **2518**, the counter is incremented to reflect the receipt of a discardable packet, whether or not the packet is to be discarded. In the presently discussed embodiment of the invention, if the counter contains its maximum value (e.g., seven) prior to being incremented, incrementing it fails resetting it to its minimum value (e.g., zero).

In state **2520**, if the packet is to be discarded the illustrated procedure continues at state **2524**. Otherwise, the procedure continues at state **2522**. In state **2522**, the packet is stored in packet queue **2400** and the illustrated procedure ends with state **2526**. In state **2524**, the packet is discarded and the illustrated procedure ends with end state **2526**.

Sun, Sun Microsystems, SPARC and Solaris are trademarks or registered trademarks of Sun Microsystems, Incorporated in the United States and other countries.

The foregoing descriptions of embodiments of the invention have been presented for purposes of illustration and description only. They are not intended to be exhaustive or to limit the invention to the forms disclosed. Many modifications and variations will be apparent to practitioners skilled in the art. Accordingly, the above disclosure is not intended to limit the invention; the scope of the invention is defined by the appended claims.

What is claimed is:

1. A method of transferring a packet to a computer system, wherein the packet is received at a communication device in a network, comprising:

parsing a header portion of a first packet received at a communication device to determine if said first packet conforms to a pre-selected protocol;

generating a flow key to identify a first communication flow that includes said first packet;

transferring said first packet to a host computer system for processing in accordance with said pre-selected protocol; and

associating an operation code with said first packet, wherein said operation code indicates a status of said first packet.

2. The method of claim **1**, wherein said parsing comprises:

copying a header portion of said first packet into a header memory; and

examining said header portion according to a series of parsing instructions;

wherein said parsing instructions are configured to reflect a set of pre-selected communication protocols.

3. The method of claim **2**, wherein said parsing instructions are updateable.

4. The method of claim **2**, further comprising copying a value from a field in a header of said header portion.

5. The method of claim **1**, wherein said parsing comprises:

extracting an identifier of a source of said first packet from said header portion; and

extracting an identifier of a destination of said first packet from said header portion.

6. The method of claim **5**, wherein said generating comprises combining said source identifier and said destination identifier.

7. The method of claim **1**, wherein said generating comprises retrieving an identifier of a communication connection from said header portion.

8. The method of claim **1**, further comprising storing said first packet in a packet memory prior to said transferring.

9. The method of claim **1**, further comprising storing said flow key in a flow database, wherein said flow database is configured to facilitate management of said first communication flow.

10. The method of claim **9**, further comprising associating a flow number with said first packet, wherein said flow number comprises an index of said flow key within said flow database.

11. The method of claim **10**, further comprising storing said flow number in a flow memory.

12. The method of claim **9**, further comprising updating an entry in said flow database associated with said flow key when a second packet in said first communication flow is received.

13. A computer readable storage medium storing instructions that, when executed by a computer, cause the computer to perform a method of transferring a packet received at a network interface from a network to a host computer system, the method comprising:

receiving a packet from a network at a network interface for a host computer system;

parsing a header portion of said packet to extract an identifier of a source entity and an identifier of a destination entity;

generating a flow key from said source identifier and said destination identifier to identify a communication flow comprising said packet;

determining whether a header in said header portion conforms to a pre-selected protocol;

storing said flow key in a database;

associating an operation code with said packet, wherein said operation code identifies a status of said packet;

storing said packet in a packet memory;

if said header conforms to said pre-selected protocol:

storing a data portion of said packet in a re-assembly buffer; and

storing said header portion in a header buffer; and

if said header conforms to a protocol other than said pre-selected protocol, storing said packet in a non-re-assembly buffer.

14. The method of claim **1**, wherein said associating comprises:

retrieving one or more header fields of said header portion; and

analyzing said header fields to determine said status of said first packet.

15. The method of claim **14**, wherein said analyzing comprises:

determining whether said first packet includes a data portion; and

if said first packet includes a data portion, determining whether said data portion exceeds a pre-determined size.

16. The method of claim **14**, wherein said analyzing comprises determining whether said first packet was received out of order in said first communication flow.

17. The method of claim **1**, further comprising storing said operation code in a control memory.

18. The method of claim **1**, wherein said first pack et is determined to conform to said pre-selected protocol, said transferring comprising:

storing a data portion of said first packet in a re-assembly storage area, wherein said re-assembly storage area is configured to only store data portions of packets in said first communication flow; and

storing one or more headers from said header portion in a header storage area.

19. The method of claim **1**, wherein said transferring comprises:

if said first packet is smaller than a predetermined threshold, storing said first packet in a first storage area; and

if said first packet is larger than said predetermined threshold, storing said first packet in a second storage area.

20. The method of claim **1**, further comprising determining whether a second packet received from said network is part of said first communication flow.

21. The method of claim **20**, wherein said determining comprises:

maintaining a packet memory configured to store one or more packets received from said network;

maintaining a flow memory configured to store, for each of said one or more packets, an identifier of a communication flow comprising said packet; and

searching said flow memory for a first identifier of said first communication flow.

22. The method of claim **21**, wherein said first identifier comprises said flow key.

23. The method of claim **21**, wherein said first identifier comprises a flow number of said first packet, wherein said flow number is an index of said flow key within a flow database.

24. The method of claim **1**, wherein said host computer system comprises a plurality of processors, further comprising:

identifying a quantity of processors in said host computer system available for processing packets; and

associating a first processor identifier with said first packet to identify a first processor in said host computer system for processing said first packet.

25. The method of claim **24**, further comprising:

receiving a second packet in said first communication flow; and

associating said first processor identifier with said second packet.

26. The method of claim **24**, further comprising:

receiving a second packet from a second communication flow; and

associating a second processor identifier with said second packet to identify a second processor in said host computer system for processing said second packet.

27. The method of claim **1**, further comprising alerting said host computer system to the arrival of said first packet.

28. The method of claim **1**, further comprising:

maintaining a packet memory configured to store packets received from said network; and

randomly discarding a packet if said packet memory contains a pre-determined level of traffic.

29. The method of claim **28**, wherein said packet is said first packet.

30. The method of claim **28**, wherein said packet memory comprises a plurality of regions, said randomly discarding comprising:

identifying one of said plurality of regions, wherein a level of traffic stored in said packet memory has reached said region;

applying a probability indicator associated with said region to determine a probability that said first packet should be discarded; and

if said probability exceeds a predetermined threshold, discarding said first packet.

31. The method of claim **1**, wherein said communication device is a network interface.

32. A method of transferring a packet received at a network interface to a host computer system, comprising:

receiving a packet from a network;

storing said packet in a packet memory;

parsing a header portion of said packet;

extracting a value stored in said header portion;

identifying a communication flow comprising said packet;

determining whether a header in said header portion conforms to a pre-selected protocol;

determining whether a second packet in said packet memory is part of said communication flow;

if the host computer system contains a plurality of processors, identifying a processor to process said packet; and

storing said packet in a host memory area.

33. A method of transferring a packet received at a network interface from a network to a host computer system, comprising:

receiving a packet from a network at a network interface for a host computer system;

parsing a header portion of said packet to extract an identifier of a source entity and an identifier of a destination entity;

generating a flow key from said source identifier and said destination identifier to identify a communication flow comprising said packet;

determining whether a header in said header portion conforms to a pre-selected protocol;

storing said flow key in a database;

associating an operation code with said packet, wherein said operation code identifies a status of said packet;

storing said packet in a packet memory;

if said header conforms to said pre-selected protocol:

storing a data portion of said packet in a re-assembly buffer; and

storing said header portion in a header buffer; and

if said header conforms to a protocol other than said pre-selected protocol, storing said packet in a non-re-assembly buffer.

34. The method of claim **33**, wherein said parsing comprises executing a series of updateable instructions configured to parse a packet header conforming to one of a set of pre-selected protocols.

35. The method of claim **33**, further comprising storing said operation code in a control memory.

36. The method of claim **33**, further comprising storing a flow number of said packet in a flow memory, wherein said flow number comprises an index of said flow key in said database.

37. The method of claim **36**, further comprising indicating whether said packet memory includes another packet with said flow number or said flow key.

38. The method of claim **33**, wherein the host computer system comprises multiple processors, further comprising identifying a first processor in the host computer system to process said packet in accordance with said pre-selected protocol.

39. The method of claim **38**, further comprising:

receiving a second packet at said network interface, wherein said second packet is part of a second communication flow; and

identifying a second processor in the host computer system to process said second packet.

40. The method of claim **33**, further comprising informing said host computer system of said receipt of said packet.

41. The method of claim **33**, wherein said packet memory comprises a plurality of regions, further comprising:

determining a level of network traffic stored in said packet memory; and

applying a probability indicator associated with one of said regions to determine whether to discard a packet received from said network.

42. An apparatus for transferring a packet to a host computer system, comprising:

a traffic classifier configured to classify a first packet received from a network by a communication flow that includes said first packet;

a packet memory configured to store said first packet;

a packet batching module configured to determine whether another packet in said packet memory belongs to said communication flow; and

a flow re-assembler configured to re-assemble a data portion of said first packet with a data portion of a second packet in said communication flow;

wherein said first packet data portion and said second packet data portion are stored in a host computer memory area to enable efficient transfer of said memory area contents.

43. The apparatus of claim **42**, wherein said traffic classifier comprises:

a parser configured to parse a header portion of said first packet;

a flow database configured to store a flow key identifying said communication flow; and

a flow database manager configured to manage said flow database;

wherein said flow key is generated from an identifier of a source of said first packet and an identifier of a destination of said first packet.

44. A computer system for receiving a packet from a network, comprising:

a memory configured to store packets received from a network; and

a communication device configured to receive a first packet from said network, the communication device comprising:

a parser configured to extract information from a header portion of a first packet;

a flow manager configured to examine said information;

a flow database configured to store an identifier of a first communication flow comprising multiple packets, including said first packet; and

a re-assembler for storing data portions of said multiple packets in a first portion of said memory; and

a processor for processing said first packet.

45. The apparatus of claim **42**, further comprising:

a load distributor for identifying a first processor within the host computer system for processing said first packet and said second packet;

wherein said load distributor identifies a second processor in the host computer system for processing a packet from a different communication flow.

46. The apparatus of claim **42**, further comprising:

a probability indicator for determining a probability of discarding a packet at said packet memory when a level of traffic stored in said packet memory is within a pre-determined region associated with said probability indicator.

47. A device for receiving a packet from a network and transferring the packet to a host computer system, comprising:

a parser configured to parse a header portion of a packet received from a network, wherein said parsing comprises:

determining whether a header within said header portion conforms to one of a set of communication protocols; and

if said header conforms to one of said communication protocols,

extracting information from said header portion to identify a communication flow to which said packet belongs;

a flow memory configured to store a flow identifier for identifying said communication flow;

a flow manager configured to assign an operation code to said packet, wherein said operation code:

indicates a status of said packet; and

indicates a manner of transferring said packet to the host computer system;

a packet memory configured to store said packet; and

a transfer module configured to transfer said packet from said packet memory to a host computer system in accordance with said operation code.

48. The device of claim **47**, wherein the device is a network interface.

49. The device of claim 47, said flow memory comprising flow database configured to store a flow key, wherein said ow key is assembled from an identifier of a source of said cket and an identifier of a destination of said packet.

50. The device of claim 47, wherein said flow manager is rther configured to update said flow memory as additional ckets in said communication flow are received from the twork.

51. The device of claim 47, said flow memory comprising flow memory configured to store a flow number, wherein id flow number comprises an index of said communication ow in a flow database.

52. The device of claim 47, further comprising a control emory configured to store said operation code.

53. A computer readable storage medium storing instructons that, when executed by a computer, cause the computer perform a method of transferring a packet from a communication device to a host computer, the method comprising:

parsing a header portion of a first packet received at a communication device to determine if said first packet conforms to a pre-selected protocol;

generating a flow key to identify a first communication flow that includes said first packet;

transferring said first packet to a host computer system for processing in accordance with said pre-selected protocol; and

associating an operation code with said first packet, wherein said operation code indicates a status of said first packet.

54. The device of claim 47, wherein said host computer stem is a multi-processor host computer system, further mprising a load distributor configured to select one of said ultiple processors for processing said packet in accordance ith one of said communication protocols.

55. The device of claim 47, wherein said transfer module configured to transfer a data portion of said packet into e of a set of host memory areas in accordance with said eration code.

56. The device of claim 47, further comprising a packet tching module configured to determine whether said ket memory contains another packet in said communion flow.

57. The device of claim 47, wherein said packet memory prises multiple regions, and wherein each of said mul- regions is associated with a probability indicator conid to indicate a probability of discarding a packet ved at the device.

58. An apparatus for transferring a packet from a network ost computer system, comprising:

a parser module configured to:

parse a header portion of a first packet received from a network to extract an identifier of a source of said first packet and an identifier of a destination of said first packet;

generate a flow key from said source identifier and said destination identifier to identify a communication flow comprising said first packet; and

determine whether a header in said header portion conforms to a pre-selected

protocol;

a flow database configured to store said flow key;

a flow database manager configured to associate an operation code with said first packet, wherein said operation code identifies a status of said first packet;

a packet memory configured to store said first packet; and

a transfer module configured to:

if said header conforms to said pre-selected protocol:

store a data portion of said first packet in a re-assembly buffer; and

store said header portion in a header buffer; and

if said header conforms to a protocol other than said pre-selected protocol,

store said packet in a non-re-assembly buffer.

59. The apparatus of claim 58, wherein said transfer module comprises a re-assembly engine configured to re-assemble, in said re-assembly buffer, a data portion of said first packet with a data portion of a second packet in said first communication flow.

60. The apparatus of claim 58, further comprising a flow memory configured to store a flow number associated with said first packet, wherein said flow number comprises an index of said flow key in said flow database.

61. The apparatus of claim 58, further comprising:

a load distributor configured to identify a first processor in said host computer system for processing said first packet, said first processor being identified on the basis of said flow key;

wherein said host computer system is a multi-processor computer system; and

wherein a second processor in said host computer system is identified for processing a packet from a communication flow other than said first communication flow.

62. The apparatus of claim 58, further comprising:

a packet batching module configured to determine whether said packet memory includes another packet in said first communication flow.

* * * * *

US006118760A

United States Patent [19]

Zaumen et al.

[11] Patent Number: 6,118,760

[45] Date of Patent: Sep. 12, 2000

[54] **MANAGEMENT OF ENTRIES IN A NETWORK ELEMENT FORWARDING MEMORY**

[75] Inventors: **William T. Zaumen**, Palo Alto, Calif.; **Donald L. Hoffman**, Portland, Oreg.; **Shree Murthy**, Sunnyvale, Calif.

[73] Assignee: **Sun Microsystems, Inc.**, Mountain View, Calif.

[21] Appl. No.: **08/884,946**

[22] Filed: **Jun. 30, 1997**

[51] **Int. Cl.**[7] **H04Q 11/04**

[52] **U.S. Cl.** **370/229**; 370/254; 370/389; 370/428; 370/475

[58] **Field of Search** 370/229, 330, 370/381, 386, 428, 469, 475, 237, 254, 389, 396, 397, 401, 402, 468; 365/49; 395/180, 182.03

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,652,874	3/1987	Loyer	340/825.05
4,850,042	7/1989	Petronio et al.	455/606
4,922,503	5/1990	Leone	370/85.13
4,933,938	6/1990	Sheehy	370/85.13
5,150,358	9/1992	Punj et al.	370/468
5,210,746	5/1993	Maher et al.	370/386
5,220,562	6/1993	Takada et al.	370/401
5,251,205	10/1993	Callon et al.	370/60
5,293,379	3/1994	Carr	370/474
5,309,437	5/1994	Perlman et al.	340/827
5,343,471	8/1994	Cassagnol	370/85.13
5,386,413	1/1995	McAuley et al.	370/54
5,392,432	2/1995	Engelstad et al.	395/180
5,394,402	2/1995	Ross	370/94.1
5,410,540	4/1995	Aiki et al.	370/390
5,422,838	6/1995	Lin	365/49
5,425,028	6/1995	Britton et al.	370/94.1
5,459,714	10/1995	Lo et al.	370/13.1
5,459,717	10/1995	Mullan et al.	370/351
5,461,624	10/1995	Mazzola	370/402
5,485,455	1/1996	Dobbins et al.	370/255
5,485,578	1/1996	Sweazey	395/200.54
5,490,139	2/1996	Baker et al.	370/60
5,490,252	2/1996	Macera et al.	395/200.01
5,500,860	3/1996	Perlman et al.	370/85.13
5,561,666	10/1996	Christensen et al.	370/434
5,572,522	11/1996	Calamvokis et al.	370/395
5,594,727	1/1997	Kolbenson et al.	370/468
5,633,865	5/1997	Short	370/412
5,636,371	6/1997	Yu	395/500
5,640,605	6/1997	Johnson et al.	395/881
5,651,002	7/1997	Van Seters et al.	370/392
5,675,741	10/1997	Aggarwal et al.	370/200.12

(List continued on next page.)

OTHER PUBLICATIONS

"Foundry Products", downloaded from Website http://www.foundrynet.com/ on Jun. 19, 1997.

Anthony J. McAuley & Paul Francis, "Fast Routing Table Lookup Using CAMs", IEEE, 1993, pp. 1382–1390.

"Gigabit Ethernet", Network Strategy Report, The Burton Group, v2, May 8, 1997 40 pages.

"IP On Speed", Erica Roberts, Internet–Draft, Data Communications on the Web, Mar. 1997, 12 pages.

"Multilayer Topology", White Paper, Internet–Draft, 13 pages, downloaded from website http://wwwbaynetworks.com on Apr. 18, 1997.

Primary Examiner—Douglas W. Olms

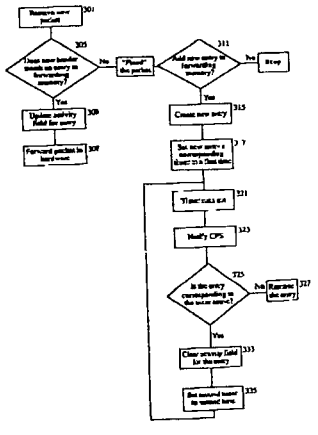
Assistant Examiner—Shick Hom

Attorney, Agent, or Firm—Blakely Sokoloff Taylor & Zafman

[57] **ABSTRACT**

The invention generally provides for a network element and methods in the network element for allowing a matching entry in a forwarding memory to be found in a single search of the memory, for determining when an entry should be placed in the memory, and for determining when an entry should be removed from the memory, in order to make more efficient use of the fixed space available in the memory. The invention is particularly useful in making more efficient use of a Content Addressable Memory (CAM) for storing flow entries, and configuring the CAM to index an associated memory that stores forwarding and quality of service information for each CAM entry.

36 Claims, 3 Drawing Sheets



U.S. PATENT DOCUMENTS

5,724,358	3/1998	Headrick et al.	370/418
5,734,865	3/1998	Yu	395/500
5,740,171	4/1998	Mazzola et al.	370/392
5,740,175	4/1998	Wakeman et al.	395/422
5,740,375	4/1998	Dunne et al.	395/200.68
5,742,604	4/1998	Edsall et al.	370/401
5,742,760	4/1998	Picazo, Jr. et al.	370/351
5,745,048	4/1998	Taguchi et al.	370/330
5,748,905	5/1998	Hauser et al.	395/200.79
5,751,967	5/1998	Raab et al.	395/200.58
5,754,540	5/1998	Liu et al.	370/315
5,764,634	6/1998	Christensen et al.	370/401
5,784,573	7/1998	Szczepanek et al.	395/200.8
5,790,546	8/1998	Dobbins et al.	370/400
5,802,052	9/1998	Venkataraman	370/395

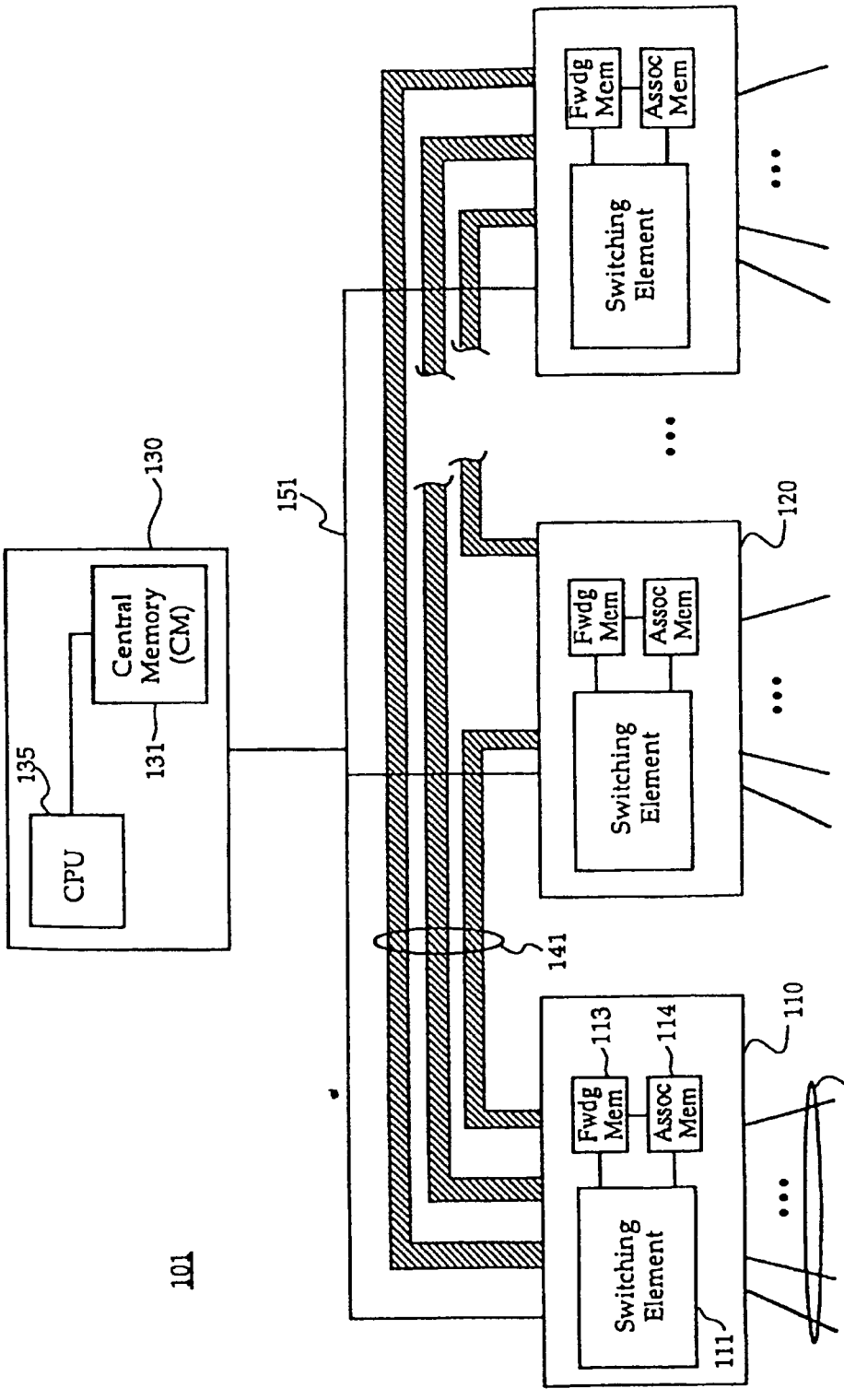

101
130
Central Memory (CM)
131
135
CPU
151
141
120
Switching Element
Fwdg Mem
Assoc Mem
110
113
114
111
117

FIG. 1

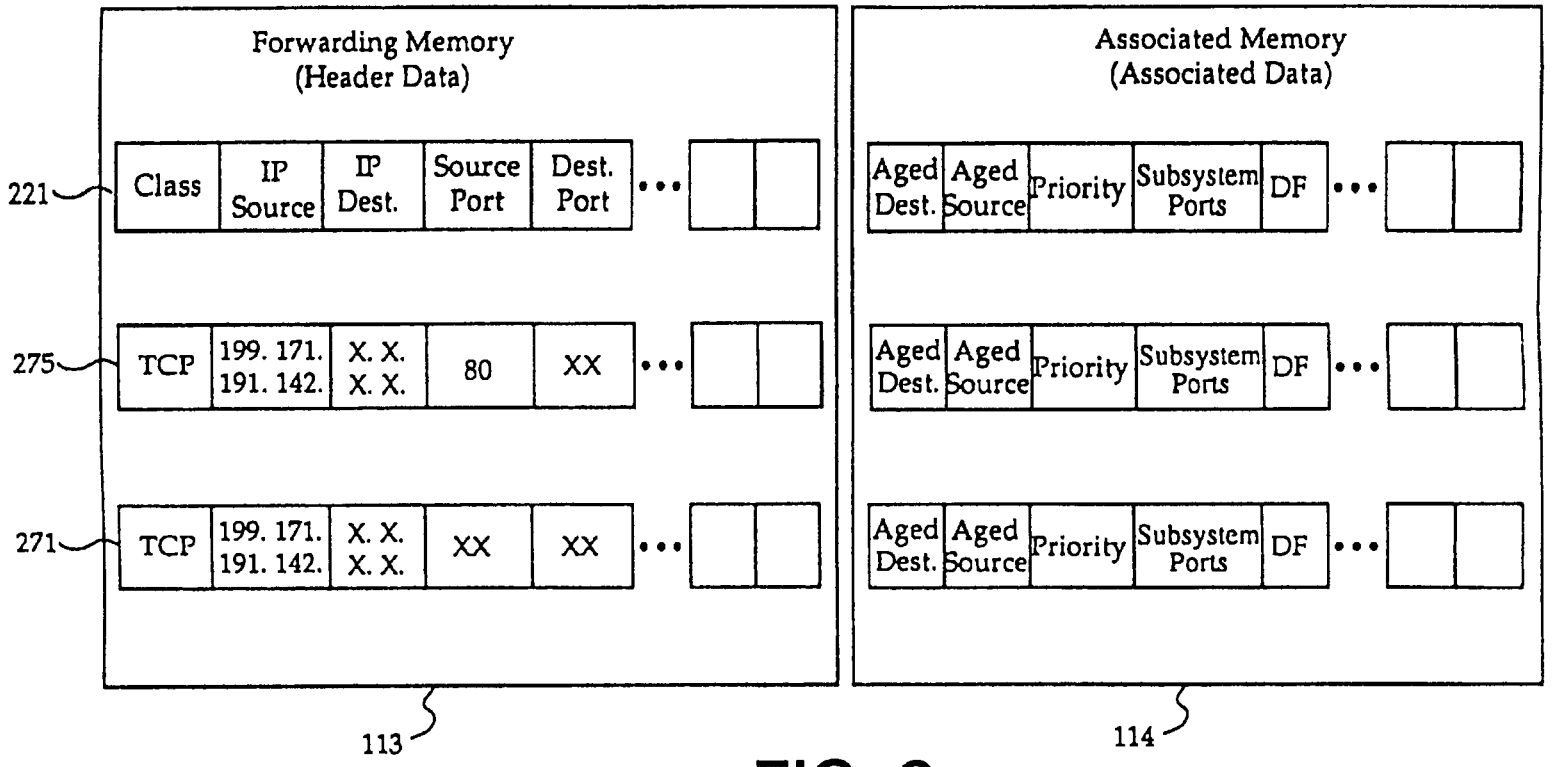

Forwarding Memory
(Header Data)
Associated Memory
(Associated Data)
221
Class
IP Source
IP Dest.
Source Port
Dest. Port
Aged Dest.
Aged Source
Priority
Subsystem Ports
DF
275
TCP
199. 171. 191. 142.
X. X. X. X.
80
XX
271
TCP
199. 171. 191. 142.
X. X. X. X.
XX
XX
113
114

FIG. 2

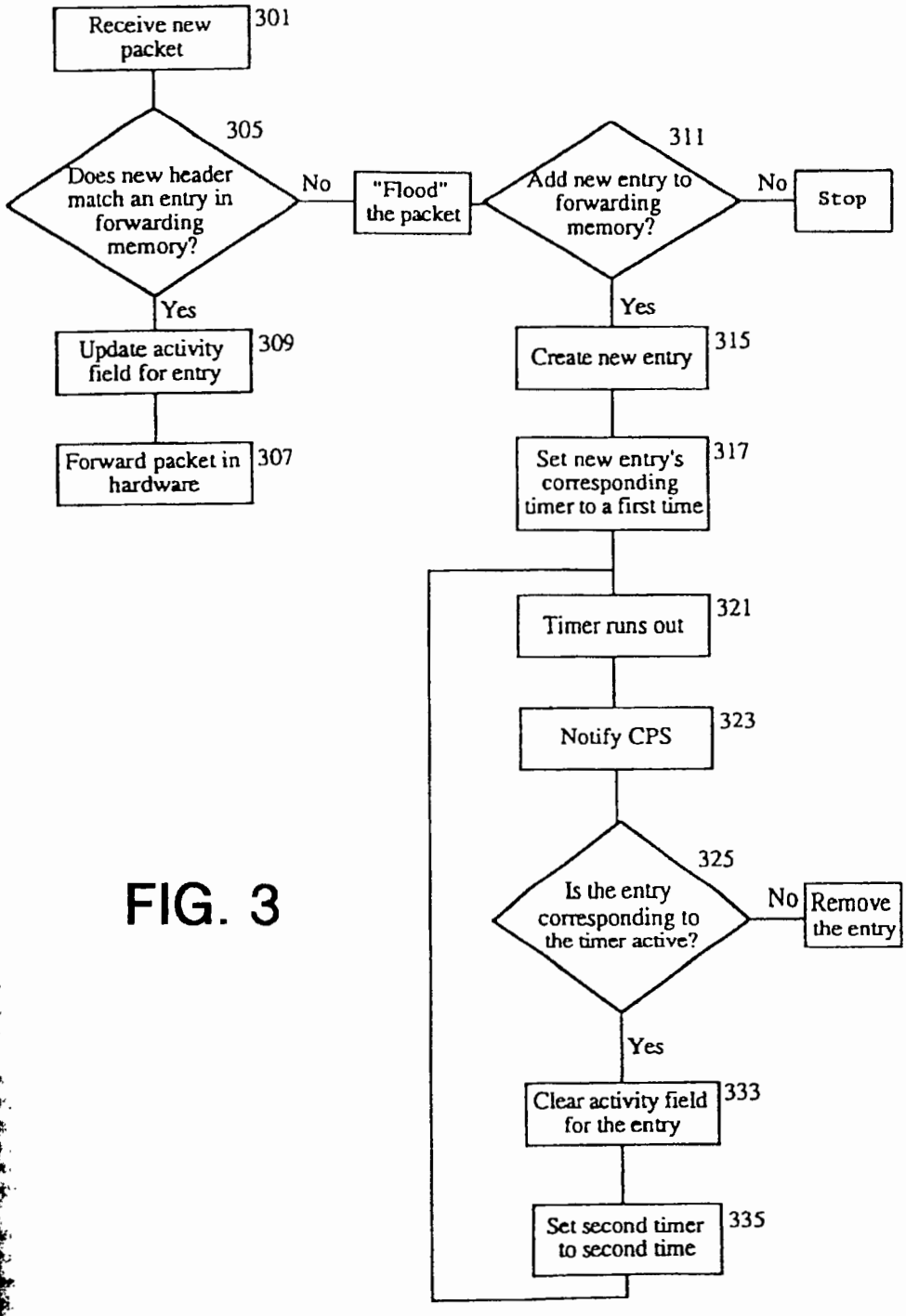

Receive new packet
301
Does new header match an entry in forwarding memory?
305
No
"Flood" the packet
Add new entry to forwarding memory?
311
No
Stop
Yes
Update activity field for entry
309
Forward packet in hardware
307
Yes
Create new entry
315
Set new entry's corresponding timer to a first time
317
Timer runs out
321
Notify CPS
323
Is the entry corresponding to the timer active?
325
No
Remove the entry
327
Yes
Clear activity field for the entry
333
Set second timer to second time
335

FIG. 3

MANAGEMENT OF ENTRIES IN A NETWORK ELEMENT FORWARDING MEMORY

BACKGROUND

1. Field of the Invention

The invention is generally related to network elements for communication between computers, and more specifically related to the creation and elimination of entries in a forwarding memory.

2. Description of Related Art

Communication between computers has become an important aspect of everyday life in both private and business environments. Computers converse with each other based upon a physical medium for transmitting the messages back and forth, and upon a set of rules implemented by electronic hardware attached to and programs running on the computers. These rules, often called protocols, define the orderly transmission and receipt of messages in a network of connected computers.

A local area network (LAN) is the most basic and simplest network that allows communication between a source computer and destination computer. The LAN can be envisioned as a cloud to which computers (also called endstations or end-nodes) that wish to communicate with one another are attached. At least one network element will connect with all the endstations in the LAN. An example of a simple network element is the repeater which is a physical layer device that forwards bits. The repeater may have a number of ports, each endstation being attached to one port. The repeater receives bits that may form a packet of data that contains a message from a source endstation, and blindly forwards the packet bit-by-bit. The bits are then received by other endstations in the LAN, including the destination. A single LAN, however, may be insufficient to meet the requirements of an organization that has many endstations, because of the limited number of physical connections available to and the limited message handling capability of the repeater. Thus, because of these physical limitations, a repeater-based approach can support only a limited number of endstations over a limited geographical area.

The capability of computer networks, however, has been extended by connecting different subnetworks to form larger networks that contain thousands of endstations communicating with each other. These LANs can in turn be connected to each other to create even larger enterprise networks, using wide area network (WAN) links.

To facilitate communication between subnets in a larger network, more complex electronic hardware and software have been proposed and are currently used in conventional networks. Also, new sets of rules for reliable and orderly communication among those endstations have been defined as standards based on the principle that the endstations interconnected by suitable network elements define a network hierarchy, where endstations within the same subnet have a common classification. A network is thus said to have a topology which defines the features and physical position of nodes and endstations within the network.

The interconnection of endstations through packet switching networks has traditionally followed a peer-to-peer network architectural abstraction. In such a model, a given source computer communicates with the same layer peer endstation (usually the destination) across the network by attaching a header to the data unit received from a higher layer, a layer provides services to enable the operation of the layer above it. A received packet will typically have several headers that were added to the original payload by the different layers operating at the source.

There are several layer partitioning schemes in the prior art, such as the Arpanet and the Open Systems Interconnect (OSI) models. The seven layer OSI model used here to describe the invention is a convenient model for mapping the functionality and detailed implementations of other models. Aspects of the Arpanet, however, (now redefined by the Internet Engineering Task Force, or IETF) will also be used in specific implementations of the invention to be discussed below.

The relevant layers for background purposes here are Layer 1 (physical), Layer 2 (data link), and Layer 3 (network), and to a limited extent Layer 4 (transport). A brief summary of the functions associated with these layers follows.

The physical layer transmits unstructured bits of information across a communication link. The repeater is an example of a network element that operates in this layer. The physical layer concerns itself with such issues as the size and shape of connectors, conversion of bits to electrical signals, and bit-level synchronization.

Layer 2 provides for transmission of frames of data and error detection. More importantly, the data link layer as referred to in this invention is typically designed to "bridge," or carry a packet of information across a single hop, i.e., a hop being the journey taken by a packet in going from one node to another. By spending only minimal time processing a received packet before sending the packet to its next destination, the data link layer can forward a packet much faster than the layers above it, which are discussed next. The data link layer provides addressing that may be used to identify a source and a destination between any computers interconnected at or below the data link layer. Examples of Layer 2 bridging protocols include those defined in IEEE 802 such as CSMA/CD, token bus, and token ring (including Fiber Distributed Data Interface, or FDDI).

Similar to Layer 2, Layer 3 also includes the ability to provide addresses of computers that communicate with each other. The network layer, however, also works with topological information about the network hierarchy. The network layer may also be configured to "route" a packet from the source to a destination using the shortest path. Finally, the network layer can control congestion by simply dropping selected packets, which the source might recognize as a request to reduce the packet rate.

Finally, Layer 4, the transport layer, provides an application program such as an electronic mail program with a "port address" which the application can use to interface with Layer 3. A key difference between the transport layer and the lower layers is that a program on the source computer carries a conversation with a similar program on the destination computer, whereas in the lower layers, the protocols are between each computer and its immediate neighbors in the network, where the ultimate source and destination endstations may be separated by a number of intermediate nodes. Examples of Layer 4 and Layer 3 protocols include the Internet suite of protocols such as TCP (Transmission Control Protocol) and IP (Internet Protocol).

Endstations are the source and ultimate destination of a packet, whereas a node refers to an intermediate point between the endstations. A node will typically include a network element which has the capability to receive and forward messages on a packet-by-packet basis.

Generally speaking, the larger and more complex networks typically rely on nodes that have higher layer (Layers 3 and 4) functionalities. A very large network consisting of several smaller subnetworks must typically use a Layer 3 network element known as a router which has knowledge of the topology of the subnetworks.

A router can form and store a topological map of the network around it based upon exchanging information with its neighbors. If a LAN is designed with Layer 3 addressing capability, then routers can be used to forward packets between LANs by taking advantage of the hierarchical routing information available from the endstations. Once a table of endstation addresses and routes has been compiled by the router, packets received by the router can be forwarded after comparing the packet's Layer 3 destination address to an existing and matching entry in the memory.

As packets are being transmitted and received between endstations, they must traverse intermediate nodes in which a network element receives a packet and forwards it towards its final destination. When a packet arrives at such an element, the packet's headers (or portions of the headers) are collected and an attempt is then made to match using hardwired logic the destination of the packet to a known Layer 2 destination address in an address table stored in a forwarding memory. Alternatively, a matching cycle may be performed in software based on the Layer 3 information contained in the received packet's header.

As a high performance device, the forwarding memory must be able to produce the matching entry from among thousands of entries in a relatively short time. This speed is important because the decision to forward the received packet to its destination must be made as soon as possible after receiving the packet to make room for other arriving packets.

Existing techniques for implementing the forwarding memory include the use of a content addressable memory (CAM) to store a Layer 2 address table. The CAM is typically implemented as a hardwired logic integrated circuit and is nicely suited for the job of storing an address table in a network element. The high performance offered by a CAM, however, comes at a high cost because of its unique and complex hardware requirements. Therefore, a CAM for use as a forwarding memory in networking applications is typically selected to provide space for only a limited number of entries.

One way to make efficient use of the forwarding memory is to reduce the number of entries by discarding stale entries after a given time has elapsed, the rationale being that since there has not been much activity in the past, there will not be much in the future. The price paid for using this approach is that if a subsequent matching packet does arrive fairly quickly after the entry is discarded, then a new identical entry must be created. This will interfere with both header processing and forwarding memory operations. It would be desirable, therefore, to devise a technique for even more efficient use of the forwarding memory in a networking application. What is needed is a network element that makes efficient use of the forwarding memory in order to forward packets faster and save costly storage space in the forwarding memory.

SUMMARY

The invention generally provides for a network element and methods in the network element for allowing a matching entry in a forwarding memory to be found in a single search of the memory, for determining when an entry should be placed in the memory, and for determining when an entry should be removed from the memory, in order to make more efficient use of the fixed space available in the memory. The invention is particularly useful in making more efficient use of a Content Addressable Memory (CAM) for storing flow entries, and configuring the CAM to index an associated memory that stores forwarding and quality of service information for each CAM entry.

A flow is defined by the network element as a series of related packets that may be represented by a single entry containing Layer 3 and higher layer information in the forwarding memory. The network element is configured to identify such packets as members of a flow. The network element selects certain fields in the entry to include wildcards. Wildcarding the entry may result in multiple flows matching the entry, which can then handle a greater number of received packets, thereby reducing the number of entries in the forwarding memory in return for coarser granularity in routing and quality of service control.

To ensure a deterministic matching result following a single search of the forwarding memory having wildcarded entries, the network element is configured to produce only the most specific match where multiple matches exist in the forwarding memory. In one embodiment, the network element is configured to order the entries in the forwarding memory most specific, so that the first matching entry will always be the most specific entry among multiple matching entries. Alternatively, the most specific match can be determined by dedicated hardware coupled to the forwarding memory.

Regardless of whether wildcards are used, entries are aged using dual time-out periods in another embodiment of the invention. When an entry is placed in the forwarding memory, a first timer is set to a first time. If no matching packets arrive before the first time elapses, the entry is marked for removal. If a matching packet does arrive before the first time elapses, then a second timer is set to a second time that is different than the first time. Subsequently, the entry is removed if either no matching packet arrives by the time the second time elapses, or a matching packet arrives late (after the second time has elapsed). Additional timers such as a third, fourth, . . . could be used which would be equivalent to changing the second timer dynamically. In the preferred embodiment of the invention, the second time is longer than the first time.

BRIEF DESCRIPTION OF THE DRAWINGS

The above aspects and features of the invention will be better understood by referring to the drawings, detailed description, and claims below, where:

FIG. 1 is a block diagram of a network element for implementing the invention.

FIG. 2 illustrates a forwarding memory and associated memory configured according to another embodiment of the invention.

FIG. 3 shows a flow diagram of steps to practice an embodiment of the invention.

DETAILED DESCRIPTION

Although as generally summarized above, the invention may be applied to manage any type of forwarding entry in a network element, the invention is particularly useful for managing end-to-end flows. A flow is a sequence of packets arriving at and identified by the network element as a single Layer 3 entry in the forwarding memory. The packets are

related in that they originate from the same end-node and normally have the same destination end-node based on their Layer 3 (network layer) and Layer 4 (transport layer) headers as defined by the Open System Interconnect (OSI) Reference Model. The packets in a flow normally receive the same quality of service (QOS) by the network element.

Membership in a flow can be based in part upon the transport layer (Layer 4) protocol used such as Transmission Control Protocol (TCP) or User Datagram Protocol (UDP), the network layer (Layer 3) addresses such as Internet Protocol (IP) addresses, and the application source and destination UDP or TCP ports (part of the transport layer protocol). Similar definitions for other protocols can be developed by those skilled in the art.

FIG. 1 illustrates an exemplary setting for practicing the invention in a network element **101**. The network element is configured to relay packets and contains a number of subsystems **110** that are interconnected by internal links **141**. A central processing system (CPS) having a programmed central processing unit (CPU) **135** and a central memory **130** couples with the different subsystems **110** through a conventional computer communication bus, such as a Peripheral Components Interconnect (PCI) bus. Each subsystem **110** has a hardware search engine that includes a hardwired logic switching element **111** coupled to a forwarding memory **113** and associated memory **114**. The address table used for forwarding packets through the network element is stored in the forwarding memory **113**, while the forwarding and quality of service (QOS) information associated with each entry is stored in the associated memory **114**. In a preferred embodiment, the forwarding memory is implemented by a CAM while the associated memory includes a random access memory (RAM). Alternatively, the associated data normally stored in the separate RAM can be stored in the CAM together with the address table. In yet another embodiment, the forwarding memory may be implemented by a programmed processor or hardwired logic circuit that implements a hash table.

The network element **101** forwards packets received by an external connection **117** to one or more other external connections based on the packets' headers and programmed criteria in the CPS and the individual subsystems **110**. For example, a newly arrived packet at the subsystem **110** will be processed by a hardware class filter in the switching element **111** which identifies the packet as a member of one of several predefined packet types. The packet's headers are then evaluated against the entries in the forwarding memory that are of the identified class. A matching entry will indicate one or more subsystem ports identified in the associated memory **114** as well as a certain QOS. The packet will be forwarded, based on the associated data, out of the subsystem **110** or to another subsystem **120**, as identified in the associated memory. In the event of a "miss" (no matching entry), the CPS **130** may be configured to handle forwarding aspects based on its software routines.

Throughout this disclosure, references are made to the CPU performing certain actions or causing certain other actions to be performed. These refer, in general, to actions or decisions by the programmed CPU **135**. However, one skilled in the art will recognize that other means for implementing those actions or decisions can be used, including hardwired logic circuits, in order to obtain certain advantages such as speed of execution.

The CPS **130** is configured to evaluate, based on previously programmed criteria, whether or not a flow entry **221** should be added to the forwarding memory **113** in response to a subsystem receiving a new packet having unmatching Layer 3 headers. As shown in FIG. **2**, the header data for a flow entry according to one embodiment of the invention include but are not limited to the following fields: class, IP source address, IP destination address, application source port, and application destination port. The flow entry **221** can alternatively be configured to store header data for message traffic that is handled in accordance with network layer, transport layer, and higher layer protocols other than TCP/IP. Thus, similar definitions for other protocols can be developed by those skilled in the art.

Regardless of the particular protocols used for each entry in the forwarding memory, the associated memory identifies internal and/or external ports of the inbound subsystem, queuing, priority, header replacement information for routing, age, and distribnted flow indication.

In the first embodiment of the invention, when the CPS **130** has determined that a flow entry should be created, wildcards are used so that some of the fields in the entry will contain "don't care" bits that will match all combinations. Exemplary entries **271** and **275** which contain certain wildcarded fields are shown in FIG. **2**. For example, by removing dependency upon the application source and destination ports, all packets transferred between the same source and destination IP addresses using TCP will match the entry **271** and will be forwarded based on the same associated data.

As another example, a flow can be defined as having wildcards in both the source and destination port fields as well as the IP destination, with the IP source identified as the address of a particular site on the Internet. Thereafter, all traffic from that site will match the flow entry and will normally receive the same priority and quality of service.

Using wild cards in the header data of an entry for defining flow membership will reduce the total number of entries in the address table as the flow is broadly defined, thus resulting in more frequent matches. This comes at the expense of coarser granularity for the network element **101**'s ability to control QOS. For finer granularity (more specific control over packet forwarding) more specific entries can be added to the forwarding memory in order to more specifically define priority and QOS for certain packets. However, doing so may result in multiple matching entries where wildcarding is used. For example, referring to the exemplary forwarding and associated memories in FIG. **2**, entry **271** will match all traffic from the specified IP source. If entry **275** is added by the network element to specifically control QOS for packets originating from application port **80**, then a subsequent packet originating with port **80** on the endstation assigned to the source IP address specified will yield two matching entries **271** and **275**. As the network element is preferably configured to forward packets based on deterministic criteria, multiple matching entries presents a problem that needs to be dealt with in a predictive manner.

An important aspect of the invention, therefore, lies in solving the multiple matching entry problem by ordering the entries from a least specific to a most specific fashion in the forwarding memory **113**. The ordering is normally set by requiring that the number of wildcard bits be lower in earlier entries, by placing entries that support resource-reservation protocols such as RSVP before those having default values, and by ordering entries to support administrative criteria. However, ordering may be different in order to satisfy certain administrative criteria.

The three TCP class entries shown in the forwarding memory **113** of FIG. **2** are given as an example of the ordering scheme. Entry **271** is less specific than entry **275**

NOAC Ex. 1015 Page 580

iuse the former includes wildcards in the application ce port field whereas the latter does not. In general, by ring the entries such that an entry has more wildcarded in a given field than the one above it, and by defining the match to be the one to use for forwarding the packet, a le sequential search of the forwarding memory **113** from to bottom will be sufficient to produce a single and ially most specific entry among multiple matching es. This may also be referred to as the longest matching r, or the one having the fewest wildcards.

lternatively, the multiple match situation can be handled roviding additional hardware that is configured specially etect and select the most specific match as the single ut, given a number of matching entries.

ius, as a first embodiment of the invention, the above ission of the forwarding memory **113** describes a tech- e for managing the entries in a forwarding memory so provide for faster forwarding of packets using only a e search so that the most specific is matched first, or by ting the most specific from a number of matching es. Also, an example was given for detecting and ing Layer 3 flows based on different granularity levels gh the use of wildcarding.

second embodiment of the invention is now discussed h focuses on how to remove Layer 3 flows to use the arding memory more efficiently. The embodiment is rated in FIG. **3** as an exemplary series of steps to be rmed in a network element having a forwarding ory **113** as in FIG. **1**. The steps are described while ring to the architecture of FIGS. **1** and **2**, and the flow am of FIG. **3**.

hen a new packet is received by the network element n block **301**, its header is passed to header matching (not shown) in switching element **111**. The header ing logic then initiates a matching cycle on the for- ing memory **113** to determine if the new headers match isting forwarding entry, as in decision block **305**.

decision block **305**, the test is whether the new headers n an entry in the forwarding memory **113**. If yes, then cket is forwarded to its destination outside the network nt **101** as between subsystems **110** (or within the nd subsystem **110**) using forwarding and quality of e information in the associated memory. The forward- emory **113** will generate a pointer to an associated on in the associated memory that contains associated or the matching entry. The associated data will include ternal or internal subsystem port, any routing informa- if the matching entry is of a hardware route type as ted in the entry's class field), and priority information euing the packet in an external port. Also, whenever a t occurs with an existing entry in the forwarding ry **113**, the activity field or aged destination bits are ed to reflect the match, as in block **309**.

urning to decision block **305**, if the received packet nknown headers, i.e., not matching any entries in the rding memory **113**, then the packet is "flooded" on all al and external ports of the inbound subsystem. In that block **311** calls for a decision as to whether or not add entry to the forwarding memory. The decision block s preferably performed by CPS **130** in response to ing relevant portions of the headers of the new packet the inbound subsystem. For packets that fall within the class, such that a Layer 3 flow entry may need to be d, the header portions sent to the CPS include the IP and destination addresses, and the application source estination ports (Layers 3 and 4). If the class filter in the subsystem **110** indicates that a hardware route is supported by the network element **101** for the received packet, then the Layer 2 source and destination addresses are also sent to the CPS. The CPS **130** then determines, based on previously programmed information in the CPS about the header fields, whether or not an entry should be added to the forwarding memory.

If the CPS **130** determines that an entry should be added to the forwarding memory as in block **311**, then the CPS **130** causes the entry to be created in the forwarding memory **113** as in block **315**. The CPS **130** can place don't care bits in any field of the entry for the desired level of granularity. The associated data of the new entry will also be provided by the CPS **130** and will include at least an output port (external and/or internal) in the subsystem **110**, queuing and priority information to be used by the output ports of the outbound subsystem, and any header replacement information required for routing. The activity or aged destination field of the new entry will also indicate that the entry is new, i.e., no match as of yet.

Once the CPS **130** has determined that a new entry should be created as in block **311**, the CPS **130** sets a timer or counter to a first time in block **317**. The timer corresponds to the new entry and is normally a software timer within the CPS **130**. The CPS **130** will include provisions such as pointers for the timer that identify its corresponding new entry as the one in forwarding memory **113** of subsystem **110**.

When the timer runs out in block **321**, the CPS **130** is notified in block **323**, normally by an interrupt to CPU **135** or, alternatively, by polling. Based on its pointer provisions, the CPS **130** identifies the corresponding entry and determines whether the entry corresponding to the timer is still active by reading the value of its activity or aged destination field in the associated memory **114**, as in decision block **325**. If the activity field indicates that a received packet has not matched the entry since the timer was set to a first time, then the CPS **130** instructs the removal of the entry in block **327**. If, however, the activity field indicated that the subsystem received a packet having headers that match the entry during the first time interval, then a second timer corresponding to the same entry is set to a second time in block **335**. Also, the activity field is cleared to indicate no matching packets as of yet.

The above procedure beginning with block **321** repeats thereafter for each subsequent time-out of the second timer. Although the first and second timers are defined as separate timers, an alternative embodiment casts the first and second timer as the same software timer having two different time intervals.

An important aspect of the invention lies in the second time being preferably longer than the first time, where preferred numbers include approximately one second for the first time and three seconds for the second time. Furthermore, the first and second times can be chosen based on network layer addresses, Layer 4 protocols, and Layer 4 port numbers.

Exceptions to the above removal of entries based on time-out periods include the situation of RSVP signaling, where a flow entry is managed explicitly by RSVP. In that case, RSVP signaling between the source and destination will determine when the flow entry should be removed from the forwarding memory **123**.

Other exceptions are packets destined to UDP port **111** (Layer 4 information). It has been observed that such packets appear to contribute greatly to packet flow entries

which result in only a single matching packet before the first time expires. The network element **101** will thus be preferably configured to not make a Layer 3 flow entry for such packets.

The use of dual time periods described above thus presents a simple yet effective technique for keeping the number of entries in a forwarding memory **113** to a minimum. The technique's simplicity in verifying the age status of an entry in response to a time-out of two different time periods expedites the entire forwarding process. However, additional timers and time periods can be used to further fine tune the invention's entry removal mechanism, which may be viewed as dynamically changing the second time upon each time-out of the second timer.

The rationale behind the shorter first and longer second intervals can perhaps be explained by analyzing the probability of packets arriving at the network element **101** which match a given entry. An assumption is made that the intervals between the arrival of such packets are independent of each other. For a given stream of arriving packets, the probability of an interpacket gap exceeding a first time t_1 is P_1, and the probability of an interpacket gap exceeding a second time t_2 is P_2. Since the assumption is that $t_1<t_2$, then $P_2<P_1$ must be true in most circumstances.

If the sole basis for removing entries from the forwarding memory is to use t_1 as a time-out period, such that if an interpacket gap exceeds t_1 then the flow entry is removed, then the probability of not prematurely removing a flow for the first n packets is $(1-P_1)^n=(1-P_1)(1-P_1)^{n-1}$.

If, however, two timeout intervals are used, t_1 for the first interval, and t_2 for subsequent intervals between arriving packets, then the probability of not prematurely removing the entry for the first n packets is $(1-P_1)(1-P_2)^{n-1}$. Since $P_2<P_1<1$, so is $(1-P_1)^{n-1}<(1-P_2)^{n-1}$, and thus the use of two interval times increases the probability of not prematurely removing a multi-packet flow, yet still effectively remove entries which only receive a single matching packet.

To summarize, the invention presents a technique in a network element for managing the entries in the forwarding memory. After identifying a first received packet as a member of a flow, an entry is created in the forwarding memory which identifies the flow. Thereafter, the entry is removed if a subsequent packet having a matching header is not received before a first time has elapsed. If, however, the subsequent packet having a matching header is received before the first time has elapsed, then the flow entry is kept and a timer is reset to a second longer time, such that the flow entry will be kept for subsequent packets arriving within the longer second time. Such a technique takes advantage of the pattern that if two packets arrive within the shorter first time, then the likelihood of more matching packets arriving within a subsequent longer time has substantially increased.

In another embodiment, yet more efficient use of a forwarding memory is made by implementing a longest-match approach, where the forwarding contains wildcarded entries. The wildcarding scheme allows for default routes to be defined based on any level of granularity desired for source and destination addresses, subnets, as well as transport layer ports.

The embodiments of the invention described above for exemplary purposes are, of course, subject to other variations in structure and implementation within the capabilities of one reasonably skilled in the art. For example, although many operations have been described as occurring sequentially, some of these can be implemented concurrently or even in a different order so long as the dependencies between them permit such deviation. Also, the invention can be implemented in a network element having a centralized forwarding memory rather than one having the distributed architecture of FIG. **1**. Thus, the details above should be interpreted as illustrative and not in a limiting sense.

What is claimed is:

1. In a network element having a forwarding memory for storing a plurality of entries used for forwarding received packets, a method of managing the storage of the entries comprising the steps of:

making a first entry in the forwarding memory based upon a header of a first received packet; and one of

marking the first entry for removal if a second packet having a header matching the first entry is not received before a first time has elapsed, and

waiting a second time not equal to the first time before marking the first entry for removal if the second packet having the header matching the first entry is received before the first time has elapsed.

2. The method as in claim **1** wherein the second time is longer than the first time.

3. The method as in claim **2** where the first time is approximately one second and the second time is approximately three seconds.

4. The method as in claim **1** further comprising the step of

marking the first entry for removal if a third packet having a third header matching the first entry is not received before the second time has elapsed.

5. The method as in claim **1** further comprising the step of waiting a third time before marking the first entry for removal if a third packet having a third header matching the first entry is received before the second time has elapsed.

6. The method as in claim **1** prior to the step of making the first entry, further comprising the step of

searching the forwarding memory for a match between the header of the first received packet and an entry in the forwarding memory,

and wherein the step of making the first entry is performed in response to not finding the match.

7. The method as in claim **5** wherein the third time is different than the second time.

8. The method as in claim **6** further comprising the step of selecting a most specific one of a number of matching entries.

9. The method as in claim **1** further comprising the step of

ordering the plurality of entries in the forwarding memory from most specific to least specific, the most specific having the fewest wildcard characters of any of the plurality of entries.

10. The method as in claim **1** wherein the header further comprises a Layer 4 address.

11. The method as in claim **10** wherein the header further comprises a Layer 4 address.

12. The method as in claim **1** wherein at least one of the first time and the second time is selected based on predefined network layer addresses.

13. In a network element having an address table for storing a plurality of entries used for forwarding received packets, and a timer associated with each entry, a method of managing the entries in the address table comprising the steps of:

receiving a first packet;

making a first entry in the address table based upon a header of the first packet;

setting a first timer to a first time; and

setting a second timer to a second time not equal to the first time if a second packet matching the first entry is received before the first time has elapsed.

14. The method as in claim **13** wherein the second time is longer than the first time.

15. The method as in claim **14** wherein the first time is approximately one second and the second time is approximately three seconds.

16. The method as in claim **13** further comprising the step of

deleting the first entry if the second packet matching the first entry is not received before the first time has elapsed.

17. The method as in claim **13** wherein the first timer is the second timer.

18. The method as in claim **13**

wherein the first entry comprises an activity field for indicating that the first entry has matched a received second packet.

19. The method as in claim **13** prior to setting the second timer further comprising the steps of:

searching the address table to find a matching entry in response to receiving the second packet, the first entry being a most specific one of a plurality of matching entries.

20. The method as in claim **13** further comprising the steps of:

ordering the plurality of entries in the forwarding memory from most specific to least specific, the most specific having the fewest wildcard characters of any of the plurality of entries; and

searching the address table to find a matching entry.

21. The method as in claim **20** wherein the step of searching the address table to find the matching entry includes searching based on Layer 3 and Layer 4 headers of a received packet.

22. A network element for managing a plurality of entries in a forwarding memory, comprising:

means for initiating a search of the forwarding memory for an entry matching a portion of a first header of a received first packet;

means for making a first entry in the forwarding memory in response to finding no entry matching the portion of the first header of the received first packet, the first entry having an activity field indicating whether the first entry has matched a received second packet;

means for setting a first timer to a first time;

means for checking the activity field in response to the first timer running out; and

means for setting a second timer to a second time not equal to the first time in response to the activity field indicating that the first entry has matched the received second packet.

23. The network element as in claim **22** wherein the second time is longer than the first time.

24. The network element as in claim **23** wherein the first time is approximately one second and the second time is approximately three seconds.

25. The network element as in claim **22** further comprising

means for invalidating the first entry in response to the activity field indicating that the first entry has not matched the received second packet.

26. The network element as in claim **22** further comprising means for setting a third timer to a third time in response to the activity field indicating that the first entry has matched a received third packet before expiration of the second time.

27. The network element as in claim **22** wherein the first header is a Layer 3 header.

28. The network element as in claim **26** wherein the third time is approximately equal to the second time.

29. The network element as in claim **22** wherein the first entry comprises Layer 4 information.

30. The network element as in claim **22** wherein the second timer is the first timer.

31. The network element as in claim **22** further comprising

memory configured for storing the forwarding memory and the plurality of entries; and

means for ordering the plurality of entries in the memory from most specific to least specific.

32. The network element as in claim **31** wherein the memory is a content addressable memory.

33. The network element as in claim **22** wherein at least one of the means for making the first entry, means for setting the first timer, means for checking the activity field, and means for setting the second timer is a programmed processor.

34. The network element as in claim **22** wherein the first timer is a software timer.

35. The network element as in claim **22** wherein the means for initiating the search is a header matching logic circuit.

36. The network element as in claim **22** further comprising means for updating the activity field in response to the first entry matching the received second packet prior to the first timer running out.

* * * * *

7/8/03

M/A
PLWB
7-15-03

Our Ref./Docket No: APPT-001-1 Patent

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant(s): Dietz, *et al.* Application No.: 09/608237 Filed: June 30, 2000 Title: METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK	Group Art Unit: 2157 Examiner: Moustafa M. Meky

RESPONSE TO OFFICE ACTION UNDER 37 CFR 1.111

Mail Stop No Fee Amendment
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Dear Commissioner:

This is a response to the Office Action of June 25, 2003.

Certificate of Facsimile Transmission under 37 CFR 1.8

I hereby certify that this response is being facsimile transmitted to the United States Patent and Trademark Office at telephone number ~~703-308-9052~~ 703-746-7239 addressed the Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450 on.

Date: 7/8/03

Signed:

Name: Dov Rosenfeld, Reg. No. 38687

A

INTRODUCTORY REMARKS:

In response to the Office Action of June 25, 2003, kindly amend this application as follows and kindly consider the following remarks.

AMENDMENT(S) TO THE ABSTRACT:

7/7/03

Kindly amend the abstract as follows:

ABSTRACT

A monitor for and a method of examining packets passing through a connection point on a computer network. Each packets conforms to one or more protocols. The method includes receiving a packet from a packet acquisition device and performing one or more parsing/extraction operations on the packet to create a parser record comprising a function of selected portions of the packet. The parsing/extraction operations depend on one or more of the protocols to which the packet conforms. The method further includes looking up a flow-entry database containing flow-entries for previously encountered conversational flows. The lookup uses the selected packet portions and determining if the packet is of an existing flow. If the packet is of an existing flow, the method classifies the packet as belonging to the found existing flow, and if the packet is of a new flow, the method stores a new flow-entry for the new flow in the flow-entry database, including identifying information for future packets to be identified with the new flow-entry. For the packet of an existing flow, the method updates the flow-entry of the existing flow. Such updating may include storing one or more statistical measures. Any stage of a flow, state is maintained, and the method performs any state processing for an identified state to further the process of identifying the flow. The method thus examines each and every packet passing through the connection point in real time until the application program associated with the conversational flow is determined. ~~The method~~

AMENDMENT(S) TO THE CLAIMS:

7/8/03

The following listing of claims will replace all prior versions, and listings, of claims on the application. Claims being amended are set forth in a larger font than all other claims. All claims are set forth below with one of the following annotations.

- (Original): Claim filed with the application following the specification.
- (Currently amended): Claim being amended in the current amendment paper.
- (Previously amended): Claim not being currently amended, but which was amended in a previous amendment paper.
- (Cancelled): Claim cancelled or deleted from the application.
- (Withdrawn): Claim still in the application, but in a non-elected status.
- (Previously added): Claim added in an earlier amendment paper.
- (New): Claim being added in the current amendment paper.
- (Reinstated - formerly claim # _): Claim deleted in an earlier amendment paper, but re-presented with a new claim number in current amendment.
- (Previously reinstated): Claim deleted in an earlier amendment and reinstated in an earlier amendment paper.
- (Re-presented - formerly dependent claim # _): Dependent claim re-presented in independent form in current amendment paper.
- (Previously re-presented): Dependent claim re-presented in independent form in an earlier amendment, but not currently amended.

CLAIMS

What is claimed is:

1. (Original) A packet monitor for examining packets passing through a connection point on a computer network in real-time, the packets provided to the packet monitor via a packet acquisition device connected to the connection point, the packet monitor comprising:

   (a) a packet-buffer memory configured to accept a packet from the packet acquisition device;

   (b) a parsing/extraction operations memory configured to store a database of parsing/extraction operations that includes information describing how to determine at least one of the protocols used in a packet from data in the packet;

(c) a parser subsystem coupled to the packet buffer and to the pattern/extraction operations memory, the parser subsystem configured to examine the packet accepted by the buffer, extract selected portions of the accepted packet, and form a function of the selected portions sufficient to identify that the accepted packet is part of a conversational flow-sequence;

(d) a memory storing a flow-entry database including a plurality of flow-entries for conversational flows encountered by the monitor;

(e) a lookup engine connected to the parser subsystem and to the flow-entry database, and configured to determine using at least some of the selected portions of the accepted packet if there is an entry in the flow-entry database for the conversational flow sequence of the accepted packet;

(f) a state patterns/operations memory configured to store a set of predefined state transition patterns and state operations such that traversing a particular transition pattern as a result of a particular conversational flow-sequence of packets indicates that the particular conversational flow-sequence is associated with the operation of a particular application program, visiting each state in a traversal including carrying out none or more predefined state operations;

(g) a protocol/state identification mechanism coupled to the state patterns/operations memory and to the lookup engine, the protocol/state identification engine configured to determine the protocol and state of the conversational flow of the packet; and

(h) a state processor coupled to the flow-entry database, the protocol/state identification engine, and to the state patterns/operations memory, the state processor, configured to carry out any state operations specified in the state patterns/operations memory for the protocol and state of the flow of the packet,

the carrying out of the state operations furthering the process of identifying which application program is associated with the conversational flow-sequence of the packet, the state processor progressing through a series of states and state operations until there are no more state operations to perform for the accepted packet, in which case the state processor updates the flow-entry, or until a final state is reached that indicates that no more analysis of the flow is required, in which case the result of the analysis is announced.

2. (Original) A packet monitor according to claim 1, wherein the flow-entry includes the state of the flow, such that the protocol/state identification mechanism determines the state of the packet from the flow-entry in the case that the lookup engine finds a flow-entry for the flow of the accepted packet.

3. (Original) A packet monitor according to claim 1, wherein the parser subsystem includes a mechanism for building a hash from the selected portions, and wherein the hash is used by the lookup engine to search the flow-entry database, the hash designed to spread the flow-entries across the flow-entry database.

4. (Original) A packet monitor according to claim 1, further comprising:

a compiler processor coupled to the parsing/extraction operations memory, the compiler processor configured to run a compilation process that includes:

receiving commands in a high-level protocol description language that describe the protocols that may be used in packets encountered by the monitor, and

translating the protocol description language commands into a plurality of parsing/extraction operations that are initialized into the parsing/extraction operations memory.

5. (Original) A packet monitor according to claim 4, wherein the protocol description language commands also describe a correspondence between a set of one or more application programs and the state transition patterns/operations that occur as a result of particular conversational flow-sequences associated with an application program, wherein the compiler processor is also coupled to the state patterns/operations memory, and wherein the compilation process further includes translating the protocol description language commands into a plurality of state patterns and state operations that are initialized into the state patterns/operations memory.

6. (Original) A packet monitor according to claim 1, further comprising:

a cache memory coupled to and between the lookup engine and the flow-entry database providing for fast access of a set of likely-to-be-accessed flow-entries from the flow-entry database.

7. (Original) A packet monitor according to claim 6, wherein the cache functions as a fully associative, least-recently-used cache memory.

8. (Original) A packet monitor according to claim 7, wherein the cache functions as a fully associative, least-recently-used cache memory and includes content addressable memories configured as a stack.

9. (Original) A packet monitor according to claim 1, wherein one or more statistical measures about a flow are stored in each flow-entry, the packet monitor further comprising:

a calculator for updating the statistical measures in a flow-entry of the accepted packet.

10. (Original) A packet monitor according to claim 9, wherein, when the application program of a flow is determined, one or more network usage metrics related to said application and determined from the statistical measures are presented to a user for network performance monitoring.

11.–59. (Cancelled)

REMARKS

7/8/03

Status of the Application:

Claims 1–59 are the claims of record of the application. Claims 1-10 have been allowed. Claims 11–59 have been rejected.

Amendment to the Abstract:

Applicants have amended the abstract to remove a spurious phase at the end, this being a typographical error.

Amendment to the Claims:

Applicants have cancelled the rejected claims.

Conclusion

The only remaining claims are those allowed by the Examiner. A notice of Allowance for the remaining claims is respectfully requested.

If the Examiner has any questions or comments that would advance the prosecution and allowance of this application, an email message to the undersigned at dov@inventek.com, or a telephone call to the undersigned at +1-510-547-3378 is requested.

Respectfully Submitted,

7/8/03
Date

Dov Rosenfeld, Reg. No. 38687

Address for correspondence:
Dov Rosenfeld
5507 College Avenue, Suite 2
Oakland, CA 94618
Tel. +1-510-547-3378
Fax: +1-510-291-2985
Email:dov@inventek.com

INVENTEK

Dov Rosenfeld
5507 College Avenue, Suite 2
Oakland, CA 94618, USA
Phone: (510)547-3378; Fax: (510)653-7992
dov@inventek.com

Fax

Patent Application Ser. No.: 09/608237	***Ref./Docket No***: APPT-001-1
Applicant(s): Dietz, *et al.*	***Examiner***.: Moustafa M. Meky
Filing Date: June 30, 2000	***Art Unit***: 2157

FAX COVER PAGE

TO: Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

United States Patent and Trademark Office
(Examiner Moustafa M. Meky, Art Unit 2157)

Fax No.: 703-~~308-9052~~ 746-7239

DATE: July 08, 2003

FROM: Dov Rosenfeld, Reg. No. 38687

RE: Response to Office Action

Number of pages including cover: 13

OFFICIAL COMMUNICATION

PLEASE URGENTLY DELIVER A COPY OF THIS RESPONSE TO EXAMINER MOUSTAFA M. MEKY, ART UNIT 2157

Certificate of Facsimile Transmission under 37 CFR 1.8

I hereby certify that this response is being facsimile transmitted to the United States Patent and Trademark Office at telephone number 703-~~308-9052~~ 746-7239 addressed the Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450

Date: 7/8/03

Signed: _____

Name: Dov Rosenfeld, Reg. No. 38687

TRANSMITTAL FORM

(to be used for all correspondence after initial filing)

Application Number	09/608237 7/8/03
Filing Date	30 Jun 2000
First Named Inventor	Dietz, Russell S.
Group Art Unit	2157
Examiner Name	Moustafa M. Meky
Attorney Docket Number	APPT-001-1

CLOSURES *(check all that apply)*

Fee Transmittal Form	☐ Assignment Papers *(for an Application)*	☐ After Allowance Communication to Group
Fee Attached	☐ Drawing(s)	☐ Appeal Communication to Board of Appeals and Interferences
Amendment / Response	☐ Licensing-related Papers	☐ Appeal Communication to Group *(Appeal Notice, Brief, Reply Brief)*
☐ After Final	☐ Petition Routing Slip (PTO/SB/69) and Accompanying Petition	☐ Proprietary Information
☐ Affidavits/declaration(s)	☐ To Convert a Provisional Application	☐ Status Letter
Extension of Time Request	☐ Power of Attorney, Revocation Change of Correspondence Address	☐ Additional Enclosure(s) *(please identify below):*
Express Abandonment Request	☐ Terminal Disclaimer	☐ Return Postcard
Information Disclosure Statement	☐ Small Entity Statement	☐
Certified Copy of Priority Document(s)	☐ Request of Refund	☐
Response to Missing Parts/ Incomplete Application	Remarks	
☐ Response to Missing Parts under 37 CFR 1.52 or 1.53		

NATURE OF APPLICANT, ATTORNEY, OR AGENT/ CORRESPONDENCE ADDRESS

or idual name	Dov Rosenfeld, Reg. No. 38687
ature	
	July 8, 2003

RESS FOR CORRESPONDENCE

dual name	Dov Rosenfeld 5507 College Avenue, Suite 2 Oakland, CA 94618, Tel: +1-510-547-3378

IFICATE OF FACSIMILE TRANSMISSION

by certify that this correspondence is being facsimile transmitted with the United States Patent and Trademark Office at hone number 703-746-7239 addressed to: Commissioner for Patents, P.O. Box 1450, Alexandria, VA -1450 on this date: July 8, 2003

r printed name	Dov Rosenfeld, Reg. No. 38687		
ure		Date	July 8, 2003

7/8/03

Our Ref./Docket No: APPT-001-1 Patent

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant(s): Dietz, *et al.* Application No.: 09/608237 Filed: June 30, 2000 Title: METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK	Group Art Unit: 2157 Examiner: Moustafa M. Meky

TRANSMITTAL: RESPONSE TO OFFICE ACTION

Mail Stop Non Fee Amendment
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Dear Commissioner:

Transmitted herewith is a response to an office action for the above referenced application. Included with the response are:

____ formal drawings (with separate letter);

This application has:

____ a small entity status. If a claim for such status has not earlier been made, consider this as a claim for small entity status.

__X__ No additional fee is required.

Certificate of Facsimile Transmission under 37 CFR 1.8

I hereby certify that this response is being facsimile transmitted to the United States Patent and Trademark Office at telephone number ~~703-308-9052~~ 703-746-7239 addressed the Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450 on.

Date: 7/8/03 Signed: _____

Name: Dov Rosenfeld, Reg. No. 38687

Received from < +1 510 291 2985 > at 7/8/03 7:43:30 PM [Eastern Daylight Time]

UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
09/608,237	06/30/2000	Russell S. Dietz	APPT-001-1	9993

7590 07/17/2003

Dov Rosenfeld
Suite 2
5507 College Avenue
Oakland, CA 94618

EXAMINER	
MEKY, MOUSTAFA M	
ART UNIT	**PAPER NUMBER**
2157	8

DATE MAILED: 07/17/2003

Determination of Patent Term Adjustment under 35 U.S.C. 154 (b)
(application filed on or after May 29, 2000)

The patent term adjustment to date is 589 days. If the issue fee is paid on the date that is three months after the mailing date of this notice and the patent issues on the Tuesday before the date that is 28 weeks (six and a half months) after the mailing date of this notice, the term adjustment will be 589 days.

If a continued prosecution application (CPA) was filed in the above-identified application, the filing date that determines patent term adjustment is the filing date of the most recent CPA.

Applicant will be able to obtain more detailed information by accessing the Patent Application Information Retrieval (PAIR) system. (http://pair.uspto.gov)

Any questions regarding the patent term extension or adjustment determination should be directed to the Office of Patent Legal Administration at (703)305-1383.

PART B - FEE(S) TRANSMITTAL

Complete and send this form, together with applicable fee(s), to: Mail Mail Stop ISSUE FEE
Commissioner for Patents
Alexandria, Virginia 22313-1450
Fax (703)746-4000

INSTRUCTIONS: This form should be used for transmitting the ISSUE FEE and PUBLICATION FEE (if required). Blocks 1 through 4 should be completed where appropriate. All further correspondence including the Patent, advance orders and notification of maintenance fees will be mailed to the current correspondence address as indicated unless corrected below or directed otherwise in Block 1, by (a) specifying a new correspondence address; and/or (b) indicating a separate "FEE ADDRESS" for maintenance fee notifications.

CURRENT CORRESPONDENCE ADDRESS (Note: Legibly mark-up with any corrections or use Block 1)

7590 07/17/2003

Dov Rosenfeld
Suite 2
5507 College Avenue
Oakland, CA 94618

Note: A certificate of mailing can only be used for domestic mailings of the Fee(s) Transmittal. This certificate cannot be used for any other accompanying papers. Each additional paper, such as an assignment or formal drawing, must have its own certificate of mailing or transmission.

Certificate of Mailing or Transmission

I hereby certify that this Fee(s) Transmittal is being deposited with the United States Postal Service with sufficient postage for first class mail in an envelope addressed to the Box Issue Fee address above, or being facsimile transmitted to the USPTO, on the date indicated below.

(Depositor's name)
(Signature)
(Date)

APPLICATION NO	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
09/608,237	06/30/2000	Russell S. Dietz	APPT-001-1	9993

TITLE OF INVENTION: METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK

APPLN TYPE	SMALL ENTITY	ISSUE FEE	PUBLICATION FEE	TOTAL FEE(S) DUE	DATE DUE
nonprovisional	NO	$1300	$0	$1300	10/17/2003

EXAMINER	ART UNIT	CLASS-SUBCLASS
MEKY, MOUSTAFA M	2157	709-224000

1. Change of correspondence address or indication of "Fee Address" (37 CFR 1.363).

❑ Change of correspondence address (or Change of Correspondence Address form PTO/SB/122) attached.

❑ "Fee Address" indication (or "Fee Address" Indication form PTO/SB/47; Rev 03-02 or more recent) attached. **Use of a Customer Number is required.**

2. For printing on the patent front page, list (1) the names of up to 3 registered patent attorneys or agents OR, alternatively, (2) the name of a single firm (having as a member a registered attorney or agent) and the names of up to 2 registered patent attorneys or agents. If no name is listed, no name will be printed.

1_____
2_____
3_____

3. ASSIGNEE NAME AND RESIDENCE DATA TO BE PRINTED ON THE PATENT (print or type)

PLEASE NOTE: Unless an assignee is identified below, no assignee data will appear on the patent. Inclusion of assignee data is only appropriate when an assignment has been previously submitted to the USPTO or is being submitted under separate cover. Completion of this form is NOT a substitute for filing an assignment.

(A) NAME OF ASSIGNEE (B) RESIDENCE: (CITY and STATE OR COUNTRY)

Please check the appropriate assignee category or categories (will not be printed on the patent) ❑ individual ❑ corporation or other private group entity ❑ government

4a. The following fee(s) are enclosed:

❑ Issue Fee
❑ Publication Fee
❑ Advance Order - # of Copies _____

4b. Payment of Fee(s):

❑ A check in the amount of the fee(s) is enclosed.
❑ Payment by credit card. Form PTO-2038 is attached.
❑ The Commissioner is hereby authorized by charge the required fee(s), or credit any overpayment, to Deposit Account Number _____(enclose an extra copy of this form).

Commissioner for Patents is requested to apply the Issue Fee and Publication Fee (if any) or to re-apply any previously paid issue fee to the application identified above.

(Authorized Signature) (Date)

NOTE: The Issue Fee and Publication Fee (if required) will not be accepted from anyone other than the applicant; a registered attorney or agent; or the assignee or other party in interest as shown by the records of the United States Patent and Trademark Office.

This collection of information is required by 37 CFR 1.311. The information is required to obtain or retain a benefit by the public which is to file (and by the USPTO to process) an application. Confidentiality is governed by 35 U.S.C. 122 and 37 CFR 1.14. This collection is estimated to take 12 minutes to complete, including gathering, preparing, and submitting the completed application form to the USPTO. Time will vary depending upon the individual case. Any comments on the amount of time you require to complete this form and/or suggestions for reducing this burden, should be sent to the Chief Information Officer, U.S. Patent and Trademark Office, U.S. Department of Commerce, Alexandria, Virginia 22313-1450. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. SEND TO: Commissioner for Patents, Alexandria, Virginia 22313-1450.

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

NOAC Ex. 1015 Page 595

TRANSMIT THIS FORM WITH FEE(S)

UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

NOTICE OF ALLOWANCE AND FEE(S) DUE

7590 07/17/2003

Dov Rosenfeld
Suite 2
5507 College Avenue
Oakland, CA 94618

EXAMINER
MEKY, MOUSTAFA M

ART UNIT	CLASS-SUBCLASS
2157	709-224000

DATE MAILED: 07/17/2003

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
09/608,237	06/30/2000	Russell S. Dietz	APPT-001-1	9993

TITLE OF INVENTION: METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK

APPLN. TYPE	SMALL ENTITY	ISSUE FEE	PUBLICATION FEE	TOTAL FEE(S) DUE	DATE DUE
nonprovisional	NO	$1300	$0	$1300	10/17/2003

THE APPLICATION IDENTIFIED ABOVE HAS BEEN EXAMINED AND IS ALLOWED FOR ISSUANCE AS A PATENT. PROSECUTION ON THE MERITS IS CLOSED. THIS NOTICE OF ALLOWANCE IS NOT A GRANT OF PATENT RIGHTS. THIS APPLICATION IS SUBJECT TO WITHDRAWAL FROM ISSUE AT THE INITIATIVE OF THE OFFICE OR UPON PETITION BY THE APPLICANT. SEE 37 CFR 1.313 AND MPEP 1308.

THE ISSUE FEE AND PUBLICATION FEE (IF REQUIRED) MUST BE PAID WITHIN THREE MONTHS FROM THE MAILING DATE OF THIS NOTICE OR THIS APPLICATION SHALL BE REGARDED AS ABANDONED. THIS STATUTORY PERIOD CANNOT BE EXTENDED. SEE 35 U.S.C. 151. THE ISSUE FEE DUE INDICATED ABOVE REFLECTS A CREDIT FOR ANY PREVIOUSLY PAID ISSUE FEE APPLIED IN THIS APPLICATION. THE PTOL-85B (OR AN EQUIVALENT) MUST BE RETURNED WITHIN THIS PERIOD EVEN IF NO FEE IS DUE OR THE APPLICATION WILL BE REGARDED AS ABANDONED.

HOW TO REPLY TO THIS NOTICE:

I. Review the SMALL ENTITY status shown above.

If the SMALL ENTITY is shown as YES, verify your current SMALL ENTITY status:
A. If the status is the same, pay the TOTAL FEE(S) DUE shown above.

B. If the status is changed, pay the PUBLICATION FEE (if required) and twice the amount of the ISSUE FEE shown above and notify the United States Patent and Trademark Office of the change in status, or

If the SMALL ENTITY is shown as NO:

A. Pay TOTAL FEE(S) DUE shown above, or

B. If applicant claimed SMALL ENTITY status before, or is now claiming SMALL ENTITY status, check the box below and enclose the PUBLICATION FEE and 1/2 the ISSUE FEE shown above.

☐ Applicant claims SMALL ENTITY status. See 37 CFR 1.27.

II. PART B - FEE(S) TRANSMITTAL should be completed and returned to the United States Patent and Trademark Office (USPTO) with your ISSUE FEE and PUBLICATION FEE (if required). Even if the fee(s) have already been paid, Part B - Fee(s) Transmittal should be completed and returned. If you are charging the fee(s) to your deposit account, section "4b" of Part B - Fee(s) Transmittal should be completed and an extra copy of the form should be submitted.

III. All communications regarding this application must give the application number. Please direct all communications prior to issuance to Box ISSUE FEE unless advised to the contrary.

IMPORTANT REMINDER: Utility patents issuing on applications filed on or after Dec. 12, 1980 may require payment of maintenance fees. It is patentee's responsibility to ensure timely payment of maintenance fees when due.

PTOL-85 (REV. 05-03) Approved for use through 04/30/2004.

Notice of Allowability	Application No.	Applicant(s)	
	09/608,237	DIETZ ET AL.	
	Examiner	Art Unit	
	Moustafa M Meky	2157	

-- *The MAILING DATE of this communication appears on the cover sheet with the correspondence address*--

All claims being allowable, PROSECUTION ON THE MERITS IS (OR REMAINS) CLOSED in this application. If not included herewith (or previously mailed), a Notice of Allowance (PTOL-85) or other appropriate communication will be mailed in due course. **THIS NOTICE OF ALLOWABILITY IS NOT A GRANT OF PATENT RIGHTS.** This application is subject to withdrawal from issue at the initiative of the Office or upon petition by the applicant. See 37 CFR 1.313 and MPEP 1308.

1. ☒ This communication is responsive to *the amendment filed 7/8/2003*.
2. ☒ The allowed claim(s) is/are *1-10*.
3. ☒ The drawings filed on *30 June 2000* are accepted by the Examiner.
4. ☐ Acknowledgment is made of a claim for foreign priority under 35 U.S.C. § 119(a)-(d) or (f).
   a) ☐ All b) ☐ Some* c) ☐ None of the:
      1. ☐ Certified copies of the priority documents have been received.
      2. ☐ Certified copies of the priority documents have been received in Application No. _____ .
      3. ☐ Copies of the certified copies of the priority documents have been received in this national stage application from the International Bureau (PCT Rule 17.2(a)).

   * Certified copies not received: _____.
5. ☒ Acknowledgment is made of a claim for domestic priority under 35 U.S.C. § 119(e) (to a provisional application).
   (a) ☐ The translation of the foreign language provisional application has been received.
6. ☐ Acknowledgment is made of a claim for domestic priority under 35 U.S.C. §§ 120 and/or 121.

Applicant has THREE MONTHS FROM THE "MAILING DATE" of this communication to file a reply complying with the requirements noted below. Failure to timely comply will result in ABANDONMENT of this application. **THIS THREE-MONTH PERIOD IS NOT EXTENDABLE**

7. ☐ A SUBSTITUTE OATH OR DECLARATION must be submitted. Note the attached EXAMINER'S AMENDMENT or NOTICE OF INFORMAL PATENT APPLICATION (PTO-152) which gives reason(s) why the oath or declaration is deficient.

8. ☐ CORRECTED DRAWINGS must be submitted.
   (a) ☐ including changes required by the Notice of Draftsperson's Patent Drawing Review (PTO-948) attached
      1) ☐ hereto or 2) ☐ to Paper No. ____.
   (b) ☐ including changes required by the proposed drawing correction filed _____, which has been approved by the Examiner.
   (c) ☐ including changes required by the attached Examiner's Amendment / Comment or in the Office action of Paper No. _____.

Identifying indicia such as the application number (see 37 CFR 1.84(c)) should be written on the drawings in the front (not the back) of each sheet.

9. ☐ DEPOSIT OF and/or INFORMATION about the deposit of BIOLOGICAL MATERIAL must be submitted. Note the attached Examiner's comment regarding REQUIREMENT FOR THE DEPOSIT OF BIOLOGICAL MATERIAL.

Attachment(s)

1☐ Notice of References Cited (PTO-892)
2☐ Notice of Informal Patent Application (PTO-152)
3☐ Notice of Draftperson's Patent Drawing Review (PTO-948)
4☐ Interview Summary (PTO-413), Paper No.____ .
5☐ Information Disclosure Statements (PTO-1449), Paper No. ____.
6☐ Examiner's Amendment/Comment
7☐ Examiner's Comment Regarding Requirement for Deposit of Biological Material
8☐ Examiner's Statement of Reasons for Allowance
9☐ Other .

Moustafa M. Meky

MOUSTAFA M. MEKY
PRIMARY EXAMINER

_X_ Applicant(s) believe(s) that no Extension of Time is required. However, this conditional petition is being made to provide for the possibility that applicant has inadvertently overlooked the need for a petition for an extension of time.

____ Applicant(s) hereby petition(s) for an Extension of Time under 37 CFR 1.136(a) of:

____ one months ($110) ____ two months ($410)

____ two months ($930) ____ four months ($1450)

If an additional extension of time is required, please consider this as a petition therefor.

____ A credit card payment form for the required fee(s) is attached.

_X_ The Commissioner is hereby authorized to charge payment of the following fees associated with this communication or credit any overpayment to Deposit Account No. 50-0292 (A DUPLICATE OF THIS TRANSMITTAL IS ATTACHED):

_X_ Any missing filing fees required under 37 CFR 1.16 for presentation of additional claims.

_X_ Any missing extension or petition fees required under 37 CFR 1.17.

Respectfully Submitted,

7/8/03
Date

Dov Rosenfeld, Reg. No. 38687

Address for correspondence:
Dov Rosenfeld
5507 College Avenue,Suite 2
Oakland, CA 94618
Tel. +1-510-547-3378; Fax: +1-510-291-2985

Received from < +1 510 291 2985 > at 7/8/03 7:43:30 PM [Eastern Daylight Time]

Our Ref./Docket No: APPT-001-1 Patent 7/8/03

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant(s): Dietz, *et al.* Application No.: 09/608237 Filed: June 30, 2000 Title: METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK	Group Art Unit: 2157 Examiner: Moustafa M. Meky

TRANSMITTAL: RESPONSE TO OFFICE ACTION

Mail Stop No Fee Amendment
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Dear Commissioner:

Transmitted herewith is a response to an office action for the above referenced application. Included with the response are:

____ formal drawings (with separate letter);

This application has:

____ a small entity status. If a claim for such status has not earlier been made, consider this as a claim for small entity status.

__X__ No additional fee is required.

Certificate of Facsimile Transmission under 37 CFR 1.8

I hereby certify that this response is being facsimile transmitted to the United States Patent and Trademark Office at telephone number 703-~~308-9052~~ 746-7239 addressed the Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450 on.

Date: 7/8/03 Signed: _____

Name: Dov Rosenfeld, Reg. No. 38687

Received from < +1 510 291 2985 > at 7/8/03 7:43:30 PM [Eastern Daylight Time]

_X_ Applicant(s) believe(s) that no Extension of Time is required. However, this conditional petition is being made to provide for the possibility that applicant has inadvertently overlooked the need for a petition for an extension of time.

____ Applicant(s) hereby petition(s) for an Extension of Time under 37 CFR 1.136(a) of:

_____ one months ($110) ____ two months ($410)

_____ two months ($930) ____ four months ($1450)

If an additional extension of time is required, please consider this as a petition therefor.

____ A credit card payment form for the required fee(s) is attached.

_X_ The Commissioner is hereby authorized to charge payment of the following fees associated with this communication or credit any overpayment to Deposit Account No. 50-0292 (A DUPLICATE OF THIS TRANSMITTAL IS ATTACHED):

_X_ Any missing filing fees required under 37 CFR 1.16 for presentation of additional claims.

_X_ Any missing extension or petition fees required under 37 CFR 1.17.

Respectfully Submitted,

7/8/03
Date

Dov Rosenfeld, Reg. No. 38687

Address for correspondence:
Dov Rosenfeld
5507 College Avenue,Suite 2
Oakland, CA 94618
Tel. +1-510-547-3378; Fax: +1-510-291-2985

Received from < +1 510 291 2985 > at 7/8/03 7:43:30 PM [Eastern Daylight Time]

UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
09/608,237	06/30/2000	Russell S. Dietz	APPT-001-1	9993

7590 07/17/2003

EXAMINER
MEKY, MOUSTAFA M

ART UNIT	PAPER NUMBER
2157	

DATE MAILED: 07/17/2003

Dov Rosenfeld
Suite 2
5507 College Avenue
Oakland, CA 94618
UNITED STATES

Notice of Fee Increase on January 1, 2003

If a reply to a "Notice of Allowance and Fee(s) Due" is filed in the Office on or after January 1, 2003, then the amount due will be higher than that set forth in the "Notice of Allowance and Fee(s) Due" since there will be an increase in fees effective on January 1, 2003. See Revision of Patent and Trademark Fees for Fiscal Year 2003: Final Rule, 67 Fed. Reg. 70847, 70849 (November 27, 2002).

The current fee schedule is accessible from: http://www.uspto.gov/main/howtofees.htm.

If the issue fee paid is the amount shown on the "Notice of Allowance and Fee(s) Due," but not the correct amount in view of the fee increase, a "Notice to Pay Balance of Issue Fee" will be mailed to applicant. In order to avoid processing delays associated with mailing of a "Notice to Pay Balance of Issue Fee," if the response to the Notice of Allowance and Fee(s) due form is to be filed on or after January 1, 2003 (or mailed with a certificate of mailing on or after January 1, 2003), the issue fee paid should be the fee that is required at the time the fee is paid. If the issue fee was previously paid, and the response to the "Notice of Allowance and Fee(s) Due" includes a request to apply a previously-paid issue fee to the issue fee now due, then the difference between the issue fee amount at the time the response is filed and the previously paid issue fee should be paid. See Manual of Patent Examining Procedure, Section 1308.01 (Eighth Edition, August 2001).

Questions relating to issue and publication fee payments should be directed to the Customer Service Center of the Office of Patent Publication at (703) 305-8283.

TOL-85 (REV. 05-03) Approved for use through 04/30/2004.

Received from <+1 510 291 2985> at 9/24/03 6:51:18 PM [Eastern Daylight Time]

OIPE JC16 SEP 24 2003 PATENT & TRADEMARK OFFICE

PART B - FEE(S) TRANSMITTAL

Complete and send this form, together with applicable fee(s), to: Mail Mail Stop ISSUE FEE
Commissioner for Patents
Alexandria, Virginia 22313-1450
Fax (703)746-4000

INSTRUCTIONS: This form should be used for transmitting the ISSUE FEE and PUBLICATION FEE (if required). Blocks 1 through 4 should be completed where appropriate. All further correspondence including the Patent, advance orders and notification of maintenance fees will be mailed to the current correspondence address as indicated unless corrected below or directed otherwise in Block 1, by (a) specifying a new correspondence address; and/or (b) indicating a separate "FEE ADDRESS" for maintenance fee notifications.

CURRENT CORRESPONDENCE ADDRESS (Note: Legibly mark-up with any corrections or use Block 1)

7590 07/17/2003

Dov Rosenfeld
Suite 2
5507 College Avenue
Oakland, CA 94618

09/26/2003 TTRAN2 00000068 09608237

01 FC:1501 1300.00 OP
02 FC:8001 30.00 OP

Note: A certificate of mailing can only be used for domestic mailings of the Fee(s) Transmittal. This certificate cannot be used for any other accompanying papers. Each additional paper, such as an assignment or formal drawing, must have its own certificate of mailing or transmission.

Certificate of Mailing or Transmission

I hereby certify that this Fee(s) Transmittal is being deposited with the United States Postal Service with sufficient postage for first class mail in an envelope addressed to the Box Issue Fee address above, or being facsimile transmitted to the USPTO, on the date indicated below.

Dov Rosenfeld	(Depositor's name)
	(Signature)
24 Sep 03	(Date)

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
09/608,237	06/30/2000	Russell S. Dietz	APFT-001-1	9993

TITLE OF INVENTION: METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK

APPLN. TYPE	SMALL ENTITY	ISSUE FEE	PUBLICATION FEE	TOTAL FEE(S) DUE	DATE DUE
nonprovisional	NO	$1300	$0	$1300	10/17/2003

EXAMINER	ART UNIT	CLASS-SUBCLASS
MEKY, MOUSTAFA M	2157	709-224000

1. Change of correspondence address or indication of "Fee Address" (37 CFR 1.363).

☐ Change of correspondence address (or Change of Correspondence Address form PTO/SB/122) attached.

☐ "Fee Address" indication (or "Fee Address" Indication form PTO/SB/47; Rev 03-02 or more recent) attached. Use of a Customer Number is required.

2. For printing on the patent front page, list (1) the names of up to 3 registered patent attorneys or agents OR, alternatively, (2) the name of a single firm (having as a member a registered attorney or agent) and the names of up to 2 registered patent attorneys or agents. If no name is listed, no name will be printed.

1 Dov Rosenfeld
2 INVENTEK
3

3. ASSIGNEE NAME AND RESIDENCE DATA TO BE PRINTED ON THE PATENT (print or type)

PLEASE NOTE: Unless an assignee is identified below, no assignee data will appear on the patent. Inclusion of assignee data is only appropriate when an assignment has been previously submitted to the USPTO or is being submitted under separate cover. Completion of this form is NOT a substitute for filing an assignment.

(A) NAME OF ASSIGNEE: Hi/fn, Inc.

(B) RESIDENCE: (CITY and STATE OR COUNTRY): Los Gatos, CA

Please check the appropriate assignee category or categories (will not be printed on the patent) ☐ individual ☒ corporation or other private group entity ☐ government

4a. The following fee(s) are enclosed:

☒ Issue Fee

☐ Publication Fee

☒ Advance Order - # of Copies 10

4b. Payment of Fee(s):

☐ A check in the amount of the fee(s) is enclosed.

☒ Payment by credit card. Form PTO-2038 is attached.

☒ The Commissioner is hereby authorized by charge the required fee(s), or credit any overpayment, to Deposit Account Number 50-0292 (enclose an extra copy of this form).

Commissioner for Patents is requested to apply the Issue Fee and Publication Fee (if any) or to re-apply any previously paid issue fee to the application identified above.

(Authorized Signature) (Date) 24 Sep 03

NOTE: The Issue Fee and Publication Fee (if required) will not be accepted from anyone other than the applicant; a registered attorney or agent; or the assignee or other party in interest as shown by the records of the United States Patent and Trademark Office.

This collection of information is required by 37 CFR 1.311. The information is required to obtain or retain a benefit by the public which is to file (and by the USPTO to process) an application. Confidentiality is governed by 35 U.S.C. 122 and 37 CFR 1.14. This collection is estimated to take 12 minutes to complete, including gathering, preparing, and submitting the completed application form to the USPTO. Time will vary depending upon the individual case. Any comments on the amount of time you require to complete this form and/or suggestions for reducing this burden, should be sent to the Chief Information Officer, U.S. Patent and Trademark Office, U.S. Department of Commerce, Alexandria, Virginia 22313-1450. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. SEND TO: Commissioner for Patents, Alexandria, Virginia 22313-1450.

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

OIPE SEP 24 2003 PATENT & TRADEMARK OFFICE JC16

TRANSMIT THIS FORM WITH FEE(S)

Sep 24 03 06:50p Dov Rosenfeld +1-510-291-2985 p.4

JC16 SEP 24 2003 PATENT & TRADEMARK OFFICE

INVENTEK

Dov Rosenfeld
6507 College Avenue, Suite 2
Oakland, CA 94618, USA
Phone: (510)547-3378; Fax: (510)653-7992
dov@inventek.com

Fax

OUR REF: APPT-001-1

TO: Mail Stop Issue Fee
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

FAX No.: (703) 746-4000

DATE: September 24, 2003

FROM: Dov Rosenfeld, Reg. No., 38,687

RE: Issue Fee for Application No.: 09/608237

Number of pages including cover: 5

OFFICIAL COMMUNICATION

ISSUE FEE PAYMENT

Included herewith are:

- A transmittal letter and copy
- Fee(s) Transmittal (form PTOL-85)
- Credit Card charge form for issue fee

Certificate of Facsimile Transmission under 37 CFR 1.8

I hereby certify that this response is being facsimile transmitted to the United States Patent and Trademark Office at telephone number (703) 746-4000 addressed to Mail Stop Issue Fee, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450 on.

Date: September 24, 2003

Signed: _____

Name: Dov Rosenfeld, Reg. No. 38687

Received from < +1 510 291 2985 > at 9/24/03 9:32:27 PM [Eastern Daylight Time]

JC16 SEP 24 2003 PATENT & TRADEMARK OFFICE

Our Ref./Docket No: APPT-001-1 Patent

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant(s): Dietz, *et al.*	Group Art Unit: 2755
Application No.: 09/608237	Examiner:
Filed: June 30, 2000	Notice of Allowance Mailed: July, 17, 2003
Title: METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK	Confirmation No: 9993

SUBMISSION OF ISSUE FEE

Mail Stop ISSUE FEE
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Dear Commissioner:

Transmitted herewith is a completed "Issue Fee Transmittal" Form. Included with the form are:

_X_ A credit card payment form for the issue fee and any advance order of copies;
____ drawing corrections (with separate letter);
____ formal drawings (with separate letter);

_X_ The Commissioner is hereby authorized to charge payment of the any missing fee or credit any overpayment to Deposit Account No. 50-0292
(A DUPLICATE OF THIS TRANSMITTAL IS ATTACHED):

Respectfully Submitted,

24 Sep 03
Date

Dov Rosenfeld, Reg. No. 38687

Address for correspondence:
Dov Rosenfeld
5507 College Avenue,Suite 2
Oakland, CA 94618
Tel. +1-510-547-3378; Fax: +1-413-638-1280

Certificate of Facsimile Transmission under 37 CFR 1.8

I hereby certify that this response is being facsimile transmitted to the United States Patent and Trademark Office at telephone number (703) 746-4000 addressed to Mail Stop Issue Fee, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450 on.

Date: September 24, 2003 Signed:

Name: Dov Rosenfeld, Reg. No. 38687

Received from < +1 510 291 2985 > at 9/24/03 9:32:27 PM [Eastern Daylight Time]

JC16 7 2004 TRADEMARK OFFICE

COFC

Our Ref./Docket No: APPT-001-1 Patent

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Inventor(s): Dietz, *et al.*

Assignee: Hi/fn, Inc.

Patent No: 6651099

Issue Date: November, 18, 2003

Application No.: 09/608237

Filed: June 30, 2000

Title: METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK

Certificate

MAR 0 5 2004

of Correction

REQUEST FOR CERTIFICATE OF CORRECTIONS

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Dear Commissioner:

The above patent contains significant error as indicated on the attached Certificate of Correction form (submitted in duplicate).

__X__ Such error arose through the fault of the Patent and Trademark Office. It is requested that the certificate be issued at no cost to the applicant.

However, if it is determined that the error arose through the fault of applicant(s), please note that such error is of clerical error or minor nature and occurred in good faith and therefore issuance of the certificate of Correction is respectfully requested. The Commissioner is authorized to charge Deposit Account No. 50-0292 any required fee. A duplicate of this request is attached.

Such errors specifically:

In col. 3, line 14, delete the word "lo"

In col. 15, line 45, kindly replace "ump" with --jump--

In col. 28, line 65, change "MEB" to --MIB--.

Certificate of Mailing under 37 CFR 1.8

I hereby certify that this response is being deposited with the United States Postal Service as first class mail in an envelope addressed to the Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450 on.

Date: 2/25/04 Signed: Amy Drury

Name: Amy Drury

8 MAR 2004

The undersigned requests being contacted at (510) 547-3378 if there are any questions or clarifications, or if there are any problems with issuance of the Certificate of Correction.

Respectfully Submitted,

Feb. 25, 2004
Date

Dov Rosenfeld, Reg. No. 38687
Agent of Record.

Address for correspondence:
Dov Rosenfeld
5507 College Avenue, Suite 2,
Oakland, CA 94618
Tel. 510-547-3378; Fax: 510-291-2985

PTO/SB/44 (10-96)
Approved for use through 6/30/99. OMB 0651-0033
Patent and Trademark Office: U.S. DEPARTMENT OF COMMERCE
Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

(Also Form PTO-1050)

UNITED STATES PATENT AND TRADEMARK OFFICE
CERTIFICATE OF CORRECTION

PATENT NO : 6651099 B1

DATED : November 18, 2003

INVENTOR(S) : Dietz, et al.

It is certified that an error appears in the above-identified patent and that said Letters Patent are hereby corrected as shown below:

In col. 3, line 14, delete the word "lo"

In col. 15, line 45, kindly replace "ump" with --jump--

In col. 28, line 65, change "MEB" to --MIB--.

MAILING ADDRESS OF SENDER (Atty/Agent of Record):
Dov Rosenfeld, Reg. No. 38687
5507 College Avenue, Suite 2
Oakland, CA 94618

PATENT NO: 6651099
No. of additional copies

 8 MAR 2004

PTO/SB/44 (10-96)
Approved for use through 6/30/99. OMB 0651-0033
Patent and Trademark Office: U.S. DEPARTMENT OF COMMERCE
Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

(Also Form PTO-1050)

UNITED STATES PATENT AND TRADEMARK OFFICE
CERTIFICATE OF CORRECTION

PATENT NO : 6651099

DATED : November 18, 2003

INVENTOR(S) : Dietz, et al.

It is certified that an error appears in the above-identified patent and that said Letters Patent are hereby corrected as shown below:

In col. 3, line 14, delete the word "lo"

In col. 15, line 45, kindly replace "ump" with --jump--

In col. 28, line 65, change "MEB" to --MIB--.

MAILING ADDRESS OF SENDER (Atty/Agent of Record):
Dov Rosenfeld, Reg. No. 38687
5507 College Avenue, Suite 2
Oakland, CA 94618

PATENT NO: 6651099
No. of additional copies

- 8 MAR 2004

US006243667B1

(12) **United States Patent**
Kerr et al.

(10) **Patent No.: US 6,243,667 B1**
(45) **Date of Patent: *Jun. 5, 2001**

(54) **NETWORK FLOW SWITCHING AND FLOW DATA EXPORT**

(75) Inventors: **Darren R. Kerr**, Union City; **Barry L. Bruins**, Los Altos, both of CA (US)

(73) Assignee: **Cisco Systems, Inc.**, San Jose, CA (US)

(*) Notice: This patent issued on a continued prosecution application filed under 37 CFR 1.53(d), and is subject to the twenty year patent term provisions of 35 U.S.C. 154(a)(2).

Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **08/655,429**

(22) Filed: **May 28, 1996**

(51) **Int. Cl.**[7] **G06F 9/34**

(52) **U.S. Cl.** **703/27**; 703/20; 370/379; 370/392; 370/389

(58) **Field of Search** 395/500, 200.01, 395/200.13, 683, 185.04; 370/352, 389, 392, 351, 410

(56) **References Cited**

U.S. PATENT DOCUMENTS

Re. 33,900	4/1992	Howson	370/105
4,131,767	12/1978	Weinstein	179/170.2
4,161,719	7/1979	Parikh et al.	340/147 SY

(List continued on next page.)

FOREIGN PATENT DOCUMENTS

0 384 758	2/1990	(EP)	H04L/12/56
91 751 A1	11/1990	(EP)	H04I/12/46
95/20850	8/1995	(WO)	H04L/12/56

OTHER PUBLICATIONS

Cormen et al., "Introduction to Algorithms", MIT Press, seventeenth edition, pp. 221–224.*
Pei et al., VLSI Implementation of Routing Tables: Tries and Cams, IEEE, 1991, pp. 515–524.*
Chandranmenon et al., "Trading Packet Headers for Packet Processing," IEEE, Apr. 1996, pp. 141–152.*
Cao et al., Performance of Hashing–Based Schemes for Internet Load Balancing, IEEE, 2000, pp. 332–341.*
Newman et al., "Flow Labelled IP: A Connectionless Approach to ATM," IEEE, Mar. 1996, pp. 1251–1260.*
Newman et al., "IP Switching and Gigabit Routers," IEEE, 1997, pp. 64–69.*
Worster et al., "Levels of Aggregation in Flow Switching Networks," IEEE, 1997, pp. 51–59.*
William Stallings, Data and Computer Communications, pp. 329–333, Prentice Hall, Upper Saddle River, New Jersey 07458.

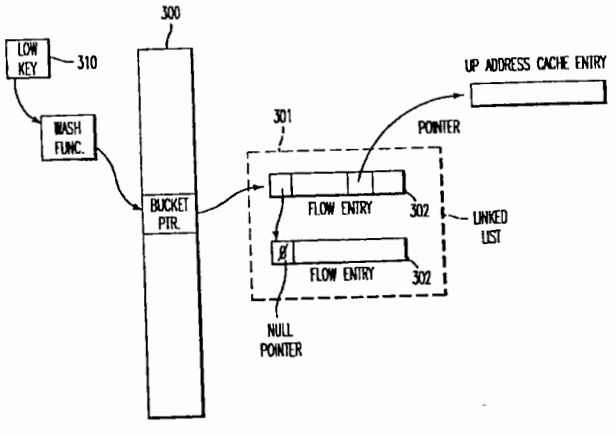
(List continued on next page.)

Primary Examiner—Kevin J. Teska
Assistant Examiner—Thai Phan
(74) *Attorney, Agent, or Firm*—Oblon, Spivak, McClelland, Maier & Neustadt, P.C.

(57) **ABSTRACT**

The invention provides a method and system for switching in networks responsive to message flow patterns. A message "flow" is defined to comprise a set of packets to be transmitted between a particular source and a particular destination. When routers in a network identify a new message flow, they determine the proper processing for packets in that message flow and cache that information for that message flow. Thereafter, when routers in a network identify a packet which is part of that message flow, they process that packet according to the proper processing for packets in that message flow. The proper processing may include a determination of a destination port for routing those packets and a determination of whether access control permits routing those packets to their indicated destination.

19 Claims, 5 Drawing Sheets



U.S. PATENT DOCUMENTS

4,316,284	2/1982	Howson	370/105
4,397,020	8/1983	Howson	370/105
4,419,728	12/1983	Larson	364/200
4,424,565	1/1984	Larson	364/200
4,437,087	3/1984	Petr	340/347 DD
4,438,511	3/1984	Baran	370/19
4,439,763	3/1984	Limb	340/825.5
4,445,213	4/1984	Baugh et al.	370/94
4,446,555	5/1984	Devault et al.	370/94
4,456,957	6/1984	Schieltz	364/200
4,464,658	8/1984	Thelen	340/825.5
4,499,576	2/1985	Fraser	370/60
4,506,358	3/1985	Montgomery	370/60
4,507,760	3/1985	Fraser	365/221
4,532,626	7/1985	Flores et al.	370/85
4,644,532	2/1987	George et al.	370/94
4,646,287	2/1987	Larson et al.	370/60
4,677,423	6/1987	Benvenuto et al.	340/347 DD
4,679,189 *	7/1987	Olson et al.	370/60
4,679,227	7/1987	Hughes-Hartogs	379/98
4,723,267	2/1988	Jones et al.	379/93
4,731,816	3/1988	Hughes-Hartogs	379/98
4,750,136	6/1988	Arpin et al.	364/514
4,757,495	7/1988	Decker et al.	370/76
4,763,191	8/1988	Gordon et al.	358/86
4,769,810	9/1988	Eckberg, Jr. et al.	370/60
4,769,811	9/1988	Eckberg, Jr. et al.	370/60
4,771,425	9/1988	Baran et al.	370/85
4,819,228	4/1989	Baran et al.	370/85
4,827,411	5/1989	Arrowhead et al.	364/300
4,833,706	5/1989	Hughes-Hartogs	379/98
4,835,737	5/1989	Herrig et al.	364/900
4,879,551	11/1989	Georgiou et al.	340/825.87
4,893,306	1/1990	Chao et al.	340/94.2
4,903,261	2/1990	Baran et al.	370/94.2
4,922,486	5/1990	Lidinsky et al.	370/60
4,933,937	6/1990	Konishi	370/85.13
4,960,310	10/1990	Cushing	350/1.7
4,962,497	10/1990	Ferenc et al.	370/60.1
4,962,532	10/1990	Kasiraj et al.	380/25
4,965,767	10/1990	Kinoshita et al. .	
4,965,772	10/1990	Daniel et al.	364/900
4,970,678	11/1990	Sladowski et al.	364/900
4,979,118 *	12/1990	Kheradpir	364/436
4,980,897	12/1990	Decker et al.	375/38
4,991,169	2/1991	Davis et al.	370/77
5,003,595	3/1991	Collins et al.	380/25
5,014,265	5/1991	Hahne et al.	370/60
5,020,058	5/1991	Holden et al.	370/109
5,033,076	7/1991	Jones et al.	379/67
5,034,919	7/1991	Sasai et al. .	
5,054,034	10/1991	Hughes-Hartogs	375/8
5,059,925	10/1991	Weisbloom	331/1 A
5,072,449	12/1991	Enns et al.	371/37.1
5,088,032	2/1992	Bosack	395/200
5,095,480 *	3/1992	Fenner et al.	370/94.1
5,115,431	5/1992	Williams et al.	370/94.1
5,128,945	7/1992	Enns et al.	371/37.1
5,136,580	8/1992	Videlock et al.	370/60
5,166,930	11/1992	Braff et al.	370/94.1
5,199,049	3/1993	Wilson	375/104
5,206,886	4/1993	Bingham	375/97
5,208,811	5/1993	Kashio et al. .	
5,212,686	5/1993	Joy et al.	370/60
5,224,099	6/1993	Corbalis et al.	370/94.2
5,226,120	7/1993	Brown et al.	395/200
5,228,062	7/1993	Bingham	375/97
5,229,994	7/1993	Balzano et al.	370/85.13
5,237,564	8/1993	Lespagnol et al.	370/60.1
5,241,682	8/1993	Bryant et al.	395/800
5,243,342	9/1993	Kattemalalavadi et al.	341/106
5,243,596	9/1993	Port et al.	370/94.1
5,247,516	9/1993	Bernstein et al.	370/82
5,249,178	9/1993	Kurano et al.	370/60
5,249,292 *	9/1993	Chiappa	395/650
5,253,251	10/1993	Aramaki .	
5,255,291	10/1993	Holden et al.	375/111
5,260,933	11/1993	Rouse	370/14
5,260,978	11/1993	Fleischer et al.	375/106
5,268,592	12/1993	Bellamy et al.	307/43
5,268,900	12/1993	Hluchyj et al.	370/94.1
5,271,004	12/1993	Proctor et al.	370/60
5,274,631	12/1993	Bhardwaj	370/60
5,274,635	12/1993	Rahman et al.	370/60.1
5,274,643	12/1993	Fisk	370/94.1
5,280,470	1/1994	Buhrke et al.	370/13
5,280,480	1/1994	Pitt et al.	370/85.13
5,280,500	1/1994	Mazzola et al.	375/17
5,283,783	2/1994	Nguyen et al.	370/16.1
5,287,103	2/1994	Kasprzyk et al.	340/825.52
5,287,453 *	2/1994	Roberts	395/200
5,291,482	3/1994	McHarg et al.	370/60
5,305,311	4/1994	Lyles	370/60
5,307,343	4/1994	Bostica et al.	370/60
5,309,437 *	5/1994	Perlman et al.	370/85.13
5,311,509	5/1994	Heddes et al.	370/60
5,313,454	5/1994	Bustini et al.	370/13
5,313,582	5/1994	Hendel et al.	395/250
5,317,562	5/1994	Nardin et al.	370/16
5,319,644	6/1994	Liang	370/85.5
5,327,421	7/1994	Hiller et al.	370/60.1
5,331,637	7/1994	Francis et al. .	
5,345,445	9/1994	Hiller et al.	370/60.1
5,345,446	9/1994	Hiller et al.	370/60.1
5,359,592	10/1994	Corbalis et al.	370/17
5,361,250	11/1994	Nguyen et al.	370/16.1
5,361,256	11/1994	Doeringer et al. .	
5,361,259	11/1994	Hunt et al.	370/84
5,365,524	11/1994	Hiller et al.	370/94.2
5,367,517	11/1994	Cidon et al.	370/54
5,371,852	12/1994	Attanasio et al.	395/200
5,386,567	1/1995	Lien et al.	395/700
5,390,170	2/1995	Sawant et al.	370/58.1
5,390,175	2/1995	Hiller et al.	370/60
5,394,394	2/1995	Crowther et al.	370/60
5,394,402	2/1995	Ross	370/94.1
5,400,325	3/1995	Chatwani et al.	370/60.1
5,408,469	4/1995	Opher et al.	370/60.1
5,416,842	5/1995	Aziz	380/30
5,422,880	6/1995	Heitkamp et al.	370/60
5,422,882	6/1995	Hiller et al.	370/60.1
5,423,002	6/1995	Hart	395/200
5,426,636	6/1995	Hiller et al.	370/60.1
5,426,637 *	6/1995	Derby et al.	370/85.13
5,428,607	6/1995	Hiller et al.	370/60.1
5,430,715	7/1995	Corbalis et al.	370/54
5,430,729	7/1995	Rahnema .	
5,442,457	8/1995	Najafi	385/400
5,442,630	8/1995	Gagliardi et al.	370/85.13
5,452,297	9/1995	Hiller et al.	370/60.1
5,473,599	12/1995	Li et al.	370/16
5,473,607	12/1995	Hausman et al.	370/85.13
5,477,541	12/1995	White et al. .	
5,485,455 *	1/1996	Dobbins et al.	370/60
5,490,140	2/1996	Abensour et al. .	
5,490,258 *	2/1996	Fenner	395/401
5,491,687	2/1996	Christensen et al.	370/17
5,491,693 *	2/1996	Britton et al.	370/85.13
5,491,804	2/1996	Heath et al.	395/275
5,497,368	3/1996	Reijnierse et al. .	
5,504,747	4/1996	Sweazey .	

5,509,006		4/1996	Wilford et al.	370/60
5,509,123	*	4/1996	Dobbins et al.	395/200.15
5,517,494		5/1996	Green .	
5,517,662	*	5/1996	Coleman et al.	395/800
5,519,704		5/1996	Farinacci et al.	370/85.13
5,519,858	*	5/1996	Walton et al.	395/600
5,524,254	*	6/1996	Morgan et al.	395/800
5,526,489		6/1996	Nilakantan et al.	395/200.02
5,530,963		6/1996	Moore et al.	395/200.15
5,535,195		7/1996	Lee	370/54
5,539,734		7/1996	Burwell et al. .	
5,541,911		7/1996	Nilakantan et al. .	
5,546,370		8/1996	Ishikawa .	
5,550,816	*	8/1996	Hardwick et al.	370/60
5,555,244		9/1996	Gupta et al.	370/60
5,561,669		10/1996	Lenney et al.	370/60.1
5,583,862		12/1996	Callon	370/397
5,592,470		1/1997	Rudrapatna et al.	370/320
5,598,581		1/1997	Daines et al.	395/872
5,600,798		2/1997	Chenrukuri et al. .	
5,602,770		2/1997	Ohira .	
5,604,868		2/1997	Komine et al.	395/200
5,608,726		3/1997	Virgile .	
5,617,417		4/1997	Sathe et al.	370/394
5,617,421		4/1997	Chin et al.	370/402
5,630,125	*	5/1997	Zellweger	395/614
5,631,908		5/1997	Saxe .	
5,632,021		5/1997	Jennings et al.	395/309
5,634,010		5/1997	Ciscon et al.	395/200
5,634,011	*	5/1997	Auerbach et al.	395/200.15
5,644,718		7/1997	Belove et al.	395/200
5,666,353		9/1997	Klausmeier et al.	370/230
5,673,265		9/1997	Gupta et al.	370/432
5,675,579	*	10/1997	Watson et al.	370/248
5,678,006		10/1997	Valizadeh et al.	395/200
5,680,116		10/1997	Hashimoto et al. .	
5,684,800	*	11/1997	Dobbins et al.	370/401
5,687,324		11/1997	Green et al. .	
5,724,351		3/1998	Chao et al. .	
5,740,097		4/1998	Satoh .	
5,748,186	*	5/1998	Raman	345/302
5,754,547		5/1998	Nakazawa .	
5,802,054		9/1998	Bellenger .	
5,835,710	*	11/1998	Nagami et al.	395/200.8
5,841,874		11/1998	Kempke et al. .	
5,854,903		12/1998	Morrison et al. .	
5,856,981		1/1999	Voelker .	
5,892,924	*	4/1999	Lyon et al.	395/200.75
5,898,686		4/1999	Virgile .	
5,903,559		5/1999	Acharya et al. .	
5,925,097	*	7/1999	Gopinath et al.	709/200

OTHER PUBLICATIONS

Chowdhury, et al., "Alternative Banddwidth Allocation Algorithms for Packet Video in ATM Networks", 1992, IEEE Infocom 92, pp. 1061–1068.

Zhang, et al., "Rate–Controlled Static–Priority Queueing", 1993, IEEE, pp. 227–236.

Doeringer, et al., "Routing on Longest–Matching Prefixes", IEEE ACM Transactions on Networking, Feb. 1, 1996, vol. 4, No. 1, pp. 86–97.

IBM, "Method and Apparatus for the Statistical Multiplexing of Voice, Data, and Image Signals", Nov., 1992, IBM Technical Data Bulletin n6, pp. 409–411.

Esaki, et al., "Datagram Delivery in an ATM–Internet," IEICE Transactions on Communications vol. E77–B, No. 3, (1994) Mar., Tokyo, Japan.

* cited by examiner

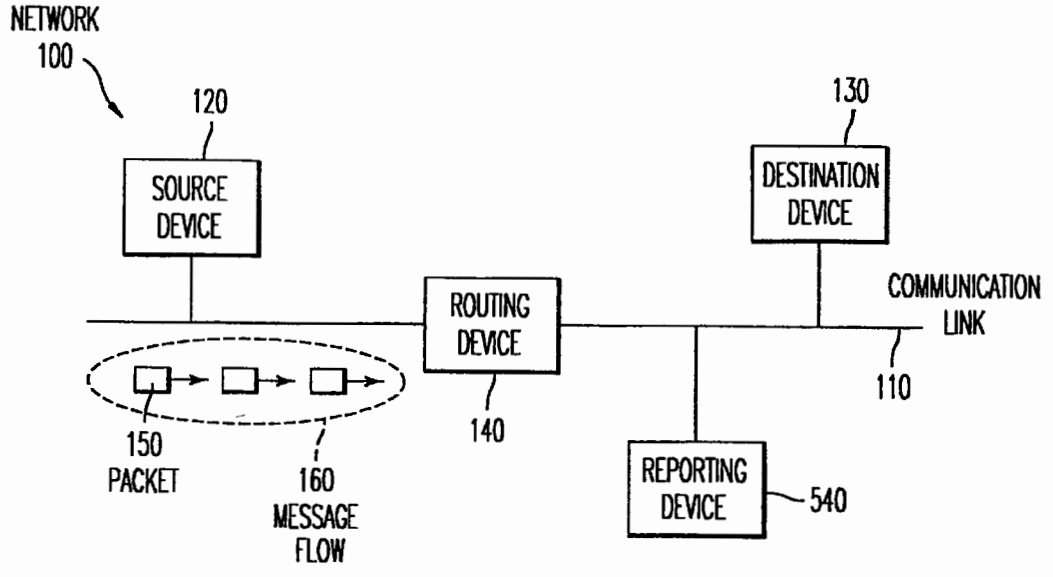

NETWORK
100
120
SOURCE DEVICE
130
DESTINATION DEVICE
ROUTING DEVICE
140
COMMUNICATION LINK
110
150
PACKET
160
MESSAGE FLOW
REPORTING DEVICE
540

FIG. 1

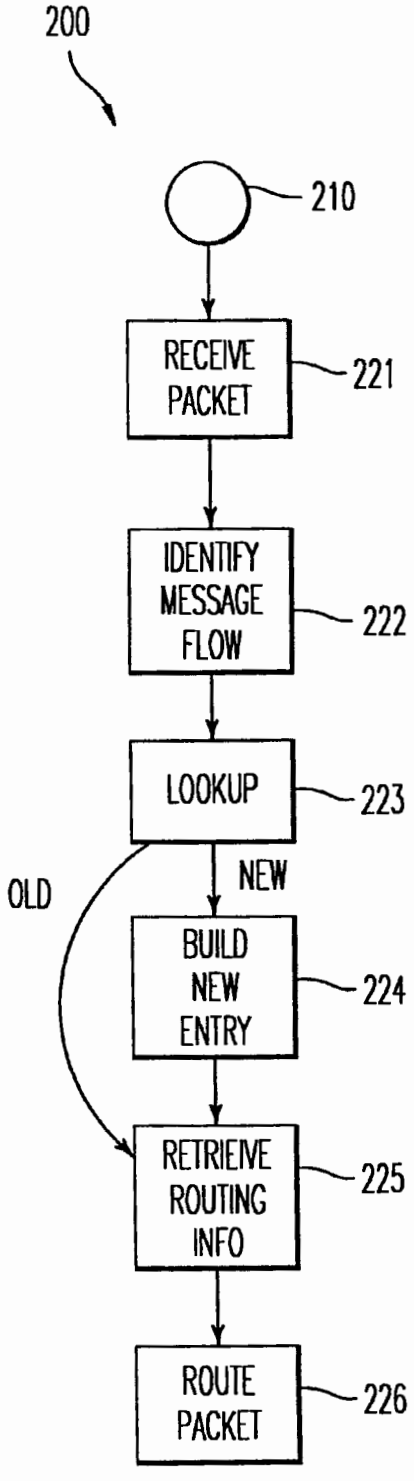

200
210
RECEIVE PACKET
221
IDENTIFY MESSAGE FLOW
222
LOOKUP
223
OLD
NEW
BUILD NEW ENTRY
224
RETRIEVE ROUTING INFO
225
ROUTE PACKET
226

FIG. 2A

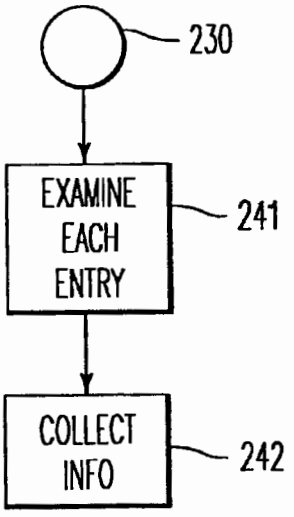

230
EXAMINE EACH ENTRY
241
COLLECT INFO
242

FIG. 2B

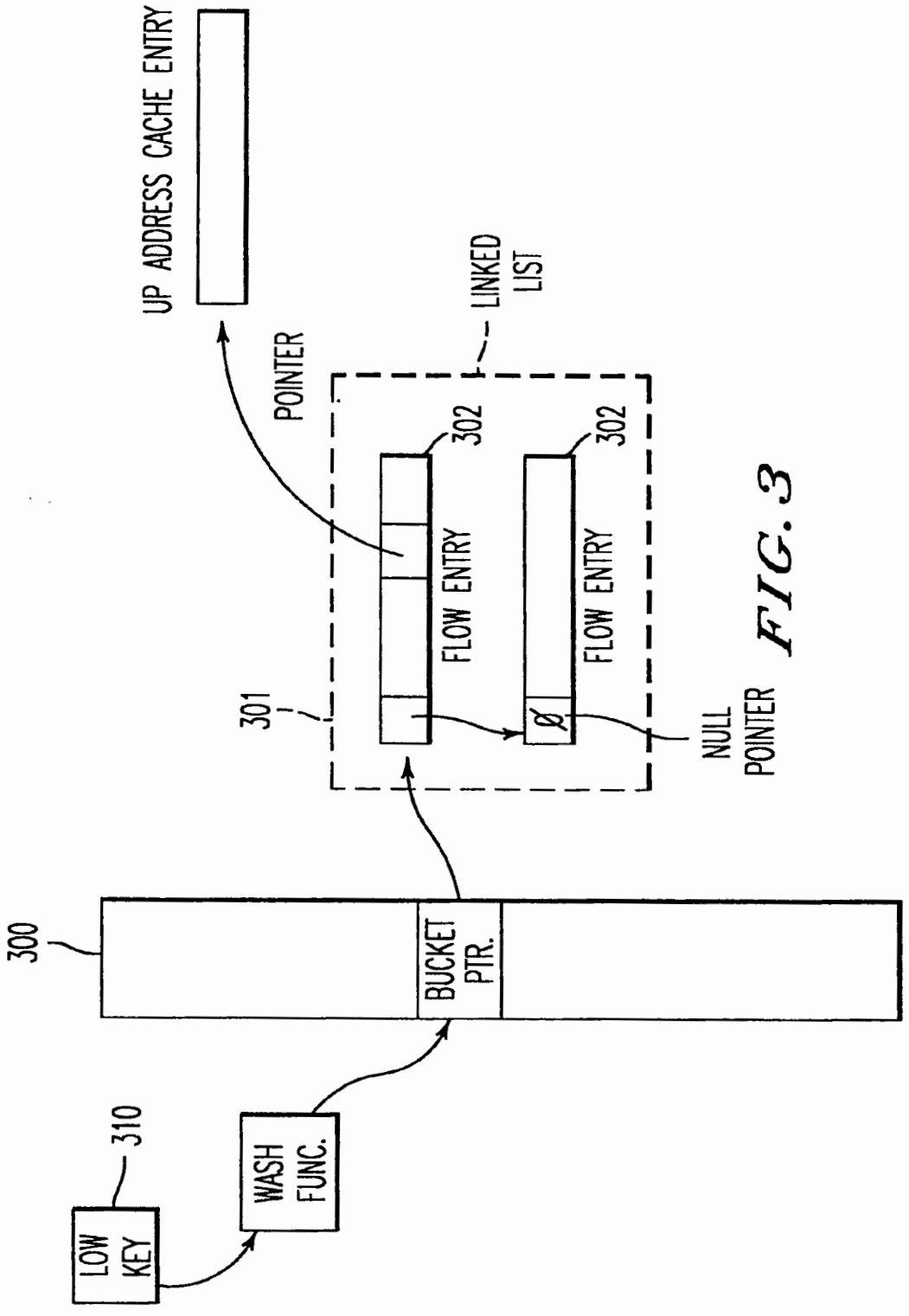

UP ADDRESS CACHE ENTRY
POINTER
LINKED LIST
302
302
FLOW ENTRY
FLOW ENTRY
301
NULL POINTER
300
BUCKET PTR.
310
WASH FUNC.
LOW KEY

FIG. 3

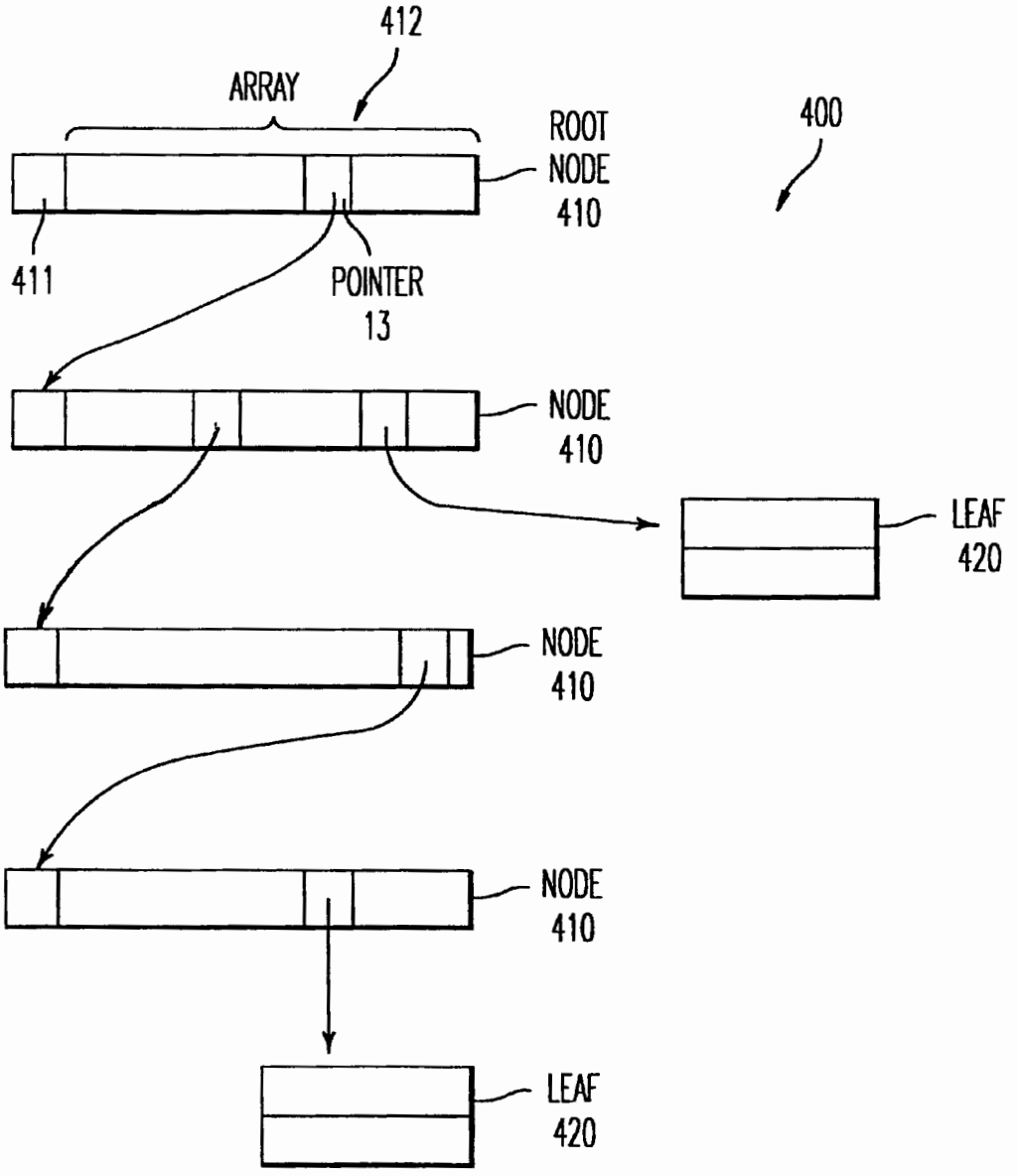

412
ARRAY
ROOT
NODE
410
400
411
POINTER
13
NODE
410
LEAF
420
NODE
410
NODE
410
LEAF
420

FIG. 4

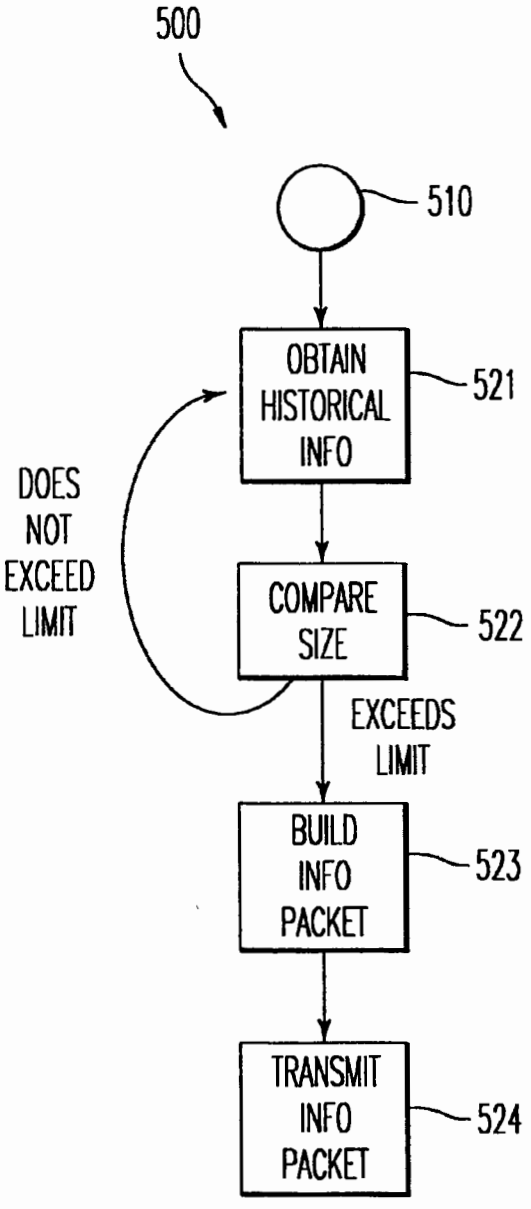

500
510
OBTAIN HISTORICAL INFO
521
DOES NOT EXCEED LIMIT
COMPARE SIZE
522
EXCEEDS LIMIT
BUILD INFO PACKET
523
TRANSMIT INFO PACKET
524

FIG. 5A

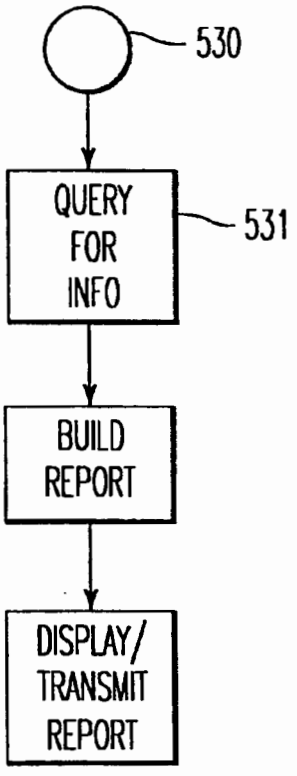

530
QUERY FOR INFO
531
BUILD REPORT
DISPLAY/ TRANSMIT REPORT

FIG. 5B

NETWORK FLOW SWITCHING AND FLOW DATA EXPORT

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to network switching and data export responsive to message flow patterns.

2. Description of Related Art

In computer networks, it commonly occurs that message traffic between a particular source and a particular destination will continue for a time with unchanged routing or switching parameters. For example, when using the file-transfer protocol "FTP" there is substantial message traffic between the file's source location and the file's destination location, comprising the transfer of many packets which have similar headers, differing in the actual data which is transmitted. During the time when message traffic continues, routing and switching devices receiving packets comprising that message traffic must examine those packets and determine the processing thereof.

One problem which has arisen in the art is that processing demands on routing and switching devices continue to grow with increased network demand. It continues to be advantageous to provide techniques for processing packets more quickly. This problem has been exacerbated by addition of more complex forms of processing, such as the use of access control lists.

It would therefore be advantageous to provide techniques in which the amount of processing required for any individual packet could be reduced. With inventive techniques described herein, information about message flow patterns is used to identify packets for which processing has already been determined, and therefore to process those packets without having to re-determine the same processing. The amount of processing required for any individual packet is therefore reduced.

Information about message flow patterns would also be valuable for providing information about use of the network, and could be used for a variety of purposes by network administrators, routing devices, service providers, and users.

Accordingly, it would be advantageous to provide a technique for network switching and data export responsive to message flow patterns.

SUMMARY OF THE INVENTION

The invention provides a method and system for switching in networks responsive to message flow patterns. A message "flow" is defined to comprise a set of packets to be transmitted between a particular source and a particular destination. When routers in a network identify a new message flow, they determine the proper processing for packets in that message flow and cache that information for that message flow. Thereafter, when routers in a network identify a packet which is part of that message flow, they process that packet according to the proper processing for packets in that message flow. The proper processing may include a determination of a destination port for routing those packets and a determination of whether access control permits routing those packets to their indicated destination.

In another aspect of the invention, information about message flow patterns is collected, responsive to identified message flows and their packets. The collected information is reported to devices on the network. The collected information is used for a variety of purposes, including: to diagnose actual or potential network problems, to determine

patterns of usage by date and time or by location, to determine which services and which users use a relatively larger or smaller amount of network resources, to determine which services are accessed by particular users, to determine which users access particular services, or to determine usage which falls within selected parameters (such as: access during particular dates or times, access to prohibited services, excessive access to particular services, excessive use of network resources, or lack of proper access).

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 shows a network in which routing responsive to message flow patterns is performed.

FIG. 2 shows a method for routing in networks responsive to message flow patterns.

FIG. 3 shows data structures for use with a method for routing in networks responsive to message flow patterns.

FIG. 4 shows an IP address cache for use with a method for routing in networks responsive to message flow patterns.

FIG. 5 shows a method for collecting and reporting information about message flow patterns.

DESCRIPTION OF THE PREFERRED EMBODIMENT

In the following description, a preferred embodiment of the invention is described with regard to preferred process steps and data structures. However, those skilled in the art would recognize, after perusal of this application, that embodiments of the invention may be implemented using a set of general purpose computers operating under program control, and that modification of a set of general purpose computers to implement the process steps and data structures described herein would not require undue invention.

Message Flows

FIG. **1** shows a network in which routing responsive to message flow patterns is performed.

A network **100** includes at least one communication link **110**, at least one source device **120**, at least one destination device **130**, and at least one routing device **140**. The routing device **140** is disposed for receiving a set of packets **150** from the source device **120** and routing them to the destination device **130**.

The communication link **110** may comprise any form of physical media layer, such as ethernet, FDDI, or HDLC serial link.

The routing device **140** comprises a routing processor for performing the process steps described herein, and may include specific hardware constructed or programmed performing the process steps described herein, a general purpose processor operating under program control, or some combination thereof.

A message flow **160** consists of a unidirectional stream of packets **150** to be transmitted between particular pairs of transport service access points (thus, network-layer addresses and port numbers). In a broad sense, a message flow **160** thus refers to a communication "circuit" between communication end-points. In a preferred embodiment, a message flow **160** is defined by a network-layer address for a particular source device **120**, a particular port number at the source device **120**, a network-layer address for a particular destination device **130**, a particular port number at the destination device **130**, and a particular transmission protocol type. For example, the transmission protocol type may

identify a known transmission protocol, such as UDP, TCP, ICMP, or IGMP (internet group management protocol).

In a preferred embodiment for use with a network of networks (an "internet"), the particular source device **120** is identified by its IP (internet protocol) address. The particular port number at the source device **120** is identified by either a port number which is specific to a particular process, or by a standard port number for the particular transmission protocol type. For example, a standard port number for the TCP protocol type is **6** and a standard port number for the UDP protocol type is **17**. Other protocols which may have standard port numbers include the FTP protocol, the TELNET protocol, an internet telephone protocol, or an internet video protocol such as the "CUSeeMe" protocol; these protocols are known in the art of networking. Similarly, the particular destination device **130** is identified by its IP (internet protocol) address; the particular port number at the destination device **130** is identified by either a port number which is specific to a particular process, or a standard port number for the particular transmission protocol type.

It will be clear to those skilled in the art, after perusing this application, that the concept of a message flow is quite broad, and encompasses a wide variety of possible alternatives within the scope and spirit of the invention. For example, in alternative embodiments, a message flow may be bi-directional instead of unidirectional, a message flow may be identified at a different protocol layer level than that of transport service access points, or a message flow may be identified responsive to other factors. These other factors may include one or more of the following: information in packet headers, packet length, time of packet transmission, or routing conditions on the network (such as relative network congestion or administrative policies with regard to routing and transmission).

Network Flow Switching

FIG. **2** shows a method for routing in networks responsive to message flow patterns.

In broad overview, the method for routing in networks responsive to message flow patterns comprises two parts. In a first part, the routing device **140** builds and uses a flow cache described in further detail with regard to FIG. **3**), in which routing information to be used for packets **150** in each particular message flow **160** is recorded and from which such routing information is retrieved for use. In a second part, the routing device **140** maintains the flow cache, such as by removing entries for message flows **160** which are no longer considered valid.

A method **200** for routing in networks responsive to message flow patterns is performed by the routing device **140**.

At a flow point **210**, the routing device **140** is disposed for building and using the flow cache.

At a step **221**, the routing device **140** receives a packet **150**.

At a step **222**, the routing device **140** identifies a message flow **160** for the packet **150**. In a preferred embodiment, the routing device **140** examines a header for the packet **150** and identifies the IP address for the source device **120**, the IP address for the destination device **130**, and the protocol type for the packet **150**. The routing device **140** determines the port number for the source device **120** and the port number for the destination device **130** responsive to the protocol type. Responsive to this set of information, the routing device **140** determines a flow key **310** (described with reference to FIG. **3**) for the message flow **160**.

At a step **223**, the routing device **140** performs a lookup in a flow cache for the identified message flow **160**. If the lookup is unsuccessful, the identified message flow **160** is a "new" message flow **160**, and the routing device **140** continues with the step **224**. If the lookup is successful, the identified message flow **160** is an "old" message flow **160**, and the routing device **140** continues with the step **225**.

In a preferred embodiment, the routing device **140** determines a hash table key responsive to the flow key **310**. This aspect of the step **223** is described in further detail with regard to FIG. **3**.

At a step **224**, the routing device **140** builds a new entry in the flow cache. The routing device **140** determines proper treatment of packets **150** in the message flow **160** and enters information regarding such proper treatment in a data structure pointed to by the new entry in the flow cache. In a preferred embodiment, the routing device **140** determines the proper treatment by performing a lookup in an IP address cache as shown in FIG. **4**.

In a preferred embodiment, the proper treatment of packets **150** in the message flow **160** includes treatment with regard to switching (thus, the routing device **140** determines an output port for switching packets **150** in the message flow **160**), with regard to access control (thus, the routing device **140** determines whether packets **150** in the message flow **160** meet the requirements of access control, as defined by access control lists in force at the routing device **140**), with regard to accounting (thus, the routing device **140** creates an accounting record for the message flow **160**), with regard to encryption (thus, the routing device **140** determines encryption treatment for packets **150** in the message flow **160**), and any special treatment for packets **150** in the message flow **160**.

In a preferred embodiment, the routing device **140** performs any special processing for new message flows **160** at this time. For example, in one preferred embodiment, the routing device **140** requires that the source device **120** or the destination device **130** must authenticate the message flow **160**. In that case, the routing device **140** transmits one or more packets **150** to the source device **120** or the destination device **130** to request information (such as a user identifier and a password) to authenticate the new message flow **160**, and receives one or more packets **150** comprising the authentication information. This technique could be useful for implementing security "firewalls" and other authentication systems.

Thereafter, the routing device **140** proceeds with the step **225**, using the information from the new entry in the flow cache, just as if the identified message flow **160** were an "old" message flow **160** and the lookup in a flow cache had been successful.

At a step **225**, the routing device **140** retrieves routing information from the entry in the flow cache for the identified message flow **160**.

In a preferred embodiment, the entry in the flow cache includes a pointer to a rewrite function for at least part of a header for the packet **150**. If this pointer is non-null, the routing device **140** invokes the rewrite function to alter the header for the packet **150**.

At a step **226**, the routing device **140** routes the packet **150** responsive to the routing information retrieved at the step **225**.

Thus, in a preferred embodiment, the routing device **140** does not separately determine, for each packet **150** in the message flow **160**, the information stored in the entry in the flow cache. Rather, when routing a packet **150** in the

message flow **160**, the routing device **140** reads the information from the entry in the flow cache and treats the packet **150** according to the information in the entry in the flow cache.

Thus, in a preferred embodiment, the routing device **140** routes the packet **150** to an output port, determines whether access is allowed for the packet **150**, determines encryption treatment for the packet **150**, and performs any special treatment for the packet **150**, all responsive to information in the entry in the flow cache.

In a preferred embodiment, the routing device **140** also enters accounting information in the entry in the flow cache for the packet **150**. When routing each packet **150** in the message flow **160**, the routing device **140** records the cumulative number of packets **150** and the cumulative number of bytes for the message flow **160**.

Because the routing device **140** processes each packet **150** in the message flow **160** responsive to the entry for the message flow **160** in the flow cache, the routing device **140** is able to implement administrative policies which are designated for each message flow **160** rather than for each packet **150**. For example, the routing device **140** is able to reserve specific amounts of bandwidth for particular message flows **160** and to queue packets **150** for transmission responsive to the bandwidth reserved for their particular message flows **160**.

Because the routing device **140** is able to associate each packet **150** with a particular message flow **160** and to associate each message flow **160** with particular network-layer source and destination addresses, the routing device **140** is able to associate network usage with particular workstations (and therefore with particular users) or with particular services available on the network. This can be used for accounting purposes, for enforcing administrative policies, or for providing usage information to interested parties.

For a first example, the routing device **140** is able to monitor and provide usage information regarding access using the HTTP protocol to world wide web pages at particular sites.

For a second example, the routing device **140** is able to monitor usage information regarding relative use of network resources, and to give priority to those message flows **160** which use relatively fewer network resources. This can occur when a first message flow **160** is using a relatively low-bandwidth transmission channel (such as a 28.8 kilobits per second modem transmission channel) and when a second message flow **160** is using a relatively high-bandwidth transmission channel (such as a T-1 transmission line).

At a flow point **230**, the routing device **140** is disposed for maintaining the flow cache.

At a step **241**, the routing device **140** examines each entry in the flow cache and compares a current time with a last time a packet **150** was routed using that particular entry. If the difference exceeds a first selected timeout, the message flow **160** represented by that entry is considered to have expired due to nonuse and thus to no longer be valid.

In a preferred embodiment, the routing device **140** also examines the entry in the flow cache and compares a current time with a first time a packet **150** was routed using that particular entry. If the difference exceeds a second selected timeout, the message flow **160** represented by that entry is considered to have expired due to age and thus to no longer be valid. The second selected timeout is preferably about one minute.

Expiring message flows **160** due to age artificially requires that a new message flow **160** must be created for the next packet **150** in the same communication session represented by the old message flow **160** which was expired. However, it is considered preferable to do so because it allows information to be collected and reported about message flows **160** without having to wait for those message flows **160** to expire from nonuse. For example, a multiple-broadcast communication session could reasonably last well beyond the time message flows **160** are expired for age, and if not so expired would mean that information about network usage would not account for significant network usage.

In a preferred embodiment, the routing device **140** also examines the entry in the flow cache and determines if the "next hop" information has changed. If so, the message flow **160** is expired due to changed conditions. Other changed conditions which might cause a message flow **160** to be expired include changes in access control lists or other changes which might affect the proper treatment of packets **150** in the message flow **160**. The routing device **140** also expires entries in the flow cache on a least-recently-used basis if the flow cache becomes too full.

If the message flow **160** is still valid, the routing device **140** continues with the next entry in the flow cache until all entries have been examined. If the message flow **160** is no longer valid, the routing device **140** continues with the step **242**.

At a step **242**, the routing device **140** collects historical information about the message flow **160** from the entry in the flow cache, and deletes the entry.

Flow Cache

FIG. **3** shows data structures for use with a method for routing in networks responsive to message flow patterns.

A flow cache **300** comprises a memory which associates flow keys **310** with information about message flows **160** identified by those flow keys **310**. The flow cache **300** includes a set of buckets **301**. Each bucket **301** includes a linked list of entries **302**. Each entry **302** includes information about a particular message flow **160**, including routing, access control, accounting, special treatment for packets **150** in that particular message flow **160**, and a pointer to information about treatment of packets **150** to the destination device **130** for that message flow **160**.

In a preferred embodiment, the flow cache **300** includes a relatively large number of buckets **301** (preferably about 16,384 buckets **301**), so as to minimize the number of entries **302** per bucket **301** and thus so as to minimize the number of memory accesses per entry **302**. Each bucket **301** comprises a four-byte pointer to a linked list of entries **302**. The linked list preferably includes only about one or two entries **302** at the most.

In a preferred embodiment, each entry **302** includes a set of routing information, a set of access control information, a set of special treatment information, and a set of accounting information, for packets **150** in the message flow **160**.

The routing information comprises the output port for routing packets **150** in the message flow **160**.

The access control information comprises whether access is permitted for packets **150** in the message flow **160**.

The accounting information comprises a time stamp for the first packet **150** in the message flow **160**, a time stamp for the most recent packet **150** in the message flow **160**, a cumulative count for the number of packets **150** in the message flow **160**, and a cumulative count for the number of bytes **150** in the message flow **160**.

IP Address Cache

FIG. **4** shows an IP address cache for use with a method for routing in networks responsive to message flow patterns.

An IP address cache **400** comprises a tree having a root node **410**, a plurality of inferior nodes **410**, and a plurality of leaf data structures **420**.

Each node **410** comprises a node/leaf indicator **411** and an array **412** of pointers **413**.

The node/leaf indicator **411** indicates whether the node **410** is a node **410** or a leaf data structure **420**; for nodes **410** it is set to a "node" value, while for leaf data structures **420** it is set to a "leaf" value.

The array **412** has room for exactly 256 pointers **413**; thus, the IP address cache **400** comprises an M-trie with a branching width of 256 at each level. M-tries are known in the art of tree structures. IP addresses comprise four bytes, each having eight bits and therefore 256 possible values. Thus, each possible IP address can be stored in the IP address cache **400** using at most four pointers **413**.

The inventors have discovered that IP addresses in actual use are unexpectedly clustered, so that the size of the IP address cache **400** is substantially less, by a factor of about five to a factor of about ten, than would be expected for a set of randomly generated four-byte IP addresses.

Each pointer **413** represents a subtree of the IP address cache **400** for its particular location in the array **412**. Thus, for the root node **410**, the pointer **413** at location **3** represents IP addresses having the form 3.xxx.xxx.xxx, where "xxx" represents any possible value from zero to 255. Similarly, in a subtree for IP addresses having the form 3.xxx.xxx.xxx, the pointer **413** at location **141** represents IP addresses having the form 3.141.xxx.xxx. Similarly, in a subtree for IP addresses having the form 3.141.xxx.xxx, the pointer **413** at location **59** represents IP addresses having the form 3.141.59.xxx. Similarly, in a subtree for IP addresses having the form 3.141.59.xxx, the pointer **413** at location **26** represents the IP address 3.141.59.26.

Each pointer **413** is either null, to indicate that there are no IP addresses for the indicated subtree, or points to an inferior node **410** or leaf data structure **420**. A least significant bit of each pointer **413** is reserved to indicate the type of the pointed-to structure; that is, whether the pointed-to structure is a node **410** or a leaf data structure **420**. In a preferred embodiment where pointers **413** must identify an address which is aligned on a four-byte boundary, the two least significant bits of each pointer **413** are unused for addressing, and reserving the least significant bit for this purpose does not reduce the scope of the pointer **413**.

Each leaf data structure comprises information about the IP address, stored in the IP address cache **400**. In a preferred embodiment this information includes the proper processing for packets **150** addressed to that IP address, such as a determination of a destination port for routing those packets and a determination of whether access control permits routing those packets to their indicated destination.

Flow Data Export

FIG. **5** shows a method for collecting and reporting information about message flow patterns.

A method **500** for collecting and reporting information about message flow patterns is performed by the routing device **140**.

At a flow point **510**, the routing device **140** is disposed for obtaining information about a message flow **160**. For example, in a preferred embodiment, as noted herein, the routing device **140** obtains historical information about a message flow **160** in the step **242**. In alternative embodiments, the routing device **140** may obtain information about message flows **160**, either in addition or instead, by occasional review of entries in the flow cache, or by directly monitoring packets **150** in message flows **160**.

It will be clear to those skilled in the art, after perusing this application, that the concept of reporting information about message flows is quite broad, and encompasses a wide variety of possible alternatives within the scope and spirit of the invention. For example, in alternative embodiments, information about message flows may include bi-directional traffic information instead of unidirectional traffic information, information about message flows may include information at a different protocol layer level other than that of transport service access points and other than that at which the message flow is itself defined, or information about message flows may include actual data transmitted as part of the message flow itself. These actual data may include one or more of the following: information in packet headers, information about files of file names transmitted during the message flow, or usage conditions of the message flow (such as whether the message flow involves steady or bursty transmission of data, or is relatively interactive or relatively unidirectional).

At a step **521**, the routing device **140** obtains historical information about a particular message flow **160**, and records that information in a flow data table.

At a step **522**, the routing device **140** determines a size of the flow data table, and compares that size with a selected size value. If the flow data table exceeds the selected size value, the routing device **140** continues with the step **523** to report flow data. If the flow data table does not exceed the selected size value, the routing device **140** returns to the step **521** to obtain historical information about a next particular message flow **160**.

At a step **523**, the routing device **140** builds an information packet, responsive to the information about message flows **160** which is recorded in the flow data table.

At a step **524**, the routing device **140** transmits the information packet to a selected destination device **130** on the network **100**. In a preferred embodiment, the selected destination device **130** is determined by an operating parameter of the routing device **140**. This operating parameter is set when the routing device **140** is initially configured, and may be altered by an operator of the routing device **140**.

In a preferred embodiment, the selected destination device **130** receives the information packet and builds (or updates) a database in the format for the RMON protocol. The RMON protocol is known in the art of network monitoring.

At a flow point **530**, a reporting device **540** on the network **100** is disposed for reporting using information about message flows **160**.

At a step **531**, the reporting device **540** queries the selected destination device **130** for information about message flows **160**. In a preferred embodiment, the reporting device **540** uses the RMON protocol to query the selected destination device **130** and to obtain information about message flows **160**.

At a step **532**, the reporting device **540** builds a report about a condition of the network **100**, responsive to information about message flows **160**.

At a step **533**, the reporting device **540** displays or transmits that report about the condition of the network **100** to interested parties.

In preferred embodiments, the report may comprise one or more of a wide variety of information, and interested parties

may use that information for one or more of a wide variety of purposes. Some possible purposes are noted herein:

Interested parties may diagnose actual or potential network problems. For example, the report may comprise information about packets **150** in particular message flows **160**, including a time stamp for a first packet **150** and a time stamp for a last packet **150** in the message flow **160**, a cumulative total number of bytes in the message flow **160**, a cumulative total number of packets **150** in the message flow **160**, or other information relevant to diagnosing actual or potential network problems.

Interested parties may determine patterns of usage of the network by date and time or by location. For example, the report may comprise information about which users or which services on the network are making relatively heavy use of resources. In a preferred embodiment, usage of the network **100** is displayed in a graphical form which shows use of the network **100** in a false-color map, so that network administrators and other interested parties may rapidly determine which services, which users, and which communication links are relatively loaded or relatively unloaded with demand.

Interested parties may determine which services are accessed by particular users, or which users access particular services. For example, the report may comprise information about which services are accessed by particular users at a particular device on the network **100**, or which users access a particular service at a particular device on the network **100**. This information may be used to market or otherwise enhance these services. In a preferred embodiment, users who access a particular world wide web page using the HTTP protocol are recorded, and information is sent to those users about changes to that web page and about further services available from the producers of that web page. Providers of the particular web page may also collect information about access to their web page in response to date and time of access, and location of accessing user.

Information about patterns of usage of the network, or about which services are accessed by particular users, or which users access particular services, may be used to implement accounting or billing for resources, or to set limits for resource usage, such as by particular users, by particular service providers, or by particular protocol types (and therefore by particular types of services).

Interested parties may determine usage which falls within (or without) selected parameters. These selected parameters may involve access during particular dates or times, such as for example access to particular services during or outside normal working hours. For example, it may be desirable to record those accesses to a company database which occur outside normal working hours.

These selected parameters may involve access to prohibited services, excessive access to particular services, or excessive use of network resources, such as for example access to particular servers using the HTTP protocol or the FTP protocol which fall within (or without) a particular administrative policy. For example, it may be desirable to record accesses to repositories of games or other recreational material, particularly those accesses which occur within normal working hours.

These selected parameters may involve or lack of proper access, such as for example access control list failures or unauthorized attempts to access secure services. For example, it may be desirable to record unauthorized attempts to access secure services, particularly those attempts which form a pattern which might indicate a concerted attempt to gain unauthorized access.

In alternative embodiments, the routing device **140** could save the actual packets **150** for the message flow **160**, or some part thereof, for later examination. For example, a TELNET session (a message flow **160** comprising use of the TELNET protocol by a user and a host) could be recorded in its entirety, or some portion thereof, for later examination, e.g., to diagnose problems noted with the network or with the particular host.

In further alternative embodiments, the routing device **140** could save the actual packets **150** for selected message flows **160** which meet certain selected parameters, such as repeated unauthorized attempts to gain access.

In embodiments where actual packets **150** of the message flow **160** are saved, it would be desirable to perform a name translation (such as a reverse DNS lookup), because the IP addresses for the source device **120** and the destination device **130** are transitory. Thus, it would be preferable to determine the symbolic names for the source device **120** and the destination device **130** from the IP addresses, so that the recorded data would have greater meaning at a later time.

Alternative Embodiments

Although preferred embodiments are disclosed herein, many variations are possible which remain within the concept, scope, and spirit of the invention, and these variations would become clear to those skilled in the art after perusal of this application.

We claim:

1. A method for routing messages in a data network wherein a set of packets is isolated for specialized policy treatment by a plurality of routing devices in the data network, the method comprising the steps of:

identifying a first one message of a first plurality of messages associated with an application layer, said first plurality of messages having at least one policy treatment in common, said first plurality of messages being identified in response to an address of a selected source device and an address of a selected destination device, wherein said policy treatment comprises at least one of the access control information, security information, queuing information, accounting information, traffic profiling information, and policy information;

generating a unique hash key by each of the routing devices that receives the first plurality of messages, the unique hash key being based upon the address of the selected source device, the address of the selected destination device, a port number associated with the selected source device, a port number associated with the selected destination device, and a protocol type corresponding to the first plurality of messages;

recording said first policy treatment by building a corresponding entry in a flow cache, wherein the first plurality of messages is identified by the unique hash key;

recording information about said first plurality of messages;

transmitting said information to at least one selected device on said network based upon a predetermined operating parameter;

identifying a second one message of said first plurality of messages; and

routing said second one message responsive to said first routing treatment.

2. A method as in claim 1, wherein

said first one message comprises a packet;

said first plurality of messages comprises a stream of packets associated with a selected source device and a selected destination device.

3. A method as in claim 2, wherein said stream of packets is associated with a first selected port number at said source device and a second selected port number at said destination device.

4. A method as in claim 1, wherein said first plurality of messages comprises a message flow.

5. A method as in claim 1, wherein said first plurality of messages comprises an ordered sequence, and said first one message has a selected position in said ordered sequence.

6. A method as in claim 1, wherein said step of recording comprises building an entry flow cache, wherein said flow cache includes a plurality of entries, one said entry for each said plurality of messages, each said entry including a unicast destination address.

7. A method as in claim 1, including a step of identifying a first packet of a second stream of packets, wherein the packets of said second stream of packets have at least one second policy treatment in common, said second routing treatment differing from said first policy treatment.

8. A method as in claim 1, wherein said policy treatment comprises a destination output port for routing said first message.

9. A method as in claim 1, wherein said information comprises

an arrival time for an initial one message in said plurality of messages;

an arrival time for most recent one message in said plurality of messages;

a cumulative count of bytes in said plurality of messages; or

a cumulative count of said one messages in said plurality of messages.

10. A method as in claim 1, comprising the steps of

receiving said information at said selected device on said network;

recording said information in a database at said selected device; and

making said information available to a second device on said network.

11. A system for routing packets in a data network wherein a set of packets is isolated for specialized policy treatment, said system comprising:

a source device for outputting a stream of packets;

a destination device for receiving said stream of packets; and

a plurality of routing devices for transporting said stream of packets from said source device to said destination device, each of said plurality of routing devices comprising,

means for receiving said stream of packets, said stream of packets comprising a plurality of message flows associated with an application layer, each said packet being associated with one selected message flow, each said message flow having at least one policy treatment in common, wherein said policy treatment comprises at least one of access control information, security information, queuing information, accounting information, traffic profiling information, and policy information;

means for associating packets with a first one of said message flows,

means for generating a unique hash key upon receipt of the stream of packets, the unique hash key being based upon an address of the source device, an address of the destination device, a port number associated with the source device, a port number associated with the destination device, and a protocol type corresponding to the first plurality of messages,

means for caching an entry associated with said first one of said message flows, wherein said first one of said message flows is identified by the unique hash key,

means for recording information about said first one of said message flows;

means for transmitting said information to the destination device on said network based upon a predetermined operating parameter, and

means for routing packets responsive to entries in said caching means.

12. A system as in claim 11, wherein said entry comprises access control information.

13. A system as in claim 12, wherein said entry comprises a destination output port for routing packets.

14. A system as in claim 11, wherein said information comprises

a transmission time for an initial one message in said plurality of messages;

a transmission time for a most recent one message in said plurality of messages;

a cumulative count of bytes in said plurality of messages; or

a cumulative count of said one messages in said plurality of messages.

15. The system as in claim 11,

wherein the caching means comprises a plurality of buckets, each bucket including a linked list that includes a maximum of two entries.

16. A method for routing messages in a data network wherein a set of packets is isolated for specialized policy treatment by plurality devices in the data network, said method comprising the steps of:

identifying a first one packet of a first stream of packets defining a first message flow associated with an application layer, wherein said first stream of packets comprise an ordered sequence and said first packet has a selected position in said ordered sequence, said first stream of packets having at least one first routing policy treatment in common, wherein said policy treatment comprises at least one of access control information, security information, queuing information, accounting information, traffic profiling information, and policy information; and

generating a unique hash key by each of the routing devices that receives the first stream of packets, the unique hash key being based upon an address of a selected source device, an address of a selected destination device, a port number associated with the selected source device, a port number associated with the selected destination device, and a selected protocol type, said first routing treatment being identified by the unique hash key;

recording said unique hash key by building an entry in a flow cache;

identifying subsequent packets of a said first stream of packets defining said first message flow;

recording information about said first stream of packets;

transmitting said information to at least one selected device on said network based upon a predetermined operating parameter; and

routing said subsequent packets responsive to said first policy treatment.

NO

17. A method as in claim 16, comprising the step of identifying a first one packet of a second stream of packets defining a second message flow, said second stream of packets having at least one second policy treatment in common, said second policy treatment differing from said first policy treatment.

18. A method as in claim 16, wherein said policy treatment further comprises a destination output port for routing said first one packet.

19. A method as in claim 16, wherein said information comprises

a transmission time for said first packet of said first stream of packets;

a transmission time for a most recent one packet in said first stream of packets;

a cumulative count of bytes in said first stream of packets; or

a cumulative count of packets in said first stream of packets.

* * * * *

US006651099B1

(12) **United States Patent**
Dietz et al.

(10) **Patent No.: US 6,651,099 B1**
(45) **Date of Patent: Nov. 18, 2003**

(54) **METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK**

(75) Inventors: **Russell S. Dietz**, San Jose, CA (US); **Joseph R. Maixner**, Aptos, CA (US); **Andrew A. Koppenhaver**, Littleton, CO (US); **William H. Bares**, Germantown, TN (US); **Haig A. Sarkissian**, San Antonio, TX (US); **James F. Torgerson**, Andover, MN (US)

(73) Assignee: **Hi/fn, Inc.**, Los Gatos, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 589 days.

(21) Appl. No.: **09/608,237**

(22) Filed: **Jun. 30, 2000**

Related U.S. Application Data

(60) Provisional application No. 60/141,903, filed on Jun. 30, 1999.

(51) **Int. Cl.**[7] **G06F 13/00**

(52) **U.S. Cl.** **709/224**; 370/389

(58) **Field of Search** 709/200, 201, 709/220, 223, 224, 231, 232, 236, 238, 239, 240, 246; 370/389, 392, 395.32

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,736,320	A	4/1988	Bristol	364/300
4,891,639	A	1/1990	Nakamura	340/825.5
5,101,402	A	3/1992	Chui et al.	370/17
5,247,517	A	9/1993	Ross et al.	370/85.5
5,247,693	A	9/1993	Bristol	395/800
5,249,292	A	9/1993	Chiappa	395/650
5,315,580	A	5/1994	Phaal	370/13
5,339,268	A	8/1994	Machida	365/49
5,351,243	A	9/1994	Kalkunte et al.	370/92
5,365,514	A	11/1994	Hershey et al.	370/17
5,375,070	A	12/1994	Hershey et al.	364/550
5,394,394	A	2/1995	Crowther et al.	370/60

(List continued on next page.)

OTHER PUBLICATIONS

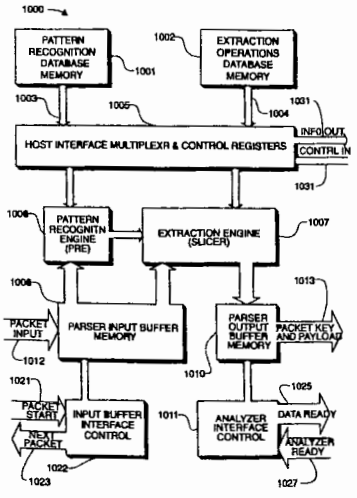
"Technical Note: the Narus System," Downloaded Apr. 29, 1999 from www.narus.com, Narus Corporation, Redwood City California.

Primary Examiner—Moustafa M. Meky
(74) *Attorney, Agent, or Firm*—Dov Rosenfeld; Inventek

(57) **ABSTRACT**

A monitor for and a method of examining packets passing through a connection point on a computer network. Each packets conforms to one or more protocols. The method includes receiving a packet from a packet acquisition device and performing one or more parsing/extraction operations on the packet to create a parser record comprising a function of selected portions of the packet. The parsing/extraction operations depend on one or more of the protocols to which the packet conforms. The method further includes looking up a flow-entry database containing flow-entries for previously encountered conversational flows. The lookup uses the selected packet portions and determining if the packet is of an existing flow. If the packet is of an existing flow, the method classifies the packet as belonging to the found existing flow, and if the packet is of a new flow, the method stores a new flow-entry for the new flow in the flow-entry database, including identifying information for future packets to be identified with the new flow entry. For the packet of an existing flow, the method updates the flow-entry of the existing flow. Such updating may include storing one or more statistical measures. Any stage of a flow, state is maintained, and the method performs any state processing for an identified state to further the process of identifying the flow. The method thus examines each and every packet passing through the connection point in real time until the application program associated with the conversational flow is determined.

10 Claims, 18 Drawing Sheets



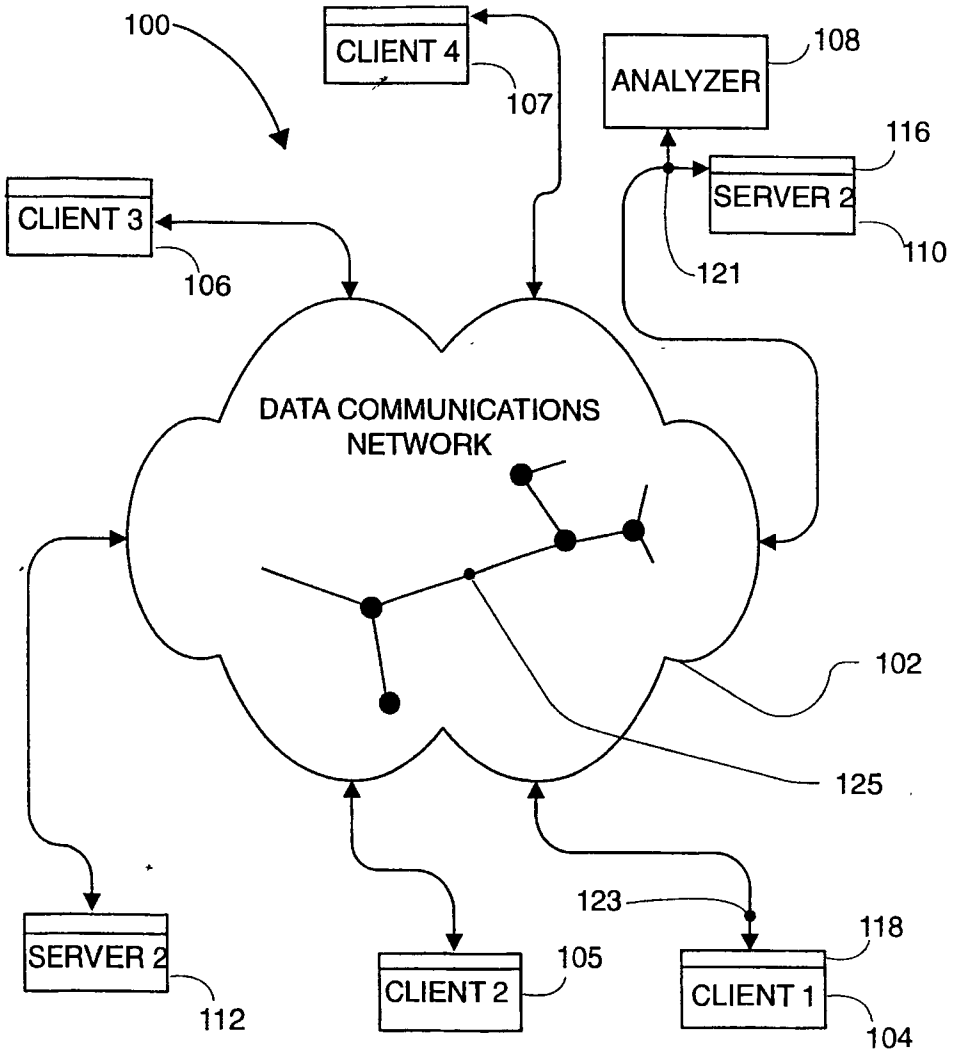

100
CLIENT 4
107
ANALYZER
108
116
SERVER 2
110
121
CLIENT 3
106
DATA COMMUNICATIONS NETWORK
102
125
123
118
SERVER 2
112
CLIENT 2
105
CLIENT 1
104

FIG. 1

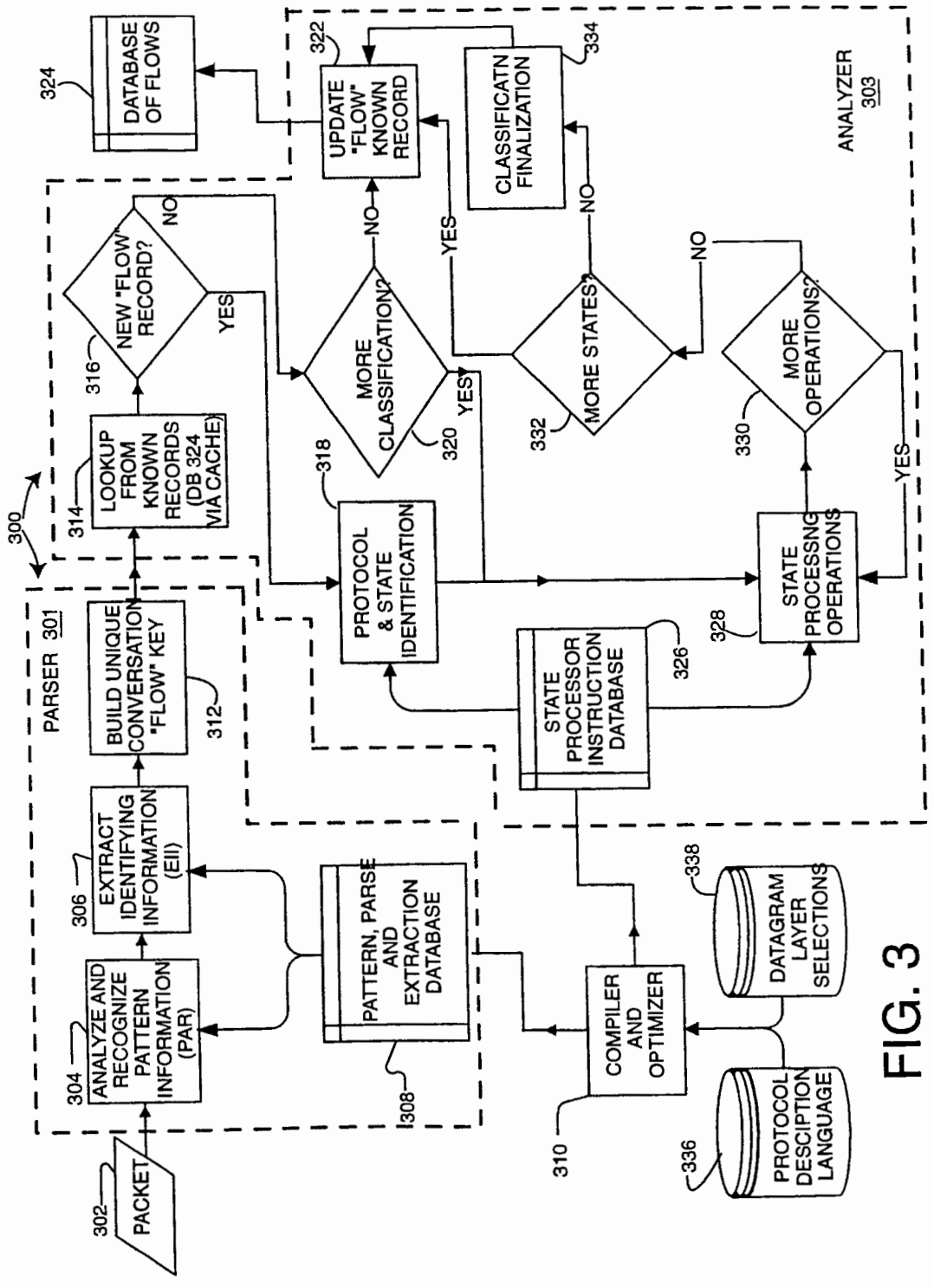

PACKET
302
PARSER 301
ANALYZE AND RECOGNIZE PATTERN INFORMATION (PAR)
304
EXTRACT IDENTIFYING INFORMATION (EII)
306
BUILD UNIQUE CONVERSATION "FLOW" KEY
312
PATTERN, PARSE AND EXTRACTION DATABASE
308
COMPILER AND OPTIMIZER
310
PROTOCOL DESCIPTION LANGUAGE
336
DATAGRAM LAYER SELECTIONS
338
ANALYZER 303
LOOKUP FROM KNOWN RECORDS (DB 324 VIA CACHE)
314
NEW "FLOW" RECORD?
316
NO
YES
PROTOCOL & STATE IDENTIFICATION
318
MORE CLASSIFICATION?
320
YES
NO
UPDATE "FLOW" KNOWN RECORD
322
DATABASE OF FLOWS
324
STATE PROCESSOR INSTRUCTION DATABASE
326
STATE PROCESSNG OPERATIONS
328
MORE OPERATIONS?
330
YES
NO
MORE STATES?
332
NO
CLASSIFICATN FINALIZATION
334
YES
NO
300

FIG. 3

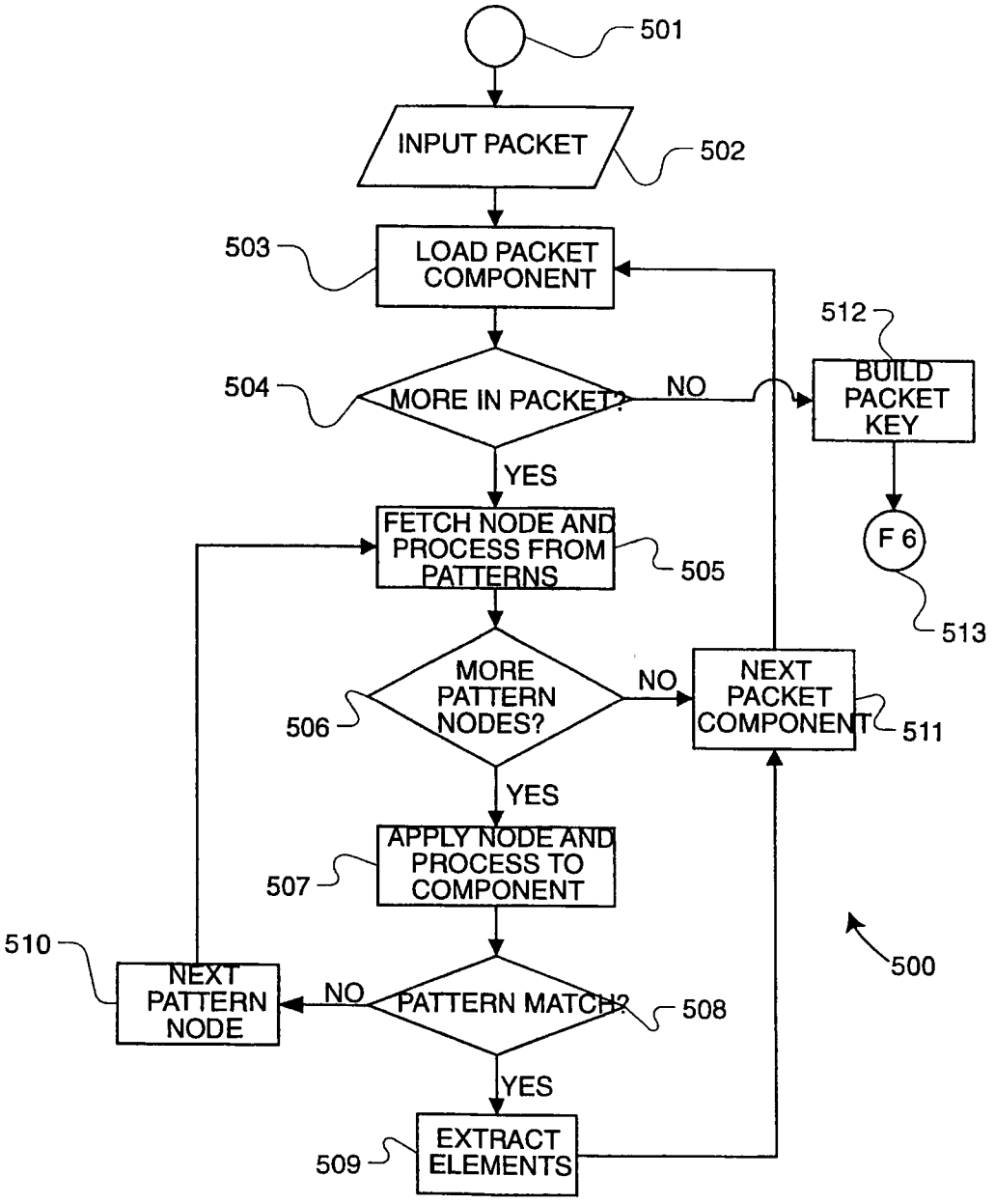

501
INPUT PACKET
502
503
LOAD PACKET COMPONENT
504
MORE IN PACKET?
NO
512
BUILD PACKET KEY
F 6
513
YES
FETCH NODE AND PROCESS FROM PATTERNS
505
MORE PATTERN NODES?
506
NO
NEXT PACKET COMPONENT
511
YES
APPLY NODE AND PROCESS TO COMPONENT
507
510
NEXT PATTERN NODE
NO
PATTERN MATCH?
508
500
YES
EXTRACT ELEMENTS
509

FIG. 5

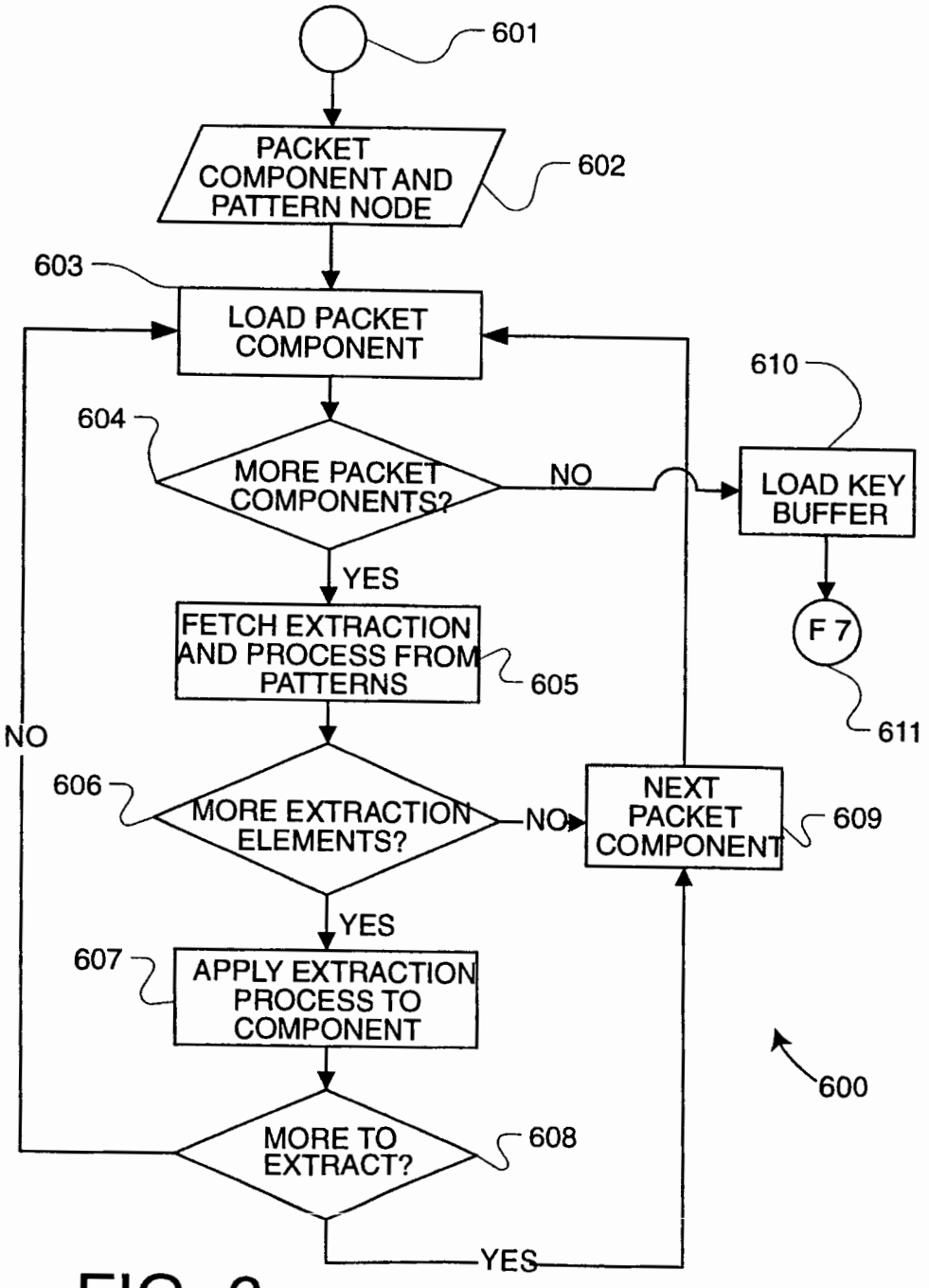

601
PACKET COMPONENT AND PATTERN NODE
602
603
LOAD PACKET COMPONENT
604
MORE PACKET COMPONENTS?
NO
610
LOAD KEY BUFFER
F 7
611
YES
FETCH EXTRACTION AND PROCESS FROM PATTERNS
605
NO
606
MORE EXTRACTION ELEMENTS?
NO
NEXT PACKET COMPONENT
609
YES
607
APPLY EXTRACTION PROCESS TO COMPONENT
600
MORE TO EXTRACT?
608
YES

FIG. 6

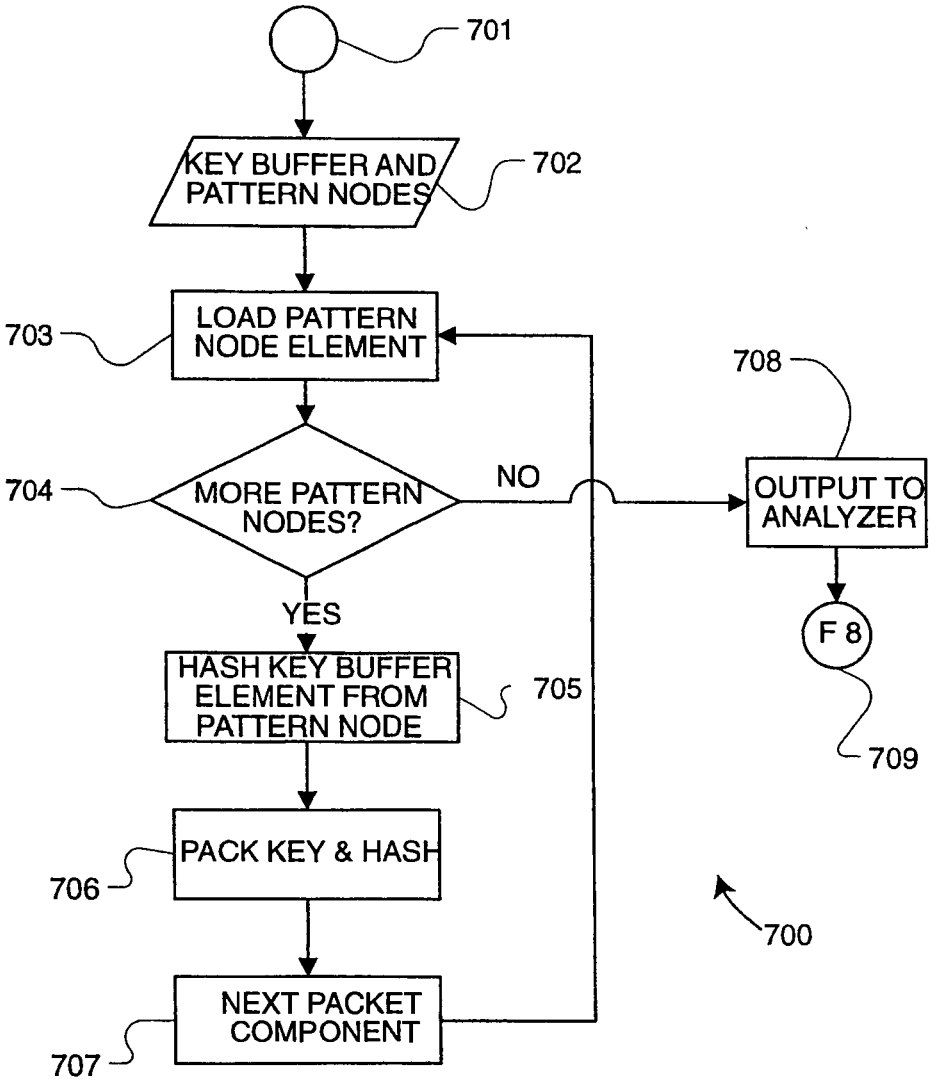

701
KEY BUFFER AND PATTERN NODES
702
LOAD PATTERN NODE ELEMENT
703
MORE PATTERN NODES?
704
NO
YES
OUTPUT TO ANALYZER
708
F 8
709
HASH KEY BUFFER ELEMENT FROM PATTERN NODE
705
PACK KEY & HASH
706
NEXT PACKET COMPONENT
707
700

FIG. 7

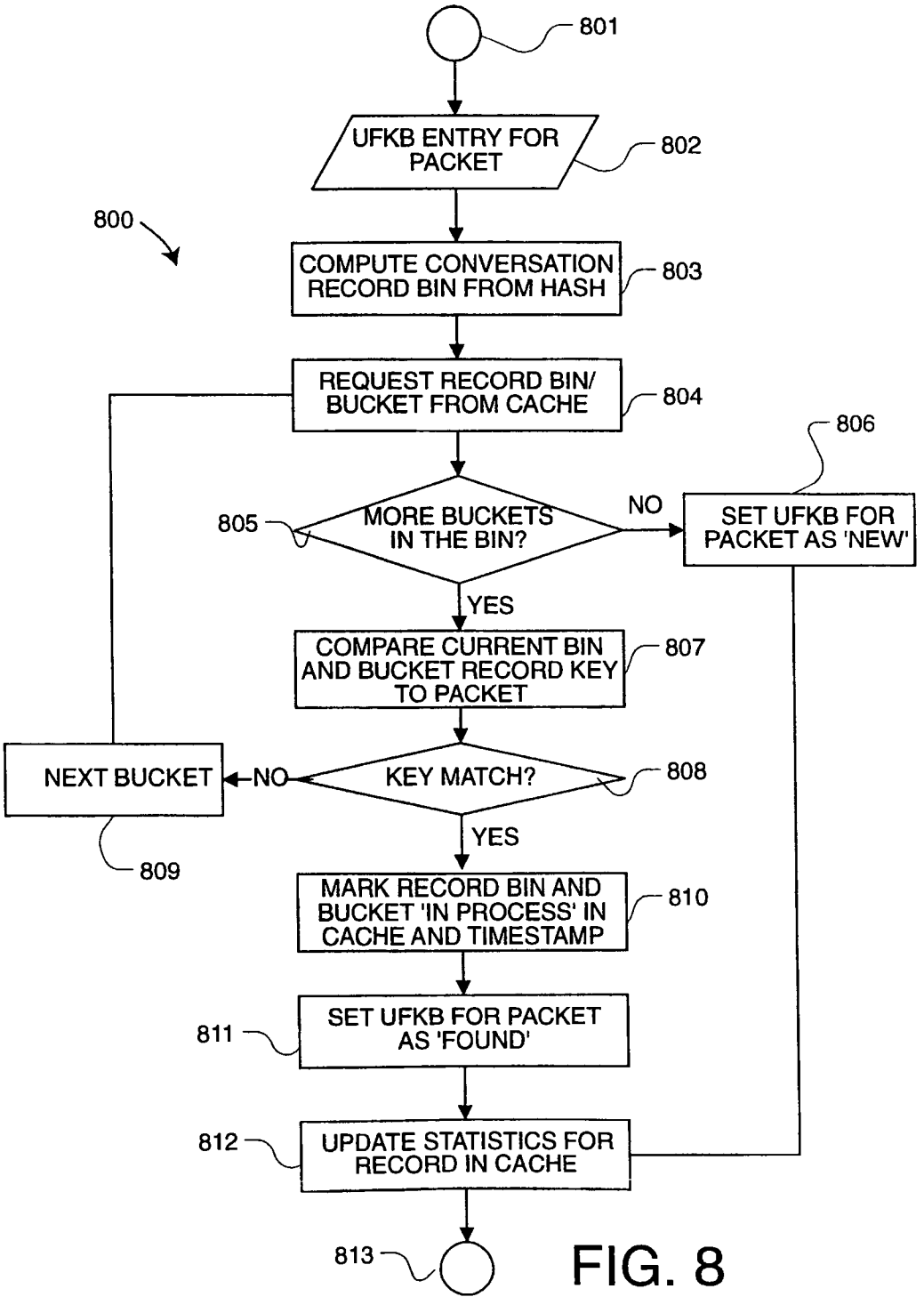

801
UFKB ENTRY FOR PACKET
802
800
COMPUTE CONVERSATION RECORD BIN FROM HASH
803
REQUEST RECORD BIN/ BUCKET FROM CACHE
804
806
805
MORE BUCKETS IN THE BIN?
NO
SET UFKB FOR PACKET AS 'NEW'
YES
COMPARE CURRENT BIN AND BUCKET RECORD KEY TO PACKET
807
NEXT BUCKET
NO
KEY MATCH?
808
YES
809
MARK RECORD BIN AND BUCKET 'IN PROCESS' IN CACHE AND TIMESTAMP
810
811
SET UFKB FOR PACKET AS 'FOUND'
812
UPDATE STATISTICS FOR RECORD IN CACHE
813

FIG. 8

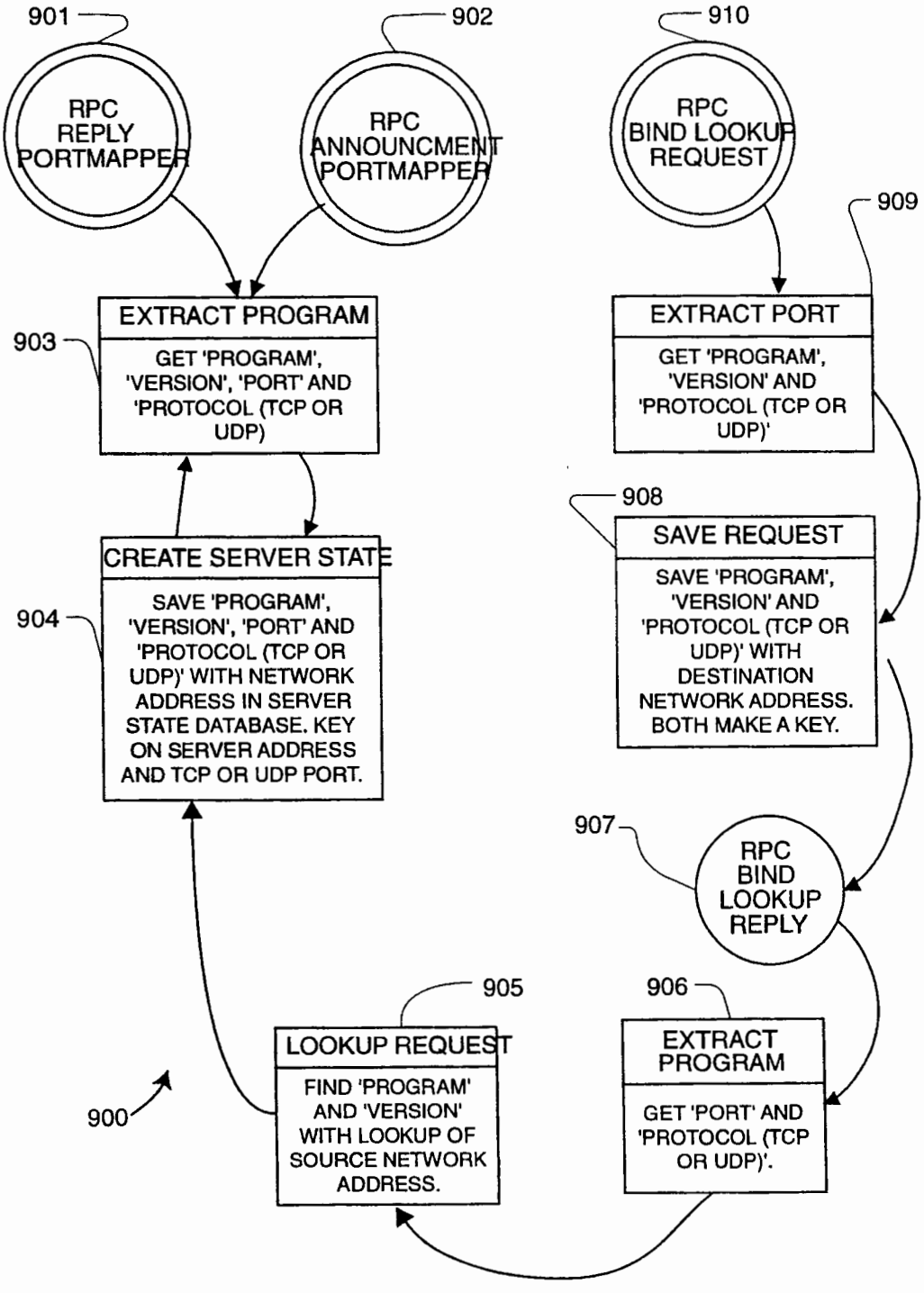

901
RPC REPLY PORTMAPPER
902
RPC ANNOUNCMENT PORTMAPPER
910
RPC BIND LOOKUP REQUEST
909
EXTRACT PROGRAM
903
GET 'PROGRAM', 'VERSION', 'PORT' AND 'PROTOCOL (TCP OR UDP)
EXTRACT PORT
GET 'PROGRAM', 'VERSION' AND 'PROTOCOL (TCP OR UDP)'
908
SAVE REQUEST
SAVE 'PROGRAM', 'VERSION' AND 'PROTOCOL (TCP OR UDP)' WITH DESTINATION NETWORK ADDRESS. BOTH MAKE A KEY.
CREATE SERVER STATE
904
SAVE 'PROGRAM', 'VERSION', 'PORT' AND 'PROTOCOL (TCP OR UDP)' WITH NETWORK ADDRESS IN SERVER STATE DATABASE. KEY ON SERVER ADDRESS AND TCP OR UDP PORT.
907
RPC BIND LOOKUP REPLY
905
906
LOOKUP REQUEST
FIND 'PROGRAM' AND 'VERSION' WITH LOOKUP OF SOURCE NETWORK ADDRESS.
EXTRACT PROGRAM
GET 'PORT' AND 'PROTOCOL (TCP OR UDP)'.
900

FIG. 9

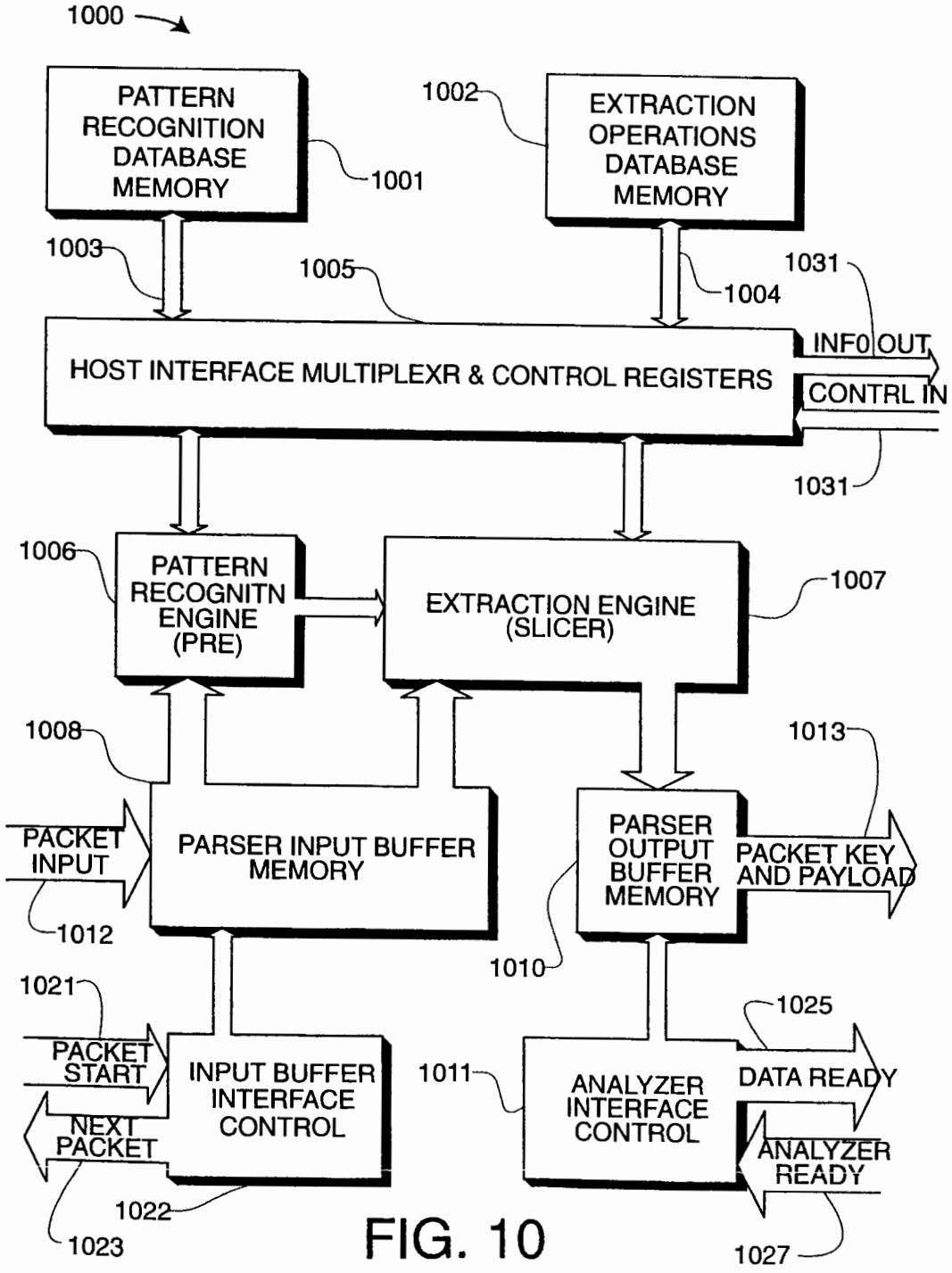

1000
PATTERN RECOGNITION DATABASE MEMORY
1001
1002
EXTRACTION OPERATIONS DATABASE MEMORY
1003
1005
1004
1031
HOST INTERFACE MULTIPLEXR & CONTROL REGISTERS
INFO OUT
CONTRL IN
1031
1006
PATTERN RECOGNITN ENGINE (PRE)
EXTRACTION ENGINE (SLICER)
1007
1008
1013
PACKET INPUT
PARSER INPUT BUFFER MEMORY
PARSER OUTPUT BUFFER MEMORY
PACKET KEY AND PAYLOAD
1012
1010
1021
1025
PACKET START
INPUT BUFFER INTERFACE CONTROL
1011
ANALYZER INTERFACE CONTROL
DATA READY
NEXT PACKET
ANALYZER READY
1022
1023
1027

FIG. 10

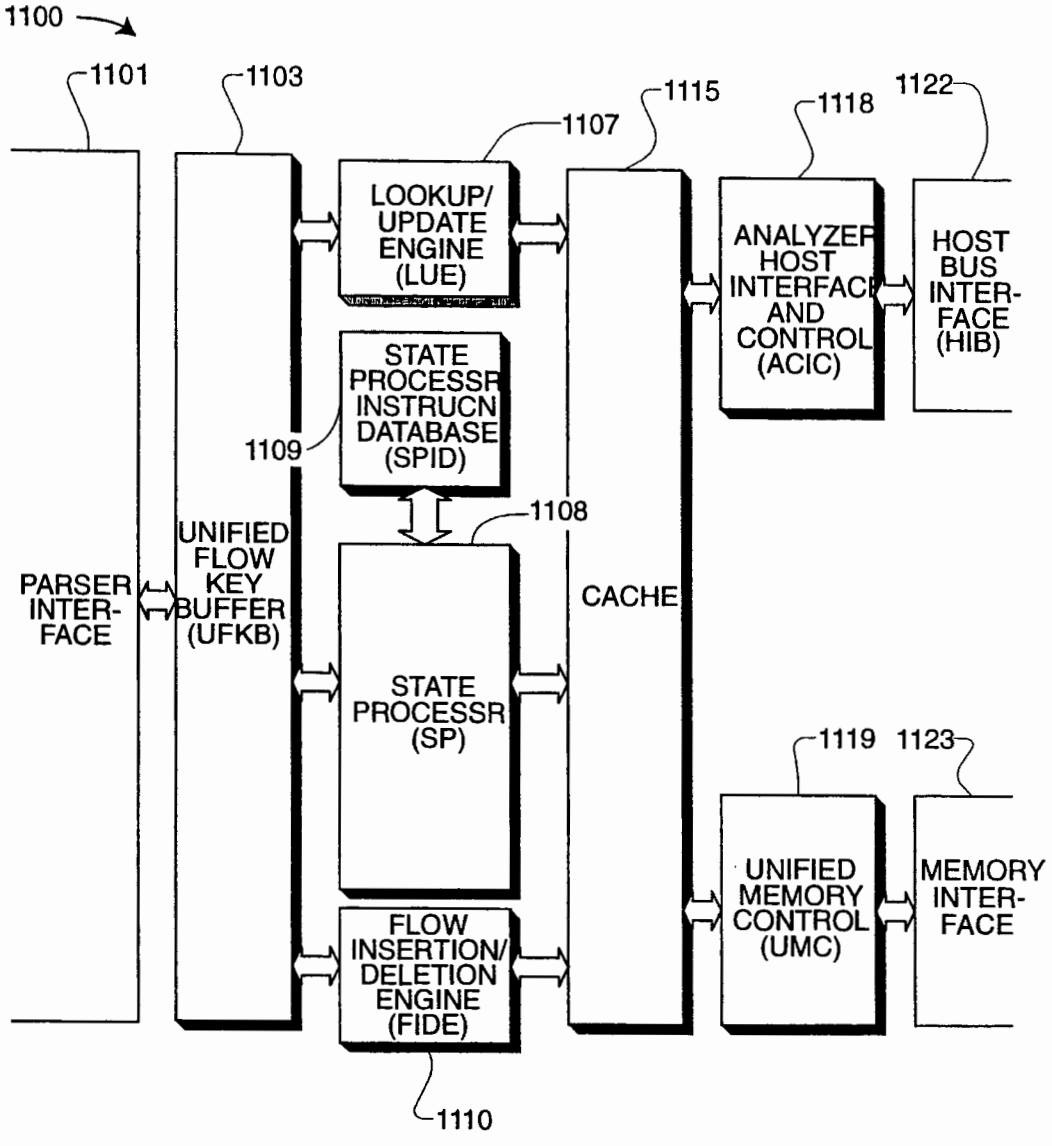

1100
1101
1103
1107
1115
1118
1122
LOOKUP/ UPDATE ENGINE (LUE)
ANALYZER HOST INTERFACE AND CONTROL (ACIC)
HOST BUS INTER-FACE (HIB)
STATE PROCESSR INSTRUCN DATABASE (SPID)
1109
1108
PARSER INTER-FACE
UNIFIED FLOW KEY BUFFER (UFKB)
CACHE
STATE PROCESSR (SP)
1119
1123
UNIFIED MEMORY CONTROL (UMC)
MEMORY INTER-FACE
FLOW INSERTION/ DELETION ENGINE (FIDE)
1110

FIG. 11

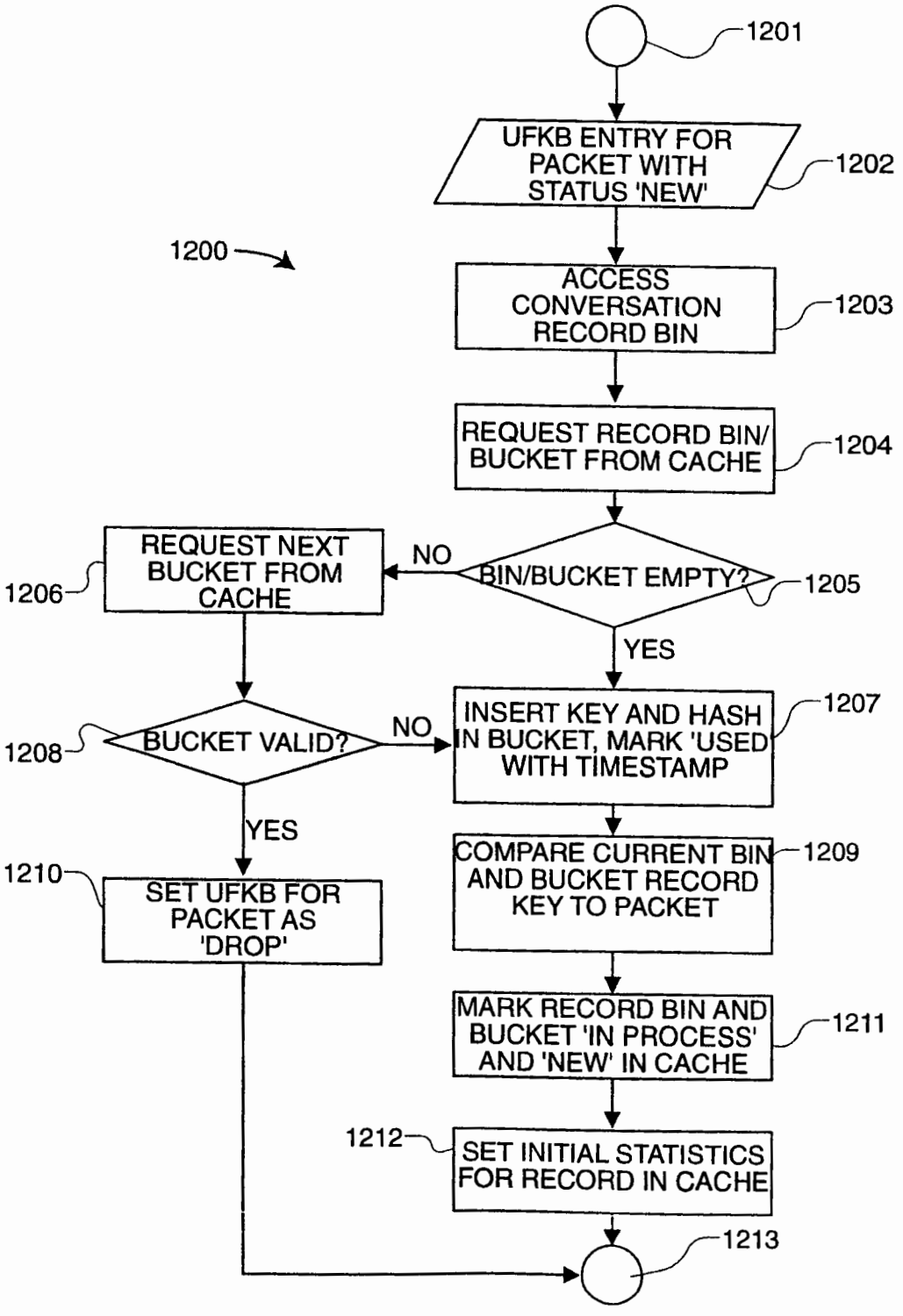

1201
UFKB ENTRY FOR PACKET WITH STATUS 'NEW'
1202
1200
ACCESS CONVERSATION RECORD BIN
1203
REQUEST RECORD BIN/ BUCKET FROM CACHE
1204
REQUEST NEXT BUCKET FROM CACHE
1206
NO
BIN/BUCKET EMPTY?
1205
YES
BUCKET VALID?
1208
NO
INSERT KEY AND HASH IN BUCKET, MARK 'USED WITH TIMESTAMP
1207
YES
COMPARE CURRENT BIN AND BUCKET RECORD KEY TO PACKET
1209
1210
SET UFKB FOR PACKET AS 'DROP'
MARK RECORD BIN AND BUCKET 'IN PROCESS' AND 'NEW' IN CACHE
1211
1212
SET INITIAL STATISTICS FOR RECORD IN CACHE
1213

FIG. 12

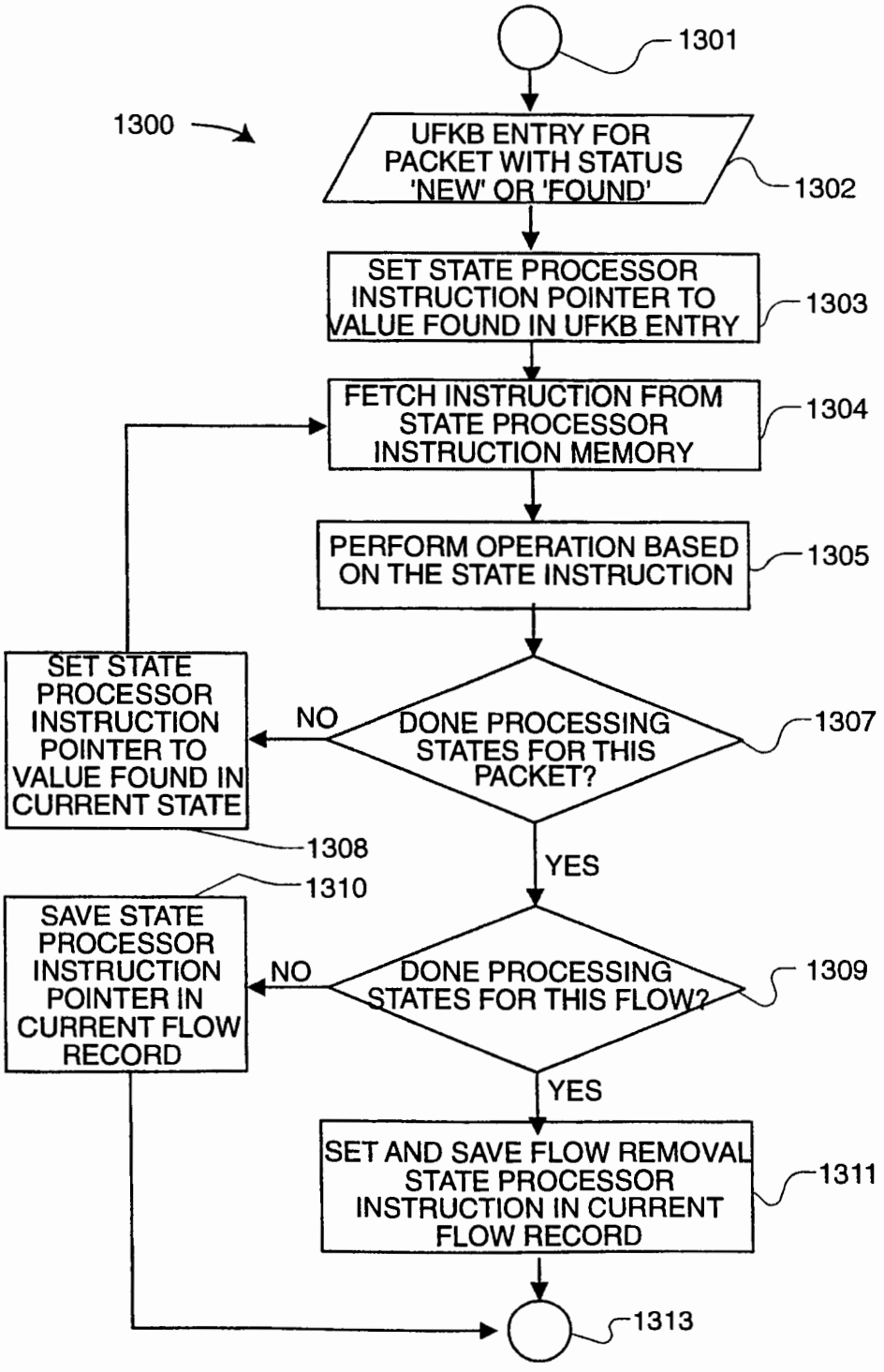

1301
1300
UFKB ENTRY FOR PACKET WITH STATUS 'NEW' OR 'FOUND'
1302
SET STATE PROCESSOR INSTRUCTION POINTER TO VALUE FOUND IN UFKB ENTRY
1303
FETCH INSTRUCTION FROM STATE PROCESSOR INSTRUCTION MEMORY
1304
PERFORM OPERATION BASED ON THE STATE INSTRUCTION
1305
DONE PROCESSING STATES FOR THIS PACKET?
1307
NO
SET STATE PROCESSOR INSTRUCTION POINTER TO VALUE FOUND IN CURRENT STATE
1308
YES
DONE PROCESSING STATES FOR THIS FLOW?
1309
NO
SAVE STATE PROCESSOR INSTRUCTION POINTER IN CURRENT FLOW RECORD
1310
YES
SET AND SAVE FLOW REMOVAL STATE PROCESSOR INSTRUCTION IN CURRENT FLOW RECORD
1311
1313

FIG. 13

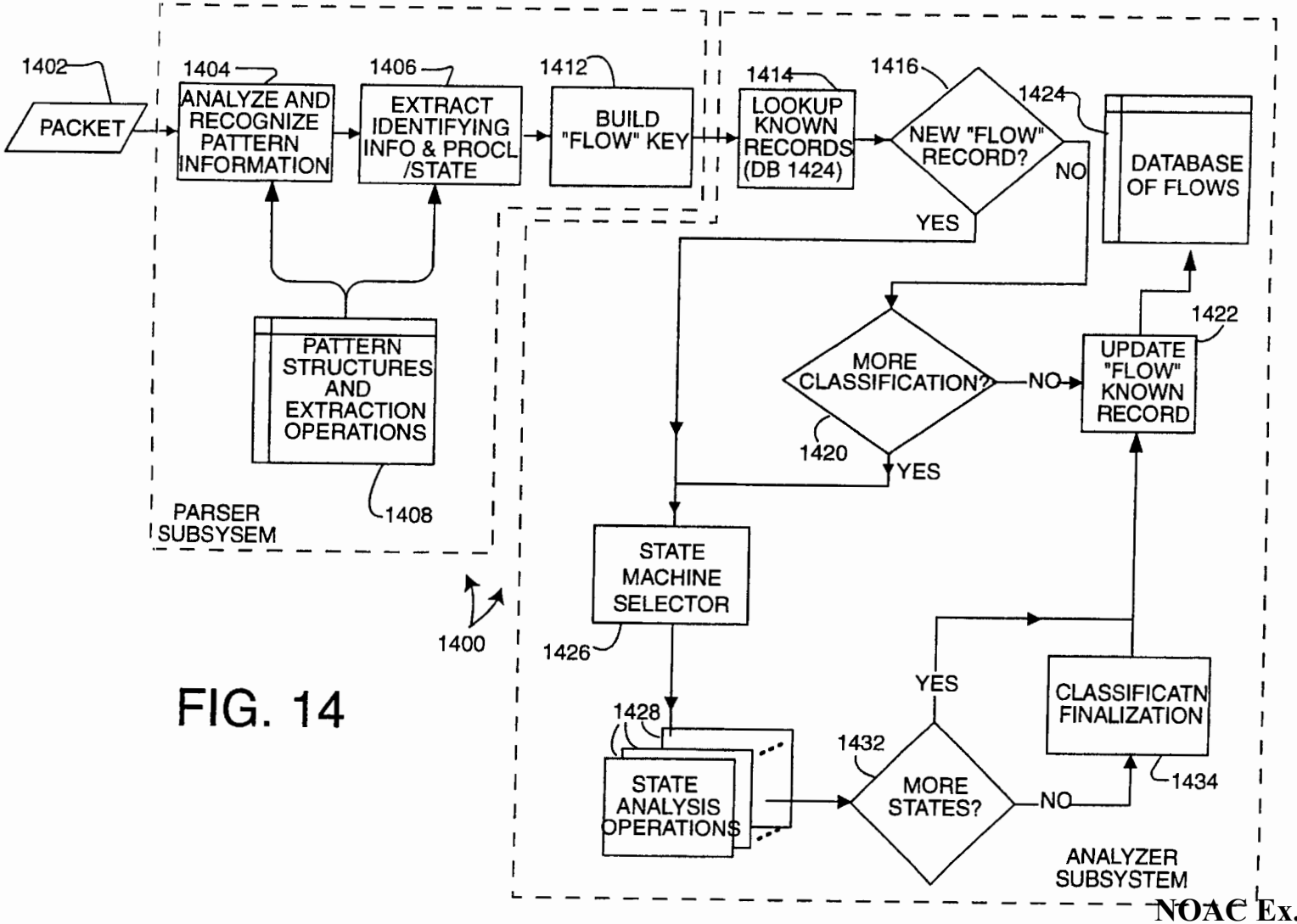

1402
PACKET
1404
ANALYZE AND RECOGNIZE PATTERN INFORMATION
1406
EXTRACT IDENTIFYING INFO & PROCL /STATE
1412
BUILD "FLOW" KEY
1414
LOOKUP KNOWN RECORDS (DB 1424)
1416
NEW "FLOW" RECORD?
NO
YES
1424
DATABASE OF FLOWS
PATTERN STRUCTURES AND EXTRACTION OPERATIONS
1408
PARSER SUBSYSEM
MORE CLASSIFICATION?
NO
YES
1420
1422
UPDATE "FLOW" KNOWN RECORD
STATE MACHINE SELECTOR
1426
1400
1428
STATE ANALYSIS OPERATIONS
1432
MORE STATES?
YES
NO
CLASSIFICATN FINALIZATION
1434
ANALYZER SUBSYSTEM

FIG. 14

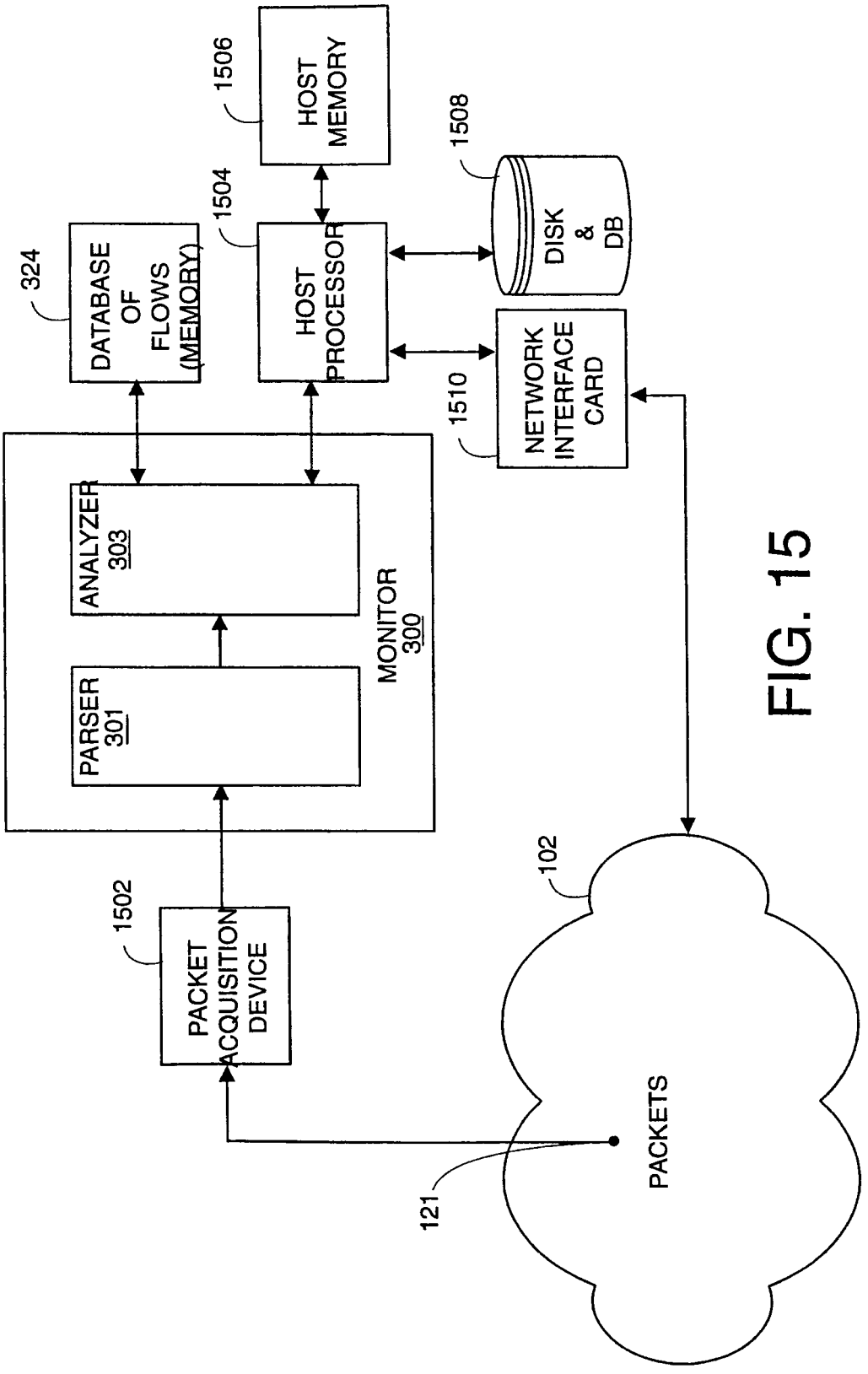

HOST MEMORY
1506
1508
DISK & DB
1504
HOST PROCESSOR
324
DATABASE OF FLOWS (MEMORY)
NETWORK INTERFACE CARD
1510
ANALYZER 303
MONITOR 300
PARSER 301
1502
PACKET ACQUISITION DEVICE
102
PACKETS
121

FIG. 15

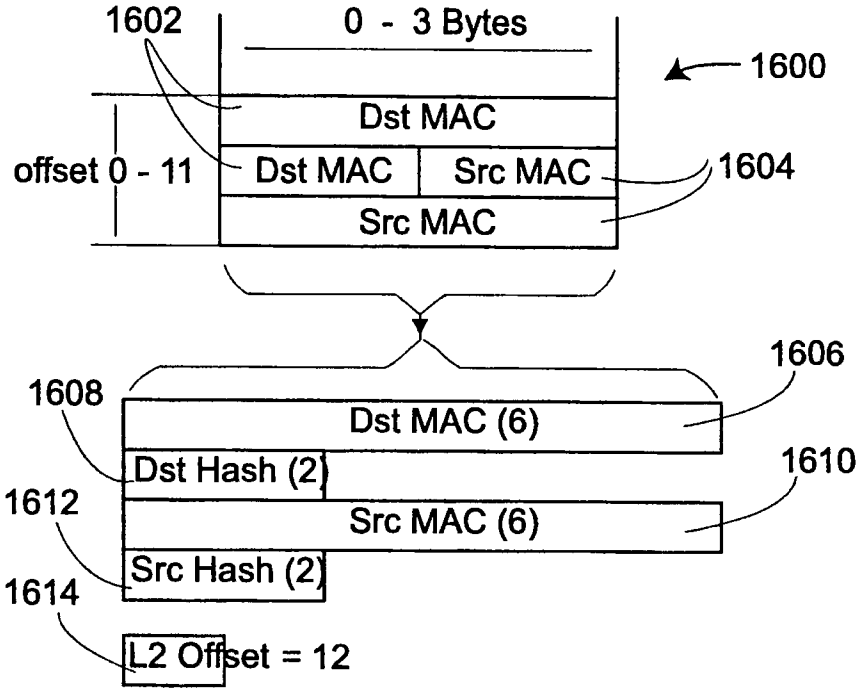

1602
0 - 3 Bytes
1600
Dst MAC
offset 0 - 11
Dst MAC
Src MAC
1604
Src MAC
1606
1608
Dst MAC (6)
Dst Hash (2)
1610
1612
Src MAC (6)
Src Hash (2)
1614
L2 Offset = 12

FIG. 16

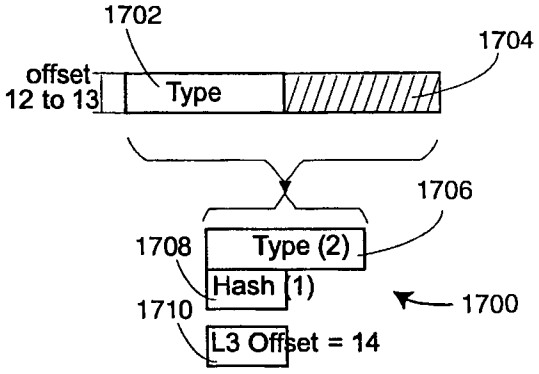


FIG. 17A

1712

IDP = 0x0600*
IP = 0x0800*
CHAOSNET = 0x0804
ARP = 0x0806
VIP = 0x0BAD*
VLOOP = 0x0BAE
VECHO = 0x0BAF
NETBIOS-3COM = 0x3C00 - 0x3C0D#
DEC-MOP = 0x6001
DEC-RC = 0x6002
DEC-DRP = 0x6003*
DEC-LAT = 0x6004
DEC-DIAG = 0x6005
DEC-LAVC = 0x6007
RARP = 0x8035
ATALK = 0x809B*
VLOOP = 0x80C4
VECHO = 0x80C5
SNA-TH = 0x80D5*
ATALKARP = 0x80F3
IPX = 0x8137*
SNMP = 0x814C#
IPv6 = 0x86DD*
LOOPBACK = 0x9000

Apple = 0x080007

* L3 Decoding
# L5 Decoding

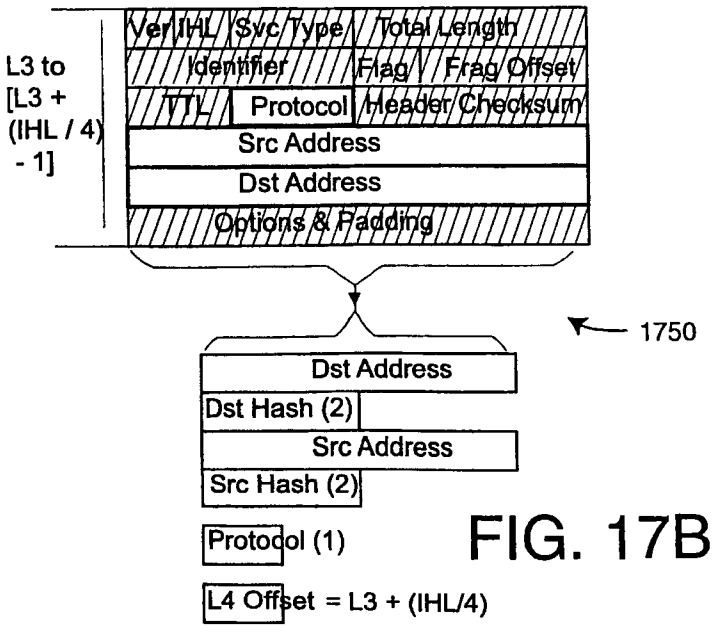


FIG. 17B

1752

ICMP = 1
IGMP = 2
GGP = 3
TCP = 6*
EGP = 8
IGRP = 9
PUP = 12
CHAOS = 16
UDP = 17*
IDP = 22#
ISO-TP4 = 29
DDP = 37#
ISO-IP = 80
VIP = 83#
EIGRP = 88
OSPF = 89

* L4 Decoding
# L3 Re-Decoding

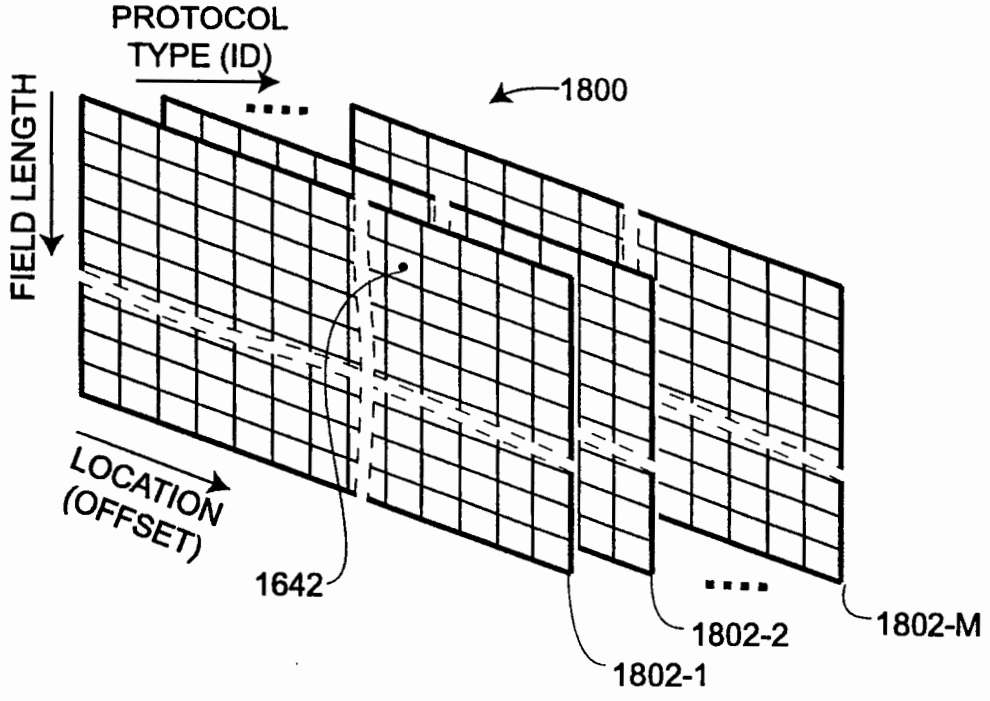


FIG. 18A

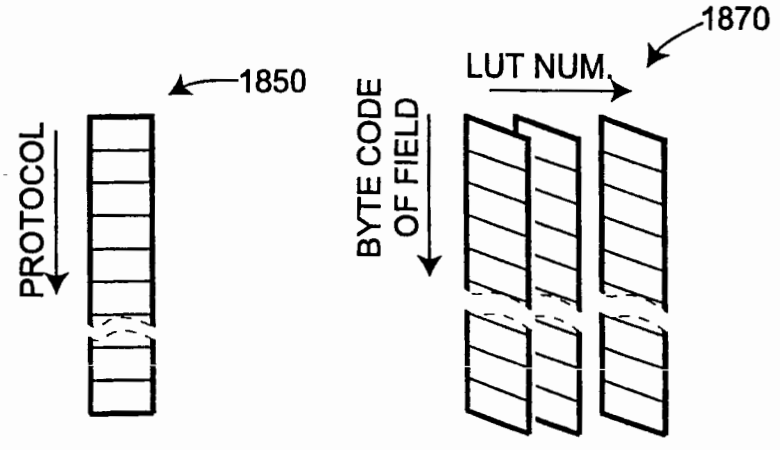


FIG. 18B

METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK

CROSS-REFERENCE TO RELATED APPLICATION

This application claims the benefit of U.S. Provisional Patent Application Ser. No.: 60/141,903 for METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK to inventors Dietz, et al., filed Jun. 30, 1999, the contents of which are incorporated herein by reference.

This application is related to the following U.S. patent applications, each filed concurrently with the present application, and each assigned to Apptitude, Inc., the assignee of the present invention:

U.S. patent application Ser. No. 09/609,179 for PROCESSING PROTOCOL SPECIFIC INFORMATION IN PACKETS SPECIFIED BY A PROTOCOL DESCRIPTION LANGUAGE, to inventors Koppenhaver, et al., filed Jun. 30, 2000, still pending, and incorporated herein by reference. U.S. patent application Ser. No. 09/608,126 for RE-USING INFORMATION FROM DATA TRANSACTIONS FOR MAINTAINING STATISTICS IN NETWORK MONITORING, to inventors Dietz, et al., filed Jun. 30, 2000, still pending, and incorporated herein by reference. U.S. patent application Ser. No. 09/608,266 for ASSOCIATIVE CACHE STRUCTURE FOR LOOKUPS AND UPDATES OF FLOW RECORDS IN A NETWORK MONITOR, to inventors Sarkissian, et al., filed Jun. 30, 2000, still penting, and incorporated herein by reference. U.S. patent application Ser. No. 09/608,267 for STATE PROCESSOR FOR PATTERN MATCHING IN A NETWORK MONITOR DEVICE, to inventors Sarkissian, et al., filed Jun. 30, 2000, still pending, and incorporated herein by reference.

FIELD OF INVENTION

The present invention relates to computer networks, specifically to the real-time elucidation of packets communicated within a data network, including classification according to protocol and application program.

BACKGROUND TO THE PRESENT INVENTION

There has long been a need for network activity monitors. This need has become especially acute, however, given the recent popularity of the Internet and other internets—an "internet" being any plurality of interconnected networks which forms a larger, single network. With the growth of networks used as a collection of clients obtaining services from one or more servers on the network, it is increasingly important to be able to monitor the use of those services and to rate them accordingly. Such objective information, for example, as which services (i.e., application programs) are being used, who is using them, how often they have been accessed, and for how long, is very useful in the maintenance and continued operation of these networks. It is especially important that selected users be able to access a network remotely in order to generate reports on network use in real time. Similarly, a need exists for a real-time network monitor that can provide alarms notifying selected users of problems that may occur with the network or site.

One prior art monitoring method uses log files. In this method, selected network activities may be analyzed retrospectively by reviewing log files, which are maintained by

network servers and gateways. Log file monitors must access this data and analyze ("mine") its contents to determine statistics about the server or gateway. Several problems exist with this method, however. First, log file information does not provide a map of real-time usage; and secondly, log file mining does not supply complete information. This method relies on logs maintained by numerous network devices and servers, which requires that the information be subjected to refining and correlation. Also, sometimes information is simply not available to any gateway or server in order to make a log file entry.

One such case, for example, would be information concerning NetMeeting® (Microsoft Corporation, Redmond, Washington) sessions in which two computers connect directly on the network and the data is never seen by a server or a gateway.

Another disadvantage of creating log files is that the process requires data logging features of network elements to be enabled, placing a substantial load on the device, which results in a subsequent decline in network performance. Additionally, log files can grow rapidly, there is no standard means of storage for them, and they require a significant amount of maintenance.

Though Netflow® (Cisco Systems, Inc., San Jose, Calif.), RMON2, and other network monitors are available for the real-time monitoring of networks, they lack visibility into application content and are typically limited to providing network layer level information.

Pattern-matching parser techniques wherein a packet is parsed and pattern filters are applied are also known, but these too are limited in how deep into the protocol stack they can examine packets.

Some prior art packet monitors classify packets into connection flows. The term "connection flow" is commonly used to describe all the packets involved with a single connection. A conversational flow, on the other hand, is the sequence of packets that are exchanged in any direction as a result of an activity—for instance, the running of an application on a server as requested by a client. It is desirable to be able to identify and classify conversational flows rather than only connection flows. The reason for this is that some conversational flows involve more than one connection, and some even involve more than one exchange of packets between a client and server. This is particularly true when using client/server protocols such as RPC, DCOMP, and SAP, which enable a service to be set up or defined prior to any use of that service.

An example of such a case is the SAP (Service Advertising Protocol), a NetWare (Novell Systems, Provo, Utah) protocol used to identify the services and addresses of servers attached to a network. In the initial exchange, a client might send a SAP request to a server for print service. The server would then send a SAP reply that identifies a particular address—for example, SAP#5—as the print service on that server. Such responses might be used to update a table in a router, for instance, known as a Server Information Table. A client who has inadvertently seen this reply or who has access to the table (via the router that has the Service Information Table) would know that SAP#5 for this particular server is a print service. Therefore, in order to print data on the server, such a client would not need to make a request for a print service, but would simply send data to be printed specifying SAP#5. Like the previous exchange, the transmission of data to be printed also involves an exchange between a client and a server, but requires a second connection and is therefore independent of the initial exchange.

In order to eliminate the possibility of disjointed conversational exchanges, it is desirable for a network packet monitor to be able to "virtually concatenate"—that is, to link—the first exchange with the second. If the clients were the same, the two packet exchanges would then be correctly identified as being part of the same conversational flow.

Other protocols that may lead to disjointed flows, include RPC (Remote Procedure Call); DCOM (Distributed Component Object Model), formerly called Network OLE (Microsoft Corporation, Redmond, Wash.); and CORBA (Common Object Request Broker Architecture). RPC is a programming interface from Sun Microsystems (Palo Alto, Calif.) that allows one program to use the services of another program in a lo remote machine. DCOM, Microsoft's counterpart to CORBA, defines the remote procedure call that allows those objects—objects are self-contained software modules—to be run remotely over the network. And CORBA, a standard from the Object Management Group (OMG) for communicating between distributed objects, provides a way to execute programs (objects) written in different programming languages running on different platforms regardless of where they reside in a network.

What is needed, therefore, is a network monitor that makes it possible to continuously analyze all user sessions on a heavily trafficked network. Such a monitor should enable non-intrusive, remote detection, characterization, analysis, and capture of all information passing through any point on the network (i.e., of all packets and packet streams passing through any location in the network). Not only should all the packets be detected and analyzed, but for each of these packets the network monitor should determine the protocol (e.g., http, ftp, H.323, VPN, etc.), the application/use within the protocol (e.g., voice, video, data, real-time data, etc.), and an end user's pattern of use within each application or the application context (e.g., options selected, service delivered, duration, time of day, data requested, etc.). Also, the network monitor should not be reliant upon server resident information such as log files. Rather, it should allow a user such as a network administrator or an Internet service provider (ISP) the means to measure and analyze network activity objectively; to customize the type of data that is collected and analyzed; to undertake real time analysis; and to receive timely notification of network problems.

Considering the previous SAP example again, because one features of the invention is to correctly identify the second exchange as being associated with a print service on that server, such exchange would even be recognized if the clients were not the same. What distinguishes this invention from prior art network monitors is that it has the ability to recognize disjointed flows as belonging to the same conversational flow.

The data value in monitoring network communications has been recognized by many inventors. Chiu, et al., describe a method for collecting information at the session level in a computer network in U.S. Pat. No. 5,101,402, titled "APPARATUS AND METHOD FOR REAL-TIME MONITORING OF NETWORK SESSIONS AND A LOCAL AREA NETWORK" (the "402 patent"). The 402 patent specifies fixed locations for particular types of packets to extract information to identify session of a packet. For example, if a DECnet packet appears, the 402 patent looks at six specific fields (at 6 locations) in the packet in order to identify the session of the packet. If, on the other hand, an IP packet appears, a different set of six different locations is specified for an IP packet. With the proliferation of protocols, clearly the specifying of all the possible places to look to determine the session becomes more and more difficult. Likewise, adding a new protocol or application is difficult. In the present invention, the locations examined and the information extracted from any packet are adaptively determined from information in the packet for the particular type of packet. There is no fixed definition of what to look for and where to look in order to form an identifying signature. A monitor implementation of the present invention, for example, adapts to handle differently IEEE 802.3 packet from the older Ethernet Type 2 (or Version 2) DIX (Digital-Intel-Xerox) packet.

The 402 patent system is able to recognize up to the session layer. In the present invention, the number of levels examined varies for any particular protocol. Furthermore, the present invention is capable of examining up to whatever level is sufficient to uniquely identify to a required level, even all the way to the application level (in the OSI model).

Other prior art systems also are known. Phael describes a network activity monitor that processes only randomly selected packets in U.S. Pat. No. 5,315,580, titled "NETWORK MONITORING DEVICE AND SYSTEM." Nakamura teaches a network monitoring system in U.S. Pat. No. 4,891,639, titled "MONITORING SYSTEM OF NETWORK." Ross, et al., teach a method and apparatus for analyzing and monitoring network activity in U.S. Pat. No. 5,247,517, titled "METHOD AND APPARATUS FOR ANALYSIS NETWORKS," McCreery, et al., describe an Internet activity monitor that decodes packet data at the Internet protocol level layer in U.S. Pat. No. 5,787,253, titled "APPARATUS AND METHOD OF ANALYZING INTERNET ACTIVITY." The McCreery method decodes IP-packets. It goes through the decoding operations for each packet, and therefore uses the processing overhead for both recognized and unrecognized flows. In a monitor implementation of the present invention, a signature is built for every flow such that future packets of the flow are easily recognized. When a new packet in the flow arrives, the recognition process can commence from where it last left off, and a new signature built to recognize new packets of the flow.

SUMMARY

In its various embodiments the present invention provides a network monitor that can accomplish one or more of the following objects and advantages:

- Recognize and classify all packets that are exchanges between a client and server into respective client/server applications.
- Recognize and classify at all protocol layer levels conversational flows that pass in either direction at a point in a network.
- Determine the connection and flow progress between clients and servers according to the individual packets exchanged over a network.
- Be used to help tune the performance of a network according to the current mix of client/server applications requiring network resources.
- Maintain statistics relevant to the mix of client/server applications using network resources.
- Report on the occurrences of specific sequences of packets used by particular applications for client/server network conversational flows.

Other aspects of embodiments of the invention are:

- Properly analyzing each of the packets exchanged between a client and a server and maintaining information relevant to the current state of each of these conversational flows. p1 providing a flexible process-

NOAC Ex. 1015 Page 642

ing system that can be tailored or adapted as new applications enter the client/server market.

Maintaining statistics relevant to the conversational flows in a client/sever network as classified by an individual application.

Reporting a specific identifier, which may be used by other network-oriented devices to identify the series of packets with a specific application for a specific client/server network conversational flow.

In general, the embodiments of the present invention overcome the problems and disadvantages of the art.

As described herein, one embodiment analyzes each of the packets passing through any point in the network in either direction, in order to derive the actual application used to communicate between a client and a server. Note that there could be several simultaneous and overlapping applications executing over the network that are independent and asynchronous.

A monitor embodiment of the invention successfully classifies each of the individual packets as they are seen on the network. The contents of the packets are parsed and selected parts are assembled into a signature (also called a key) that may then be used identify further packets of the same conversational flow, for example to further analyze the flow and ultimately to recognize the application program. Thus the key is a function of the selected parts, and in the preferred embodiment, the function is a concatenation of the selected parts. The preferred embodiment forms and remembers the state of any conversational flow, which is determined by the relationship between individual packets and the entire conversational flow over the network. By remembering the state of a flow in this way, the embodiment determines the context of the conversational flow, including the application program it relates to and parameters such as the time, length of the conversational flow, data rate, etc.

The monitor is flexible to adapt to future applications developed for client/server networks. New protocols and protocol combinations may be incorporated by compiling files written in a high-level protocol description language.

The monitor embodiment of the present invention is preferably implemented in application-specific integrated circuits (ASIC) or field programmable gate arrays (FPGA). In one embodiment, the monitor comprises a parser subsystem that forms a signature from a packet. The monitor further comprises an analyzer subsystem that receives the signature from the parser subsystem.

A packet acquisition device such as a media access controller (MAC) or a segmentation and reassemble module is used to provide packets to the parser subsystem of the monitor.

In a hardware implementation, the parsing subsystem comprises two sub-parts, the pattern analysis and recognition engine (PRE), and an extraction engine (slicer). The PRE interprets each packet, and in particular, interprets individual fields in each packet according to a pattern database.

The different protocols that can exist in different layers may be thought of as nodes of one or more trees of linked nodes. The packet type is the root of a tree. Each protocol is either a parent node or a terminal node. A parent node links a protocol to other protocols (child protocols) that can be at higher layer levels. For example, An Ethernet packet (the root node) may be an Ethertype packet—also called an Ethernet Type/Version 2 and a DIX (DIGITAL-Intel-Xerox packet)—or an IEEE 802.3 packet. Continuing with the IEEE 802.3-type packet, one of the children nodes may be the IP protocol, and one of the children of the IP protocol may be the TCP protocol.

The pattern database includes a description of the different headers of packets and their contents, and how these relate to the different nodes in a tree. The PRE traverses the tree as far as it can. If a node does not include a link to a deeper level, pattern matching is declared complete. Note that protocols can be the children of several parents. If a unique node was generated for each of the possible parent/child trees, the pattern database might become excessively large. Instead, child nodes are shared among multiple parents, thus compacting the pattern database.

Finally the PRE can be used on its own when only protocol recognition is required.

For each protocol recognized, the slicer extracts important packet elements from the packet. These form a signature (i.e., key) for the packet. The slicer also preferably generates a hash for rapidly identifying a flow that may have this signature from a database of known flows.

The flow signature of the packet, the hash and at least some of the payload are passed to an analyzer subsystem. In a hardware embodiment, the analyzer subsystem includes a unified flow key buffer (UFKB) for receiving parts of packets from the parser subsystem and for storing signatures in process, a lookup/update engine (LUE) to lookup a database of flow records for previously encountered conversational flows to determine whether a signature is from an existing flow, a state processor (SP) for performing state processing, a flow insertion and deletion engine (FIDE) for inserting new flows into the database of flows, a memory for storing the database of flows, and a cache for speeding up access to the memory containing the flow database. The LUE, SP, and FIDE are all coupled to the UFKB, and to the cache.

The unified flow key buffer thus contains the flow signature of the packet, the hash and at least some of the payload for analysis in the analyzer subsystem. Many operations can be performed to further elucidate the identity of the application program content of the packet involved in the client/server conversational flow while a packet signature exists in the unified flow signature buffer. In the particular hardware embodiment of the analyzer subsystem several flows may be processed in parallel, and multiple flow signatures from all the packets being analyzed in parallel may be held in the one UFKB.

The first step in the packet analysis process of a packet from the parser subsystem is to lookup the instance in the current database of known packet flow signatures. A lookup/update engine (LUE) accomplishes this task using first the hash, and then the flow signature. The search is carried out in the cache and if there is no flow with a matching signature in the cache, the lookup engine attempts to retrieve the flow from the flow database in the memory. The flow-entry for previously encountered flows preferably includes state information, which is used in the state processor to execute any operations defined for the state, and to determine the next state. A typical state operation may be to search for one or more known reference strings in the payload of the packet stored in the UFKB.

Once the lookup processing by the LUE has been completed a flag stating whether it is found or is new is set within the unified flow signature buffer structure for this packet flow signature. For an existing flow, the flow-entry is updated by a calculator component of the LUE that adds values to counters in the flow-entry database used to store one or more statistical measures of the flow. The counters are used for determining network usage metrics on the flow.

After the packet flow signature has been looked up and contents of the current flow signature are in the database, a

state processor can begin analyzing the packet payload to further elucidate the identity of the application program component of this packet. The exact operation of the state processor and functions performed by it will vary depending on the current packet sequence in the stream of a conversational flow. The state processor moves to the next logical operation stored from the previous packet seen with this same flow signature. If any processing is required on this packet, the state processor will execute instructions from a database of state instruction for this state until there are either no more left or the instruction signifies processing.

In the preferred embodiment, the state processor functions are programmable to provide for analyzing new application programs, and new sequences of packets and states that can arise from using such application.

If during the lookup process for this particular packet flow signature, the flow is required to be inserted into the active database, a flow insertion and deletion engine (FIDE) is initiated. The state processor also may create new flow signatures and thus may instruct the flow insertion and deletion engine to add a new flow to the database as a new item.

In the preferred hardware embodiment, each of the LUE, state processor, and FIDE operate independently from the other two engines.

BRIEF DESCRIPTION OF THE DRAWINGS

Although the present invention is better understood by referring to the detailed preferred embodiments, these should not be taken to limit the present invention to any specific embodiment because such embodiments are provided only for the purposes of explanation. The embodiments, in turn, are explained with the aid of the following figures.

FIG. **1** is a functional block diagram of a network embodiment of the present invention in which a monitor is connected to analyze packets passing at a connection point.

FIG. **2** is a diagram representing an example of some of the packets and their formats that might be exchanged in starting, as an illustrative example, a conversational flow between a client and server on a network being monitored and analyzed. A pair of flow signatures particular to this example and to embodiments of the present invention is also illustrated. This represents some of the possible flow signatures that can be generated and used in the process of analyzing packets and of recognizing the particular server applications that produce the discrete application packet exchanges.

FIG. **3** is a functional block diagram of a process embodiment of the present invention that can operate as the packet monitor shown in FIG. **1**. This process may be implemented in software or hardware.

FIG. **4** is a flowchart of a high-level protocol language compiling and optimization process, which in one embodiment may be used to generate data for monitoring packets according to versions of the present invention.

FIG. **5** is a flowchart of a packet parsing process used as part of the parser in an embodiment of the inventive packet monitor.

FIG. **6** is a flowchart of a packet element extraction process that is used as part of the parser in an embodiment of the inventive packet monitor.

FIG. **7** is a flowchart of a flow-signature building process that is used as part of the parser in the inventive packet monitor.

FIG. **8** is a flowchart of a monitor lookup and update process that is used as part of the analyzer in an embodiment of the inventive packet monitor.

FIG. **9** is a flowchart of an exemplary Sun Microsystems Remote Procedure Call application than may be recognized by the inventive packet monitor.

FIG. **10** is a functional block diagram of a hardware parser subsystem including the pattern recognizer and extractor that can form part of the parser module in an embodiment of the inventive packet monitor.

FIG. **11** is a functional block diagram of a hardware analyzer including a state processor that can form part of an embodiment of the inventive packet monitor.

FIG. **12** is a functional block diagram of a flow insertion and deletion engine process that can form part of the analyzer in an embodiment of the inventive packet monitor.

FIG. **13** is a flowchart of a state processing process that can form part of the analyzer in an embodiment of the inventive packet monitor.

FIG. **14** is a simple functional block diagram of a process embodiment of the present invention that can operate as the packet monitor shown in FIG. **1**. This process may be implemented in software.

FIG. **15** is a functional block diagram of how the packet monitor of FIG. **3** (and FIGS. **10** and **11**) may operate on a network with a processor such as a microprocessor.

FIG. **16** is an example of the top (MAC) layer of an Ethernet packet and some of the elements that may be extracted to form a signature according to one aspect of the invention.

FIG. **17A** is an example of the header of an Ethertype type of Ethernet packet of FIG. **16** and some of the elements that may be extracted to form a signature according to one aspect of the invention.

FIG. **17B** is an example of an IP packet, for example, of the Ethertype packet shown in FIGS. **16** and **17A**, and some of the elements that may be extracted to form a signature according to one aspect of the invention.

FIG. **18A** is a three dimensional structure that can be used to store elements of the pattern, parse and extraction database used by the parser subsystem in accordance to one embodiment of the invention.

FIG. **18B** is an alternate form of storing elements of the pattern, parse and extraction database used by the parser subsystem in accordance to another embodiment of the invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Note that this document includes hardware diagrams and descriptions that may include signal names. In most cases, the names are sufficiently descriptive, in other cases however the signal names are not needed to understand the operation and practice of the invention.

Operation in a Network

FIG. **1** represents a system embodiment of the present invention that is referred to herein by the general reference numeral **100**. The system **100** has a computer network **102** that communicates packets (e.g., IP datagrams) between various computers, for example between the clients **104–107** and servers **110** and **112**. The network is shown schematically as a cloud with several network nodes and links shown in the interior of the cloud. A monitor **108** examines the packets passing in either direction past its connection point **121** and, according to one aspect of the invention, can elucidate what application programs are associated with

each packet. The monitor **108** is shown examining packets (i.e., datagrams) between the network interface **116** of the server **110** and the network. The monitor can also be placed at other points in the network, such as connection point **123** between the network **102** and the interface **118** of the client **104**, or some other location, as indicated schematically by connection point **125** somewhere in network **102**. Not shown is a network packet acquisition device at the location **123** on the network for converting the physical information on the network into packets for input into monitor **108**. Such packet acquisition devices are common.

Various protocols may be employed by the network to establish and maintain the required communication, e.g., TCP/IP, etc. Any network activity—for example an application program run by the client **104** (CLIENT 1) communicating with another running on the server **110** (SERVER 2)—will produce an exchange of a sequence of packets over network **102** that is characteristic of the respective programs and of the network protocols. Such characteristics may not be completely revealing at the individual packet level. It may require the analyzing of many packets by the monitor **108** to have enough information needed to recognize particular application programs. The packets may need to be parsed then analyzed in the context of various protocols, for example, the transport through the application session layer protocols for packets of a type conforming to the ISO layered network model.

Communication protocols are layered, which is also referred to as a protocol stack. The ISO (International Standardization Organization) has defined a general model that provides a framework for design of communication protocol layers. This model, shown in tables form below, serves as a basic reference for understanding the functionality of existing communication protocols.

ISO MODEL

Layer	Functionality	Example
7	Application	Telnet, NFS, Novell NCP, HTTP, H.323
6	Presentation	XDR
5	Session	RPC, NETBIOS, SNMP, etc
4	Transport	TCP, Novel SPX, UDP, etc.
3	Network	IP, Novell IPX, VIP, AppleTalk, etc.
2	Data Link	Network Interface Card (Hardware Interface). MAC layer
1	Physical	Ethernet, Token Ring, Frame Relay, ATM, T1 (Hardware Connection)

Different communication protocols employ different levels of the ISO model or may use a layered model that is similar to but which does not exactly conform to the ISO model. A protocol in a certain layer may not be visible to protocols employed at other layers. For example, an application (Level 7) may not be able to identify the source computer for a communication attempt (Levels 2–3).

In some communication arts, the term "frame" generally refers to encapsulated data at OSI layer **2**, including a destination address, control bits for flow control, the data or payload, and CRC (cyclic redundancy check) data for error checking. The term "packet" generally refers to encapsulated data at OSI layer **3**. In the TCP/IP world, the term "datagram" is also used. In this specification, the term "packet" is intended to encompass packets, datagrams, frames, and cells. In general, a packet format or frame format refers to how data is encapsulated with various fields and headers for transmission across a network. For example, a data packet typically includes an address destination field, a length field, an error correcting code (ECC) field, or cyclic redundancy check (CRC) field, as well as headers and footers to identify the beginning and end of the packet. The terms "packet format" and "frame format," also referred to as "cell format,"**0** are generally synonymous.

Monitor **108** looks at every packet passing the connection point **121** for analysis. However, not every packet carries the same information useful for recognizing all levels of the protocol. For example, in a conversational flow associated with a particular application, the application will cause the server to send a type-A packet, but so will another. If, though, the particular application program always follows a type-A packet with the sending of a type-B packet, and the other application program does not, then in order to recognize packets of that application's conversational flow, the monitor can be available to recognize packets that match the type-B packet to associate with the type-A packet. If such is recognized after a type-A packet, then the particular application program's conversational flow has started to reveal itself to the monitor **108**.

Further packets may need to be examined before the conversational flow can be identified as being associated with the application program. Typically, monitor **108** is simultaneously also in partial completion of identifying other packet exchanges that are parts of conversational flows associated with other applications. One aspect of monitor **108** is its ability to maintain the state of a flow. The state of a flow is an indication of all previous events in the flow that lead to recognition of the content of all the protocol levels, e.g., the ISO model protocol levels. Another aspect of the invention is forming a signature of extracted characteristic portions of the packet that can be used to rapidly identify packets belonging to the same flow.

In real-world uses of the monitor **108**, the number of packets on the network **102** passing by the monitor **108**'s connection point can exceed a million per second. Consequently, the monitor has very little time available to analyze and type each packet and identify and maintain the state of the flows passing through the connection point. The monitor **108** therefore masks out all the unimportant parts of each packet that will not contribute to its classification. However, the parts to mask-out will change with each packet depending on which flow it belongs to and depending on the state of the flow.

The recognition of the packet type, and ultimately of the associated application programs according to the packets that their executions produce, is a multi-step process within the monitor **108**. At a first level, for example, several application programs will all produce a first kind of packet. A first "signature" is produced from selected parts of a packet that will allow monitor **108** to identify efficiently any packets that belong to the same flow. In some cases, that packet type may be sufficiently unique to enable the monitor to identify the application that generated such a packet in the conversational flow. The signature can then be used to efficiently identify all future packets generated in traffic related to that application.

In other cases, that first packet only starts the process of analyzing the conversational flow, and more packets are necessary to identify the associated application program. In such a case, a subsequent packet of a second type—but that potentially belongs to the same conversational flow—is recognized by using the signature. At such a second level, then, only a few of those application programs will have

conversational flows that can produce such a second packet type. At this level in the process of classification, all application programs that are not in the set of those that lead to such a sequence of packet types may be excluded in the process of classifying the conversational flow that includes these two packets. Based on the known patterns for the protocol and for the possible applications, a signature is produced that allows recognition of any future packets that may follow in the conversational flow.

It may be that the application is now recognized, or recognition may need to proceed to a third level of analysis using the second level signature. For each packet, therefore, the monitor parses the packet and generates a signature to determine if this signature identified a previously encountered flow, or shall be used to recognize future packets belonging to the same conversational flow. In real time, the packet is further analyzed in the context of the sequence of previously encountered packets (the state), and of the possible future sequences such a past sequence may generate in conversational flows associated with different applications. A new signature for recognizing future packets may also be generated. This process of analysis continues until the applications are identified. The last generated signature may then be used to efficiently recognize future packets associated with the same conversational flow. Such an arrangement makes it possible for the monitor **108** to cope with millions of packets per second that must be inspected.

Another aspect of the invention is adding Eavesdropping. In alternative embodiments of the present invention capable of eavesdropping, once the monitor **108** has recognized the executing application programs passing through some point in the network **102** (for example, because of execution of the applications by the client **105** or server **110**), the monitor sends a message to some general purpose processor on the network that can input the same packets from the same location on the network, and the processor then loads its own executable copy of the application program and uses it to read the content being exchanged over the network. In other words, once the monitor **108** has accomplished recognition of the application program, eavesdropping can commence.

The Network Monitor

FIG. **3** shows a network packet monitor **300**, in an embodiment of the present invention that can be implemented with computer hardware and/or software. The system **300** is similar to monitor **108** in FIG. **1**. A packet **302** is examined, e.g., from a packet acquisition device at the location **121** in network **102** (FIG. **1**), and the packet evaluated, for example in an attempt to determine its characteristics, e.g., all the protocol information in a multi-level model, including what server application produced the packet.

The packet acquisition device is a common interface that converts the physical signals and then decodes them into bits, and into packets, in accordance with the particular network (Ethernet, frame relay, ATM, etc.). The acquisition device indicates to the monitor **108** the type of network of the acquired packet or packets.

Aspects shown here include: (1) the initialization of the monitor to generate what operations need to occur on packets of different types—accomplished by compiler and optimizer **310**, (2) the processing—parsing and extraction of selected portions—of packets to generate an identifying signature—accomplished by parser subsystem **301**, and (3) the analysis of the packets—accomplished by analyzer **303**.

The purpose of compiler and optimizer **310** is to provide protocol specific information to parser subsystem **301** and to analyzer subsystem **303**. The initialization occurs prior to operation of the monitor, and only needs to re-occur when new protocols are to be added.

A flow is a stream of packets being exchanged between any two addresses in the network. For each protocol there are known to be several fields, such as the destination (recipient), the source (the sender), and so forth, and these and other fields are used in monitor **300** to identify the flow. There are other fields not important for identifying the flow, such as checksums, and those parts are not used for identification.

Parser subsystem **301** examines the packets using pattern recognition process **304** that parses the packet and determines the protocol types and associated headers for each protocol layer that exists in the packet **302**. An extraction process **306** in parser subsystem **301** extracts characteristic portions (signature information) from the packet **302**. Both the pattern information for parsing and the related extraction operations, e.g., extraction masks, are supplied from a parsing-pattern-structures and extraction-operations database (parsing/extractions database) **308** filled by the compiler and optimizer **310**.

The protocol description language (PDL) files **336** describes both patterns and states of all protocols that an occur at any layer, including how to interpret header information, how to determine from the packet header information the protocols at the next layer, and what information to extract for the purpose of identifying a flow, and ultimately, applications and services. The layer selections database **338** describes the particular layering handled by the monitor. That is, what protocols run on top of what protocols at any layer level. Thus **336** and **338** combined describe how one would decode, analyze, and understand the information in packets, and, furthermore, how the information is layered. This information is input into compiler and optimizer **310**.

When compiler and optimizer **310** executes, it generates two sets of internal data structures. The first is the set of parsing/extraction operations **308**. The pattern structures include parsing information and describe what will be recognized in the headers of packets; the extraction operations are what elements of a packet are to be extracted from the packets based on the patterns that get matched. Thus, database **308** of parsing/extraction operations includes information describing how to determine a set of one or more protocol dependent extraction operations from data in the packet that indicate a protocol used in the packet.

The other internal data structure that is built by compiler **310** is the set of state patterns and processes **326**. These are the different states and state transitions that occur in different conversational flows, and the state operations that need to be performed (e.g., patterns that need to be examined and new signatures that need to be built) during any state of a conversational flow to further the task of analyzing the conversational flow.

Thus, compiling the PDL files and layer selections provides monitor **300** with the information it needs to begin processing packets. In an alternate embodiment, the contents of one or more of databases **308** and **326** may be manually or otherwise generated. Note that in some embodiments the layering selections information is inherent rather than explicitly described. For example, since a PDL file for a protocol includes the child protocols, the parent protocols also may be determined.

In the preferred embodiment, the packet **302** from the acquisition device is input into a packet buffer. The pattern recognition process **304** is carried out by a pattern analysis

NOAC Ex. 1015 Page 646

and recognition (PAR) engine that analyzes and recognizes patterns in the packets. In particular, the PAR locates the next protocol field in the header and determines the length of the header, and may perform certain other tasks for certain types of protocol headers. An example of this is type and length comparison to distinguish an IEEE 802.3 (Ethernet) packet from the older type 2 (or Version 2) Ethernet packet, also called a DIGITAL-Intel-Xerox (DIX) packet. The PAR also uses the pattern structures and extraction operations database **308** to identify the next protocol and parameters associated with that protocol that enables analysis of the next protocol layer. Once a pattern or a set of patterns has been identified, it/they will be associated with a set of none or more extraction operations. These extraction operations (in the form of commands and associated parameters) are passed to the extraction process **306** implemented by an extracting and information identifying (EII) engine that extracts selected parts of the packet, including identifying information from the packet as required for recognizing this packet as part of a flow. The extracted information is put in sequence and then processed in block **312** to build a unique flow signature (also called a "key") for this flow. A flow signature depends on the protocols used in the packet. For some protocols, the extracted components may include source and destination addresses. For example, Ethernet frames have end-point addresses that are useful in building a better flow signature. Thus, the signature typically includes the client and server address pairs. The signature is used to recognize further packets that are or may be part of this flow.

In the preferred embodiment, the building of the flow key includes generating a hash of the signature using a hash function. The purpose if using such a hash is conventional—to spread flow-entries identified by the signature across a database for efficient searching. The hash generated is preferably based on a hashing algorithm and such hash generation is known to those in the art.

In one embodiment, the parser passes data from the packet—a parser record—that includes the signature (i.e., selected portions of the packet), the hash, and the packet itself to allow for any state processing that requires further data from the packet. An improved embodiment of the parser subsystem might generate a parser record that has some predefined structure and that includes the signature, the hash, some flags related to some of the fields in the parser record, and parts of the packet's payload that the parser subsystem has determined might be required for further processing, e.g., for state processing.

Note that alternate embodiments may use some function other than concatenation of the selected portions of the packet to make the identifying signature. For example, some "digest function" of the concatenated selected portions may be used.

The parser record is passed onto lookup process **314** which looks in an internal data store of records of known flows that the system has already encountered, and decides (in **316**) whether or not this particular packet belongs to a known flow as indicated by the presence of a flow-entry matching this flow in a database of known flows **324**. A record in database **324** is associated with each encountered flow.

The parser record enters a buffer called the unified flow key buffer (UFKB). The UFKB stores the data on flows in a data structure that is similar to the parser record, but that includes a field that can be modified. In particular, one or the UFKB record fields stores the packet sequence number, and another is filled with state information in the form of a program counter for a state processor that implements state processing **328**.

The determination (**316**) of whether a record with the same signature already exists is carried out by a lookup engine (LUE) that obtains new UFKB records and uses the hash in the UFKB record to lookup if there is a matching known flow. In the particular embodiment, the database of known flows **324** is in an external memory. A cache is associated with the database **324**. A lookup by the LUE for a known record is carried out by accessing the cache using the hash, and if the entry is not already present in the cache, the entry is looked up (again using the hash) in the external memory.

The flow-entry database **324** stores flow-entries that include the unique flow-signature, state information, and extracted information from the packet for updating flows, and one or more statistical about the flow. Each entry completely describes a flow. Database **324** is organized into bins that contain a number, denoted N, of flow-entries (also called flow-entries, each a bucket), with N being 4 in the preferred embodiment. Buckets (i.e., flow-entries) are accessed via the hash of the packet from the parser subsystem **301** (i.e., the hash in the UFKB record). The hash spreads the flows across the database to allow for fast lookups of entries, allowing shallower buckets. The designer selects the bucket depth N based on the amount of memory attached to the monitor, and the number of bits of the hash data value used. For example, in one embodiment, each flow-entry is 128 bytes long, so for 128K flow-entries, 16 Mbytes are required. Using a is 16-bit hash gives two flow-entries per bucket. Empirically, this has been shown to be more than adequate for the vast majority of cases. Note that another embodiment uses flow-entries that are 256 bytes long.

Herein, whenever an access to database **324** is described, it is to be understood that the access is via the cache, unless otherwise stated or clear from the context.

If there is no flow-entry found matching the signature, i.e., the signature is for a new flow, then a protocol and state identification process **318** further determines the state and protocol. That is, process **318** determines the protocols and where in the state sequence for a flow for this protocol's this packet belongs. Identification process **318** uses the extracted information and makes reference to the database **326** of state patterns and processes. Process **318** is then followed by any state operations that need to be executed on this packet by a state processor **328**.

If the packet is found to have a matching flow-entry in the database **324** (e.g., in the cache), then a process **320** determines, from the looked-up flow-entry, if more classification by state processing of the flow signature is necessary. If not, a process **322** updates the flow-entry in the flow-entry database **324** (e.g., via the cache). Updating includes updating one or more statistical measures stored in the flow-entry. In our embodiment, the statistical measures are stored in counters in the flow-entry.

If state processing is required, state process **328** is commenced. State processor **328** carries out any state operations specified for the state of the flow and updates the state to the next state according to a set of state instructions obtained form the state pattern and processes database **326**.

The state processor **328** analyzes both new and existing flows in order to analyze all levels of the protocol stack, ultimately classifying the flows by application (level **7** in the ISO model). It does this by proceeding from state-to-state based on predefined state transition rules and state opera-

NOAC Ex. 1015 Page 647

tions as specified in state processor instruction database **326**. A state transition rule is a rule typically containing a test followed by the next-state to proceed to if the test result is true. An operation is an operation to be performed while the state processor is in a particular state—for example, in order to evaluate a quantity needed to apply the state transition rule. The state processor goes through each rule and each state process until the test is true, or there are no more tests to perform.

In general, the set of state operations may be none or more operations on a packet, and carrying out the operation or operations may leave one in a state that causes exiting the system prior to completing the identification, but possibly knowing more about what state and state processes are needed to execute next, i.e., when a next packet of this flow is encountered. As an example, a state process (set of state operations) at a particular state may build a new signature for future recognition packets of the next state.

By maintaining the state of the flows and knowing that new flows may be set up using the information from previously encountered flows, the network traffic monitor **300** provides for (a) single-packet protocol recognition of flows, and (b) multiple-packet protocol recognition of flows. Monitor **300** can even recognize the application program from one or more disjointed sub-flows that occur in server announcement type flows. What may seem to prior art monitors to be some unassociated flow, may be recognized by the inventive monitor using the flow signature to be a sub-flow associated with a previously encountered sub-flow.

Thus, state processor **328** applies the first state operation to the packet for this particular flow-entry. A process **330** decides if more operations need to be performed for this state. If so, the analyzer continues looping between block **330** and **328** applying additional state operations to this particular packet until all those operations are completed—that is, there are no more operations for this packet in this state. A process **332** decides if there are further states to be analyzed for this type of flow according to the state of the flow and the protocol, in order to fully characterize the flow. If not, the conversational flow has now been fully characterized and a process **334** finalizes the classification of the conversational flow for the flow.

In the particular embodiment, the state processor **328** starts the state processing by using the last protocol recognized by the parser as an offset into a jump table (jump vector). The jump table finds the state processor instructions to use for that protocol in the state patterns and processes database **326**. Most instructions test something in the unified flow key buffer, or the flow-entry in the database of known flows **324**, if the entry exists. The state processor may have to test bits, do comparisons, add, or subtract to perform the test. For example, a common operation carried out by the state processor is searching for one or more patterns in the payload part of the UFKB.

Thus, in **332** in the classification, the analyzer decides whether the flow is at an end state. If not at an end state, the flow-entry is updated (or created if a new flow) for this flow-entry in process **322**.

Furthermore, if the flow is known and if in **332** it is determined that there are further states to be processed using later packets, the flow-entry is updated in process **322**.

The flow-entry also is updated after classification finalization so that any further packets belonging to this flow will be readily identified from their signature as belonging to this fully analyzed conversational flow.

After updating, database **324** therefore includes the set of all the conversational flows that have occurred.

Thus, the embodiment of present invention shown in FIG. **3** automatically maintains flow-entries, which in one aspect includes storing states. The monitor of FIG. **3** also generates characteristic parts of packets—the signatures—that can be used to recognize flows. The flow-entries may be identified and accessed by their signatures. Once a packet is identified to be from a known flow, the state of the flow is known and this knowledge enables state transition analysis to be performed in real time for each different protocol and application. In a complex analysis, state transitions are traversed as more and more packets are examined. Future packets that are part of the same conversational flow have their state analysis continued from a previously achieved state. When enough packets related to an application of interest have been processed, a final recognition state is ultimately reached, i.e., a set of states has been traversed by state analysis to completely characterize the conversational flow. The signature for that final state enables each new incoming packet of the same conversational flow to be individually recognized in real time.

In this manner, one of the great advantages of the present invention is realized. Once a particular set of state transitions has been traversed for the first time and ends in a final state, a short-cut recognition pattern—a signature—can be generated that will key on every new incoming packet that relates to the conversational flow. Checking a signature involves a simple operation, allowing high packet rates to be successfully monitored on the network.

In improved embodiments, several state analyzers are run in parallel so that a large number of protocols and applications may be checked for. Every known protocol and application will have at least one unique set of state transitions, and can therefore be uniquely identified by watching such transitions.

When each new conversational flow starts, signatures that recognize the flow are automatically generated on-the-fly, and as further packets in the conversational flow are encountered, signatures are updated and the states of the set of state transitions for any potential application are further traversed according to the state transition rules for the flow. The new states for the flow—those associated with a set of state transitions for one or more potential applications—are added to the records of previously encountered states for easy recognition and retrieval when a new packet in the flow is encountered.

Detailed Operation

FIG. **4** diagrams an initialization system **400** that includes the compilation process. That is, part of the initialization generates the pattern structures and extraction operations database **308** and the state instruction database **328**. Such initialization can occur off-line or from a central location.

The different protocols that can exist in different layers may be thought of as nodes of one or more trees of linked nodes. The packet type is the root of a tree (called level **0**). Each protocol is either a parent node or a terminal node. A parent node links a protocol to other protocols (child protocols) that can be at higher layer levels. Thus a protocol may have zero or more children. Ethernet packets, for example, have several variants, each having a basic format that remains substantially the same. An Ethernet packet (the root or level **0** node) may be an Ethertype packet—also called an Ethernet Type/Version 2 and a DIX (DIGITAL-Intel-Xerox packet)—or an IEEE 803.2 packet. Continuing with the IEEE 802.3 packet, one of the children nodes may be the IP protocol, and one of the children of the IP protocol may be the TCP protocol.

FIG. **16** shows the header **1600** (base level **1**) of a complete Ethernet frame (i.e., packet) of information and includes information on the destination media access control address (Dst MAC **1602**) and the source media access control address (Src MAC **1604**). Also shown in FIG. **16** is some (but not all) of the information specified in the PDL files for extraction the signature.

FIG. **17**A now shows the header information for the next level (level-**2**) for an Ethertype packet **1700**. For an Ethertype packet **1700**, the relevant information from the packet that indicates the next layer level is a two-byte type field **1702** containing the child recognition pattern for the next level. The remaining information **1704** is shown hatched because it not relevant for this level. The list **1712** shows the possible children for an Ethertype packet as indicated by what child recognition pattern is found offset **12**. FIG. **17**B shows the structure of the header of one of the possible next levels, that of the IP protocol. The possible children of the IP protocol are shown in table **1752**.

The pattern, parse, and extraction database (pattern recognition database, or PRD) **308** generated by compilation process **310**, in one embodiment, is in the form of a three dimensional structure that provides for rapidly searching packet headers for the next protocol. FIG. **18**A shows such a 3-D representation **1800** (which may be considered as an indexed set of 2-D representations). A compressed form of the 3-D structure is preferred.

An alternate embodiment of the data structure used in database **308** is illustrated in FIG. **18**B. Thus, like the 3-D structure of FIG. **18**A, the data structure permits rapid searches to be performed by the pattern recognition process **304** by indexing locations in a memory rather than performing address link computations. In this alternate embodiment, the PRD **308** includes two parts, a single protocol table **1850** (PT) which has an entry for each protocol known for the monitor, and a series of Look Up Tables **1870** (LUT's) that are used to identify known protocols and their children. The protocol table includes the parameters needed by the pattern analysis and recognition process **304** (implemented by PRE **1006**) to evaluate the header information in the packet that is associated with that protocol, and parameters needed by extraction process **306** (implemented by slicer **1007**) to process the packet header. When there are children, the PT describes which bytes in the header to evaluate to determine the child protocol. In particular, each PT entry contains the header length, an offset to the child, a slicer command, and some flags.

The pattern matching is carried out by finding particular "child recognition codes" in the header fields, and using these codes to index one or more of the LUT's. Each LUT entry has a node code that can have one of four values, indicating the protocol that has been recognized, a code to indicate that the protocol has been partially recognized (more LUT lookups are needed), a code to indicate that this is a terminal node, and a null node to indicate a null entry. The next LUT to lookup is also returned from a LUT lookup.

Compilation process is described in FIG. **4**. The source-code information in the form of protocol description files is shown as **402**. In the particular embodiment, the high level decoding descriptions includes a set of protocol description files **336**, one for each protocol, and a set of packet layer selections **338**, which describes the particular layering (sets of trees of protocols) that the monitor is to be able to handle.

A compiler **403** compiles the descriptions. The set of packet parse-and-extract operations **406** is generated (**404**), and a set of packet state instructions and operations **407** is generated (**405**) in the form of instructions for the state processor that implements state processing process **328**. Data files for each type of application and protocol to be recognized by the analyzer are downloaded from the pattern, parse, and extraction database **406** into the memory systems of the parser and extraction engines. (See the parsing process **500** description and FIG. **5**; the extraction process **600** description and FIG. **6**; and the parsing subsystem hardware description and FIG. **10**). Data files for each type of application and protocol to be recognized by the analyzer are also downloaded from the state-processor instruction database **407** into the state processor. (see the state processor **1108** description and FIG. **11**.).

Note that generating the packet parse and extraction operations builds and links the three dimensional structure (one embodiment) or the or all the lookup tables for the PRD.

Because of the large number of possible protocol trees and subtrees, the compiler process **400** includes optimization that compares the trees and subtrees to see which children share common parents. When implemented in the form of the LUT's, this process can generate a single LUT from a plurality of LUT's. The optimization process further includes a compaction process that reduces the space needed to store the data of the PRD.

As an example of compaction, consider the 3-D structure of FIG. **18**A that can be thought of as a set of 2-D structures each representing a protocol. To enable saving space by using only one array per protocol which may have several parents, in one embodiment, the pattern analysis subprocess keeps a "current header" pointer. Each location (offset) index for each protocol 2-D array in the 3-D structure is a relative location starting with the start of header for the particular protocol. Furthermore, each of the two-dimensional arrays is sparse. The next step of the optimization, is checking all the 2-D arrays against all the other 2-D arrays to find out which ones can share memory. Many of these 2-D arrays are often sparsely populated in that they each have only a small number of valid entries. So, a process of "folding" is next used to combine two or more 2-D arrays together into one physical 2-D array without losing the identity of any of the original 2-D arrays (i.e., all the 2-D arrays continue to exist logically). Folding can occur between any 2-D arrays irrespective of their location in the tree as long as certain conditions are met. Multiple arrays may be combined into a single array as long as the individual entries do not conflict with each other. A fold number is then used to associate each element with its original array. A similar folding process is used for the set of LUTs **1850** in the alternate embodiment of FIG. **18**B.

In **410**, the analyzer has been initialized and is ready to perform recognition.

FIG. **5** shows a flowchart of how actual parser subsystem **301** functions. Starting at **501**, the packet **302** is input to the packet buffer in step **502**. Step **503** loads the next (initially the first) packet component from the packet **302**. The packet components are extracted from each packet **302** one element at a time. A check is made (**504**) to determine if the load-packet-component operation **503** succeeded, indicating that there was more in the packet to process. If not, indicating all components have been loaded, the parser subsystem **301** builds the packet signature (**512**)—the next stage (FIG. **6**).

If a component is successfully loaded in **503**, the node and processes are fetched (**505**) from the pattern, parse and extraction database **308** to provide a set of patterns and

NOAC Ex. 1015 Page 649

processes for that node to apply to the loaded packet component. The parser subsystem **301** checks (**506**) to determine if the fetch pattern-node operation **505** completed successfully, indicating there was a pattern node that loaded in **505**. If not, step **511** moves to the next packet component. If yes, then the node and pattern matching process are applied in **507** to the component extracted in **503**. A pattern match obtained in **507** (as indicated by test **508**) means the parser subsystem **301** has found a node in the parsing elements; the parser subsystem **301** proceeds to step **509** to extract the elements.

If applying the node process to the component does not produce a match (test **508**), the parser subsystem **301** moves (**510**) to the next pattern node from the pattern database **308** and to step **505** to fetch the next node and process. Thus, there is an "applying patterns" loop between **508** and **505**. Once the parser subsystem **301** completes all the patterns and has either matched or not, the parser subsystem **301** moves to the next packet component (**511**).

Once all the packet components have been the loaded and processed from the input packet **302**, then the load packet will fail (indicated by test **504**), and the parser subsystem **301** moves to build a packet signature which is described in FIG. **6**

FIG. **6** is a flow chart for extracting the information from which to build the packet signature. The flow starts at **601**, which is the exit point **513** of FIG. **5**. At this point parser subsystem **301** has a completed packet component and a pattern node available in a buffer (**602**). Step **603** loads the packet component available from the pattern analysis process of FIG. **5**. If the load completed (test **604**), indicating that there was indeed another packet component, the parser subsystem **301** fetches in **605** the extraction and process elements received from the pattern node component in **602**. If the fetch was successful (test **606**), indicating that there are extraction elements to apply, the parser subsystem **301** in step **607** applies that extraction process to the packet component based on an extraction instruction received from that pattern node. This removes and saves an element from the packet component.

In step **608**, the parser subsystem **301** checks if there is more to extract from this component, and if not, the parser subsystem **301** moves back to **603** to load the next packet component at hand and repeats the process. If the answer is yes, then the parser subsystem **301** moves to the next packet component ratchet. That new packet component is then loaded in step **603**. As the parser subsystem **301** moved through the loop between **608** and **603**, extra extraction processes are applied either to the same packet component if there is more to extract, or to a different packet component if there is no more to extract.

The extraction process thus builds the signature, extracting more and more components according to the information in the patterns and extraction database **308** for the particular packet. Once loading the next packet component operation **603** fails (test **604**), all the components have been extracted. The built signature is loaded into the signature buffer (**610**) and the parser subsystem **301** proceeds to FIG. **7** to complete the signature generation process.

Referring now to FIG. **7**, the process continues at **701**. The signature buffer and the pattern node elements are available (**702**). The parser subsystem **301** loads the next pattern node element. If the load was successful (test **704**) indicating there are more nodes, the parser subsystem **301** in **705** hashes the signature buffer element based on the hash elements that are found in the pattern node that is in the element database. In **706** the resulting signature and the hash are packed. In **707** the parser subsystem **301** moves on to the next packet component which is loaded in **703**.

The **703** to **707** loop continues until there are no more patterns of elements left (test **704**). Once all the patterns of elements have been hashed, processes **304**, **306** and **312** of parser subsystem **301** are complete. Parser subsystem **301** has generated the signature used by the analyzer subsystem **303**.

A parser record is loaded into the analyzer, in particular, into the UFKB in the form of a UFKB record which is similar to a parser record, but with one or more different fields.

FIG. **8** is a flow diagram describing the operation of the lookup/update engine (LUE) that implements lookup operation **314**. The process starts at **801** from FIG. **7** with the parser record that includes a signature, the hash and at least parts of the payload. In **802** those elements are shown in the form of a UFKB-entry in the buffer. The LUE, the lookup engine **314** computes a "record bin number" from the hash for a flow-entry. A bin herein may have one or more "buckets" each containing a flow-entry. The preferred embodiment has four buckets per bin.

Since preferred hardware embodiment includes the cache, all data accesses to records in the flowchart of FIG. **8** are stated as being to or from the cache.

Thus, in **804**, the system looks up the cache for a bucket from that bin using the hash. If the cache successfully returns with a bucket from the bin number, indicating there are more buckets in the bin, the lookup/update engine compares (**807**) the current signature (the UFKB-entry's signature) from that in the bucket (i.e., the flow-entry signature). If the signatures match (test **808**), that record (in the cache) is marked in step **810** as "in process" and a timestamp added. Step **811** indicates to the UFKB that the UFKB-entry in **802** has a status of "found." The "found" indication allows the state processing **328** to begin processing this UFKB element. The preferred hardware embodiment includes one or more state processors, and these can operate in parallel with the lookup/update engine.

In the preferred embodiment, a set of statistical operations is performed by a calculator for every packet analyzed. The statistical operations may include one or more of counting the packets associated with the flow; determining statistics related to the size of packets of the flow; compiling statistics on differences between packets in each direction, for example using times tamps; and determining statistical relationships of timestamps of packets in the same direction. The statistical measures are kept in the flow-entries. Other statistical measures also may be compiled. These statistics may be used singly or in combination by a statistical processor component to analyze many different aspects of the flow. This may include determining network usage metrics from the statistical measures, for example to ascertain the network's ability to transfer information for this application. Such analysis provides for measuring the quality of service of a conversation, measuring how well an application is performing in the network, measuring network resources consumed by an application, and so forth.

To provide for such analyses, the lookup/update engine updates one or more counters that are part of the flow-entry (in the cache) in step **812**. The process exits at **813**. In our embodiment, the counters include the total packets of the flow, the time, and a differential time from the last timestamp to the present timestamp.

It may be that the bucket of the bin did not lead to a signature match (test **808**). In such a case, the analyzer in

809 moves to the next bucket for this bin. Step 804 again looks up the cache for another bucket from that bin. The lookup/update engine thus continues lookup up buckets of the bin until there is either a match in 808 or operation 804 is not successful (test 805), indicating that there are no more buckets in the bin and no match was found.

If no match was found, the packet belongs to a new (not previously encountered) flow. In 806 the system indicates that the record in the unified flow key buffer for this packet is new, and in 812, any statistical updating operations are performed for this packet by updating the flow-entry in the cache. The update operation exits at 813. A flow insertion/deletion engine (FIDE) creates a new record for this flow (again via the cache).

Thus, the update/lookup engine ends with a UFKB-entry for the packet with a "new" status or a "found" status.

Note that the above system uses a hash to which more than one flow-entry can match. A longer hash may be used that corresponds to a single flow-entry. In such an embodiment, the flow chart of FIG. 8 is simplified as would be clear to those in the art.

The Hardware System

Each of the individual hardware elements through which the data flows in the system are now described with reference to FIGS. 10 and 11. Note that while we are describing a particular hardware implementation of the invention embodiment of FIG. 3, it would be clear to one skilled in the art that the flow of FIG. 3 may alternatively be implemented in software running on one or more general-purpose processors, or only partly implemented in hardware. An implementation of the invention that can operate in software is shown in FIG. 14. The hardware embodiment (FIGS. 10 and 11) can operate at over a million packets per second, while the software system of FIG. 14 may be suitable for slower networks. To one skilled in the art it would be clear that more and more of the system may be implemented in software as processors become faster.

FIG. 10 is a description of the parsing subsystem (301, shown here as subsystem 1000) as implemented in hardware. Memory 1001 is the pattern recognition database memory, in which the patterns that are going to be analyzed are stored. Memory 1002 is the extraction-operation database memory, in which the extraction instructions are stored. Both 1001 and 1002 correspond to internal data structure 308 of FIG. 3. Typically, the system is initialized from a microprocessor (not shown) at which time these memories are loaded through a host interface multiplexor and control register 1005 via the internal buses 1003 and 1004. Note that the contents of 1001 and 1002 are preferably obtained by compiling process 310 of FIG. 3.

A packet enters the parsing system via 1012 into a parser input buffer memory 1008 using control signals 1021 and 1023, which control an input buffer interface controller 1022. The buffer 1008 and interface control 1022 connect to a packet acquisition device (not shown). The buffer acquisition device generates a packet start signal 1021 and the interface control 1022 generates a next packet (i.e., ready to receive data) signal 1023 to control the data flow into parser input buffer memory 1008. Once a packet starts loading into the buffer memory 1008, pattern recognition engine (PRE) 1006 carries out the operations on the input buffer memory described in block 304 of FIG. 3. That is, protocol types and associated headers for each protocol layer that exist in the packet are determined.

The PRE searches database 1001 and the packet in buffer 1008 in order to recognize the protocols the packet contains.

In one implementation, the database 1001 includes a series of linked lookup tables. Each lookup table uses eight bits of addressing. The first lookup table is always at address zero. The Pattern Recognition Engine uses a base packet offset from a control register to start the comparison. It loads this value into a current offset pointer (COP). It then reads the byte at base packet offset from the parser input buffer and uses it as an address into the first lookup table.

Each lookup table returns a word that links to another lookup table or it returns a terminal flag. If the lookup produces a recognition event the database also returns a command for the slicer. Finally it returns the value to add to the COP.

The PRE 1006 includes of a comparison engine. The comparison engine has a first stage that checks the protocol type field to determine if it is an 802.3 packet and the field should be treated as a length. If it is not a length, the protocol is checked in a second stage. The first stage is the only protocol level that is not programmable. The second stage has two full sixteen bit content addressable memories (CAMs) defined for future protocol additions.

Thus, whenever the PRE recognizes a pattern, it also generates a command for the extraction engine (also called a "slicer") 1007. The recognized patterns and the commands are sent to the extraction engine 1007 that extracts information from the packet to build the parser record. Thus, the operations of the extraction engine are those carried out in blocks 306 and 312 of FIG. 3. The commands are sent from PRE 1006 to slicer 1007 in the form of extraction instruction pointers which tell the extraction engine 1007 where to a find the instructions in the extraction operations database memory (i.e., slicer instruction database) 1002.

Thus, when the PRE 1006 recognizes a protocol it outputs both the protocol identifier and a process code to the extractor. The protocol identifier is added to the flow signature and the process code is used to fetch the first instruction from the instruction database 1002. Instructions include an operation code and usually source and destination offsets as well as a length. The offsets and length are in bytes. A typical operation is the MOVE instruction. This instruction tells the slicer 1007 to copy n bytes of data unmodified from the input buffer 1008 to the output buffer 1010. The extractor contains a byte-wise barrel shifter so that the bytes moved can be packed into the flow signature. The extractor contains another instruction called HASH. This instruction tells the extractor to copy from the input buffer 1008 to the HASH generator.

Thus these instructions are for extracting selected element (s) of the packet in the input buffer memory and transferring the data to a parser output buffer memory 1010. Some instructions also generate a hash.

The extraction engine 1007 and the PRE operate as a pipeline. That is, extraction engine 1007 performs extraction operations on data in input buffer 1008 already processed by PRE 1006 while more (i.e., later arriving) packet information is being simultaneously parsed by PRE 1006. This provides high processing speed sufficient to accommodate the high arrival rate speed of packets.

Once all the selected parts of the packet used to form the signature are extracted, the hash is loaded into parser output buffer memory 1010. Any additional payload from the packet that is required for further analysis is also included. The parser output memory 1010 is interfaced with the analyzer subsystem by analyzer interface control 1011. Once all the information of a packet is in the parser output buffer memory 1010, a data ready signal 1025 is asserted by

analyzer interface control. The data from the parser subsystem **1000** is moved to the analyzer subsystem via **1013** when an analyzer ready signal **1027** is asserted.

FIG. **11** shows the hardware components and dataflow for the analyzer subsystem that performs the functions of the analyzer subsystem **303** of FIG. **3**. The analyzer is initialized prior to operation, and initialization includes loading the state processing information generated by the compilation process **310** into a database memory for the state processing, called state processor instruction database (SPID) memory **1109**.

The analyzer subsystem **1100** includes a host bus interface **1122** using an analyzer host interface controller **1118**, which in turn has access to a cache system **1115**. The cache system has bi-directional access to and from the state processor of the system **1108**. State processor **1108** is responsible for initializing the state processor instruction database memory **1109** from information given over the host bus interface **1122**.

With the SPID **1109** loaded, the analyzer subsystem **1100** receives parser records comprising packet signatures and payloads that come from the parser into the unified flow key buffer (UFKB) **1103**. UFKB is comprised of memory set up to maintain UFKB records. A UFKB record is essentially a parser record; the UFKB holds records of packets that are to be processed or that are in process. Furthermore, the UFKB provides for one or more fields to act as modifiable status flags to allow different processes to run concurrently.

Three processing engines run concurrently and access records in the UFKB **1103**: the lookup/update engine (LUE) **1107**, the state processor (SP) **1108**, and the flow insertion and deletion engine (FIDE) **1110**. Each of these is implemented by one or more finite state machines (FSM's). There is bi-directional access between each of the finite state machines and the unified flow key buffer **1103**. The UFKB record includes a field that stores the packet sequence number, and another that is filled with state information in the form of a program counter for the state processor **1108** that implements state processing **328**. The status flags of the UFKB for any entry includes that the LUE is done and that the LUE is transferring processing of the entry to the state processor. The LUE done indicator is also used to indicate what the next entry is for the LUE. There also is provided a flag to indicate that the state processor is done with the current flow and to indicate what the next entry is for the state processor. There also is provided a flag to indicate the state processor is transferring processing of the UFKB-entry to the flow insertion and deletion engine.

A new UFKB record is first processed by the LUE **1107**. A record that has been processed by the LUE **1107** may be processed by the state processor **1108**, and a UFKB record data may be processed by the flow insertion/deletion engine **1110** after being processed by the state processor **1108** or only by the LUE. Whether or not a particular engine has been applied to any unified flow key buffer entry is determined by status fields set by the engines upon completion. In one embodiment, a status flag in the UFKB-entry indicates whether an entry is new or found. In other embodiments, the LUE issues a flag to pass the entry to the state processor for processing, and the required operations for a new record are included in the SP instructions.

Note that each UFKB-entry may not need to be processed by all three engines. Furthermore, some UFKB entries may need to be processed more than once by a particular engine.

Each of these three engines also has bi-directional access to a cache subsystem **1115** that includes a caching engine. Cache **1115** is designed to have information flowing in and out of it from five different points within the system: the three engines, external memory via a unified memory controller (UMC) **1119** and a memory interface **1123**, and a microprocessor via analyzer host interface and control unit (ACIC) **1118** and host interface bus (HIB) **1122**. The analyzer microprocessor (or dedicated logic processor) can thus directly insert or modify data in the cache.

The cache subsystem **1115** is an associative cache that includes a set of content addressable memory cells (CAMs) each including an address portion and a pointer portion pointing to the cache memory (e.g., RAM) containing the cached flow-entries. The CAMs are arranged as a stack ordered from a top CAM to a bottom CAM. The bottom CAM's pointer points to the least recently used (LRU) cache memory entry. Whenever there is a cache miss, the contents of cache memory pointed to by the bottom CAM are replaced by the flow-entry from the flow-entry database **324**. This now becomes the most recently used entry, so the contents of the bottom CAM are moved to the top CAM and all CAM contents are shifted down. Thus, the cache is an associative cache with a true LRU replacement policy.

The LUE **1107** first processes a UFKB-entry, and basically performs the operation of blocks **314** and **316** in FIG. **3**. A signal is provided to the LUE to indicate that a "new" UFKB-entry is available. The LUE uses the hash in the UFKB-entry to read a matching bin of up to four buckets from the cache. The cache system attempts to obtain the matching bin. If a matching bin is not in the cache, the cache **1115** makes the request to the UMC **1119** to bring in a matching bin from the external memory.

When a flow-entry is found using the hash, the LUE **1107** looks at each bucket and compares it using the signature to the signature of the UFKB-entry until there is a match or there are no more buckets.

If there is no match, or if the cache failed to provide a bin of flow-entries from the cache, a time stamp in set in the flow key of the UFKB record, a protocol identification and state determination is made using a table that was loaded by compilation process **310** during initialization, the status for the record is set to indicate the LUE has processed the record, and an indication is made that the UFKB-entry is ready to start state processing. The identification and state determination generates a protocol identifier which in the preferred embodiment is a "jump vector" for the state processor which is kept by the UFKB for this UFKB-entry and used by the state processor to start state processing for the particular protocol. For example, the jump vector jumps to the subroutine for processing the state.

If there was a match, indicating that the packet of the UFKB-entry is for a previously encountered flow, then a calculator component enters one or more statistical measures stored in the flow-entry, including the timestamp. In addition, a time difference from the last stored timestamp may be stored, and a packet count may be updated. The state of the flow is obtained from the flow-entry is examined by looking at the protocol identifier stored in the flow-entry of database **324**. If that value indicates that no more classification is required, then the status for the record is set to indicate the LUE has processed the record. In the preferred embodiment, the protocol identifier is a jump vector for the state processor to a subroutine to state processing the protocol, and no more classification is indicated in the preferred embodiment by the jump vector being zero. If the protocol identifier indicates more processing, then an indication is made that the UFKB-entry is ready to start state

processing and the status for the record is set to indicate the LUE has processed the record.

The state processor **1108** processes information in the cache system according to a UFKB-entry after the LUE has completed. State processor **1108** includes a state processor program counter SPPC that generates the address in the state processor instruction database **1109** loaded by compiler process **310** during initialization. It contains an Instruction Pointer (SPIP) which generates the SPID address. The instruction pointer can be incremented or loaded from a Jump Vector Multiplexor which facilitates conditional branching. The SPIP can be loaded from one of three sources: (1) A protocol identifier from the UFKB, (2) an immediate jump vector form the currently decoded instruction, or (3) a value provided by the arithmetic logic unit (SPALU) included in the state processor.

Thus, after a Flow Key is placed in the UFKB by the LUE with a known protocol identifier, the Program Counter is initialized with the last protocol recognized by the Parser. This first instruction is a jump to the subroutine which analyzes the protocol that was decoded.

The State Processor ALU (SPALU) contains all the Arithmetic, Logical and String Compare functions necessary to implement the State Processor instructions. The main blocks of the SPALU are: The A and B Registers, the Instruction Decode & State Machines, the String Reference Memory the Search Engine, an Output Data Register and an Output Control Register.

The Search Engine in turn contains the Target Search Register set, the Reference Search Register set, and a Compare block which compares two operands by exclusive-or-ing them together.

Thus, after the UFKB sets the program counter, a sequence of one or more state operations are be executed in state processor **1108** to further analyze the packet that is in the flow key buffer entry for this particular packet.

FIG. **13** describes the operation of the state processor **1108**. The state processor is entered at **1301** with a unified flow key buffer entry to be processed. The UFKB-entry is new or corresponding to a found flow-entry. This UFKB-entry is retrieved from unified flow key buffer **1103** in **1301**. In **1303**, the protocol identifier for the UFKB-entry is used to set the state processor's instruction counter. The state processor **1108** starts the process by using the last protocol recognized by the parser subsystem **301** as an offset into a jump table. The jump table takes us to the instructions to use for that protocol. Most instructions test something in the unified flow key buffer or the flow-entry if it exists. The state processor **1108** may have to test bits, do comparisons, add or subtract to perform the test.

The first state processor instruction is fetched in **1304** from the state processor instruction database memory **1109**. The state processor performs the one or more fetched operations (**1304**). In our implementation, each single state processor instruction is very primitive (e.g., a move, a compare, etc.), so that many such instructions need to be performed on each unified flow key buffer entry. One aspect of the state processor is its ability to search for one or more (up to four) reference strings in the payload part of the UFKB entry. This is implemented by a search engine component of the state processor responsive to special searching instructions.,

In **1307**, a check is made to determine if there are any more instructions to be performed for the packet. If yes, then in **1308** the system sets the state processor instruction pointer (SPIP) to obtain the next instruction. The SPIP may be set by an immediate jump vector in the currently decoded instruction, or by a value provided by the SPALU during processing.

The next instruction to be performed is now fetched (**1304**) for execution. This state processing loop between **1304** and **1307** continues until there are no more instructions to be performed.

At this stage, a check is made in **1309** if the processing on this particular packet has resulted in a final state. That is, is the analyzer is done processing not only for this particular packet, but for the whole flow to which the packet belongs, and the flow is fully determined. If indeed there are no more states to process for this flow, then in **1311** the processor finalizes the processing. Some final states may need to put a state in place that tells the system to remove a flow—for example, if a connection disappears from a lower level connection identifier. In that case, in **1311**, a flow removal state is set and saved in the flow-entry. The flow removal state may be a NOP (no-op) instruction which means there are no removal instructions.

Once the appropriate flow removal instruction as specified for this flow (a NOP or otherwise) is set and saved, the process is exited at **1313**. The state processor **1108** can now obtain another unified flow key buffer entry to process.

If at **1309** it is determined that processing for this flow is not completed, then in **1310** the system saves the state processor instruction pointer in the current flow-entry in the current flow-entry. That will be the next operation that will be performed the next time the LRE **1107** finds packet in the UFKB that matches this flow. The processor now exits processing this particular unified flow key buffer entry at **1313**.

Note that state processing updates information in the unified flow key buffer **1103** and the flow-entry in the cache. Once the state processor is done, a flag is set in the UFKB for the entry that the state processor is done. Furthermore, If the flow needs to be inserted or deleted from the database of flows, control is then passed on to the flow insertion/deletion engine **1110** for that flow signature and packet entry. This is done by the state processor setting another flag in the UFKB for this UFKB-entry indicating that the state processor is passing processing of this entry to the flow insertion and deletion engine.

The flow insertion and deletion engine **1110** is responsible for maintaining the flow-entry database. In particular, for creating new flows in the flow database, and deleting flows from the database so that they can be reused.

The process of flow insertion is now described with the aid of FIG. **12**. Flows are grouped into bins of buckets by the hash value. The engine processes a UFKB-entry that may be new or that the state processor otherwise has indicated needs to be created. FIG. **12** shows the case of a new entry being created. A conversation record bin (preferably containing 4 buckets for four records) is obtained in **1203**. This is a bin that matches the hash of the UFKB, so this bin may already have been sought for the UFKB-entry by the LUE. In **1204** the FIDE **1110** requests that the record bin/bucket be maintained in the cache system **1115**. If in **1205** the cache system **1115** indicates that the bin/bucket is empty, step **1207** inserts the flow signature (with the hash) into the bucket and the bucket is marked "used" in the cache engine of cache **1115** using a timestamp that is maintained throughout the process. In **1209**, the FIDE **1110** compares the bin and bucket record flow signature to the packet to verify that all the elements are in place to complete the record. In **1211** the system marks the record bin and bucket as "in process" and as new in the

NOAC Ex. 1015 Page 653

cache system (and hence in the external memory). In **1212**, the initial statistical measures for the flow-record are set in the cache system. This in the preferred embodiment clears the set of counters used to maintain statistics, and may perform other procedures for statistical operations requires by the analyzer for the first packet seen for a particular flow.

Back in step **1205**, if the bucket is not empty, the FIDE **1110** requests the next bucket for this particular bin in the cache system. If this succeeds, the processes of **1207**, **1209**, **1211** and **1212** are repeated for this next bucket. If at **1208**, there is no valid bucket, the unified flow key buffer entry for the packet is set as "drop," indicating that the system cannot process the particular packet because there are no buckets left in the system. The process exits at **1213**. The FIDE **1110** indicates to the UFKB that the flow insertion and deletion operations are completed for this UFKB-entry. This also lets the UFKB provide the FIDE with the next UFKB record.

Once a set of operations is performed on a unified flow key buffer entry by all of the engines required to access and manage a particular packet and its flow signature, the unified flow key buffer entry is marked as "completed." That element will then be used by the parser interface for the next packet and flow signature coming in from the parsing and extracting system.

All flow-entries are maintained in the external memory and some are maintained in the cache **1115**. The cache system **1115** is intelligent enough to access the flow database and to understand the data structures that exists on the other side of memory interface **1123**. The lookup/update engine **1107** is able to request that the cache system pull a particular flow or "buckets" of flows from the unified memory controller **1119** into the cache system for further processing. The state processor **1108** can operate on information found in the cache system once it is looked up by means of the lookup/update engine request, and the flow insertion/deletion engine **1110** can create new entries in the cache system if required based on information in the unified flow key buffer **1103**. The cache retrieves information as required from the memory through the memory interface **1123** and the unified memory controller **1119**, and updates information as required in the memory through the memory controller **1119**.

There are several interfaces to components of the system external to the module of FIG. **11** for the particular hardware implementation. These include host bus interface **1122**, which is designed as a generic interface that can operate with any kind of external processing system such as a microprocessor or a multiplexor (MUX) system. Consequently, one can connect the overall traffic classification system of FIGS. **11** and **12** into some other processing system to manage the classification system and to extract data gathered by the system.

The memory interface **1123** is designed to interface to any of a variety of memory systems that one may want to use to store the flow-entries. One can use different types of memory systems like regular dynamic random access memory (DRAM), synchronous DRAM, synchronous graphic memory (SGRAM), static random access memory (SRAM), and so forth.

FIG. **10** also includes some "generic" interfaces. There is a packet input interface **1012**—a general interface that works in tandem with the signals of the input buffer interface control **1022**. These are designed so that they can be used with any kind of generic systems that can then feed packet information into the parser. Another generic interface is the interface of pipes **1031** and **1033** respectively out of and into host interface multiplexor and control registers **1005**. This enables the parsing system to be managed by an external system, for example a microprocessor or another kind of external logic, and enables the external system to program and otherwise control the parser.

The preferred embodiment of this aspect of the invention is described in a hardware description language (HDL) such as VHDL or Verilog. It is designed and created in an HDL so that it may be used as a single chip system or, for instance, integrated into another general-purpose system that is being designed for purposes related to creating and analyzing traffic within a network. Verilog or other HDL implementation is only one method of describing the hardware.

In accordance with one hardware implementation, the elements shown in FIGS. **10** and **11** are implemented in a set of six field programmable logic arrays (FPGA's). The boundaries of these FPGA's are as follows. The parsing subsystem of FIG. **10** is implemented as two FPGAS; one FPGA, and includes blocks **1006**, **1008** and **1012**, parts of **1005**, and memory **1001**. The second FPGA includes **1002**, **1007**, **1013**, **1011** parts of **1005**. Referring to FIG. **11**, the unified look-up buffer **1103** is implemented as a single FPGA. State processor **1108** and part of state processor instruction database memory **1109** is another FPGA. Portions of the state processor instruction database memory **1109** are maintained in external SRAM's. The lookup/update engine **1107** and the flow insertion/deletion engine **1110** are in another FPGA. The sixth FPGA includes the cache system **1115**, the unified memory control **1119**, and the analyzer host interface and control **1118**.

Note that one can implement the system as one or more VSLI devices, rather than as a set of application specific integrated circuits (ASIC's) such as FPGA's. It is anticipated that in the future device densities will continue to increase, so that the complete system may eventually form a sub-unit (a "core") of a larger single chip unit.

Operation of the Invention

FIG. **15** shows how an embodiment of the network monitor **300** might be used to analyze traffic in a network **102**. Packet acquisition device **1502** acquires all the packets from a connection point **121** on network **102** so that all packets passing point **121** in either direction are supplied to monitor **300**. Monitor **300** comprises the parser sub-system **301**, which determines flow signatures, and analyzer sub-system **303** that analyzes the flow signature of each packet. A memory **324** is used to store the database of flows that are determined and updated by monitor **300**. A host computer **1504**, which might be any processor, for example, a general-purpose computer, is used to analyze the flows in memory **324**. As is conventional, host computer **1504** includes a memory, say RAM, shown as host memory **1506**. In addition, the host might contain a disk. In one application, the system can operate as an RMON probe, in which case the host computer is coupled to a network interface card **1510** that is connected to the network **102**.

The preferred embodiment of the invention is supported by an optional Simple Network Management Protocol (SNMP) implementation. FIG. **15** describes how one would, for example, implement an RMON probe, where a network interface card is used to send RMON information to the network. Commercial SNMP implementations also are available, and using such an implementation can simplify the process of porting the preferred embodiment of the invention to any platform.

In addition, MEB Compilers are available. An MIB Compiler is a tool that greatly simplifies the creation and maintenance of proprietary MIB extensions.

NOAC Ex. 1015 Page 654

Examples of Packet Elucidation

Monitor **300**, and in particular, analyzer **303** is capable of carrying out state analysis for packet exchanges that are commonly referred to as "server announcement" type exchanges. Server announcement is a process used to ease communications between a server with multiple applications that can all be simultaneously accessed from multiple clients. Many applications use a server announcement process as a means of multiplexing a single port or socket into many applications and services. With this type of exchange, messages are sent on the network, in either a broadcast or multicast approach, to announce a server and application, and all stations in the network may receive and decode these messages. The messages enable the stations to derive the appropriate connection point for communicating that particular application with the particular server. Using the server announcement method, a particular application communicates using a service channel, in the form of a TCP or UDP socket or port as in the IP protocol suite, or using a SAP as in the Novell IPX protocol suite.

The analyzer **303** is also capable of carrying out "in-stream analysis" of packet exchanges. The "in-stream analysis" method is used either as a primary or secondary recognition process. As a primary process, in-stream analysis assists in extracting detailed information which will be used to further recognize both the specific application and application component. A good example of in-stream analysis is any Web-based application. For example, the commonly used PointCast Web information application can be recognized using this process; during the initial connection between a PointCast server and client, specific key tokens exist in the data exchange that will result in a signature being generated to recognize PointCast.

The in-stream analysis process may also be combined with the server announcement process. In many cases in-stream analysis will augment other recognition processes. An example of combining in-stream analysis with server announcement can be found in business applications such as SAP and BAAN.

"Session tracking" also is known as one of the primary processes for tracking applications in client/server packet exchanges. The process of tracking sessions requires an initial connection to a predefined socket or port number. This method of communication is used in a variety of transport layer protocols. It is most commonly seen in the TCP and UDP transport protocols of the IP protocol.

During the session tracking, a client makes a request to a server using a specific port or socket number. This initial request will cause the server to create a TCP or UDP port to exchange the remainder of the data between the client and the server. The server then replies to the request of the client using this newly created port. The original port used by the client to connect to the server will never be used again during this data exchange.

One example of session tracking is TFTP (Trivial File Transfer Protocol), a version of the TCP/IP FTP protocol that has no directory or password capability. During the client/server exchange process of TFTP, a specific port (port number **69**) is always used to initiate the packet exchange. Thus, when the client begins the process of communicating, a request is made to UDP port **69**. Once the server receives this request, a new port number is created on the server. The server then replies to the client using the new port. In this example, it is clear that in order to recognize TFTP; network monitor **300** analyzes the initial request from the client and generates a signature for it. Monitor **300** uses that signature to recognize the reply. Monitor **300** also analyzes the reply from the server with the key port information, and uses this to create a signature for monitoring the remaining packets of this data exchange.

Network monitor **300** can also understand the current state of particular connections in the network. Connection-oriented exchanges often benefit from state tracking to correctly identify the application. An example is the common TCP transport protocol that provides a reliable means of sending information between a client and a server. When a data exchange is initiated, a TCP request for synchronization message is sent. This message contains a specific sequence number that is used to track an acknowledgement from the server. Once the server has acknowledged the synchronization request, data may be exchanged between the client and the server. When communication is no longer required, the client sends a finish or complete message to the server, and the server acknowledges this finish request with a reply containing the sequence numbers from the request. The states of such a connection-oriented exchange relate to the various types of connection and maintenance messages.

Server Announcement Example

The individual methods of server announcement protocols vary. However, the basic underlying process remains similar. A typical server announcement message is sent to one or more clients in a network. This type of announcement message has specific content, which, in another aspect of the invention, is salvaged and maintained in the database of flow-entries in the system. Because the announcement is sent to one or more stations, the client involved in a future packet exchange with the server will make an assumption that the information announced is known, and an aspect of the inventive monitor is that it too can make the same assumption.

Sun-RPC is the implementation by Sun Microsystems, Inc. (Palo Alto, Calif.) of the Remote Procedure Call (RPC), a programming interface that allows one program to use the services of another on a remote machine. A Sun-RPC example is now used to explain how monitor **300** can capture server announcements.

A remote program or client that wishes to use a server or procedure must establish a connection, for which the RPC protocol can be used.

Each server running the Sun-RPC protocol must maintain a process and database called the port Mapper. The port Mapper creates a direct association between a Sun-RPC program or application and a TCP or UDP socket or port (for TCP or UDP implementations). An application or program number is a 32-bit unique identifier assigned by ICANN (the Internet Corporation for Assigned Names and Numbers, www.icann.org), which manages the huge number of parameters associated with Internet protocols (port numbers, router protocols, multicast addresses, etc.) Each port Mapper on a Sun-RPC server can present the mappings between a unique program number and a specific transport socket through the use of specific request or a directed announcement. According to ICANN, port number **111** is associated with Sun RPC.

As an example, consider a client (e.g., CLIENT **3** shown as **106** in FIG. **1**) making a specific request to the server (e.g., SERVER **2** of FIG. **1**, shown as **110**) on a predefined UDP or TCP socket. Once the port Mapper process on the sun RPC server receives the request, the specific mapping is returned in a directed reply to the client.

1. A client (CLIENT **3**, **106** in FIG. **1**) sends a TCP packet to SERVER **2** (**110** in FIG. **1**) on port **111**.

Lookup Request (rpcBindLookup). TCP or UDP port **111** is always associated Sun RPC. This request specifies the program (as a program identifier), version, and might specify the protocol (UDP or TCP).

2. The server SERVER **2** (**110** in FIG. **1**) extracts the program identifier and version identifier from the request. The server also uses the fact that this packet came in using the TCP transport and that no protocol was specified, and thus will use the TCP protocol for its reply.

3. The server **110** sends a TCP packet to port number **111**, with an RPC Bind Lookup Reply. The reply contains the specific port number (e.g., port number 'port') on which future transactions will be accepted for the specific RPC program identifier (e.g., Program 'program') and the protocol (UDP or TCP) for use.

It is desired that from now on every time that port number 'port' is used, the packet is associated with the application program 'program' until the number 'port' no longer is to be associated with the program 'program'. Network monitor **300** by creating a flow-entry and a signature includes a mechanism for remembering the exchange so that future packets that use the port number 'port' will be associated by the network monitor with the application program 'program'.

In addition to the Sun RPC Bind Lookup request and reply, there are other ways that a particular program—say 'program'-might be associated with a particular port number, for example number 'port'. One is by a broadcast announcement of a particular association between an application service and a port number, called a Sun RPC portMapper Announcement. Another, is when some server—say the same SERVER **2**—replies to some client—say CLIENT **1**—requesting some portMapper assignment with a RPC portMapper Reply. Some other client—say CLIENT **2**—might inadvertently see this request, and thus know that for this particular server, SERVER **2**, port number 'port' is associated with the application service 'program', It is desirable for the network monitor **300** to be able to associate any packets to SERVER **2** using port number 'port' with the application program 'program',

FIG. **9** represents a dataflow **900** of some operations in the monitor **300** of FIG. **3** for Sun Remote Procedure Call. Suppose a client **106** (e.g., CLIENT **3** in FIG. **1**) is communicating via its interface to the network **118** to a server **110** (e.g., SERVER **2** in FIG. **1**) via the server's interface to the network **116**. Further assume that Remote Procedure Call is used to communicate with the server **110**. One path in the data flow **900** starts with a step **910** that a Remote Procedure Call bind lookup request is issued by client **106** and ends with the server state creation step **904**. Such RPC bind lookup request includes values for the 'program,' 'version,' and 'protocol' to use, e.g., TCP or UDP. The process for Sun RPC analysis in the network monitor **300** includes the following aspects.:

Process **909**: Extract the 'program,' 'version,' and 'protocol' (UDP or TCP).

Extract the TCP or UDP port (process **909**) which is **111** indicating Sun RPC.

Process **908**: Decode the Sun RPC packet. Check RPC type field for ID. If value is portMapper, save paired socket (i.e., dest for destination address, src for source address). Decode ports and mapping, save ports with socket/addr key. There may be more than one pairing per mapper packet. Form a signature (e.g., a key). A flow-entry is created in database **324**. The saving of the request is now complete.

At some later time, the server (process **907**) issues a RPC bind lookup reply. The packet monitor **300** will extract a signature from the packet and recognize it from the previously stored flow. The monitor will get the protocol port number (**906**) and lookup the request (**905**). A new signature (i.e., a key) will be created and the creation of the server state (**904**) will be stored as an entry identified by the new signature in the flow-entry database. That signature now may be used to identify packets associated with the server.

The server state creation step **904** can be reached not only from a Bind Lookup Request/Reply pair, but also from a RPC Reply portMapper packet shown as **901** or an RPC Announcement portMapper shown as **902**. The Remote Procedure Call protocol can announce that it is able to provide a particular application service. Embodiments of the present invention preferably can analyze when an exchange occurs between a client and a server, and also can track those stations that have received the announcement of a service in the network.

The RPC Announcement portMapper announcement **902** is a broadcast. Such causes various clients to execute a similar set of operations, for example, saving the information obtained from the announcement. The RPC Reply portMapper step **901** could be in reply to a portMapper request, and is also broadcast. It includes all the service parameters.

Thus monitor **300** creates and saves all such states for later classification of flows that relate to the particular service 'program',

FIG. **2** shows how the monitor **300** in the example of Sun RPC builds a signature and flow states. A plurality of packets **206–209** are exchanged, e.g., in an exemplary Sun Microsystems Remote Procedure Call protocol. A method embodiment of the present invention might generate a pair of flow signatures, "signature-**1**" **210** and "signature-**2**" **212**, from information found in the packets **206** and **207** which, in the example, correspond to a Sun RPC Bind Lookup request and reply, respectively.

Consider first the Sun RPC Bind Lookup request. Suppose packet **206** corresponds to such a request sent from CLIENT **3** to SERVER **2**. This packet contains important information that is used in building a signature according to an aspect of the invention. A source and destination network address occupy the first two fields of each packet, and according to the patterns in pattern database **308**, the flow signature (shown as KEY**1** **230** in FIG. **2**) will also contain these two fields, so the parser subsystem **301** will include these two fields in signature KEY **1** (**230**). Note that in FIG. **2**, if an address identifies the client **106** (shown also as **202**), the label used in the drawing is "C_1". If such address identifies the server **110** (shown also as server **204**), the label used in the drawing is "S_1". The first two fields **214** and **215** in packet **206** are "S_1" and "C_1" because packet **206** is provided from the server **110** and is destined for the client **106**. Suppose for this example, "S_1" is an address numerically less than address "C_1". A third field "p^1" **216** identifies the particular protocol being used, e.g., TCP, UDP, etc.

In packet **206**, a fourth field **217** and a fifth field **218** are used to communicate port numbers that are used. The conversation direction determines where the port number field is. The diagonal pattern in field **217** is used to identify a source-port pattern, and the hash pattern in field **218** is used to identify the destination-port pattern. The order indicates the client-server message direction. A sixth field denoted "i^1" **219** is an element that is being requested by the client from the server. A seventh field denoted "s_1a" **220** is the service requested by the client from server **110**. The

NOAC Ex. 1015 Page 656

following eighth field "QA" **221** (for question mark) indicates that the client **106** wants to know what to use to access application "s_1a". A tenth field "QP" **223** is used to indicate that the client wants the server to indicate what protocol to use for the particular application.

Packet **206** initiates the sequence of packet exchanges, e.g., a RPC Bind Lookup Request to SERVER **2**. It follows a well-defined format, as do all the packets, and is transmitted to the server **110** on a well-known service connection identifier (port **111** indicating Sun RPC).

Packet **207** is the first sent in reply to the client **106** from the server. It is the RPC Bind Lookup Reply as a result of the request packet **206**.

Packet **207** includes ten fields **224–233**. The destination and source addresses are carried in fields **224** and **225**, e.g., indicated "C_1" and "S_1", respectively. Notice the order is now reversed, since the client-server message direction is from the server **110** to the client **106**. The protocol "p^1" is used as indicated in field **226**. The request "i^1" is in field **229**. Values have been filled in for the application port number, e.g., in field **233** and protocol "p^2" in field **233**.

The flow signature and flow states built up as a result of this exchange are now described. When the packet monitor **300** sees the request packet **206** from the client, a first flow signature **210** is built in the parser subsystem **301** according to the pattern and extraction operations database **308**. This signature **210** includes a destination and a source address **240** and **241**. One aspect of the invention is that the flow keys are built consistently in a particular order no matter what the direction of conversation. Several mechanisms may be used to achieve this. In the particular embodiment, the numerically lower address is always placed before the numerically higher address. Such least to highest order is used to get the best spread of signatures and hashes for the lookup operations. In this case, therefore, since we assume "S_1"<"C_1", the order is address "S_1" followed by client address "C_1". The next field used to build the signature is a protocol field **242** extracted from packet **206**'s field **216**, and thus is the protocol "p^1". The next field used for the signature is field **243**, which contains the destination source port number shown as a crosshatched pattern from the field **218** of the packet **206**. This pattern will be recognized in the payload of packets to derive how this packet or sequence of packets exists as a flow. In practice, these may be TCP port numbers, or a combination of TCP port numbers. In the case of the Sun RPC example, the crosshatch represents a set of port numbers of UDS for p^1 that will be used to recognize this flow (e.g., port **111**). Port **111** indicates this is Sun RPC. Some applications, such as the Sun RPC Bind Lookups, are directly determinable ("known") at the parser level. So in this case, the signature KEY-1 points to a known application denoted "a^1" (Sun RPC Bind Lookup), and a next—state that the state processor should proceed to for more complex recognition jobs, denoted as state "st_D" is placed in the field **245** of the flow-entry.

When the Sun RPC Bind Lookup reply is acquired, a flow signature is again built by the parser. This flow signature is identical to KEY-1. Hence, when the signature enters the analyzer subsystem **303** from the parser subsystem **301**, the complete flow-entry is obtained, and in this flow-entry indicates state "st_D". The operations for state "st_D" in the state processor instruction database **326** instructs the state processor to build and store a new flow signature, shown as KEY-2 (**212**) in FIG. **2**. This flow signature built by the state processor also includes the destination and a source addresses **250** and **251**, respectively, for server "S_1" followed by (the numerically higher address) client "C_1". A protocol field **252** defines the protocol to be used, e.g., "p^2" which is obtained from the reply packet. A field **253** contains a recognition pattern also obtained from the reply packet. In this case, the application is Sun RPC, and field **254** indicates this application "a^2". A next-state field **255** defines the next state that the state processor should proceed to for more complex recognition jobs, e.g., a state "st^1". In this particular example, this is a final state. Thus, KEY-2 may now be used to recognize packets that are in any way associated with the application "a^2". Two such packets **208** and **209** are shown, one in each direction. They use the particular application service requested in the original Bind Lookup Request, and each will be recognized because the signature KEY-2 will be built in each case.

The two flow signatures **210** and **212** always order the destination and source address fields with server "S_1" followed by client "C_1". Such values are automatically filled in when the addresses are first created in a particular flow signature. Preferably, large collections of flow signatures are kept in a lookup table in a least-to-highest order for the best spread of flow signatures and hashes.

Thereafter, the client and server exchange a number of packets, e.g., represented by request packet **208** and response packet **209**. The client **106** sends packets **208** that have a destination and source address S_1 and C_1, in a pair of fields **260** and **261**. A field **262** defines the protocol as "p^2", and a field **263** defines the destination port number.

Some network-server application recognition jobs are so simple that only a single state transition has to occur to be able to pinpoint the application that produced the packet.

Others require a sequence of state transitions to occur in order to match a known and predefined climb from state-to-state.

Thus the flow signature for the recognition of application "a^2" is automatically set up by predefining what packet-exchange sequences occur for this example when a relatively simple Sun Microsystems Remote Procedure Call bind lookup request instruction executes. More complicated exchanges than this may generate more than two flow signatures and their corresponding states. Each recognition may involve setting up a complex state transition diagram to be traversed before a "final" resting state such as "st_1" in field **255** is reached. All these are used to build the final set of flow signatures for recognizing a particular application in the future.

Embodiments of the present invention automatically generate flow signatures with the necessary recognition patterns and state transition climb procedure. Such comes from analyzing packets according to parsing rules, and also generating state transitions to search for. Applications and protocols, at any level, are recognized through state analysis of sequences of packets.

Note that one in the art will understand that computer networks are used to connect many different types of devices, including network appliances such as telephones, "Internet" radios, pagers, and so forth. The term computer as used herein encompasses all such devices and a computer network as used herein includes networks of such computers.

Although the present invention has been described in terms of the presently preferred embodiments, it is to be understood that the disclosure is not to be interpreted as limiting. Various alterations and modifications will no doubt become apparent to those or ordinary skill in the art after having read the above disclosure. Accordingly, it is intended that the claims be interpreted as covering all alterations and modifications as fall within the true spirit and scope of the present invention.

NOAC Ex. 1015 Page 657

What is claimed is:

1. A packet monitor for examining packets passing through a connection point on a computer network in real-time, the packets provided to the packet monitor via a packet acquisition device connected to the connection point, the packet monitor comprising:

(a) a packet-buffer memory configured to accept a packet from the packet acquisition device;

(b) a parsing/extraction operations memory configured to store a database of parsing/extraction operations that includes information describing how to determine at least one of the protocols used in a packet from data in the packet;

(c) a parser subsystem coupled to the packet buffer and to the pattern/extraction operations memory, the parser subsystem configured to examine the packet accepted by the buffer, extract selected portions of the accepted packet, and form a function of the selected portions sufficient to identify that the accepted packet is part of a conversational flow-sequence;

(d) a memory storing a flow-entry database including a plurality of flow-entries for conversational flows encountered by the monitor;

(e) a lookup engine connected to the parser subsystem and to the flow-entry database, and configured to determine using at least some of the selected portions of the accepted packet if there is an entry in the flow-entry database for the conversational flow sequence of the accepted packet;

(f) a state patterns/operations memory configured to store a set of predefined state transition patterns and state operations such that traversing a particular transition pattern as a result of a particular conversational flow-sequence of packets indicates that the particular conversational flow-sequence is associated with the operation of a particular application program, visiting each state in a traversal including carrying out none or more predefined state operations;

(g) a protocol/state identification mechanism coupled to the state patterns/operations memory and to the lookup engine, the protocol/state identification engine configured to determine the protocol and state of the conversational flow of the packet; and

(h) a state processor coupled to the flow-entry database, the protocol/state identification engine, and to the state patterns/operations memory, the state processor, configured to carry out any state operations specified in the state patterns/operations memory for the protocol and state of the flow of the packet,

the carrying out of the state operations furthering the process of identifying which application program is associated with the conversational flow-sequence of the packet, the state processor progressing through a series of states and state operations until there are no more state operations to perform for the accepted packet, in which case the state processor updates the flow-entry, or until a final state is reached that indicates that no more analysis of the flow is required, in which case the result of the analysis is announced.

2. A packet monitor according to claim 1, wherein the flow-entry includes the state of the flow, such that the protocol/state identification mechanism determines the state of the packet from the flow-entry in the case that the lookup engine finds a flow-entry for the flow of the accepted packet.

3. A packet monitor according to claim 1, wherein the parser subsystem includes a mechanism for building a hash from the selected portions, and wherein the hash is used by the lookup engine to search the flow-entry database, the hash designed to spread the flow-entries across the flow-entry database.

4. A packet monitor according to claim 1, further comprising:

a compiler processor coupled to the parsing/extraction operations memory, the compiler processor configured to run a compilation process that includes:

receiving commands in a high-level protocol description language that describe the protocols that may be used in packets encountered by the monitor, and

translating the protocol description language commands into a plurality of parsing/extraction operations that are initialized into the parsing/extraction operations memory.

5. A packet monitor according to claim 4, wherein the protocol description language commands also describe a correspondence between a set of one or more application programs and the state transition patterns/operations that occur as a result of particular conversational flow-sequences associated with an application program, wherein the compiler processor is also coupled to the state patterns/operations memory, and wherein the compilation process further includes translating the protocol description language commands into a plurality of state patterns and state operations that are initialized into the state patterns/operations memory.

6. A packet monitor according to claim 1, further comprising:

a cache memory coupled to and between the lookup engine and the flow-entry database providing for fast access of a set of likely-to-be-accessed flow-entries from the flow-entry database.

7. A packet monitor according to claim 6, wherein the cache functions as a fully associative, least-recently-used cache memory.

8. A packet monitor according to claim 7, wherein the cache functions as a fully associative, least-recently-used cache memory and includes content addressable memories configured as a stack.

9. A packet monitor according to claim 1, wherein one or more statistical measures about a flow are stored in each flow-entry, the packet monitor further comprising:

a calculator for updating the statistical measures in a flow-entry of the accepted packet.

10. A packet monitor according to claim 9, wherein, when the application program of a flow is determined, one or more network usage metrics related to said application and determined from the statistical measures are presented to a user for network performance monitoring.

* * * * *

PRINT OF DRAWINGS
AS ORIGINALLY FILED

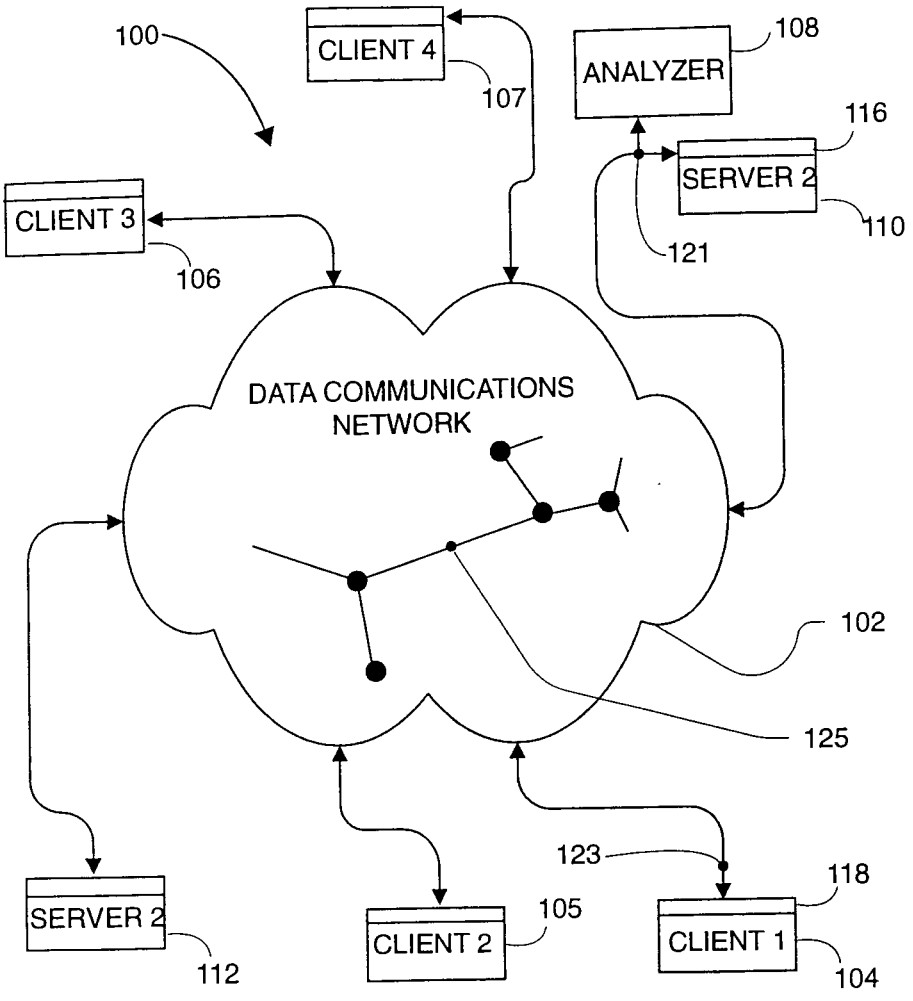


FIG. 1

AS ORIGINALLY FILED 001-1

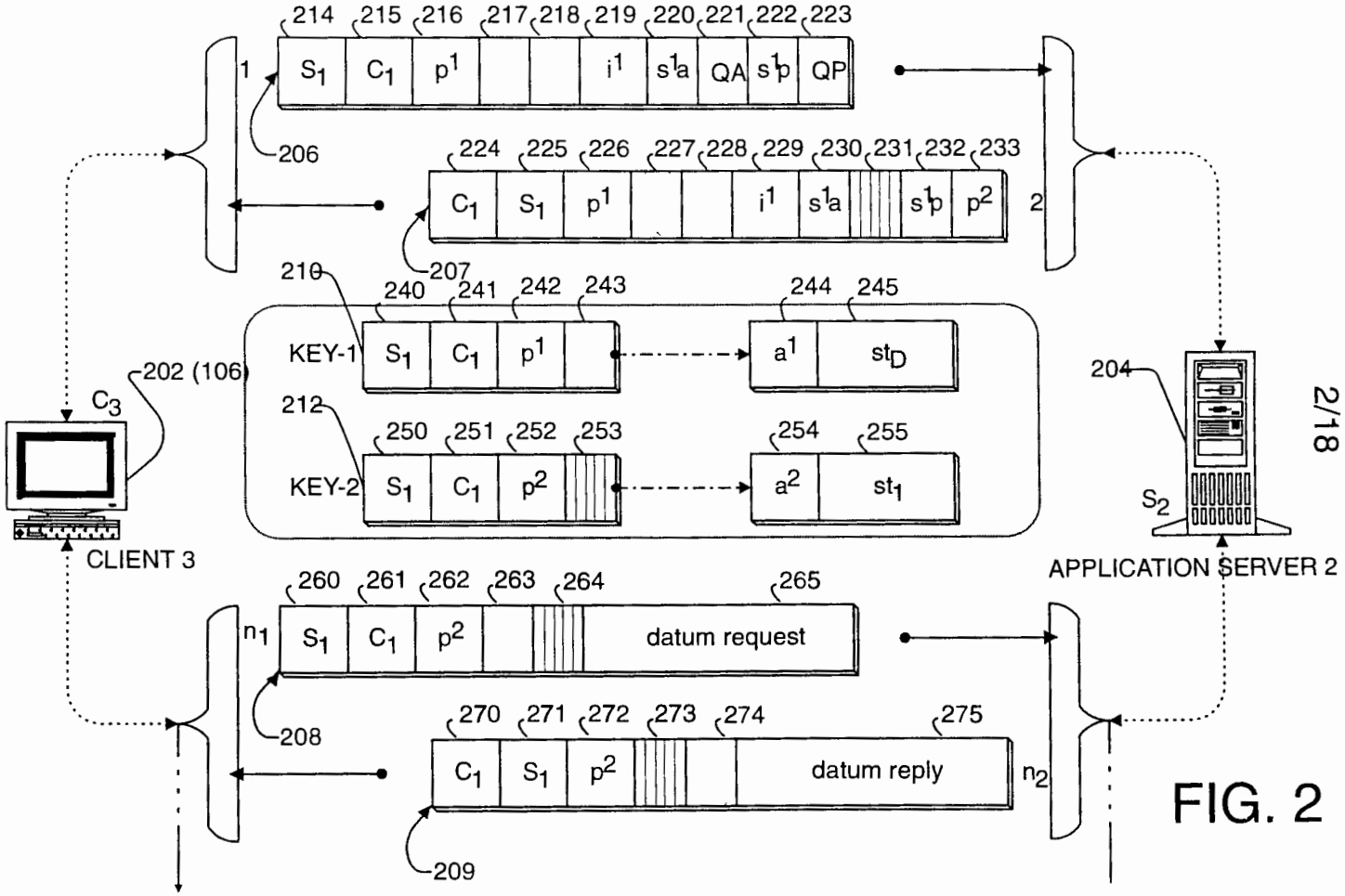


FIG. 2

2/18

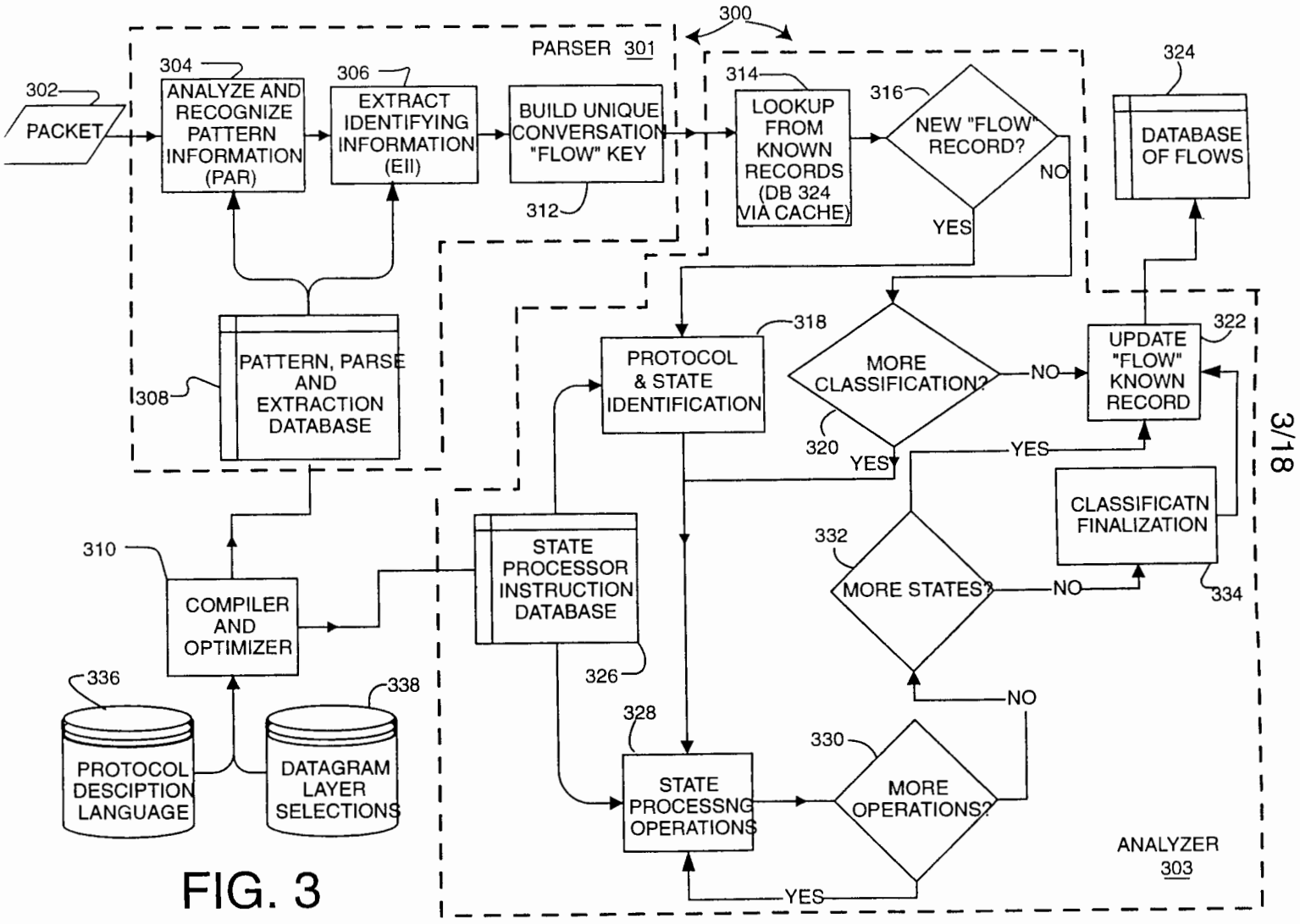


FIG. 3

3/18

PRINT OF DRAWINGS AS ORIGINALLY FILED

PRINT OF DRAWINGS
AS ORIGINALLY FILED

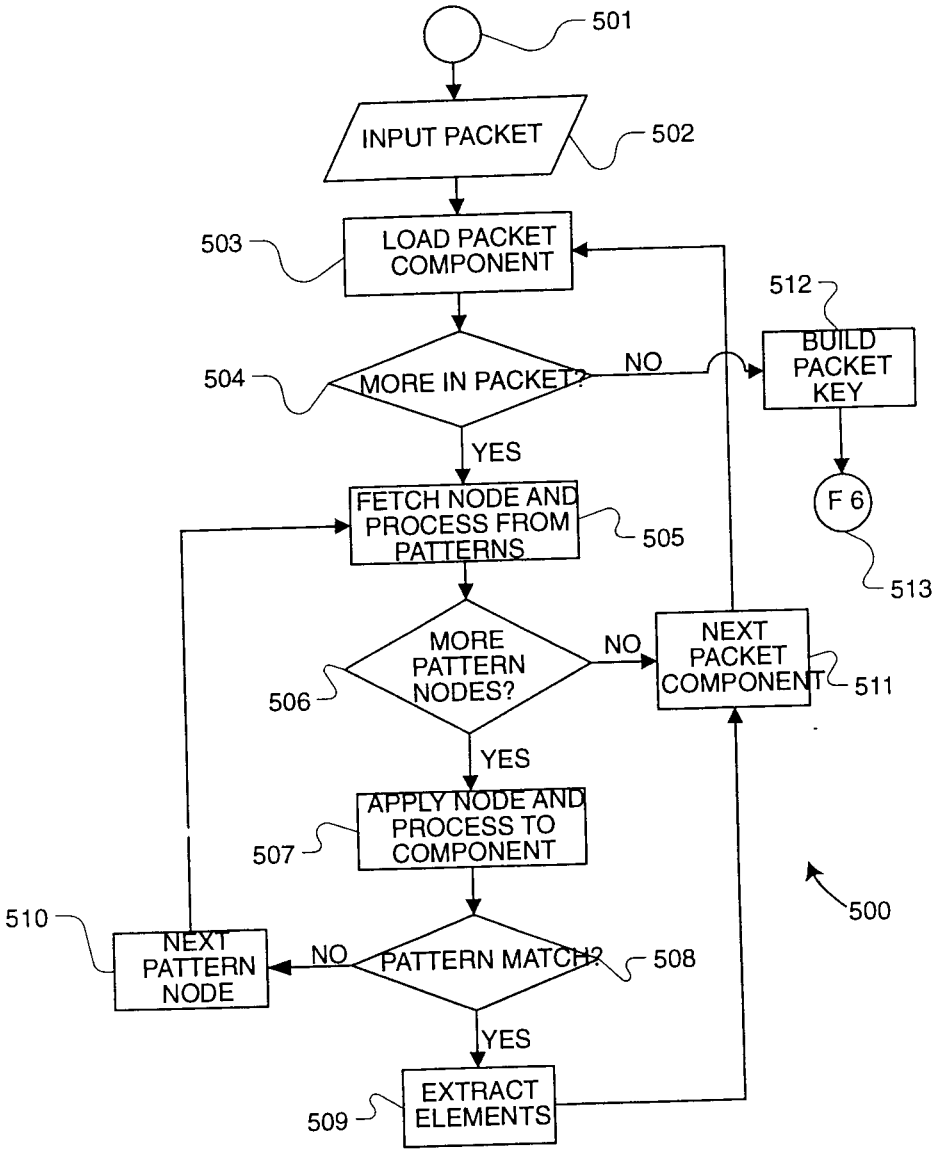


FIG. 5

PRINT OF DRAWINGS
AS ORIGINALLY FILED 001-1

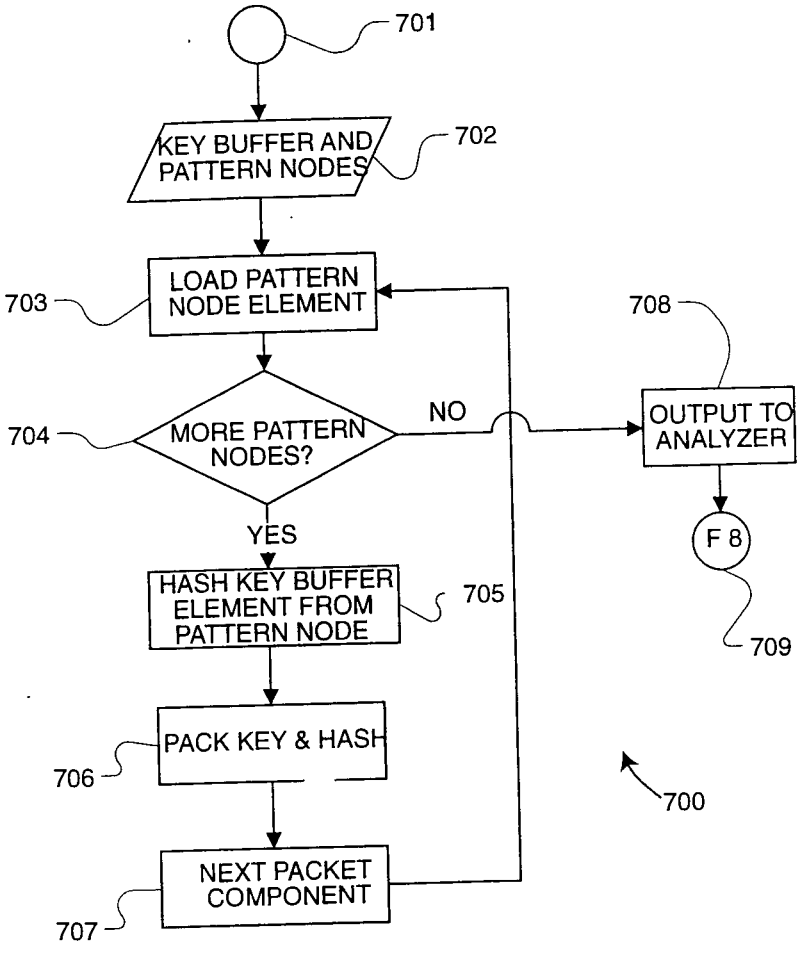


FIG. 7

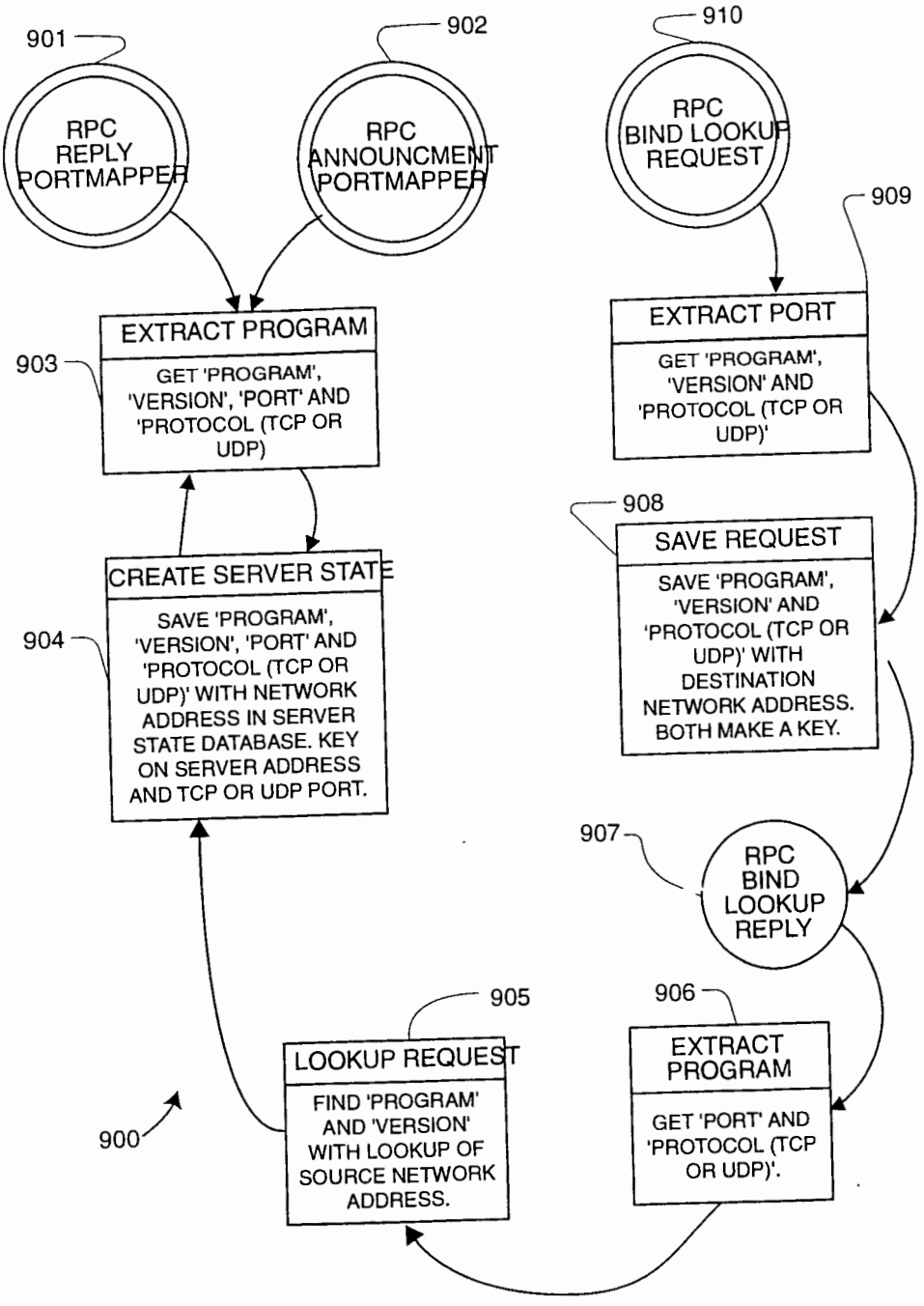

901
RPC REPLY PORTMAPPER
902
RPC ANNOUNCMENT PORTMAPPER
910
RPC BIND LOOKUP REQUEST
909
EXTRACT PROGRAM
GET 'PROGRAM', 'VERSION', 'PORT' AND 'PROTOCOL (TCP OR UDP)
903
EXTRACT PORT
GET 'PROGRAM', 'VERSION' AND 'PROTOCOL (TCP OR UDP)'
908
SAVE REQUEST
SAVE 'PROGRAM', 'VERSION' AND 'PROTOCOL (TCP OR UDP)' WITH DESTINATION NETWORK ADDRESS. BOTH MAKE A KEY.
CREATE SERVER STATE
904
SAVE 'PROGRAM', 'VERSION', 'PORT' AND 'PROTOCOL (TCP OR UDP)' WITH NETWORK ADDRESS IN SERVER STATE DATABASE. KEY ON SERVER ADDRESS AND TCP OR UDP PORT.
907
RPC BIND LOOKUP REPLY
905
906
LOOKUP REQUEST
FIND 'PROGRAM' AND 'VERSION' WITH LOOKUP OF SOURCE NETWORK ADDRESS.
EXTRACT PROGRAM
GET 'PORT' AND 'PROTOCOL (TCP OR UDP)'.
900

FIG. 9

PRINT OF DRAWINGS
AS ORIGINALLY FILED

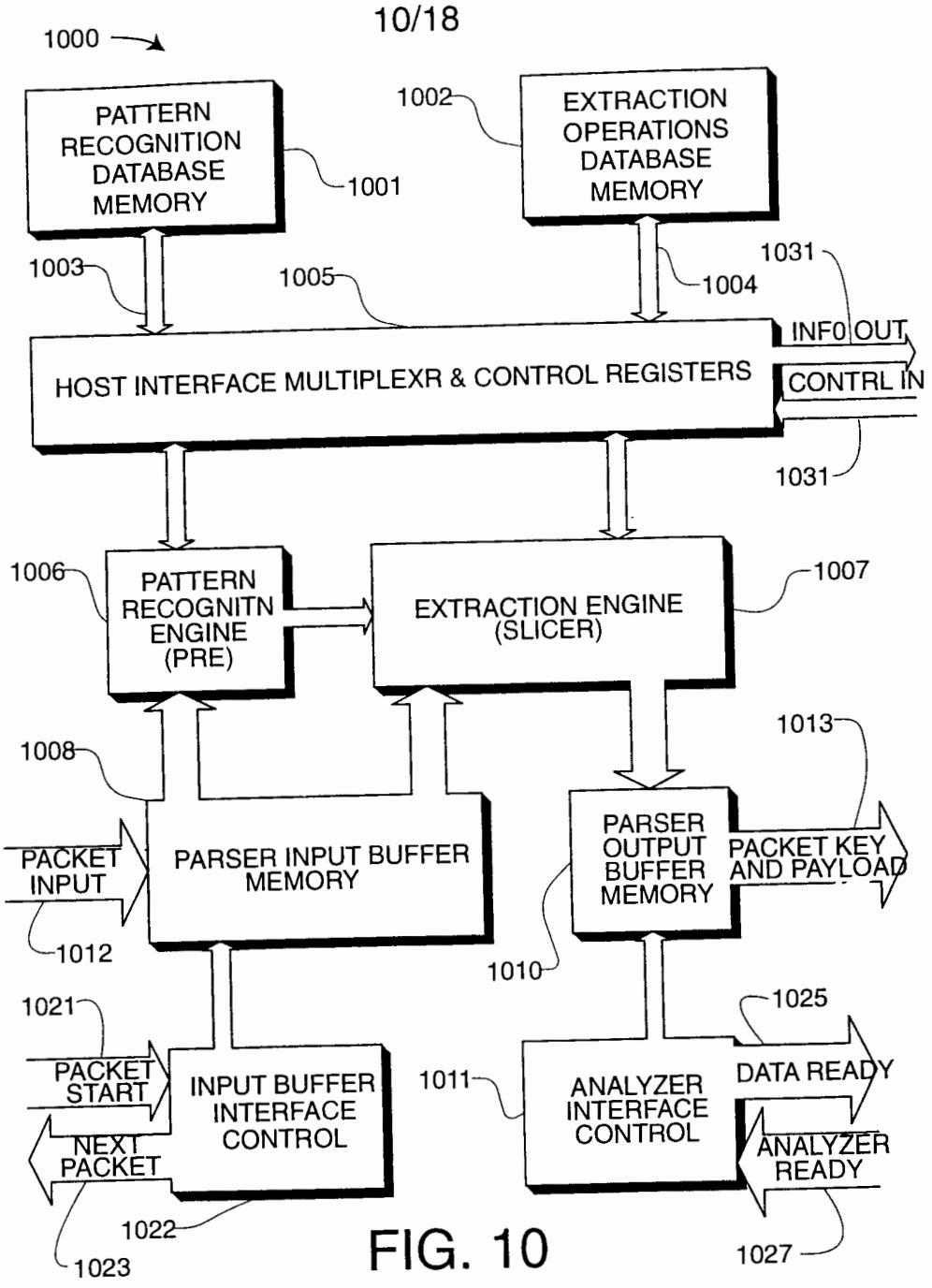


FIG. 10

PRINT OF DRAWINGS
AS ORIGINALLY FILED 01-1

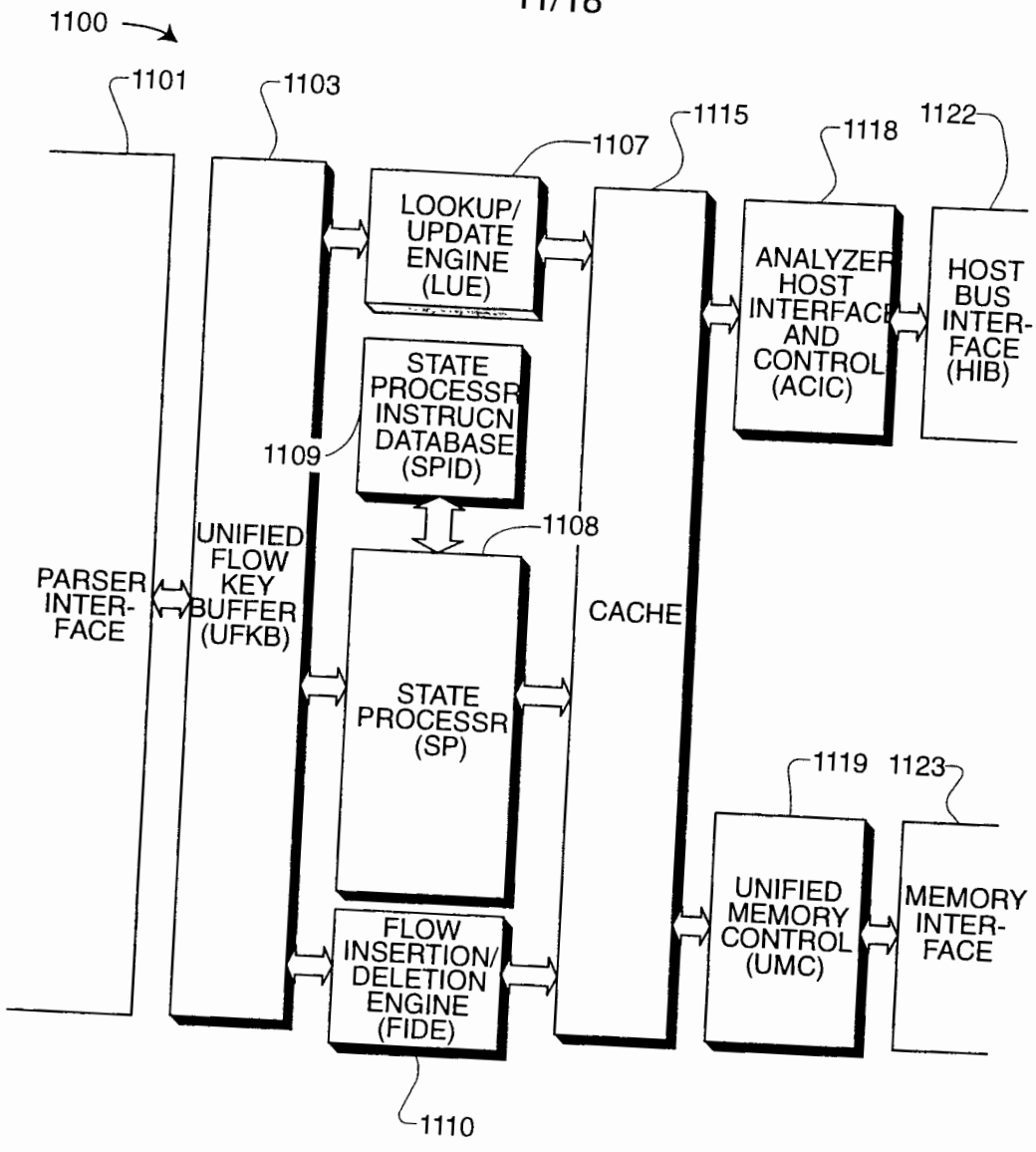


FIG. 11

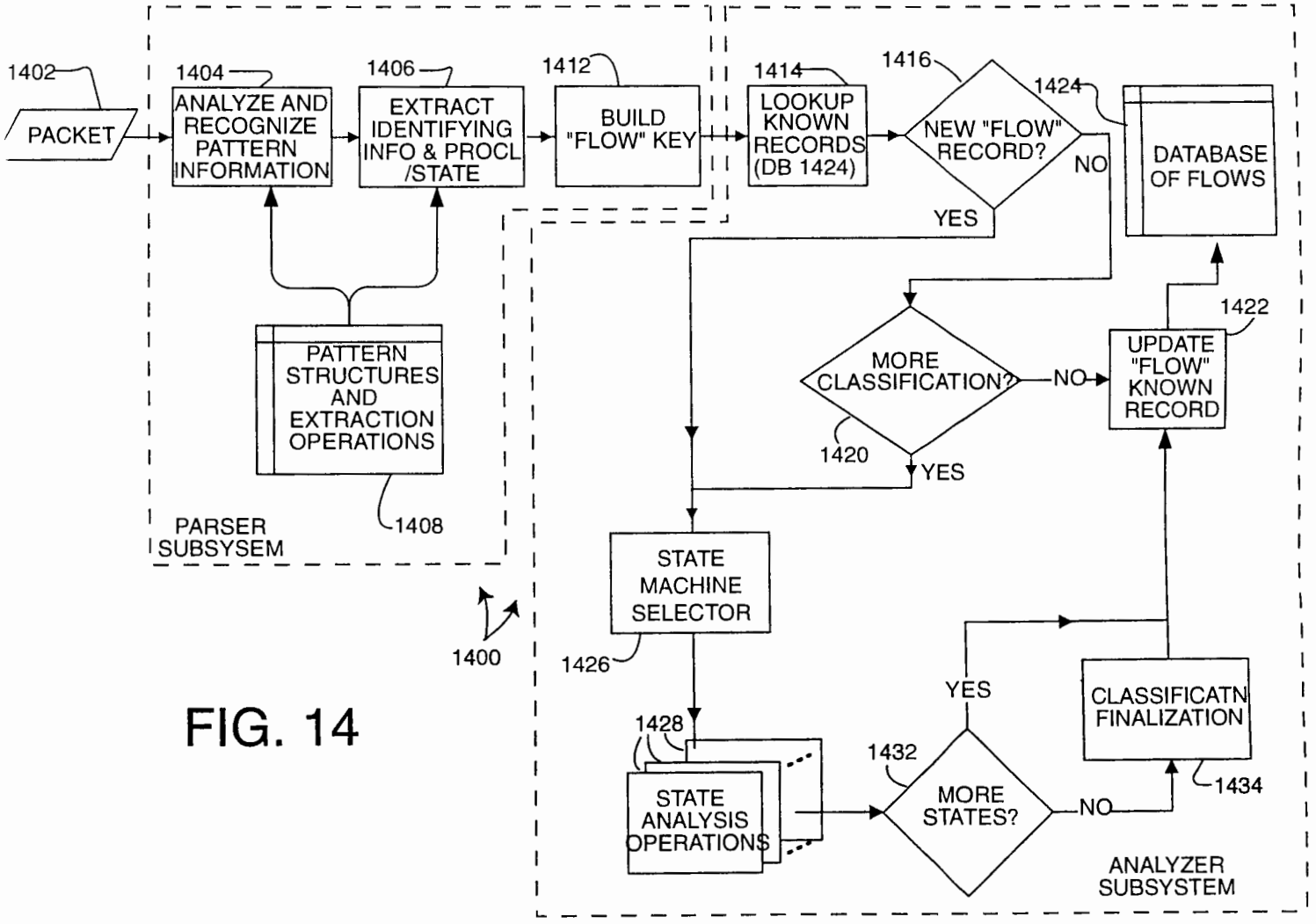


FIG. 14

PRINT OF DRAWINGS AS ORIGINALLY FILED

NOAC Ex. 1015 Page 667

PRINT OF DRAWINGS
AS ORIGINALLY FILED 101-1

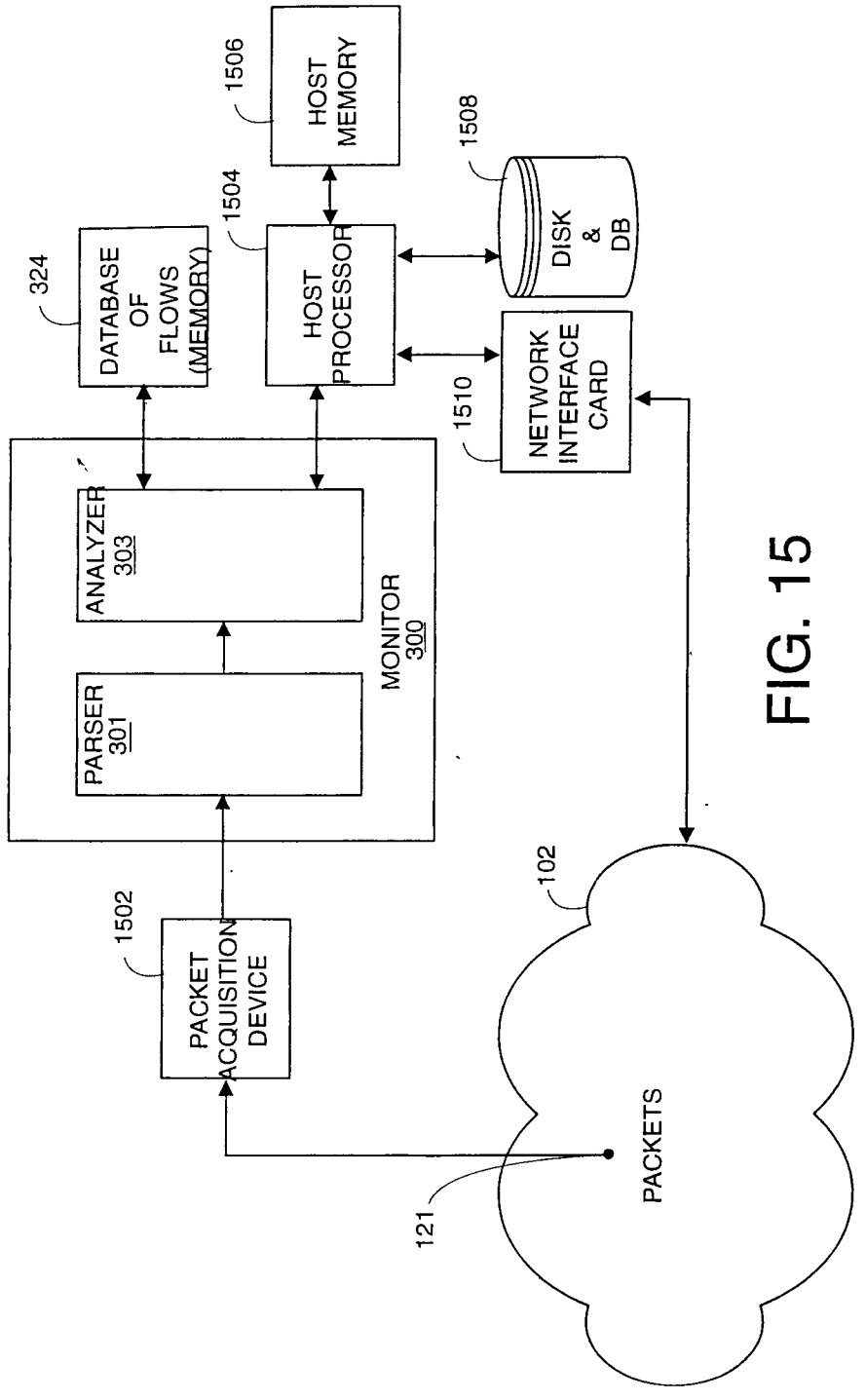


FIG. 15

PRINT OF DRAWINGS
AS ORIGINALLY FILED 001-1

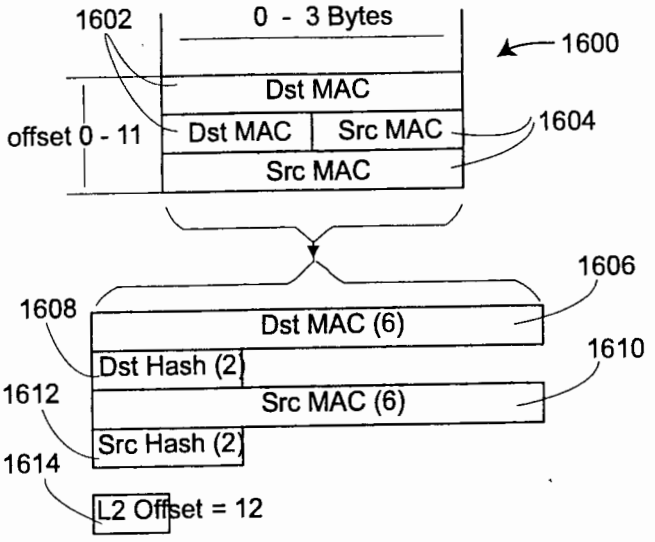


FIG. 16

PRINT OF DRAWINGS AS ORIGINALLY FILED

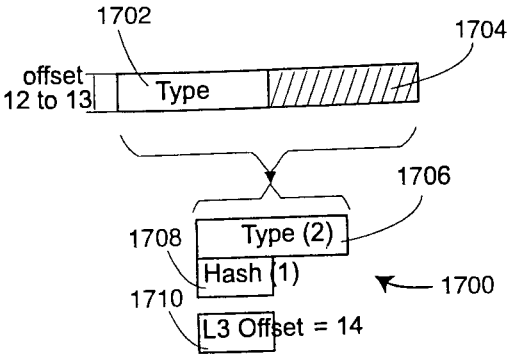


FIG. 17A

1712

IDP = 0x0600*
IP = 0x0800*
CHAOSNET = 0x0804
ARP = 0x0806
VIP = 0x0BAD*
VLOOP = 0x0BAE
VECHO = 0x0BAF
NETBIOS-3COM = 0x3C00 - 0x3C0D#
DEC-MOP = 0x6001
DEC-RC = 0x6002
DEC-DRP = 0x6003*
DEC-LAT = 0x6004
DEC-DIAG = 0x6005
DEC-LAVC = 0x6007
RARP = 0x8035
ATALK = 0x809B*
VLOOP = 0x80C4
VECHO = 0x80C5
SNA-TH = 0x80D5*
ATALKARP = 0x80F3
IPX = 0x8137*
SNMP = 0x814C#
IPv6 = 0x86DD*
LOOPBACK = 0x9000

Apple = 0x080007

* L3 Decoding
# L5 Decoding

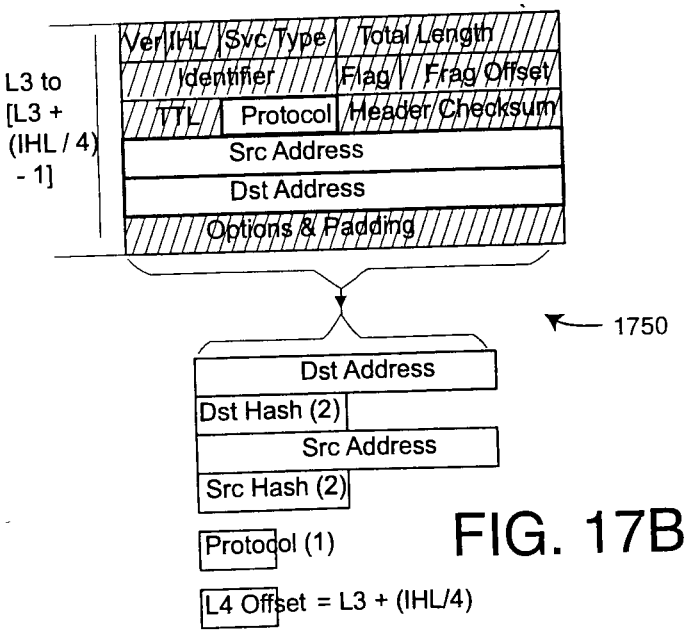


FIG. 17B

1752

ICMP = 1
IGMP = 2
GGP = 3
TCP = 6*
EGP = 8
IGRP = 9
PUP = 12
CHAOS = 16
UDP = 17*
IDP = 22#
ISO-TP4 = 29
DDP = 37#
ISO-IP = 80
VIP = 83#
EIGRP = 88
OSPF = 89

* L4 Decoding
# L3 Re-Decoding

PRINT OF DRAWINGS
AS ORIGINALLY FILED -001-1

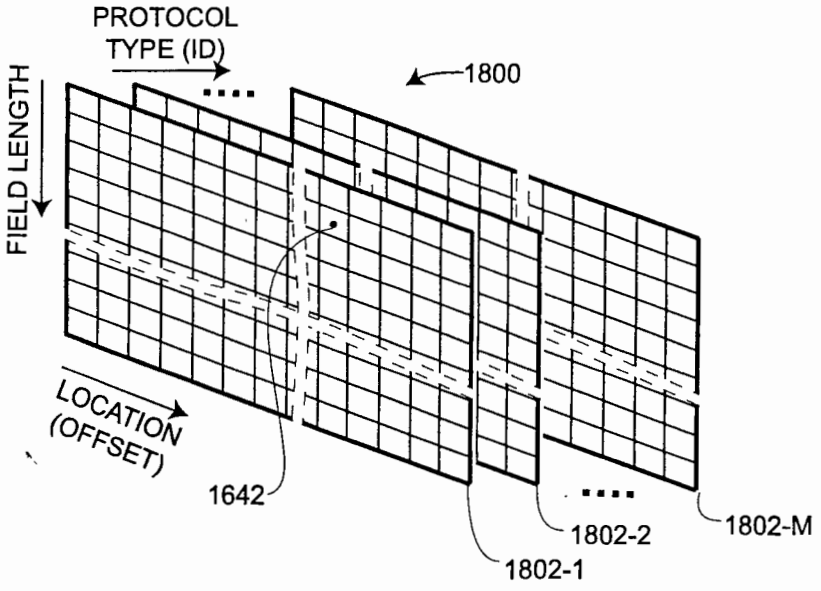


FIG. 18A

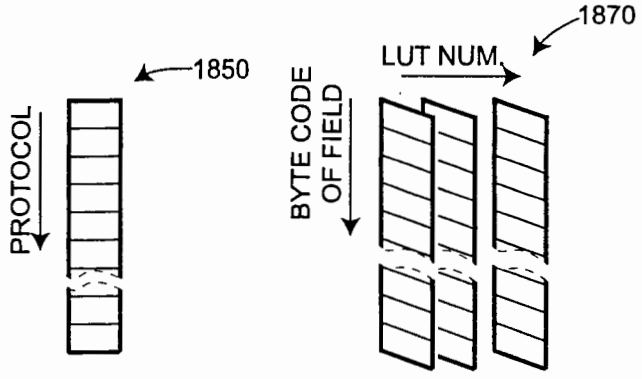


FIG. 18B

__X__ The Commissioner is hereby authorized to charge payment of any missing fees associated with this communication or credit any overpayment to Deposit Account No. 50-0292

(A DUPLICATE OF THIS TRANSMITTAL IS ATTACHED):

Respectfully Submitted,

Nov 1, 2000
Date

Dov Rosenfeld, Reg. No. 38687

Address for correspondence:
Dov Rosenfeld
5507 College Avenue, Suite 2
Oakland, CA 94618
Tel. (510) 547-3378; Fax: (510) 653-7992

Our Docket/Ref. No.: APPT-C[illegible]-1 Patent

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

OIPE
APR 12 2001
PATENT & TRADEMARK OFFICE

Applicant(s): Dietz et al. Serial No.: 09/608237 Filed: June 30, 2000 Title: METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK	Group Art Unit: 2755 Examiner:

RECEIVED
APR 16 2001
Technology Center 2100

Commissioner for Patents
Washington, D.C. 20231

TRANSMITTAL: INFORMATION DISCLOSURE STATEMENT

Dear Commissioner:

Transmitted herewith are:

X An Information Disclosure Statement for the above referenced patent application, together with PTO form 1449 and a copy of each reference cited in form 1449.

___ A check for petition fees.

X Return postcard.

X The commissioner is hereby authorized to charge payment of any missing fee associated with this communication or credit any overpayment to Deposit Account 50-0292.

A DUPLICATE OF THIS TRANSMITTAL IS ATTACHED

Respectfully submitted,

Date: April 9, 2001

Dov Rosenfeld
Attorney/Agent for Applicant(s)
Reg. No. 38687

Correspondence Address:
Dov Rosenfeld
5507 College Avenue, Suite 2
Oakland, CA 94618
Telephone No.: +1-510-547-3378

Certificate of Mailing under 37 CFR 1.18

I hereby certify that this correspondence is being deposited with the United States Postal Service as first class mail in an envelope addressed to: Commissioner for Patents, Washington, D.C. 20231.

Date of Deposit: Apr 9, 2001

Signature: _____

Dov Rosenfeld, Reg No 38,687

OIPE SC42
APR 18 2002
PATENT & TRADEMARK OFFICE

Our Docket/Ref. No.: APPT-00 Patent

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant(s): Dietz et al. Serial No.: 09/608237 Filed: June 30, 2000 Title: METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK	Group Art Unit: 2755 Examiner:

RECEIVED
APR 22 2002
Technology Center 2100

Commissioner for Patents
Washington, D.C. 20231

TRANSMITTAL: INFORMATION DISCLOSURE STATEMENT

Dear Commissioner:

Transmitted herewith are:

X An Information Disclosure Statement for the above referenced patent application, together with PTO form 1449 and a copy of each reference cited in form 1449.

___ A check for petition fees.

X Return postcard.

X The commissioner is hereby authorized to charge payment of any missing fee associated with this communication or credit any overpayment to Deposit Account 50-0292.

A DUPLICATE OF THIS TRANSMITTAL IS ATTACHED

Respectfully submitted,

Date: 30 Mar 2002

Dov Rosenfeld
Attorney/Agent for Applicant(s)
Reg. No. 38687

Correspondence Address:
Dov Rosenfeld
5507 College Avenue, Suite 2
Oakland, CA 94618
Telephone No.: +1-510-547-3378

Certificate of Mailing under 37 CFR 1.18

I hereby certify that this correspondence is being deposited with the United States Postal Service as first class mail in an envelope addressed to: Commissioner for Patents, Washington, D.C. 20231.

Date of Deposit: 30 Mar 2002 Signature: _____
Dov Rosenfeld, Reg. No. 38,687

INS
A1

OIPE JC160 FEB 27 2004 PATENT & TRADEMARK OFFICE

Our Ref./Docket No.: APPT-001-1 Patent

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Inventor(s): Dietz, *et al.* Assignee: Hi/fn, Inc. Patent No: 6651099 Issue Date: November, 18, 2003 Application No.: 09/608237 Filed: June 30, 2000 Title: METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK	

REQUEST FOR CERTIFICATE OF CORRECTIONS

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Dear Commissioner:

The above patent contains significant error as indicated on the attached Certificate of Correction form (submitted in duplicate).

__X__ Such error arose through the fault of the Patent and Trademark Office. It is requested that the certificate be issued at no cost to the applicant.

However, if it is determined that the error arose through the fault of applicant(s), please note that such error is of clerical error or minor nature and occurred in good faith and therefore issuance of the certificate of Correction is respectfully requested. The Commissioner is authorized to charge Deposit Account No. 50-0292 any required fee. A duplicate of this request is attached.

Such errors specifically:

In col. 3, line 14, delete the word "lo"

In col. 15, line 45, kindly replace "ump" with --jump--

In col. 28, line 65, change "MEB" to --MIB--.

Certificate of Mailing under 37 CFR 1.8

I hereby certify that this response is being deposited with the United States Postal Service as first class mail in an envelope addressed to the Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450 on.

Date. Feb. 25, 2004 Signed: Amy Drury

Name: Amy Drury

The undersigned requests being contacted at (510) 547-3378 if there are any questions or clarifications, or if there are any problems with issuance of the Certificate of Correction.

Respectfully Submitted,

Feb. 25, 2004
Date

Dov Rosenfeld, Reg. No. 38687
Agent of Record.

Address for correspondence:
Dov Rosenfeld
5507 College Avenue, Suite 2,
Oakland, CA 94618
Tel. 510-547-3378; Fax: 510-291-2985

INVENTEK

Fax

Dov Rosenfeld
5507 College Avenue, Suite 2
Oakland, CA 94618, USA
Phone: (510)547-3378; Fax: (510)653-7992
dov@inventek.com

Patent Application Ser. No.: 09/608237	***Ref./Docket No***: APPT-001-1
Applicant(s): Dietz, *et al.*	***Examiner.***: Moustafa M. Meky
Filing Date: June 30, 2000	***Art Unit***: 2157

FAX COVER PAGE

TO: Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

United States Patent and Trademark Office
(Examiner Moustafa M. Meky, Art Unit 2157)

Fax No.: 703-~~308-9052~~ 746-7239

DATE: July 08, 2003

FROM: Dov Rosenfeld, Reg. No. 38687

RE: Response to Office Action

Number of pages including cover: *13*

OFFICIAL COMMUNICATION

PLEASE URGENTLY DELIVER A COPY OF THIS RESPONSE TO EXAMINER MOUSTAFA M. MEKY, ART UNIT 2157

Certificate of Facsimile Transmission under 37 CFR 1.8

I hereby certify that this response is being facsimile transmitted to the United States Patent and Trademark Office at telephone number 703-~~308-9052~~ 746-7239 addressed the Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450 on.

Date: 7/8/03

Signed: _____

Name: Dov Rosenfeld, Reg. No. 38687

Received from < +1 510 291 2985 > at 7/8/03 7:36:18 PM [Eastern Daylight Time]

TRANSMITTAL FORM

(to be used for all correspondence after initial filing)

Application Number	09/608237
Filing Date	30 Jun 2000
First Named Inventor	Dietz, Russell S.
Group Art Unit	2157
Examiner Name	Moustafa M. Meky
Attorney Docket Number	APPT-001-1

ENCLOSURES *(check all that apply)*

- ☐ Fee Transmittal Form
- ☐ Fee Attached
- ☒ Amendment / Response
- ☐ ☐ After Final
- ☐ ☐ Affidavits/declaration(s)
- ☐ Extension of Time Request
- ☐ Express Abandonment Request
- ☐ Information Disclosure Statement
- ☐ Certified Copy of Priority Document(s)
- ☐ Response to Missing Parts/ Incomplete Application
- ☐
- ☐ ☐ Response to Missing Parts under 37 CFR 1.52 or 1.53
- ☐ Assignment Papers *(for an Application)*
- ☐ Drawing(s)
- ☐ Licensing-related Papers
- ☐ Petition Routing Slip (PTO/SB/69) and Accompanying Petition
- ☐ To Convert a Provisional Application
- ☐ Power of Attorney, Revocation Change of Correspondence Address
- ☐ Terminal Disclaimer
- ☐ Small Entity Statement
- ☐ Request of Refund
- ☐ After Allowance Communication to Group
- ☐ Appeal Communication to Board of Appeals and Interferences
- ☐ Appeal Communication to Group *(Appeal Notice, Bnef, Reply Brief)*
- ☐ Proprietary Information
- ☐ Status Letter
- ☐ Additional Enclosure(s) *(please identify below):*
- ☐ Return Postcard
- ☐
- ☐

Remarks

SIGNATURE OF APPLICANT, ATTORNEY, OR AGENT/ CORRESPONDENCE ADDRESS

Firm *or* Individual name	Dov Rosenfeld, Reg. No. 38687
Signature	
Date	July 8, 2003

ADDRESS FOR CORRESPONDENCE

Firm *or* Individual name	Dov Rosenfeld 5507 College Avenue, Suite 2 Oakland, CA 94618, Tel: +1-510-547-3378

CERTIFICATE OF FACSIMILE TRANSMISSION

I hereby certify that this correspondence is being facsimile transmitted with the United States Patent and Trademark Office at Telephone number 703-746-7239 addressed to: Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450 on this date: July 8, 2003

Type or printed name	Dov Rosenfeld, Reg. No. 38687		
Signature		Date	July 8, 2003

Our Ref./Docket No: APPT-001-1 Patent

7/8/03

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant(s): Dietz, *et al.* Application No.: 09/608237 Filed: June 30, 2000 Title: METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK	Group Art Unit: 2157 Examiner: Moustafa M. Meky

TRANSMITTAL: RESPONSE TO OFFICE ACTION

Mail Stop Non Fee Amendment
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Dear Commissioner:

Transmitted herewith is a response to an office action for the above referenced application. Included with the response are:

_____ formal drawings (with separate letter);

This application has:

_____ a small entity status. If a claim for such status has not earlier been made, consider this as a claim for small entity status.

__X__ No additional fee is required.

Certificate of Facsimile Transmission under 37 CFR 1.8

I hereby certify that this response is being facsimile transmitted to the United States Patent and Trademark Office at telephone number ~~703-308-9052~~ 703-746-7239 addressed the Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450 on. 703-746-7239

Date: 7/8/03 Signed: _____

Name: Dov Rosenfeld, Reg. No. 38687

Received from < +1 510 291 2985 > at 7/8/03 7:36:18 PM [Eastern Daylight Time]

__X__ Applicant(s) believe(s) that no Extension of Time is required. However, this conditional petition is being made to provide for the possibility that applicant has inadvertently overlooked the need for a petition for an extension of time.

____ Applicant(s) hereby petition(s) for an Extension of Time under 37 CFR 1.136(a) of:

____ one months ($110) ____ two months ($410)

____ two months ($930) ____ four months ($1450)

If an additional extension of time is required, please consider this as a petition therefor.

____ A credit card payment form for the required fee(s) is attached.

__X__ The Commissioner is hereby authorized to charge payment of the following fees associated with this communication or credit any overpayment to Deposit Account No. 50-0292 (A DUPLICATE OF THIS TRANSMITTAL IS ATTACHED):

__X__ Any missing filing fees required under 37 CFR 1.16 for presentation of additional claims.

__X__ Any missing extension or petition fees required under 37 CFR 1.17.

Respectfully Submitted,

7/8/03
Date

Dov Rosenfeld, Reg. No. 38687

Address for correspondence:
Dov Rosenfeld
5507 College Avenue,Suite 2
Oakland, CA 94618
Tel. +1-510-547-3378; Fax: +1-510-291-2985

Received from < +1 510 291 2985 > at 7/8/03 7:36:18 PM [Eastern Daylight Time]

7/8/03

Our Ref./Docket No: APPT-001-1 Patent

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant(s): Dietz, *et al.* Application No.: 09/608237 Filed: June 30, 2000 Title: METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK	Group Art Unit: 2157 Examiner: Moustafa M. Meky

TRANSMITTAL: RESPONSE TO OFFICE ACTION

Mail Stop No Fee Amendment
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Dear Commissioner:

Transmitted herewith is a response to an office action for the above referenced application. Included with the response are:

____ formal drawings (with separate letter);

This application has:

____ a small entity status. If a claim for such status has not earlier been made, consider this as a claim for small entity status.

__X__ No additional fee is required.

Certificate of Facsimile Transmission under 37 CFR 1.8

I hereby certify that this response is being facsimile transmitted to the United States Patent and Trademark Office at telephone number 703-~~308-9052~~ 746-7239 addressed the Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450 on.

Date: 7/8/03

Signed: _____

Name: Dov Rosenfeld, Reg. No. 38687

_X_ Applicant(s) believe(s) that no Extension of Time is required. However, this conditional petition is being made to provide for the possibility that applicant has inadvertently overlooked the need for a petition for an extension of time.

____ Applicant(s) hereby petition(s) for an Extension of Time under 37 CFR 1.136(a) of:

_____ one months ($110) ____ two months ($410)

_____ two months ($930) ____ four months ($1450)

If an additional extension of time is required, please consider this as a petition therefor.

____ A credit card payment form for the required fee(s) is attached.

_X_ The Commissioner is hereby authorized to charge payment of the following fees associated with this communication or credit any overpayment to Deposit Account No. 50-0292 (A DUPLICATE OF THIS TRANSMITTAL IS ATTACHED):

_X_ Any missing filing fees required under 37 CFR 1.16 for presentation of additional claims.

_X_ Any missing extension or petition fees required under 37 CFR 1.17.

Respectfully Submitted,

7/8/03
Date

Dov Rosenfeld, Reg. No. 38687

Address for correspondence:
Dov Rosenfeld
5507 College Avenue,Suite 2
Oakland, CA 94618
Tel. +1-510-547-3378; Fax: +1-510-291-2985

Received from < +1 510 291 2985 > at 7/8/03 7:36:18 PM [Eastern Daylight Time]

1 H 29 54

Application or Docket Number

PATENT APPLICATION FEE DETERMINATION RECORD
Effective December 29, 1999

CLAIMS AS FILED - PART I

	(Column 1) NUMBER FILED	(Column 2) NUMBER EXTRA
FOR		
BASIC FEE		
TOTAL CLAIMS	54 minus 20=	* 39
INDEPENDENT CLAIMS	4 minus 3 =	* 1
MULTIPLE DEPENDENT CLAIM PRESENT		

SMALL ENTITY TYPE ☐ RATE	FEE	OR	OTHER THAN SMALL ENTITY RATE	FEE
	345.00	OR		690.00
X$ 9=		OR	X$18=	702
X39=		OR	X78=	78
+130=		OR	+260=	
TOTAL		OR	TOTAL	1470

* If the difference in column 1 is less than zero, enter "0" in column 2

CLAIMS AS AMENDED - PART II

AMENDMENT A

	(Column 1) CLAIMS REMAINING AFTER AMENDMENT		(Column 2) HIGHEST NUMBER PREVIOUSLY PAID FOR	(Column 3) PRESENT EXTRA
Total	* 0	Minus	**	=
Independent	*	Minus	***	=
FIRST PRESENTATION OF MULTIPLE DEPENDENT CLAIM				

SMALL ENTITY RATE	ADDITIONAL FEE	OR	OTHER THAN SMALL ENTITY RATE	ADDITIONAL FEE
X$ 9=		OR	X$18=	
X39=		OR	X78=	
+130=		OR	+260=	
TOTAL ADDIT. FEE		OR	TOTAL ADDIT. FEE	

AMENDMENT B

	(Column 1) CLAIMS REMAINING AFTER AMENDMENT		(Column 2) HIGHEST NUMBER PREVIOUSLY PAID FOR	(Column 3) PRESENT EXTRA
Total	*	Minus	**	=
Independent	*	Minus	***	=
FIRST PRESENTATION OF MULTIPLE DEPENDENT CLAIM				

SMALL ENTITY RATE	ADDITIONAL FEE	OR	OTHER THAN SMALL ENTITY RATE	ADDITIONAL FEE
X$ 9=		OR	X$18=	
X39=		OR	X78=	
+130=		OR	+260=	
TOTAL ADDIT. FEE		OR	TOTAL ADDIT. FEE	

AMENDMENT C

	(Column 1) CLAIMS REMAINING AFTER AMENDMENT		(Column 2) HIGHEST NUMBER PREVIOUSLY PAID FOR	(Column 3) PRESENT EXTRA
Total	*	Minus	**	=
Independent	*	Minus	***	=
FIRST PRESENTATION OF MULTIPLE DEPENDENT CLAIM				

SMALL ENTITY RATE	ADDITIONAL FEE	OR	OTHER THAN SMALL ENTITY RATE	ADDITIONAL FEE
X$ 9=		OR	X$18=	
X39=		OR	X78=	
+130=		OR	+260=	
TOTAL ADDIT. FEE		OR	TOTAL ADDIT. FEE	

* If the entry in column 1 is less than the entry in column 2, write "0" in column 3.
** If the "Highest Number Previously Paid For" IN THIS SPACE is less than 20, enter "20."
***If the "Highest Number Previously Paid For" IN THIS SPACE is less than 3, enter "3."
The "Highest Number Previously Paid For" (Total or Independent) is the highest number found in the appropriate box in column 1.

NOAC Ex. 1015 Page 684

ISSUE SLIP STAPLE AREA (for additional cross references)

POSITION	INITIALS	ID NO.	DATE
FEE DETERMINATION	MH		7/14/15
C.I.P.E. CLASSIFIER	MH		7/15
FORMALITY REVIEW	L	JC851	08-21-0
RESPONSE FORMALITY REVIEW			

INDEX OF CLAIMS

✔ ..... Rejected
= ..... Allowed
— (Through numeral) .. Canceled
÷ ..... Restricted
N ..... Non-elected
I ..... Interference
A ..... Appeal
O ..... Objected

Claim Final	Claim Original	Date
	1	6/22/2023
2	2	7/16/2023
3	3	
4	4	
5	5	
6	6	
7	7	
8	8	
9	9	
	10	
	11	
	12	
	13	
	14	
	15	
	16	
	17	
	18	
	19	
	20	
	21	
	22	
	23	
	24	
	25	
	26	
	27	
	28	
	29	
	30	
	31	
	32	
	33	
	34	
	35	
	36	
	37	
	38	
	39	
	40	
	41	
	42	
	43	
	44	
	45	
	46	
	47	
	48	
	49	
	50	

Claim Final	Claim Original	Date
	51	6/22/2023
	52	
	53	
	54	
	55	
	56	
	57	
	58	
	59	
	60	
	61	
	62	
	63	
	64	
	65	
	66	
	67	
	68	
	69	
	70	
	71	
	72	
	73	
	74	
	75	
	76	
	77	
	78	
	79	
	80	
	81	
	82	
	83	
	84	
	85	
	86	
	87	
	88	
	89	
	90	
	91	
	92	
	93	
	94	
	95	
	96	
	97	
	98	
	99	
	100	

Claim Final	Claim Original	Date
	101	
	102	
	103	
	104	
	105	
	106	
	107	
	108	
	109	
	110	
	111	
	112	
	113	
	114	
	115	
	116	
	117	
	118	
	119	
	120	
	121	
	122	
	123	
	124	
	125	
	126	
	127	
	128	
	129	
	130	
	131	
	132	
	133	
	134	
	135	
	136	
	137	
	138	
	139	
	140	
	141	
	142	
	143	
	144	
	145	
	146	
	147	
	148	
	149	
	150	

If more than 150 claims or 10 actions staple additional sheet here

(LEFT INSIDE)

SEARCHED

Class	Sub.	Date	Exmr.
709	200		
	201		
	220		
	223		
	224		
	231		
	232		
	236		
	238	6/21/2003	MMM
	239		
	240		
	246		
370	389		
	392		
	395.32		
update		7/16/2003	MMM

INTERFERENCE SEARCHED

Class	Sub.	Date	Exmr.
709	224	7/16/2003	MMM
370	389		

SEARCH NOTES
(INCLUDING SEARCH STRATEGY)

	Date	Exmr.
West	6/21/2003	MMM

(RIGHT OUTSIDE)

IF 1439F 09608237 RET 232724102—000004 SP DUP RT ID: VD

PATENT APPLICATION

09608237

09/608231

06/30/00

INITIALS PH

CONTENTS

	Date Received (Incl. C. of M.) or Date Mailed		Date Received (Incl. C. of M.) or Date Mailed
1. Application 18 Pgs papers.		42.	
2. Error Dec missing	08-25-00	43.	
3. Decl, fee, EOT	11-6-00	44.	
4. IDS	4/2/01	45.	
5. I.D.S. w/References	4-18-02	46.	
6. 103 Rejection-3 mos	6-25-03	47.	
7. Amdt A	7-8-03	48.	
8. Notice of Allowability	7-17-03	49.	
9. Reg. for C & C	2-22-04	50.	
10.		51.	
11.		52.	
12.		53.	
13.		54.	
14.		55.	
15.		56.	
16.		57.	
17.		58.	
18.		59.	
19.		60.	
20.		61.	
21.		62.	
22.		63.	
23.		64.	
24.		65.	
25.		66.	
26.		67.	
27.		68.	
28.		69.	
29.		70.	
30.		71.	
31.		72.	
32.		73.	
33.		74.	
34.		75.	
35.		76.	
36.		77.	
37.		78.	
38.		79.	
39.		80.	
40.		81.	
41.		82.	

(LEFT OUTSIDE)

Our Ref./Docket No: AP-T-001-1 Patent

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant(s): Dietz, *et al.* Application No.: 09/608237 Filed: June 30, 2000 Title: METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK	Group Art Unit: 2755 Examiner: (Unassigned)

OIPE JC66 NOV 06 2000 PATENT & TRADEMARK OFFICE

RESPONSE TO NOTICE TO FILE MISSING PARTS OF APPLICATION

Assistant Commissioner for Patents
Washington, D.C. 20231
Attn: Box Missing Parts

Dear Assistant Commissioner:

This is in response to a Notice to File Missing Parts of Application under 37 CFR 1.53(f). Enclosed is a copy of said Notice and the following documents and fees to complete the filing requirements of the above-identified application:

__X__ Executed Declaration and Power of Attorney. The above-identified application is the same application which the inventor executed by signing the enclosed declaration;

__X__ Executed Assignment with assignment cover sheet.

__X__ A credit card payment form in the amount of $__1772.00__ is attached, being for:

- __X__ Statutory basic filing fee: $ 710
- __X__ Additional claim fee of $ 782
- __X__ Assignment recordation fee of $ 40
- __X__ Extension Fee First Month of $ 110
- __X__ Missing Parts Surcharge $ 130

____ Applicant(s) believe(s) that no Extension of Time is required. However, this conditional petition is being made to provide for the possibility that applicant has inadvertently overlooked the need for a petition for an extension of time.

__X__ Applicant(s) hereby petition(s) for an Extension of Time under 37 CFR 1.136(a) of:

__X__ one months ($110) ____ two months ($380)

____ two months ($870) ____ four months ($1360)

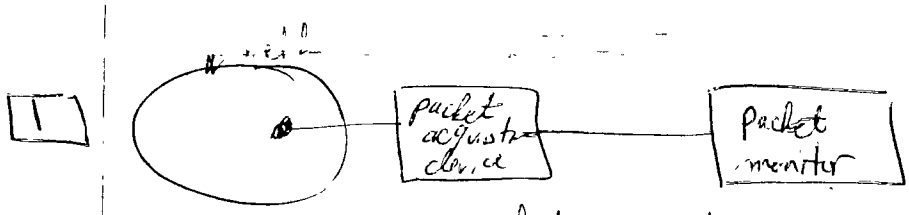
If an additional extension of time is required, please consider this as a petition therefor.

Certificate of Mailing under 37 CFR 1.8

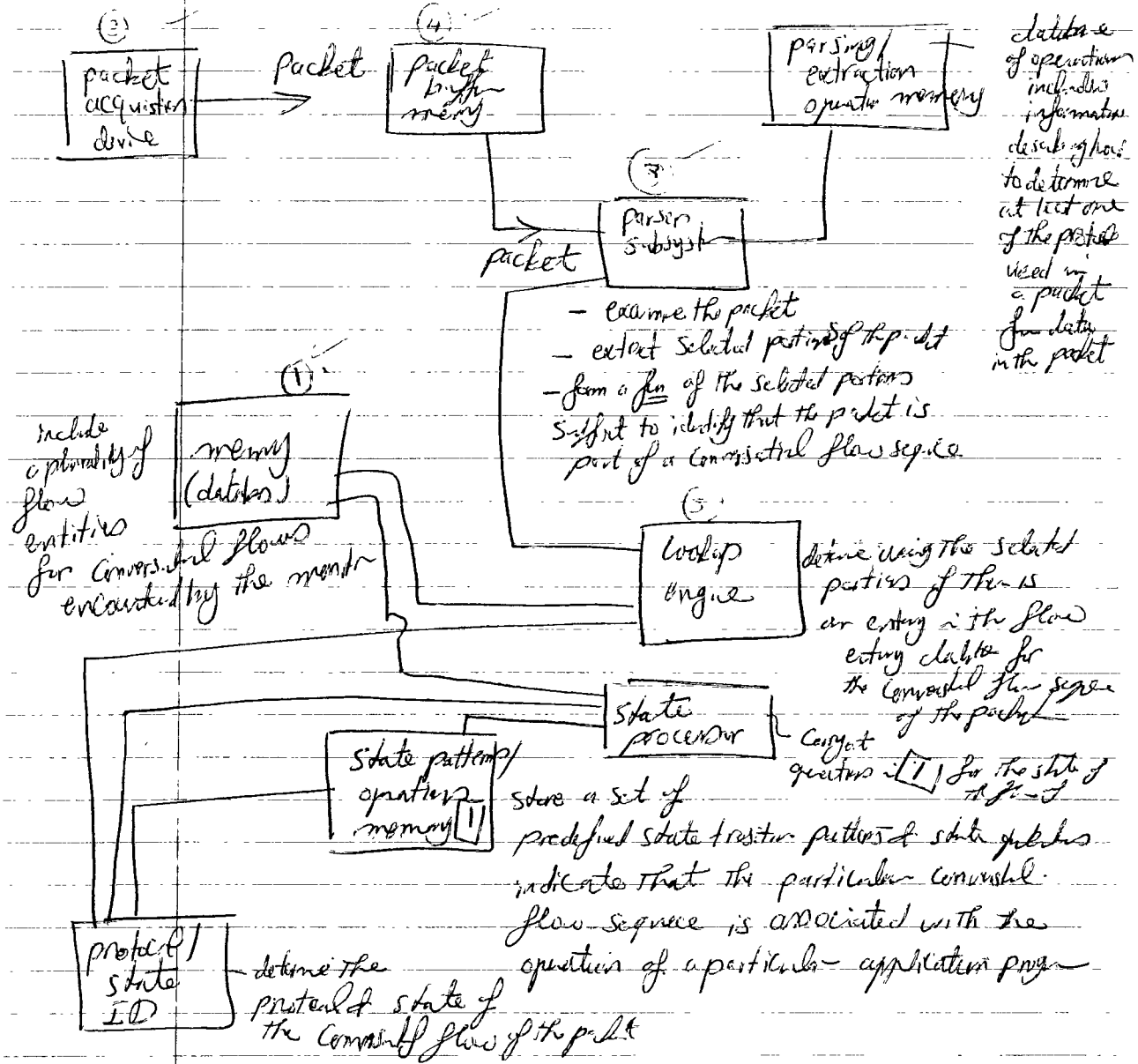
I hereby certify that this response is being deposited with the United States Postal Service as first class mail in an envelope addressed to the Assistant Commissioner for Patents, Washington, D.C. 20231 on.

Date: Nov 1, 2000 Signed: _____

Name: Dov Rosenfeld, Reg. No. 38687

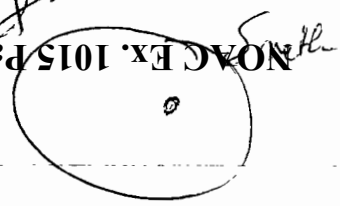


The packet monitor



- identify which application program is associated with the connectional flow segment of the packet, update the flow entry

[29/1/54]



device → (a) packet → (b) perform parsing/extraction operation on the packet to create a parser record comprising a fn of selected portions of the packet

(c) Looking up a flow entry dataset having entry (entries) for previously encountered conversational flows, the looking up using at least some of the selected packet portions and determine if the packet is of an existing flow

→ if the packet is of an existing flow, classify the packet as belong to the existing flow

→ if a new flow, storing new flow entry for the new flow entry dataset relying information for fn packets

Freeform Search

Database: US Patents Full-Text Database; US Pre-Grant Publication Full-Text Database; JPO Abstracts Database; EPO Abstracts Database; Derwent World Patents Index; IBM Technical Disclosure Bulletins

Term: 14 same 15

Display: 10 **Documents in Display Format:** TI **Starting with Number** 1

Generate: ○ Hit List ◉ Hit Count ○ Side by Side ○ Image

Search | Clear | Help | Logout | Interrupt

Main Menu | Show S Numbers | Edit S Numbers | Preferences | Cases

Search History

DATE: Saturday, June 21, 2003 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
DB=USPT; PLUR=YES; OP=ADJ			
L6	14 same 15	50	L6
L5	(packet or datagram) near5 flow	3376	L5
L4	flow near3 (entry or entries)	4883	L4
L3	11 same L2	18	L3
L2	(source or destination) adj3 address$2	15666	L2
L1	(packet or datagram) near3 signature	199	L1

END OF SEARCH HISTORY