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 (1) OF (2) PART(S)

09/608237
06/30/00

ISSUE CLASSIFICATION	
Class	Subclass
709	224

PATENT NUMBER
6651099
6651099

U.S. **UTILITY** Patent Application

O.I.P.E.	PATENT DATE
SCANNED ~~[illegible]~~ Q.A.	NOV 18 2003

APPLICATION NO.	CONT/PRIOR	CLASS	SUBCLASS	ART UNIT	EXAMINER
09/608237	D	709	224	~~2153~~ 2157	~~Burgess~~ Meky

APPLICANTS
Russell Dietz
Joseph Maixner
Andrew Koppenhaver
William Bares

TITLE
Method and apparatus for monitoring traffic in a network

Certificate
MAY 04 2004
of Correction

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
7/16/2003
(Primary Examiner) (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 09/608,237	FILING DATE 06/30/2000 RULE	CLASS 709	GROUP ART UNIT 2755	ATTORNEY DOCKET NO. 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 ********************
None, MMM

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

Foreign Priority claimed □ yes ☒ no 35 USC 119 (a-d) conditions met □ yes ☒ no □ Met after Allowance Verified and Acknowledged ______ Examiner's Signature MMM Initials	STATE OR COUNTRY CA	SHEETS DRAWING 18	TOTAL CLAIMS 59	INDEPENDENT CLAIMS 4

ADDRESS

Dov Rosenfeld
Suite 2
5507 College Avenue
Oakland ,CA 94618

TITLE

Method and apparatus for monitoring traffic in a network

FILING FEE RECEIVED 1622	FEES: Authority has been given in Paper 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

A

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

	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

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:

Name: Dov Rosenfeld, Reg. No. 38687

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, still pending ~~Attorney/Agent Reference Number APPT-001-2~~; 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, still pending ~~Attorney/Agent Reference Number APPT-001-3~~, 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, still pending ~~Attorney/Agent Reference Number APPT-001-4~~, 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, still pending ~~Attorney/Agent Reference Number APPT-001-5,~~ 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).

A1

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.

A1

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

6651099

1/18

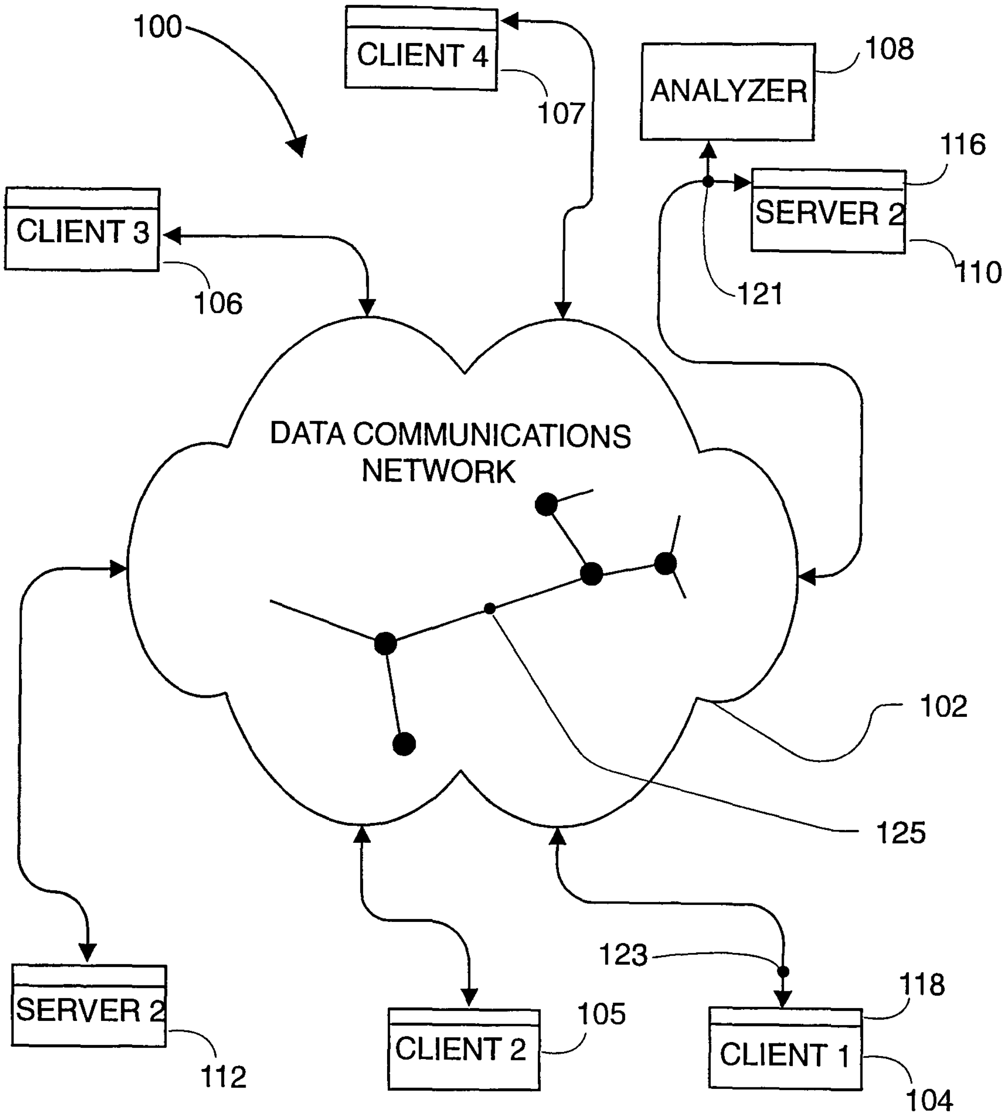


FIG. 1

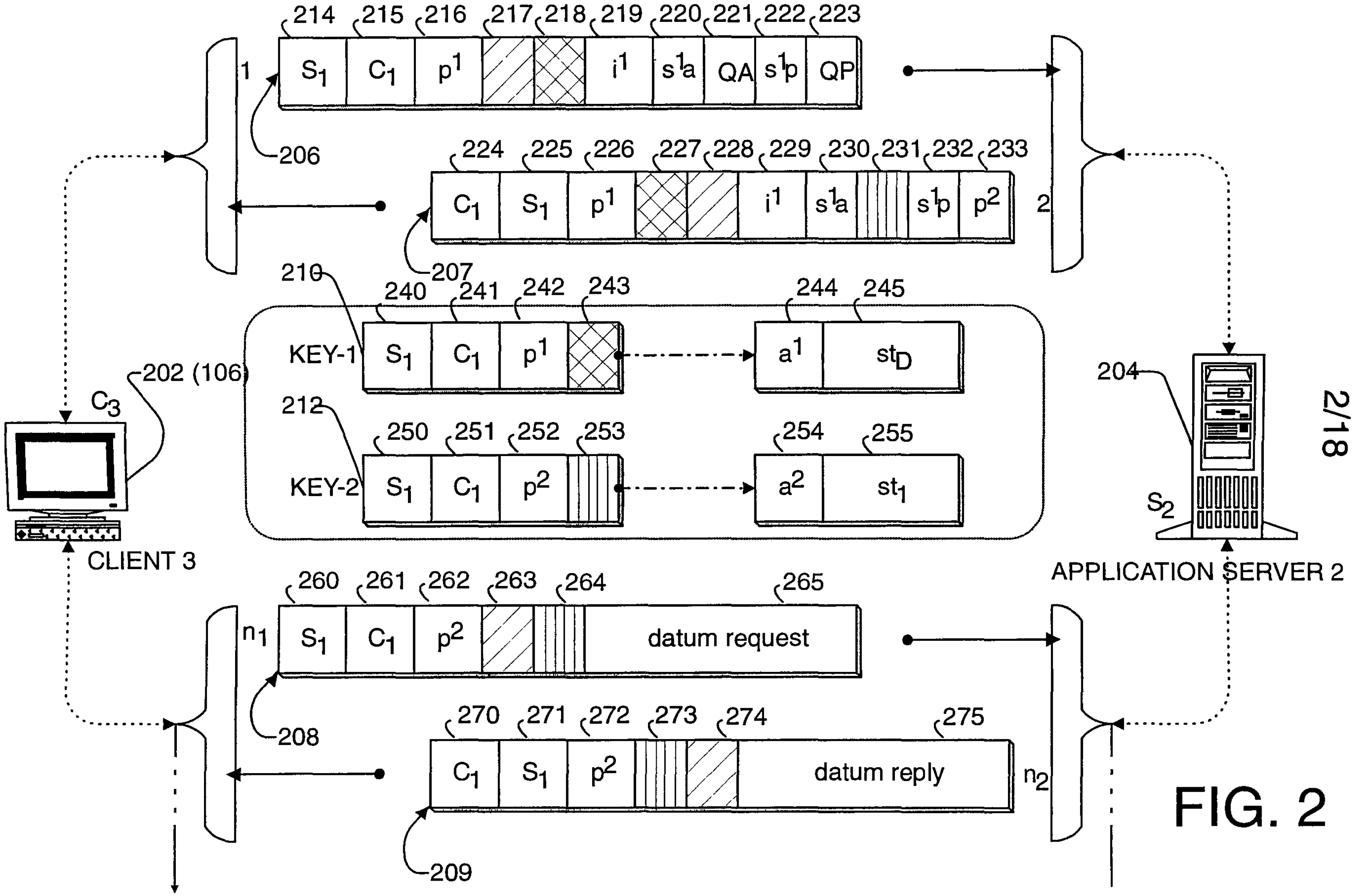

214
215
216
217
218
219
220
221
222
223
1
S1
C1
p1
i1
s1a
QA
s1p
QP
206
224
225
226
227
228
229
230
231
232
233
C1
S1
p1
i1
s1a
s1p
p2
2
207
210
240
241
242
243
244
245
KEY-1
S1
C1
p1
a1
stD
212
250
251
252
253
254
255
KEY-2
S1
C1
p2
a2
st1
202 (106)
C3
CLIENT 3
204
S2
APPLICATION SERVER 2
260
261
262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

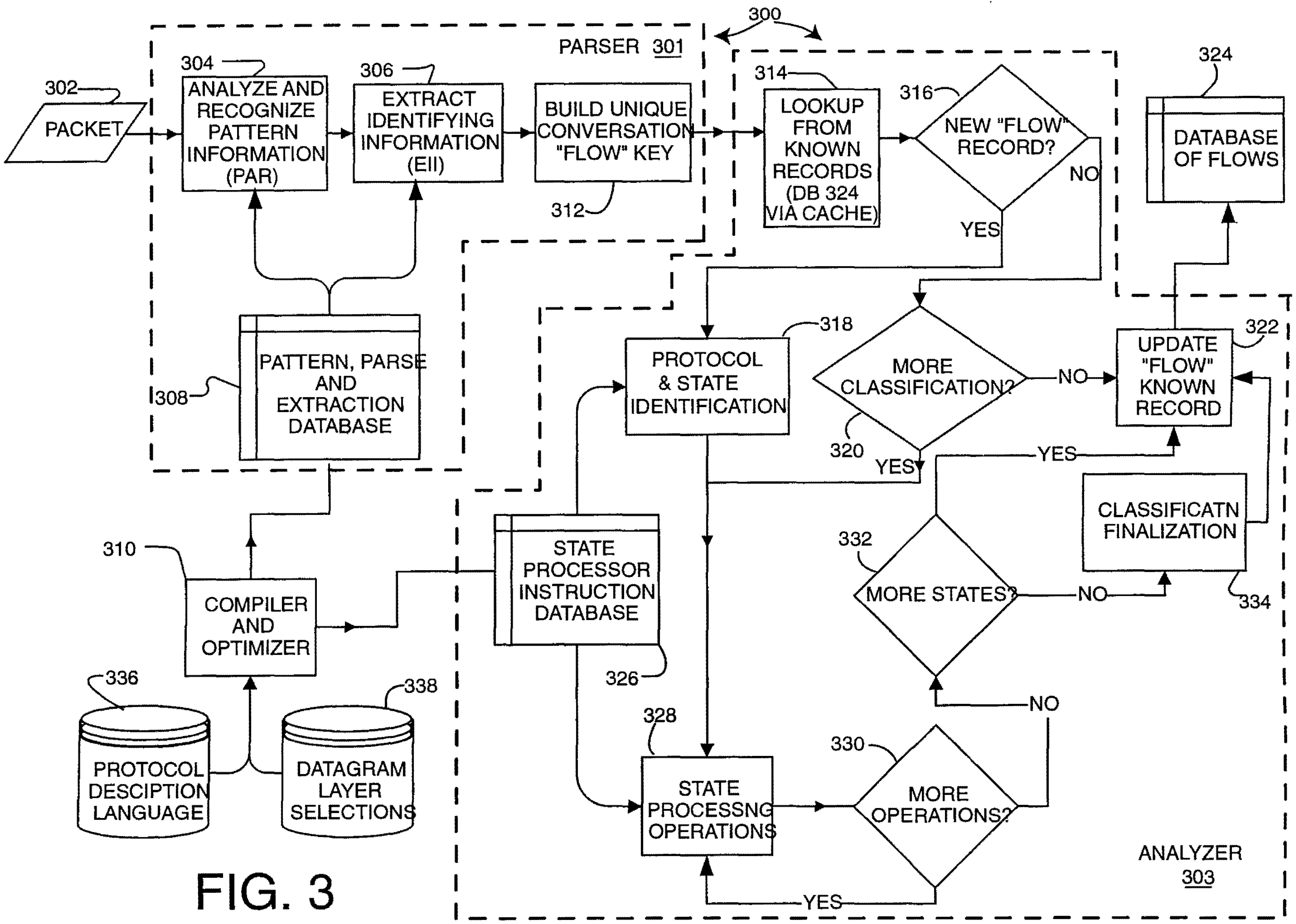

300
PARSER 301
302
PACKET
304
ANALYZE AND RECOGNIZE PATTERN INFORMATION (PAR)
306
EXTRACT IDENTIFYING INFORMATION (EII)
BUILD UNIQUE CONVERSATION "FLOW" KEY
312
308
PATTERN, PARSE AND EXTRACTION DATABASE
314
LOOKUP FROM KNOWN RECORDS (DB 324 VIA CACHE)
316
NEW "FLOW" RECORD?
NO
YES
324
DATABASE OF FLOWS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

FIG. 3

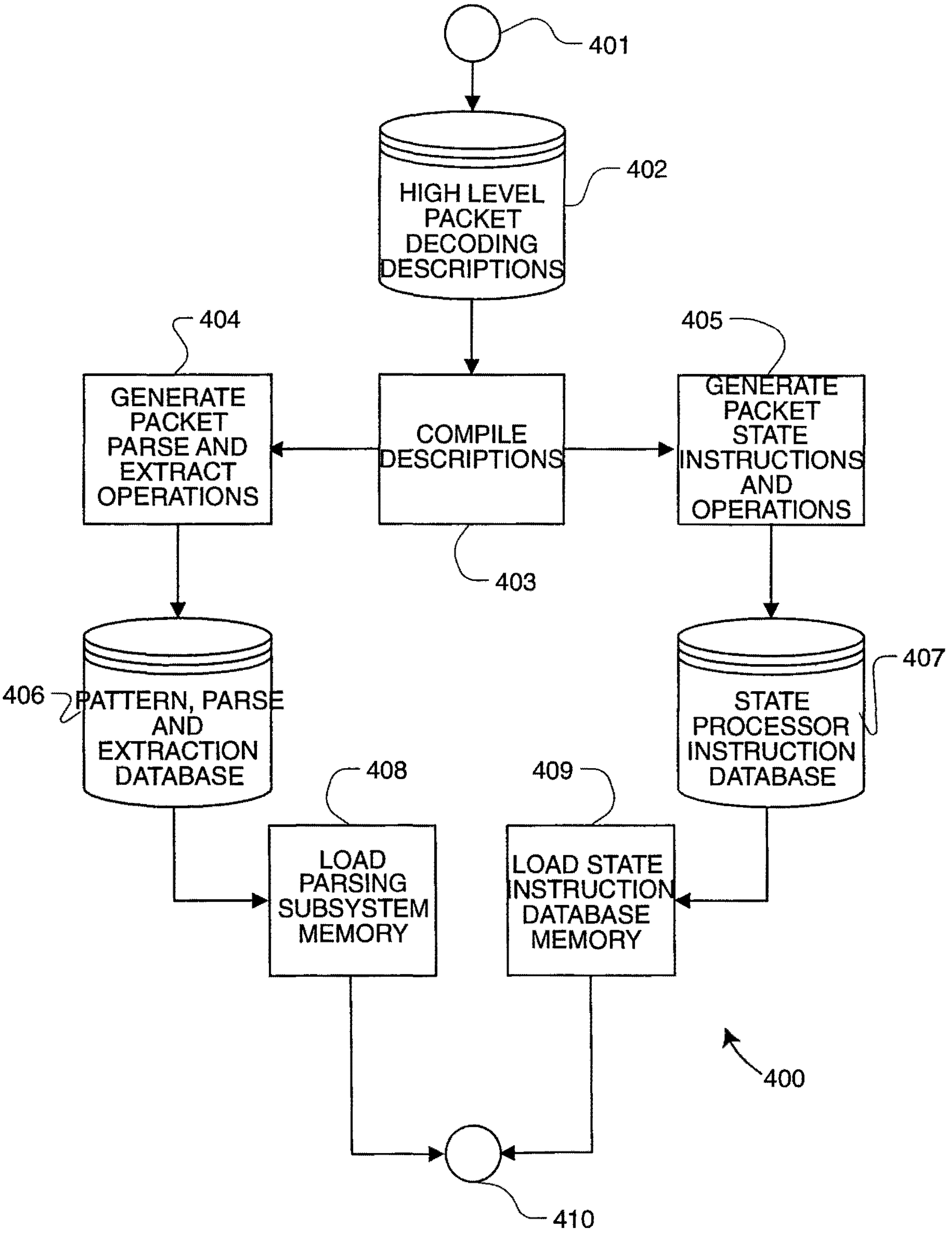
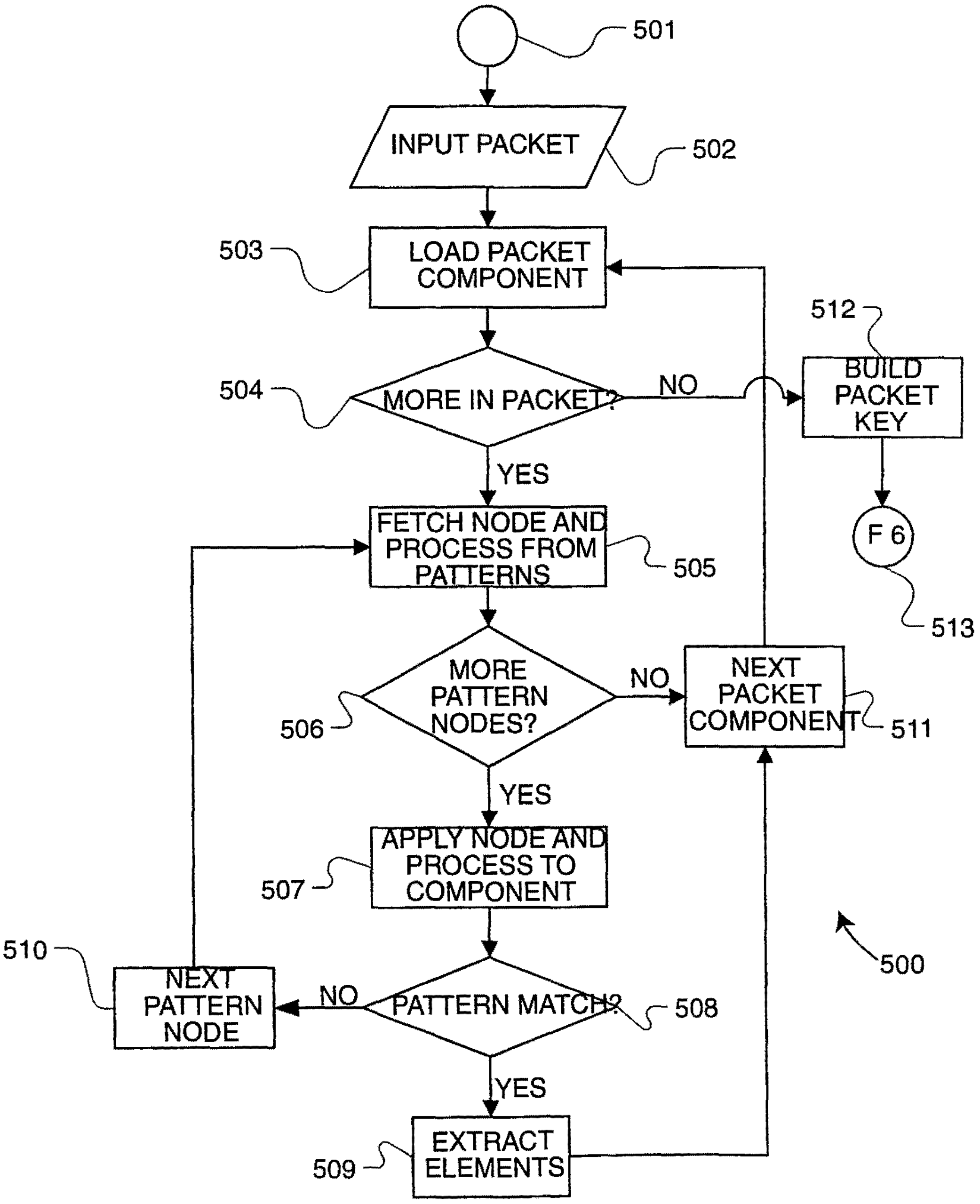

401
HIGH LEVEL PACKET DECODING DESCRIPTIONS
402
404
405
GENERATE PACKET PARSE AND EXTRACT OPERATIONS
COMPILE DESCRIPTIONS
GENERATE PACKET STATE INSTRUCTIONS AND OPERATIONS
403
406
PATTERN, PARSE AND EXTRACTION DATABASE
407
STATE PROCESSOR INSTRUCTION DATABASE
408
409
LOAD PARSING SUBSYSTEM MEMORY
LOAD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

FIG. 4



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

FIG. 5

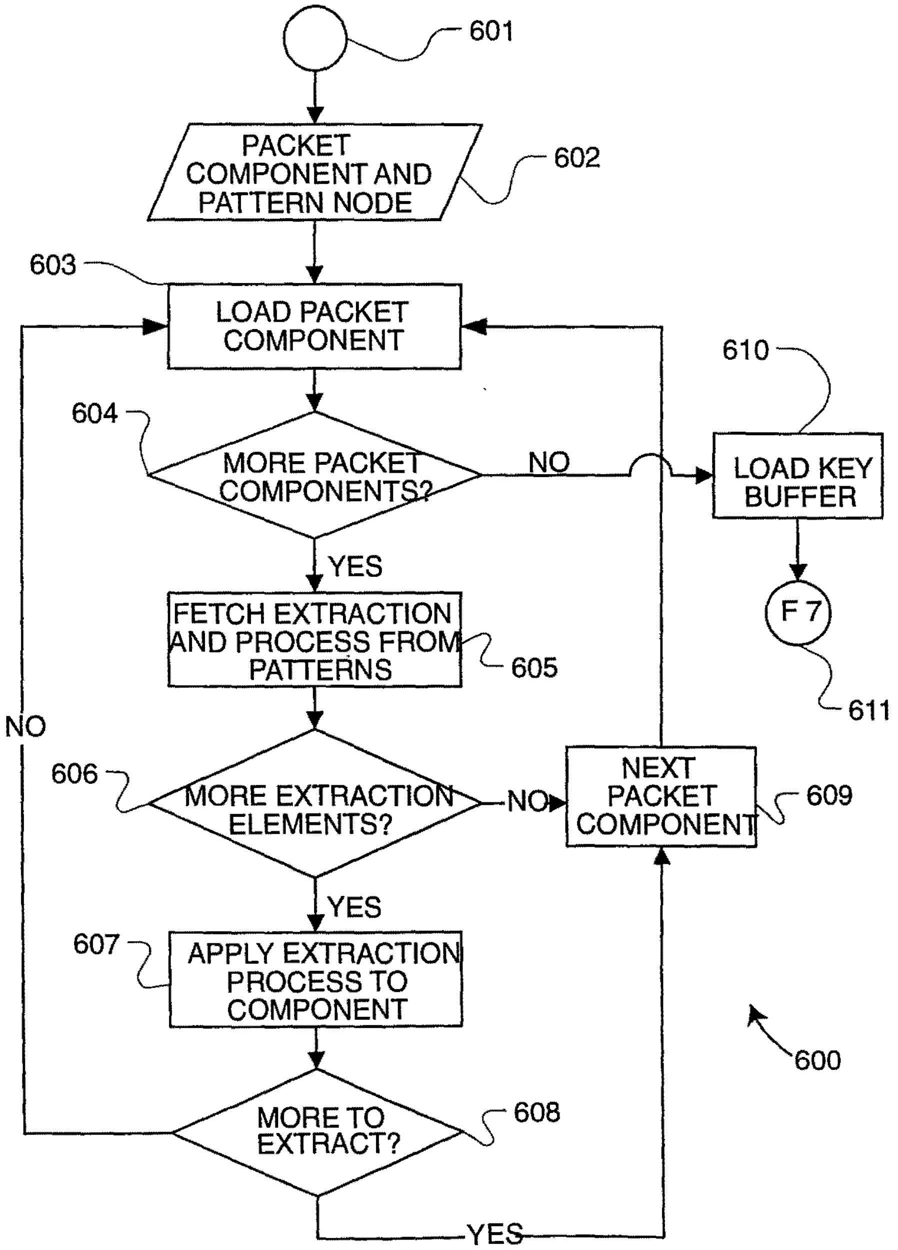

601
PACKET COMPONENT AND PATTERN NODE
602
603
LOAD PACKET COMPONENT
604
MORE PACKET COMPONENTS?
NO
610
LOAD KEY BUFFER
F 7
611
YES
FETCH EXTRACTION AND PROCESS FROM PATTERNS
605
NO
606
MORE EXTRACTION ELEMENTS?
NO
NEXT PACKET COMPONENT
609
YES
607
APPLY EXTRACTION PROCESS TO COMPONENT
600
MORE TO EXTRACT?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

FIG. 6

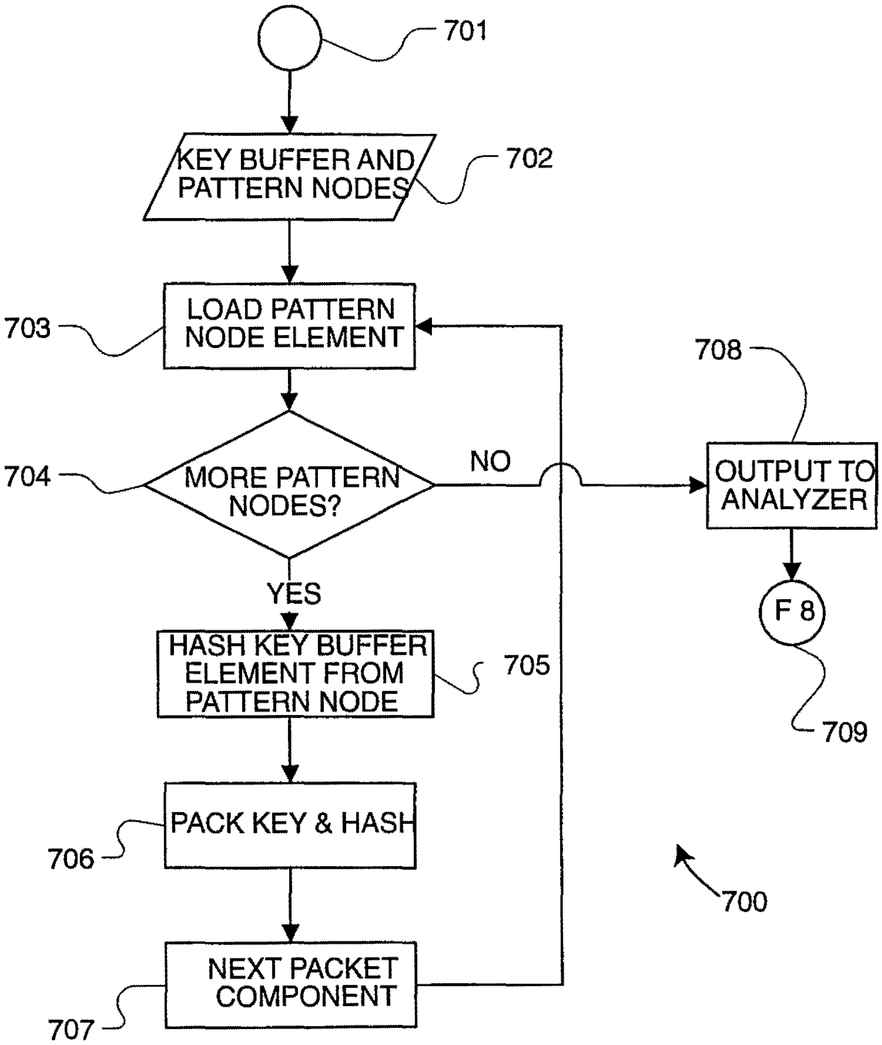

701
KEY BUFFER AND PATTERN NODES
702
LOAD PATTERN NODE ELEMENT
703
MORE PATTERN NODES?
704
NO
YES
OUTPUT TO ANALYZER
708
F 8
709
HASH KEY BUFFER ELEMENT FROM PATTERN NODE
705
PACK KEY & HASH
706
NEXT PACKET COMPONENT
707
700

FIG. 7

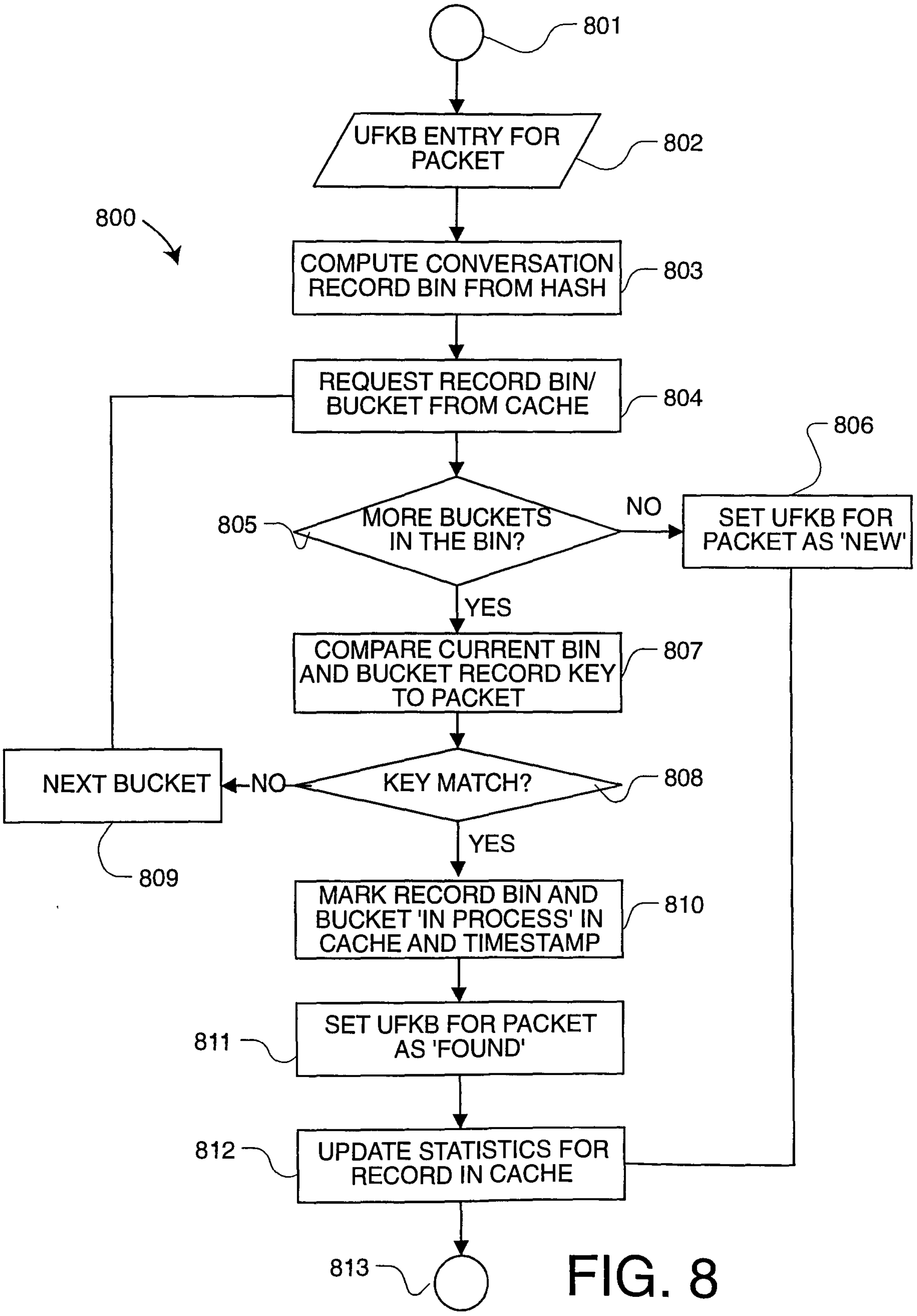

801
UFKB ENTRY FOR PACKET
802
800
COMPUTE CONVERSATION RECORD BIN FROM HASH
803
REQUEST RECORD BIN/ BUCKET FROM CACHE
804
806
805
MORE BUCKETS IN THE BIN?
NO
SET UFKB FOR PACKET AS 'NEW'
YES
COMPARE CURRENT BIN AND BUCKET RECORD KEY TO PACKET
807
NEXT BUCKET
NO
KEY MATCH?
808
YES
809
MARK RECORD BIN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

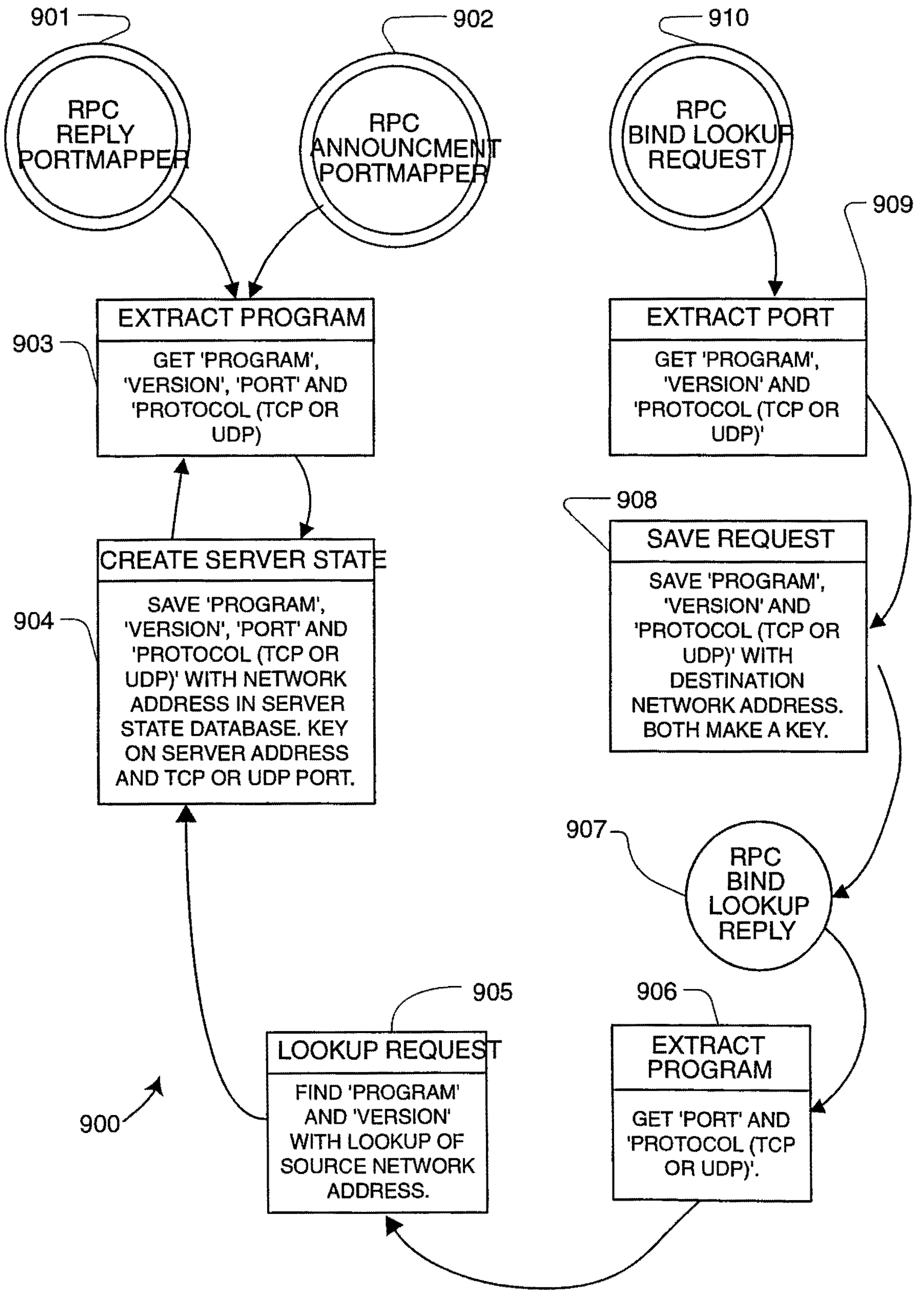

901
RPC REPLY PORTMAPPER
902
RPC ANNOUNCMENT PORTMAPPER
910
RPC BIND LOOKUP REQUEST
909
903
EXTRACT PROGRAM
GET 'PROGRAM', 'VERSION', 'PORT' AND 'PROTOCOL (TCP OR UDP)
EXTRACT PORT
GET 'PROGRAM', 'VERSION' AND 'PROTOCOL (TCP OR UDP)'
908
SAVE REQUEST
SAVE 'PROGRAM', 'VERSION' AND 'PROTOCOL (TCP OR UDP)'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

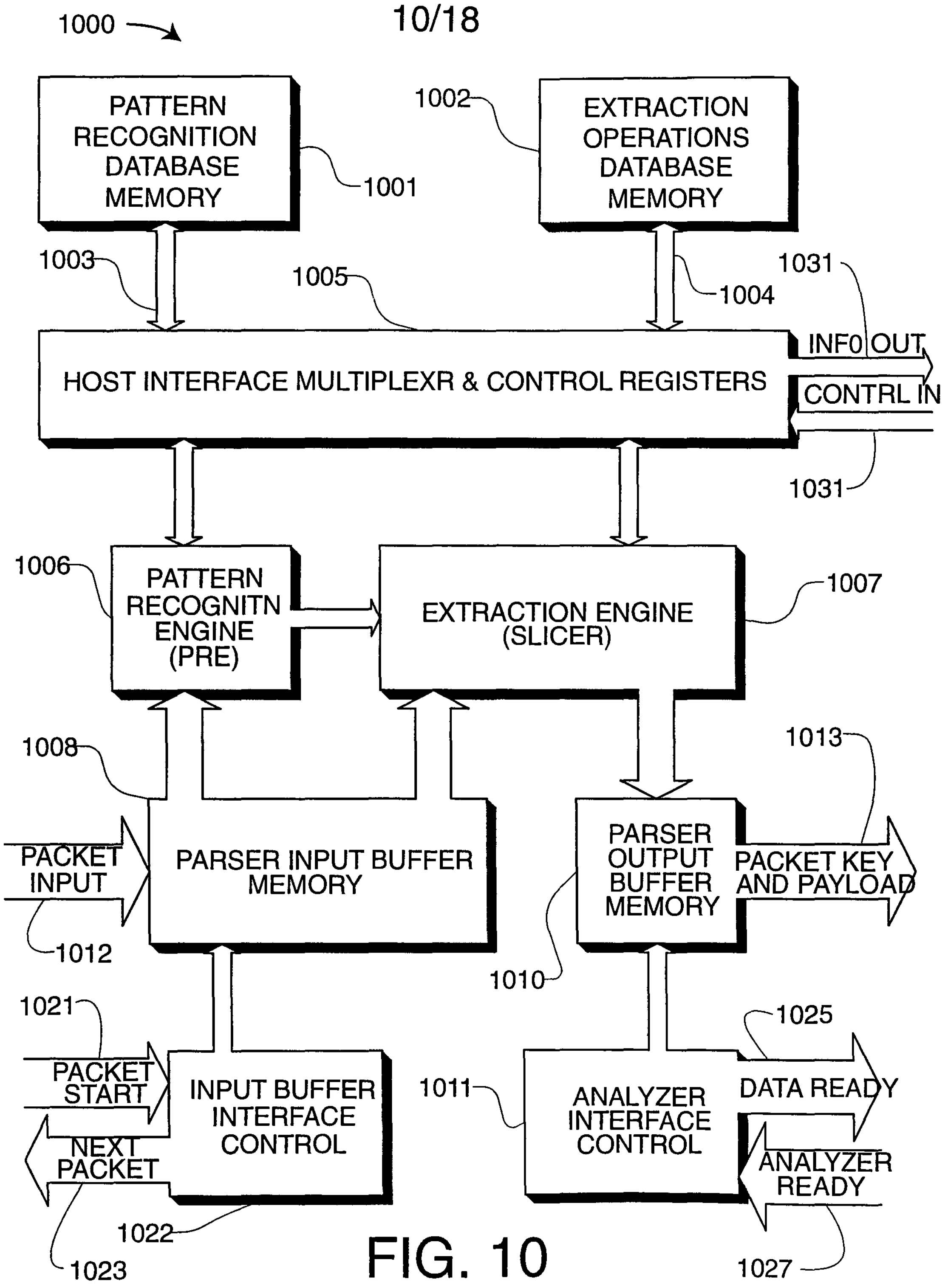

1000
PATTERN RECOGNITION DATABASE MEMORY
1001
1002
EXTRACTION OPERATIONS DATABASE MEMORY
1003
1005
1004
1031
INF0 OUT
CONTRL IN
HOST INTERFACE MULTIPLEXR & CONTROL REGISTERS
1031
1006
PATTERN RECOGNITN ENGINE (PRE)
EXTRACTION ENGINE (SLICER)
1007
1008
1013
PACKET INPUT
PARSER INPUT BUFFER MEMORY
PARSER 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

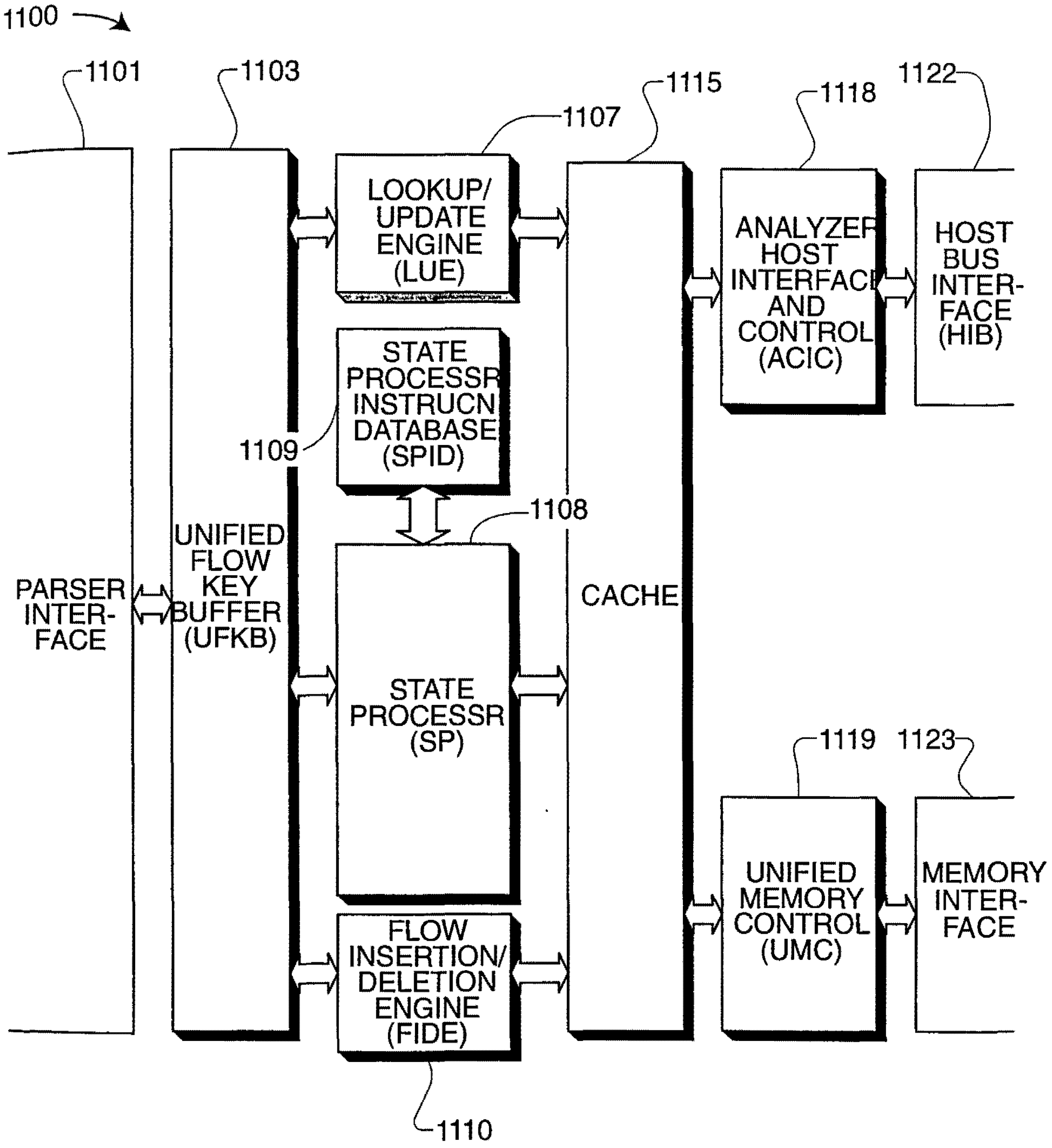

1100
1101
1103
1107
1115
1118
1122
PARSER INTER- FACE
UNIFIED FLOW KEY BUFFER (UFKB)
LOOKUP/ UPDATE ENGINE (LUE)
STATE PROCESSR INSTRUCN DATABASE (SPID)
1109
1108
STATE PROCESSR (SP)
FLOW INSERTION/ DELETION ENGINE (FIDE)
1110
CACHE
ANALYZER HOST INTERFACE AND CONTROL (ACIC)
HOST BUS INTER- FACE (HIB)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

FIG. 11

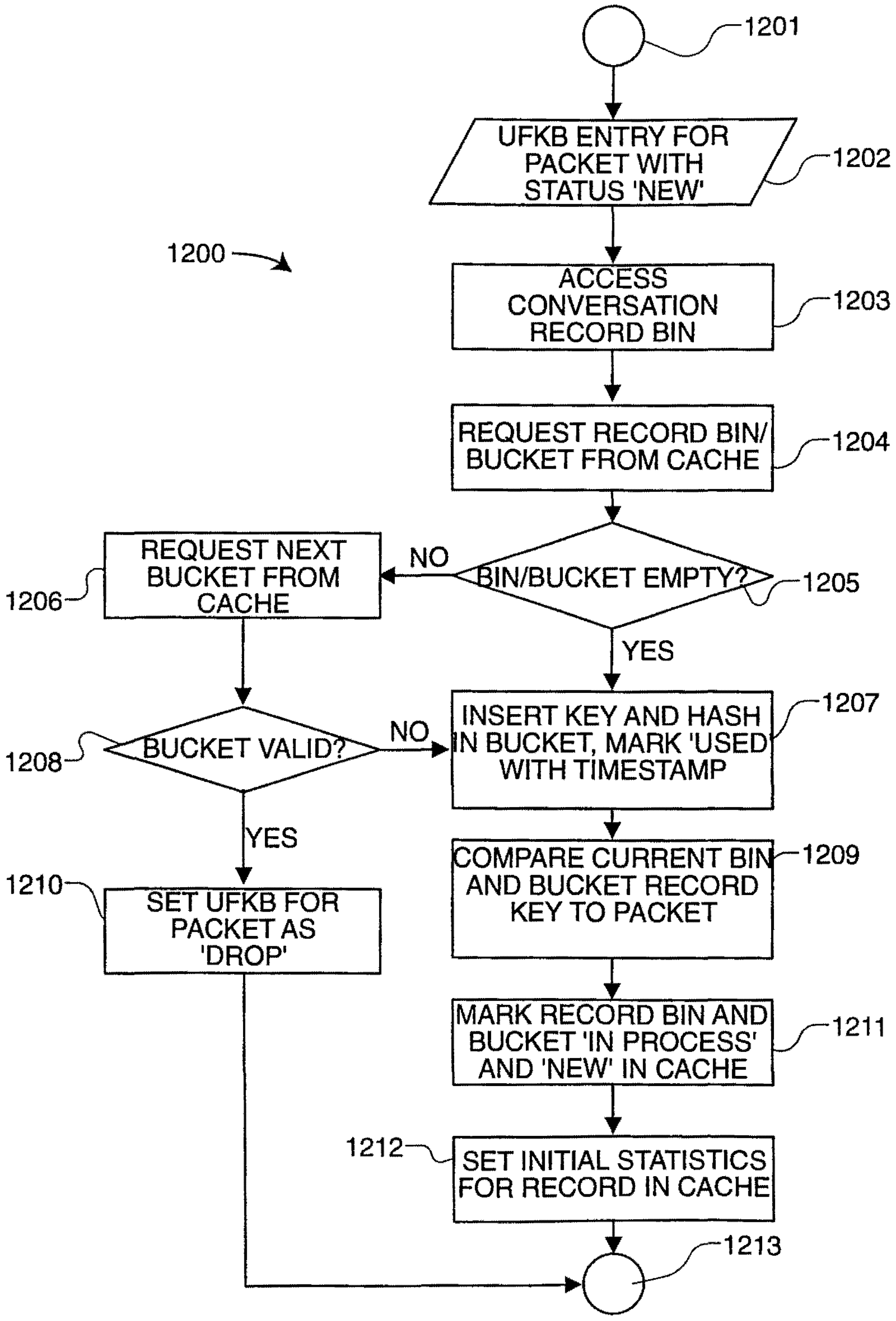

1201
UFKB ENTRY FOR PACKET WITH STATUS 'NEW'
1202
1200
ACCESS CONVERSATION RECORD BIN
1203
REQUEST RECORD BIN/ BUCKET FROM CACHE
1204
REQUEST NEXT BUCKET FROM CACHE
1206
NO
BIN/BUCKET EMPTY?
1205
YES
BUCKET VALID?
1208
NO
INSERT KEY AND HASH IN BUCKET, MARK 'USED' WITH TIMESTAMP
1207
YES
COMPARE CURRENT 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

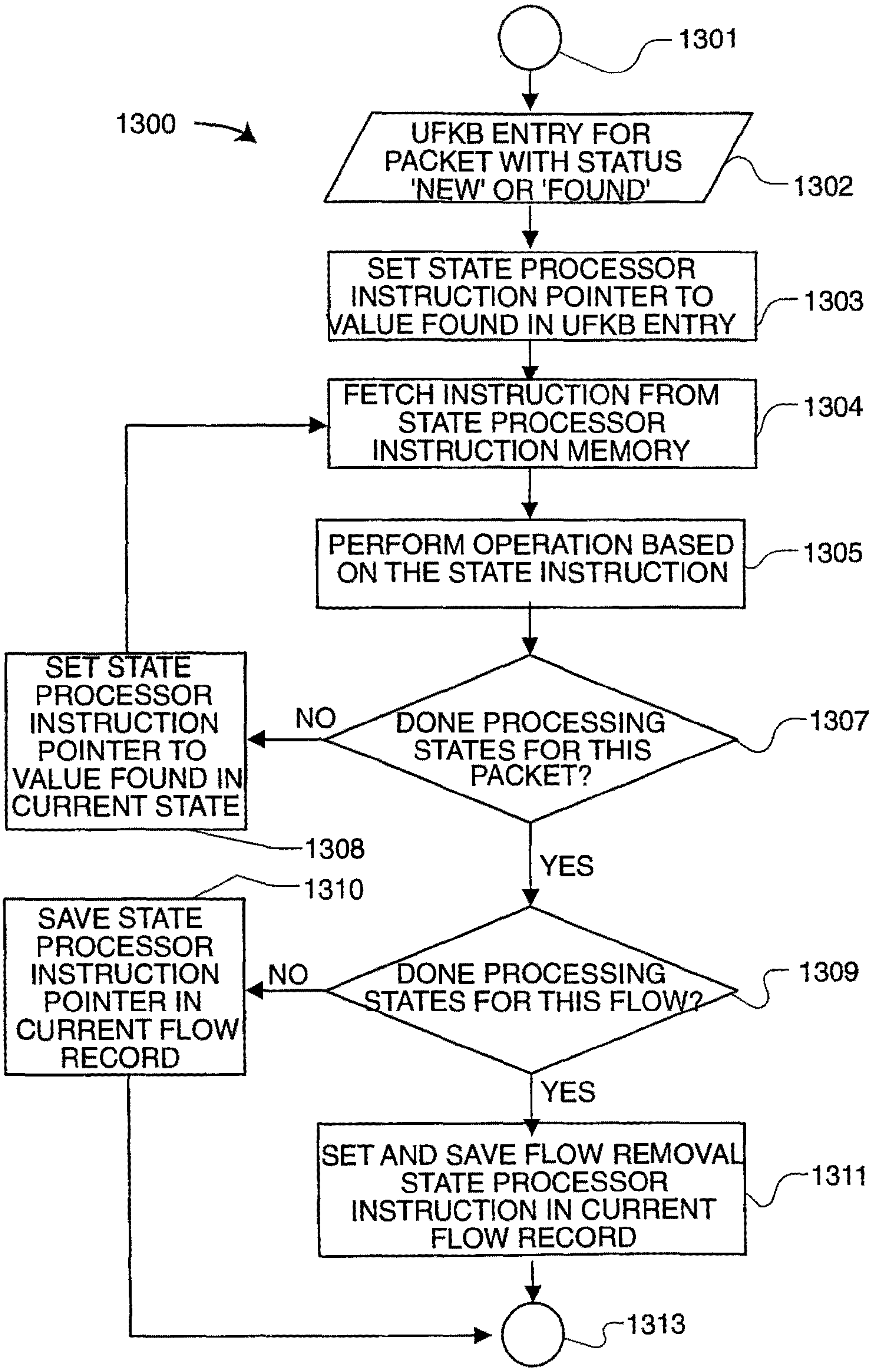

1301
1300
UFKB ENTRY FOR PACKET WITH STATUS 'NEW' OR 'FOUND'
1302
SET STATE PROCESSOR INSTRUCTION POINTER TO VALUE FOUND IN UFKB ENTRY
1303
FETCH INSTRUCTION FROM STATE PROCESSOR INSTRUCTION MEMORY
1304
PERFORM OPERATION BASED ON THE STATE INSTRUCTION
1305
DONE PROCESSING STATES FOR THIS PACKET?
1307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

FIG. 13

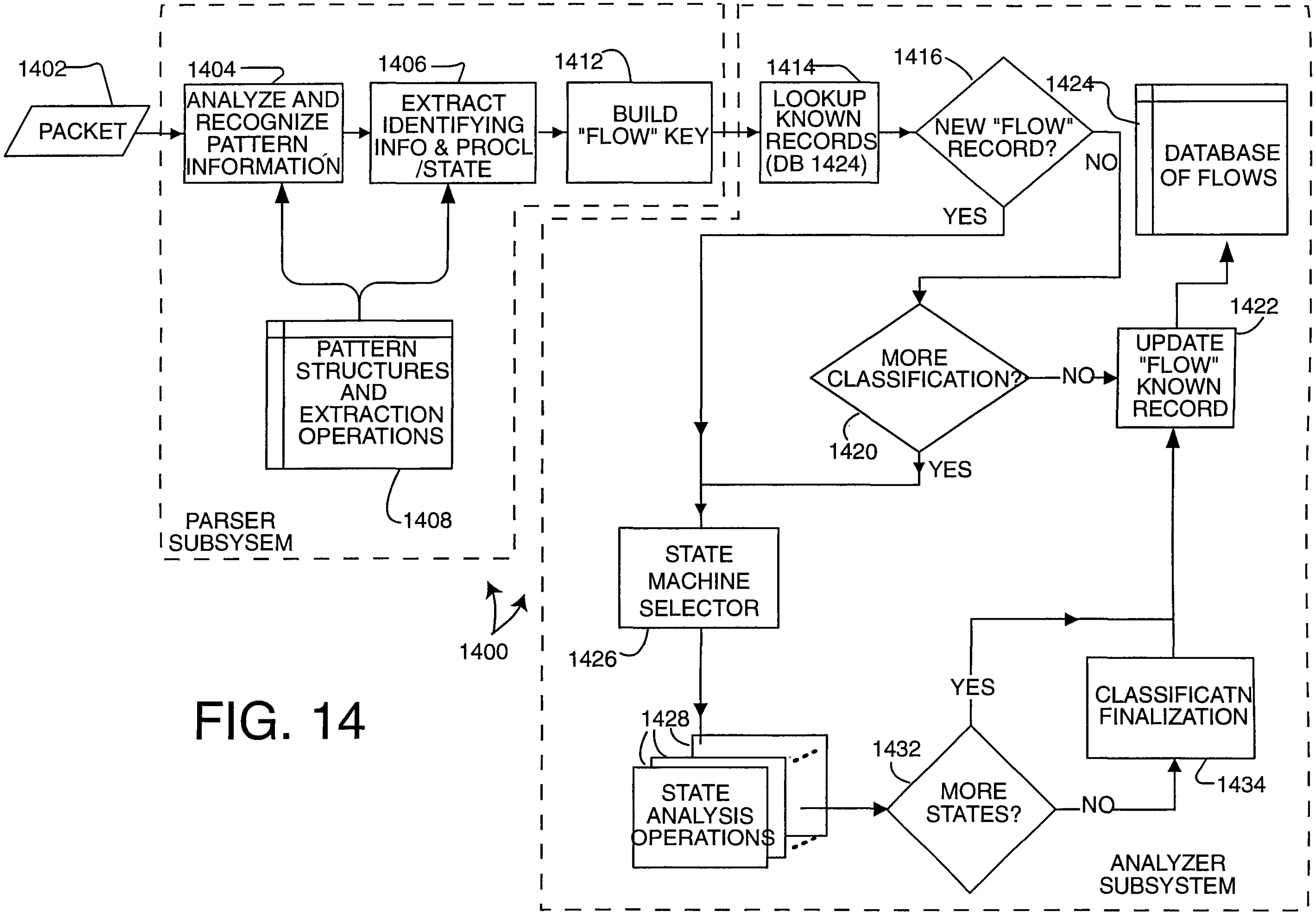

1402
PACKET
1404
ANALYZE AND RECOGNIZE PATTERN INFORMATION
1406
EXTRACT IDENTIFYING INFO & PROCL /STATE
1412
BUILD "FLOW" KEY
1414
LOOKUP KNOWN RECORDS (DB 1424)
1416
NEW "FLOW" RECORD?
NO
YES
1424
DATABASE OF FLOWS
PATTERN STRUCTURES AND EXTRACTION OPERATIONS
1408
PARSER SUBSYSEM
MORE CLASSIFICATION?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

FIG. 14

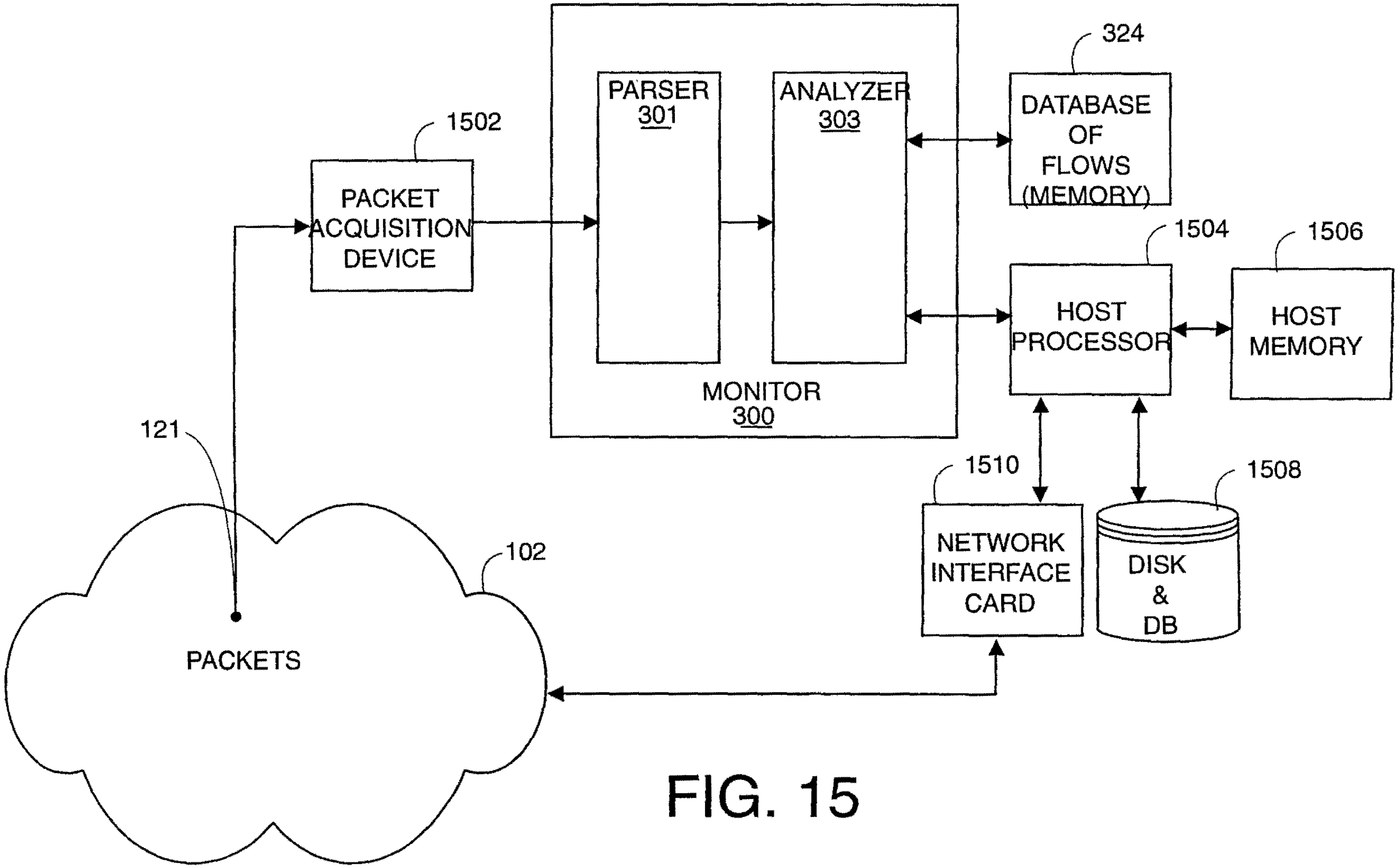

324
DATABASE OF FLOWS (MEMORY)
1502
PACKET ACQUISITION DEVICE
PARSER 301
ANALYZER 303
MONITOR 300
1504
HOST PROCESSOR
1506
HOST MEMORY
121
1510
NETWORK INTERFACE CARD
1508
DISK & DB
102
PACKETS

FIG. 15

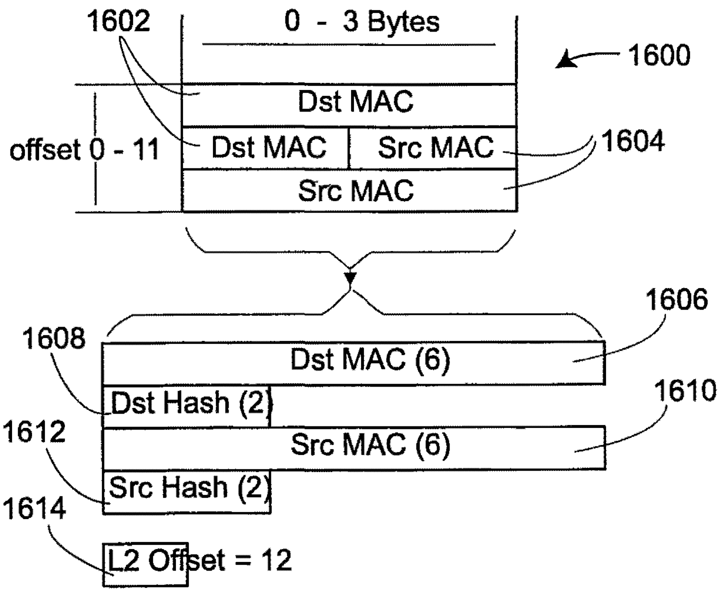

1602
0 - 3 Bytes
1600
Dst MAC
offset 0 - 11
Dst MAC
Src MAC
1604
Src MAC
1606
1608
Dst MAC (6)
Dst Hash (2)
1610
1612
Src MAC (6)
Src Hash (2)
1614
L2 Offset = 12

FIG. 16

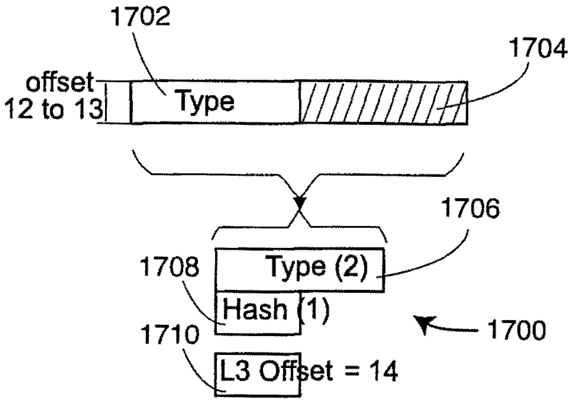


FIG. 17A

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

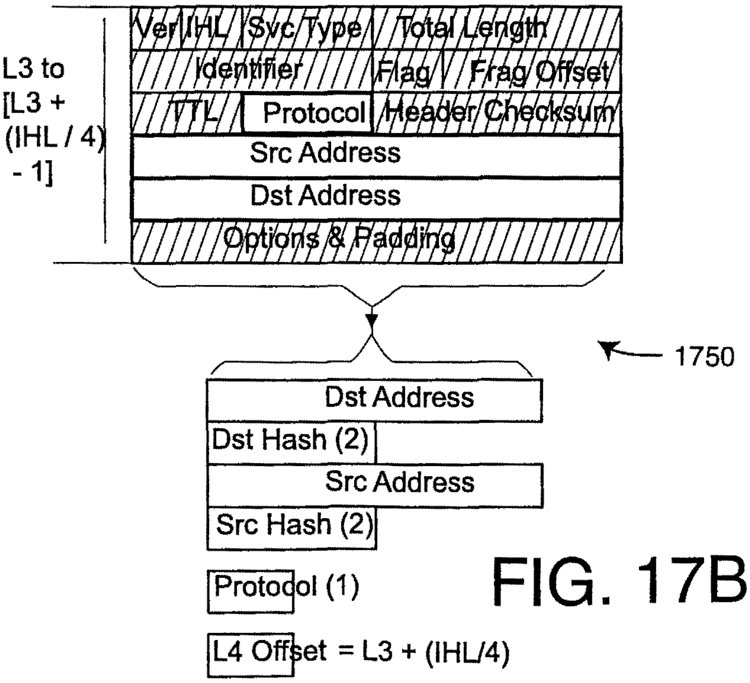


FIG. 17B

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

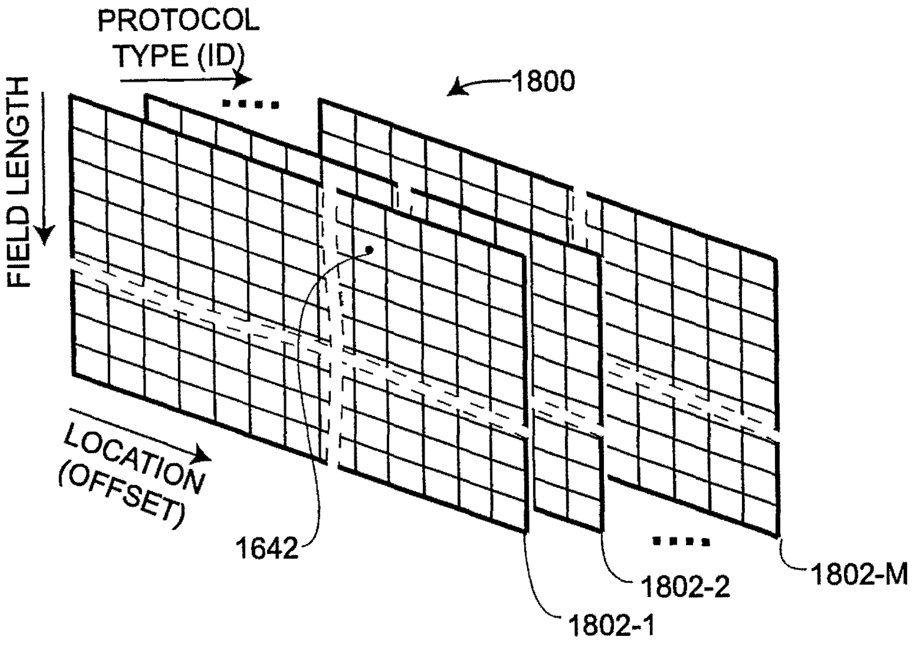


FIG. 18A

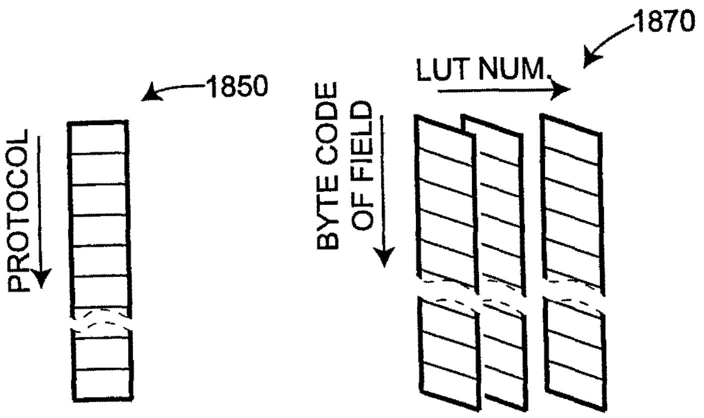


FIG. 18B

)01-1

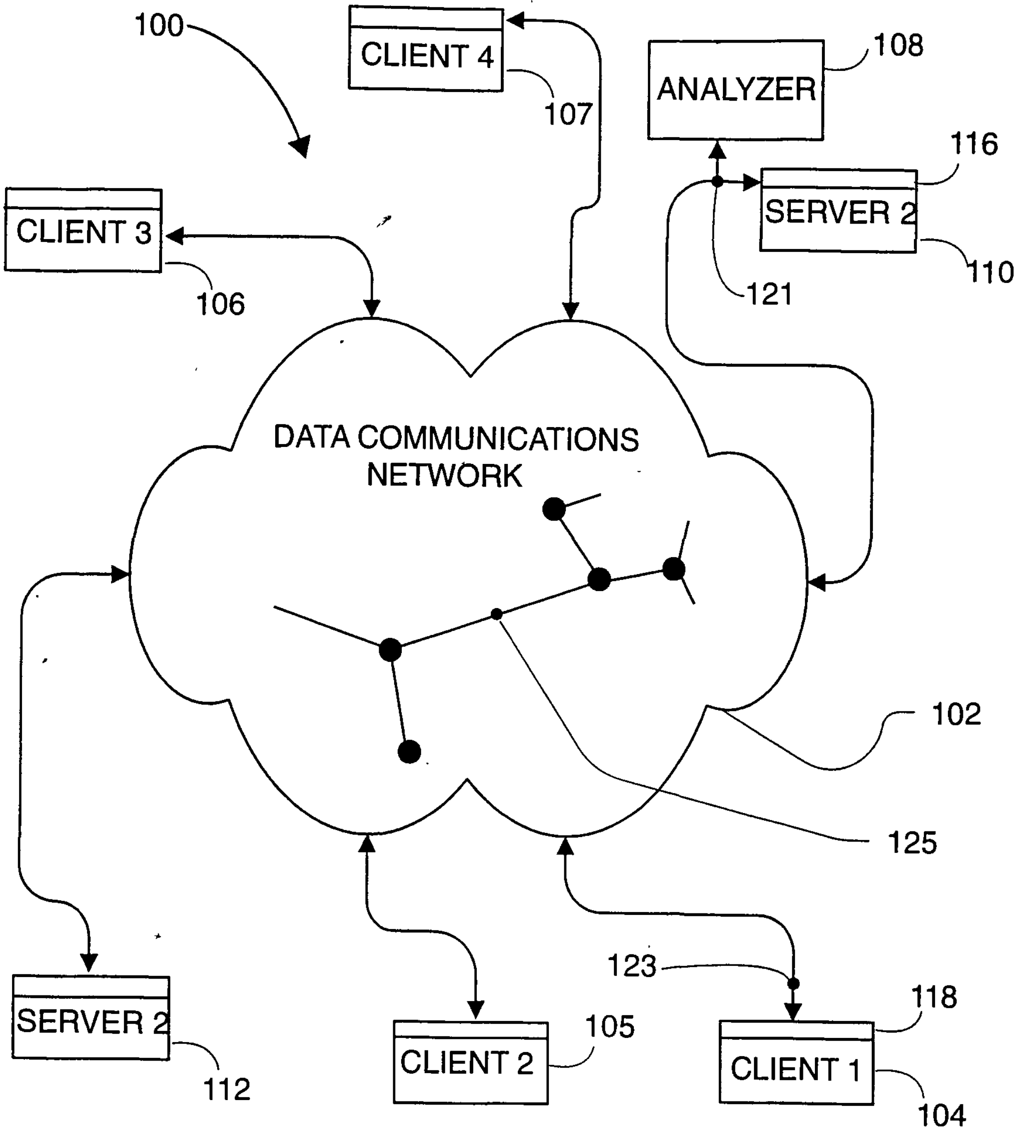

100
CLIENT 4
107
ANALYZER
108
116
SERVER 2
110
121
CLIENT 3
106
DATA COMMUNICATIONS
NETWORK
102
125
123
118
SERVER 2
112
CLIENT 2
105
CLIENT 1
104

FIG. 1

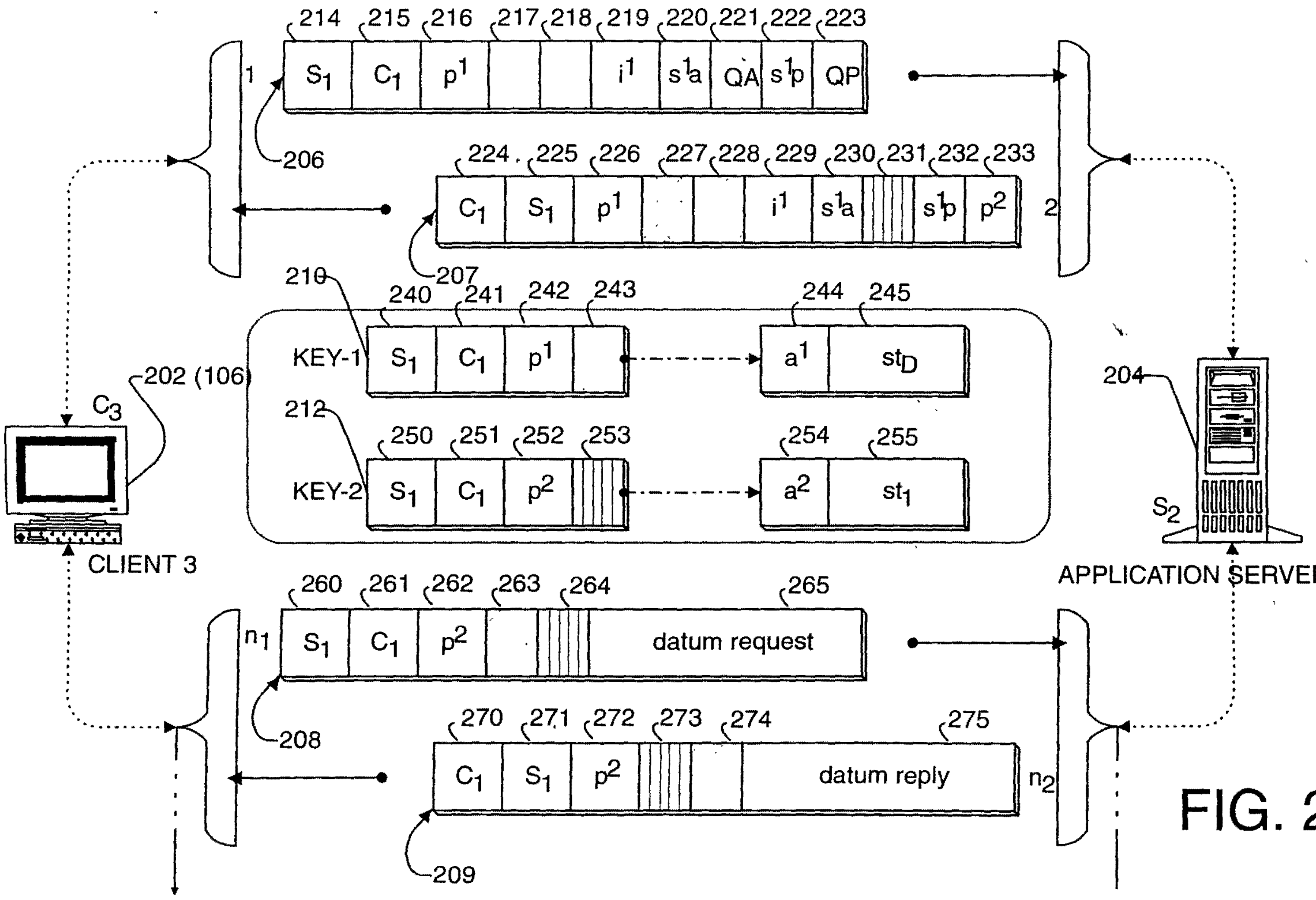

214
215
216
217
218
219
220
221
222
223
1
S1
C1
p1
i1
s1a
QA
s1p
QP
206
224
225
226
227
228
229
230
231
232
233
C1
S1
p1
i1
s1a
s1p
p2
2
207
210
240
241
242
243
244
245
KEY-1
S1
C1
p1
a1
stD
212
250
251
252
253
254
255
KEY-2
S1
C1
p2
a2
st1
202 (106)
C3
CLIENT 3
204
S2
APPLICATION SERVER 2
260
261
262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

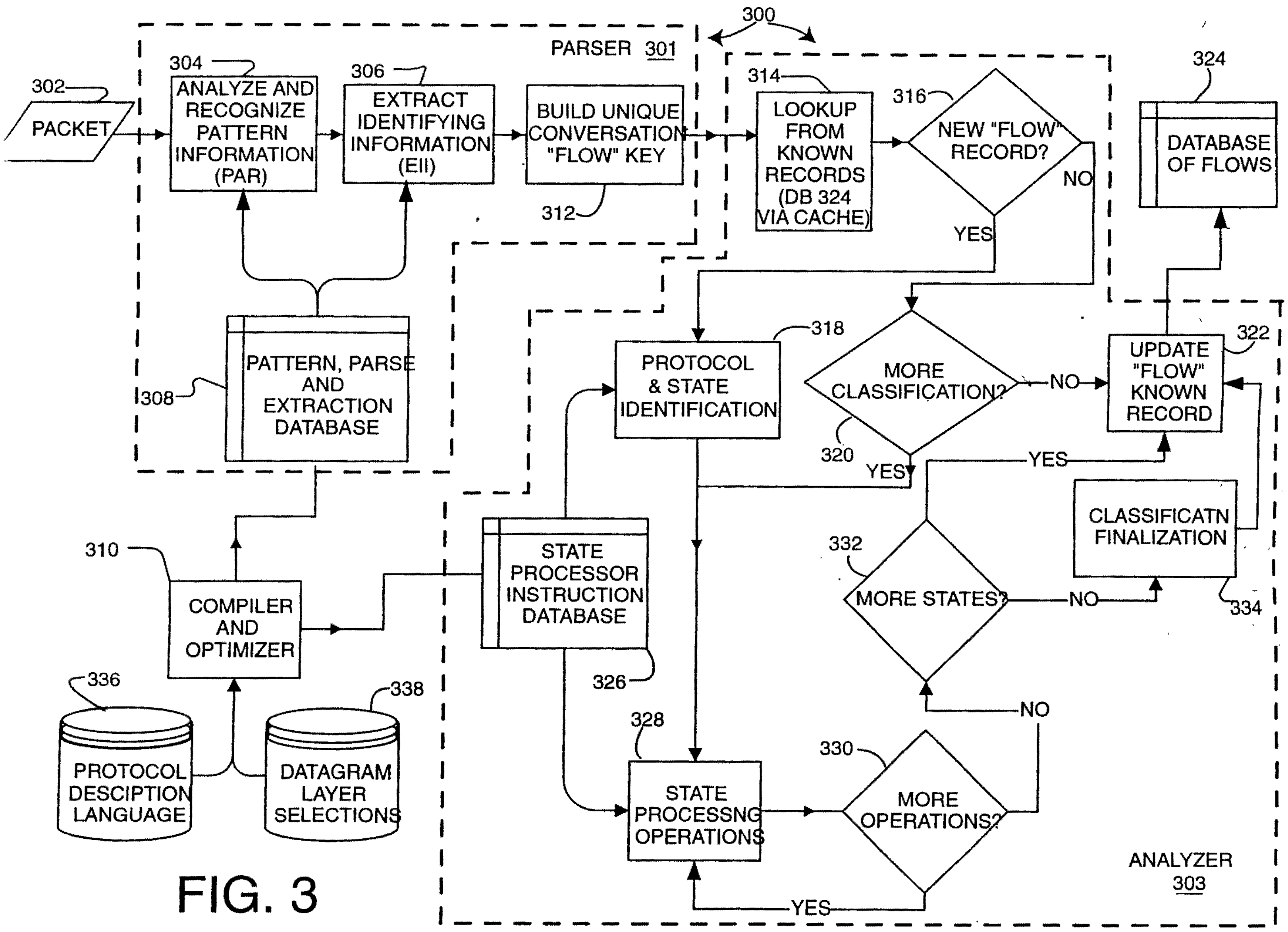

300
PARSER 301
302
PACKET
304
ANALYZE AND RECOGNIZE PATTERN INFORMATION (PAR)
306
EXTRACT IDENTIFYING INFORMATION (EII)
BUILD UNIQUE CONVERSATION "FLOW" KEY
312
308
PATTERN, PARSE AND EXTRACTION DATABASE
314
LOOKUP FROM KNOWN RECORDS (DB 324 VIA CACHE)
316
NEW "FLOW" RECORD?
NO
YES
324
DATABASE OF FLOWS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

FIG. 3

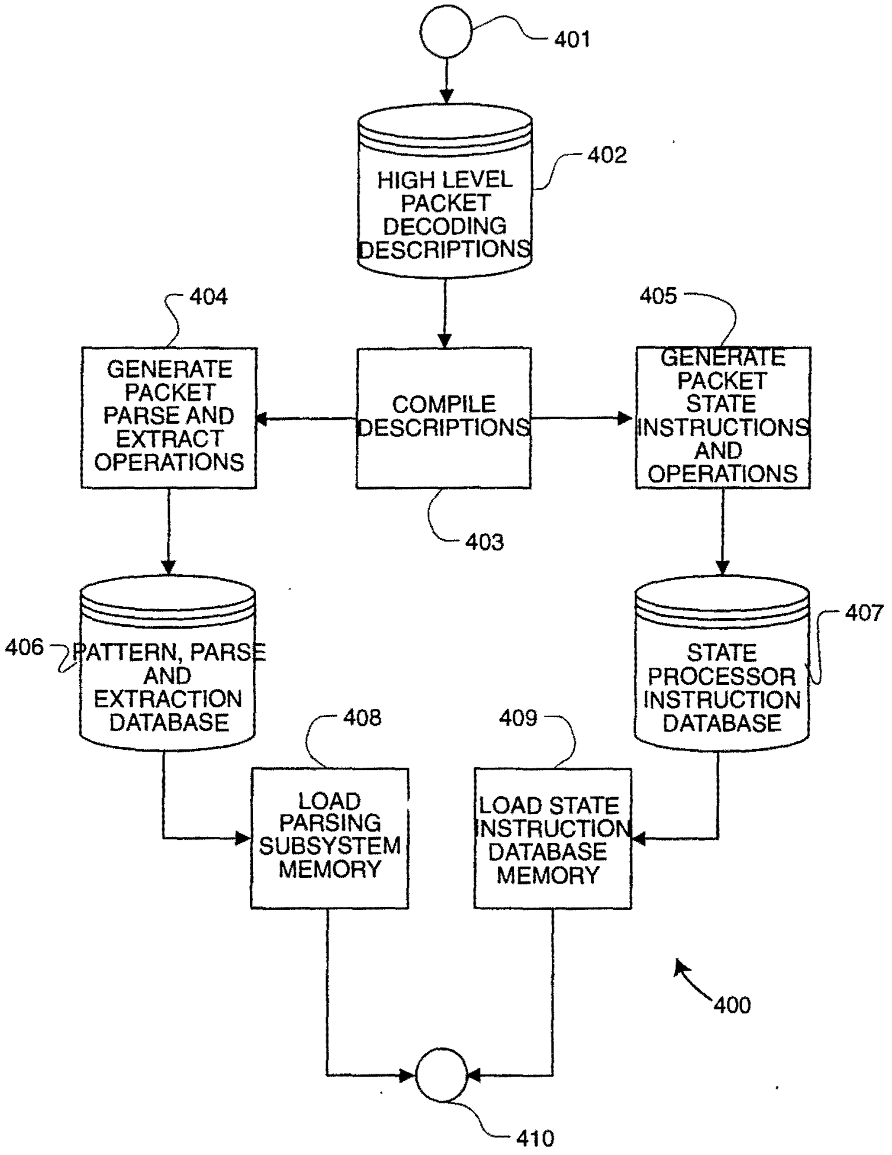

401
HIGH LEVEL PACKET DECODING DESCRIPTIONS
402
COMPILE DESCRIPTIONS
403
GENERATE PACKET PARSE AND EXTRACT OPERATIONS
404
GENERATE PACKET STATE INSTRUCTIONS AND OPERATIONS
405
PATTERN, PARSE AND EXTRACTION DATABASE
406
STATE PROCESSOR INSTRUCTION DATABASE
407
LOAD PARSING SUBSYSTEM MEMORY
408
LOAD STATE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

FIG. 4

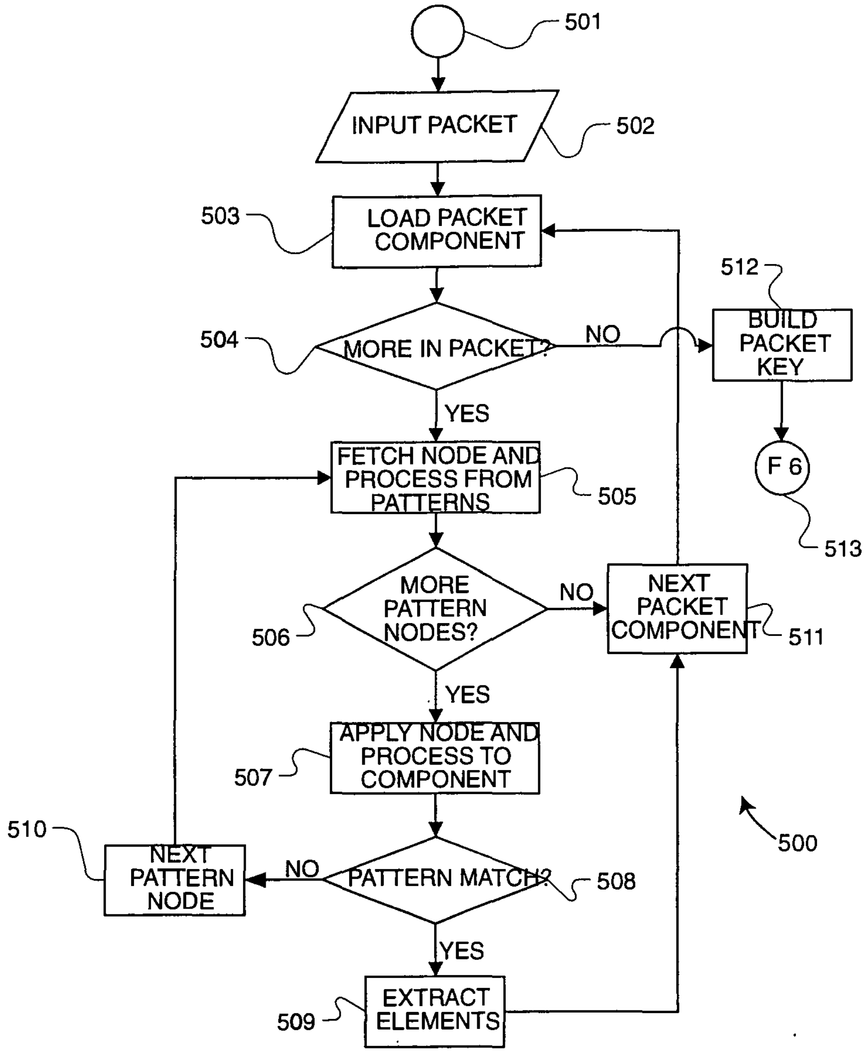

501
INPUT PACKET
502
503
LOAD PACKET COMPONENT
504
MORE IN PACKET?
NO
YES
512
BUILD PACKET KEY
F 6
513
FETCH NODE AND PROCESS FROM PATTERNS
505
MORE PATTERN NODES?
506
NO
NEXT PACKET COMPONENT
511
YES
APPLY NODE AND PROCESS TO COMPONENT
507
500
510
NEXT PATTERN NODE
NO
PATTERN MATCH?
508
YES
EXTRACT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

FIG. 5

-001-1

6/18

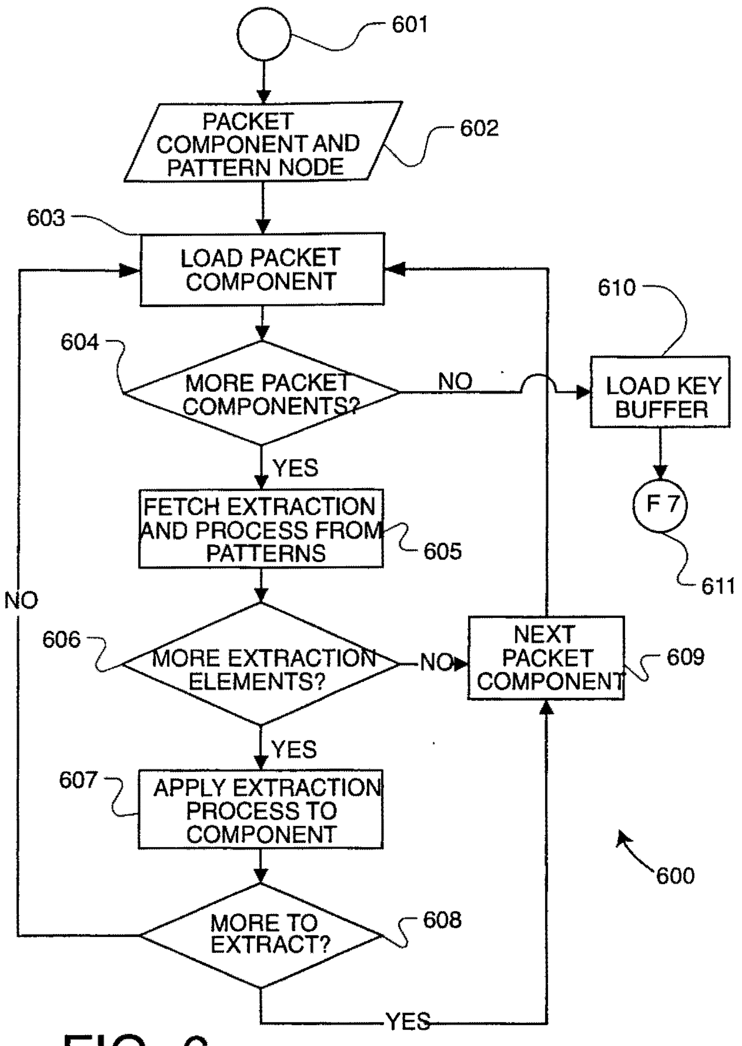


FIG. 6

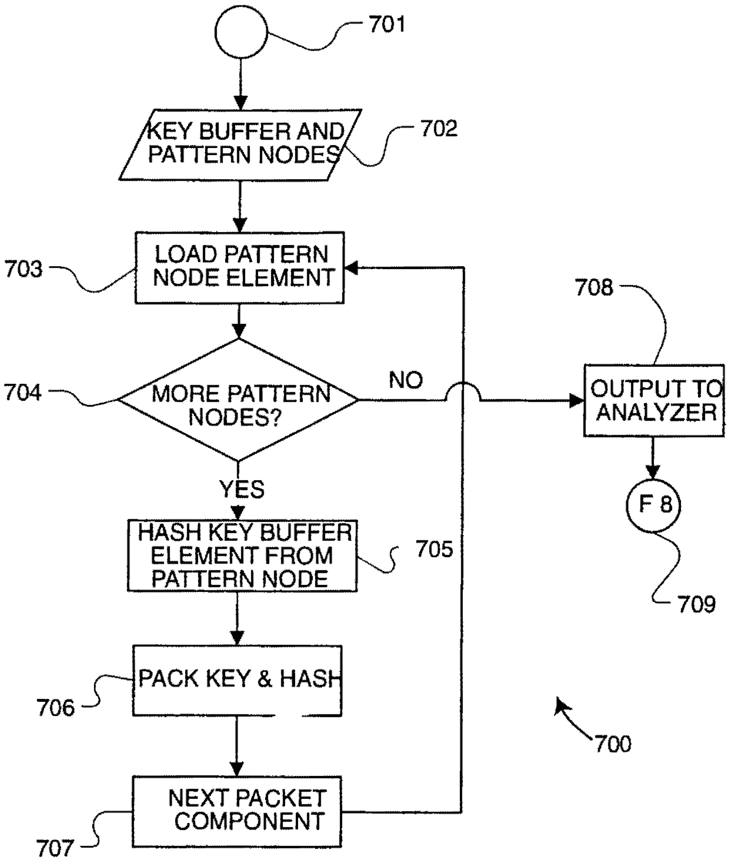

701
KEY BUFFER AND PATTERN NODES
702
LOAD PATTERN NODE ELEMENT
703
MORE PATTERN NODES?
704
NO
YES
OUTPUT TO ANALYZER
708
F 8
709
HASH KEY BUFFER ELEMENT FROM PATTERN NODE
705
PACK KEY & HASH
706
NEXT PACKET COMPONENT
707
700

FIG. 7

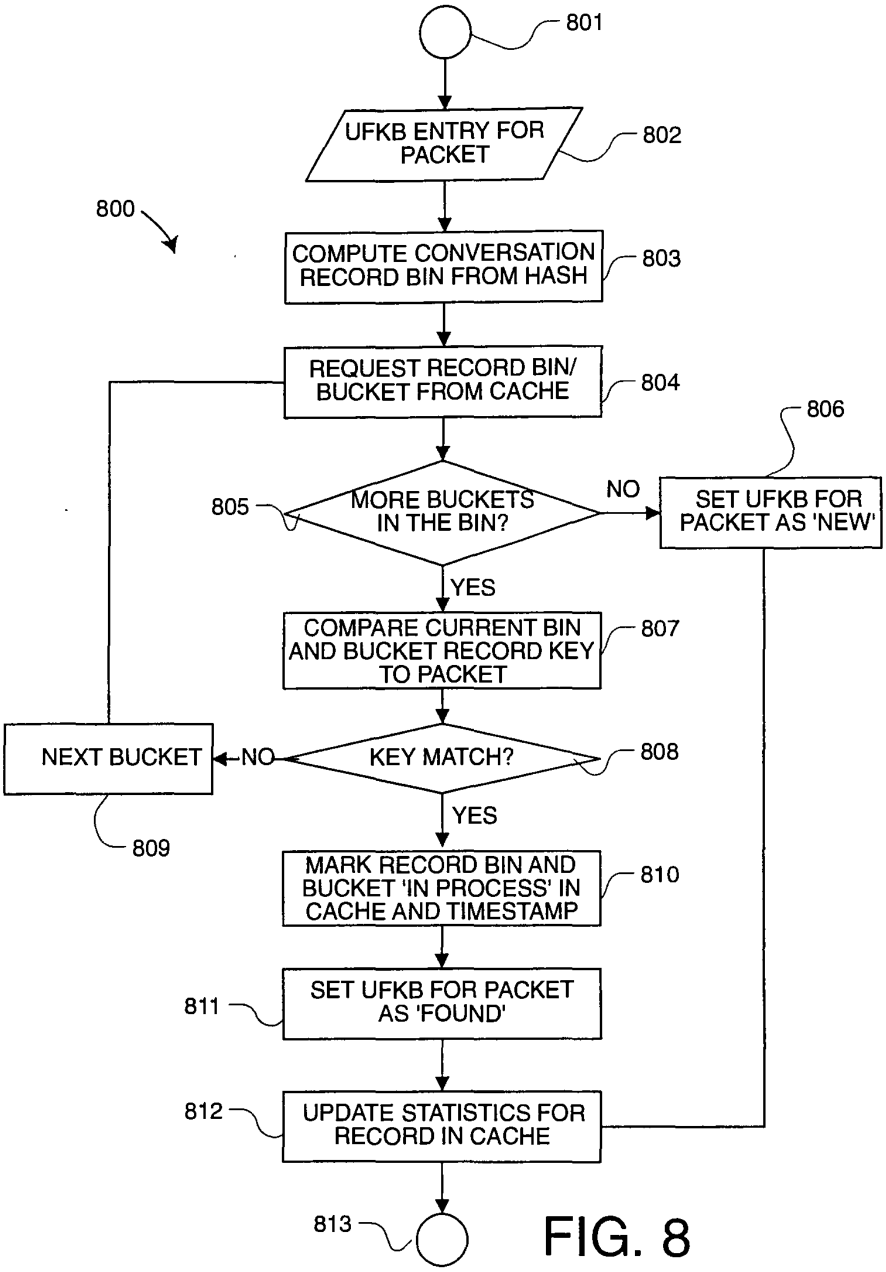
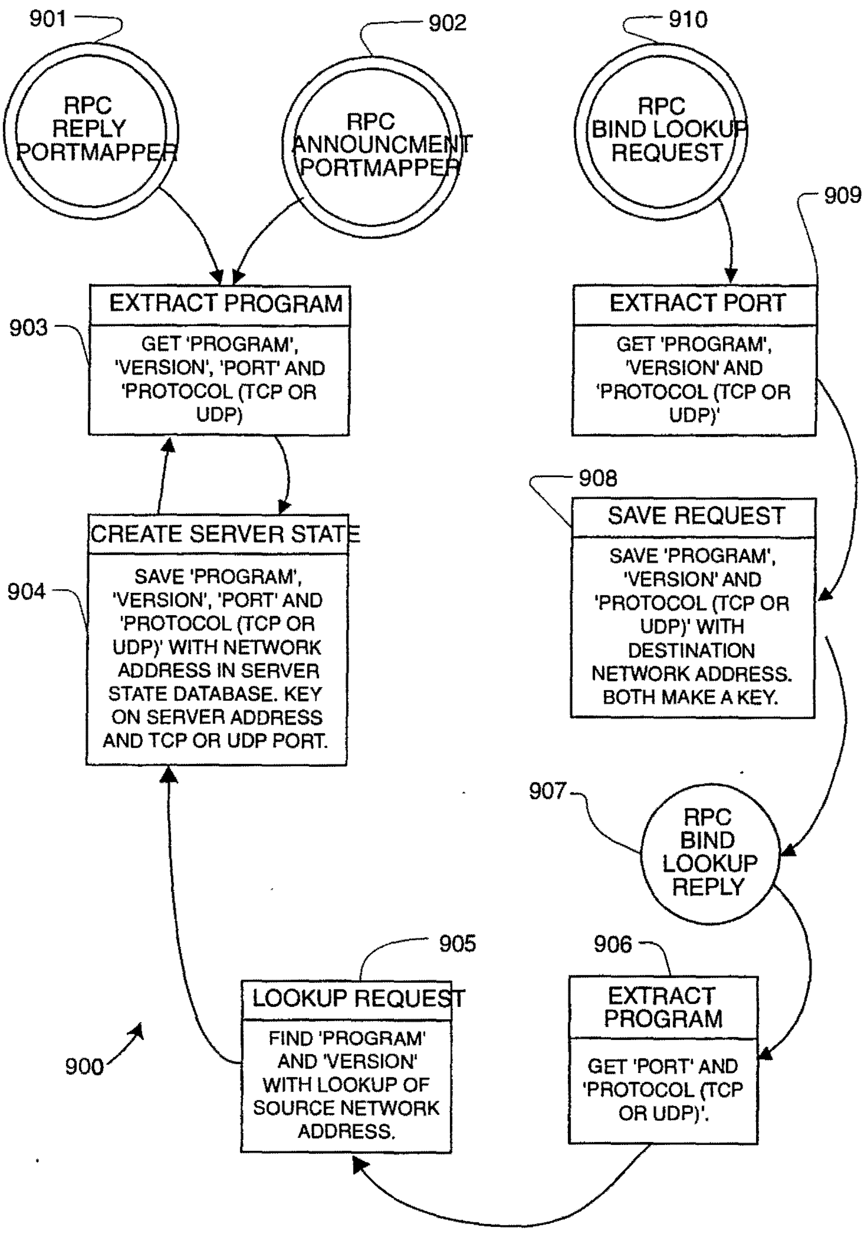

801
UFKB ENTRY FOR PACKET
802
800
COMPUTE CONVERSATION RECORD BIN FROM HASH
803
REQUEST RECORD BIN/ BUCKET FROM CACHE
804
806
805
MORE BUCKETS IN THE BIN?
NO
SET UFKB FOR PACKET AS 'NEW'
YES
COMPARE CURRENT BIN AND BUCKET RECORD KEY TO PACKET
807
NEXT BUCKET
NO
KEY MATCH?
808
YES
809
MARK RECORD BIN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

FIG. 8

PRINT OF DRAWINGS
AS ORIGINALLY FILED T-001-1

9/18



901
RPC REPLY PORTMAPPER
902
RPC ANNOUNCMENT PORTMAPPER
910
RPC BIND LOOKUP REQUEST
909
903
EXTRACT PROGRAM
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.
904
CREATE SERVER STATE
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

)1-1

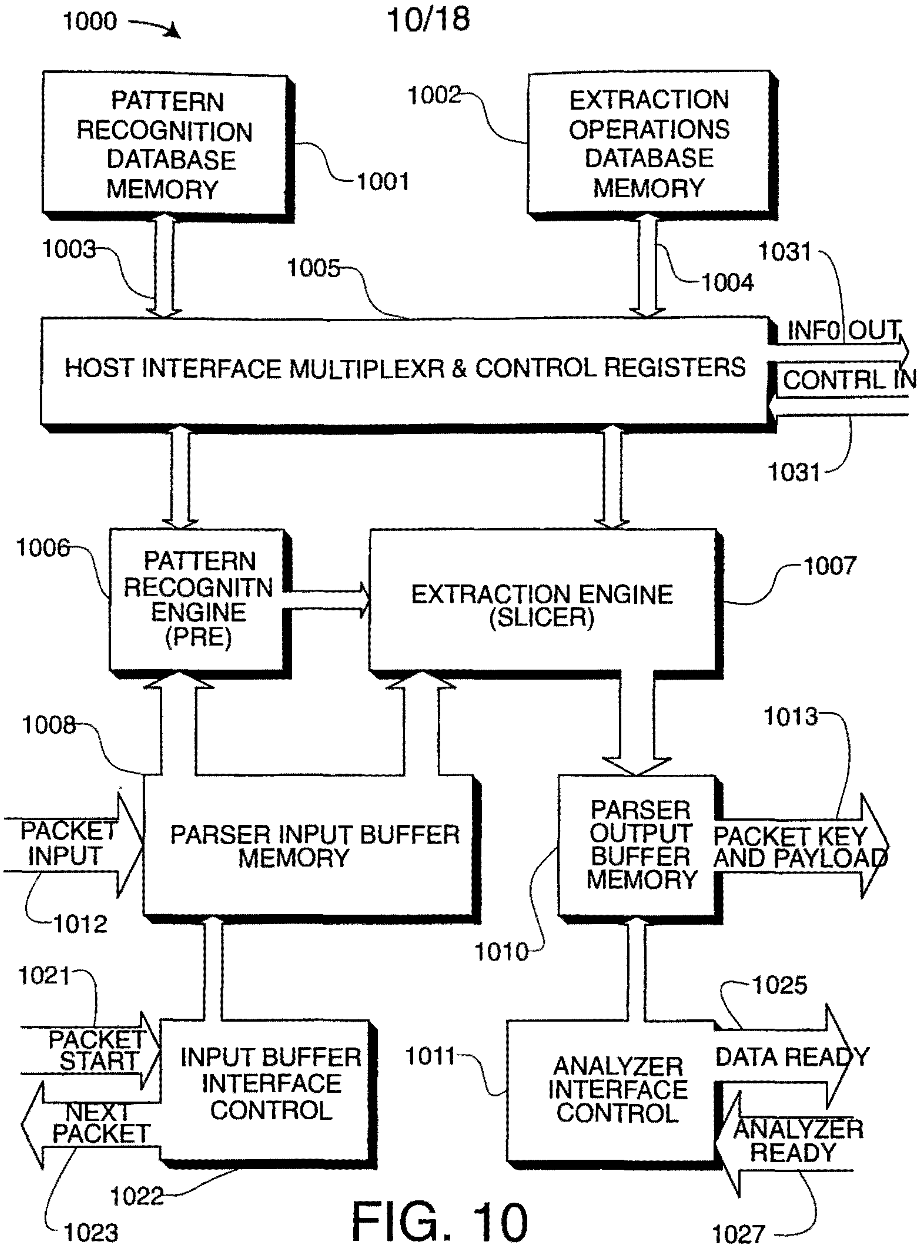

1000
PATTERN RECOGNITION DATABASE MEMORY
1001
1002
EXTRACTION OPERATIONS DATABASE MEMORY
1003
1005
1004
1031
HOST INTERFACE MULTIPLEXR & CONTROL REGISTERS
INFO OUT
CONTRL IN
1031
1006
PATTERN RECOGNITN ENGINE (PRE)
EXTRACTION ENGINE (SLICER)
1007
1008
1013
PACKET INPUT
PARSER INPUT BUFFER MEMORY
PARSER 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

001-1

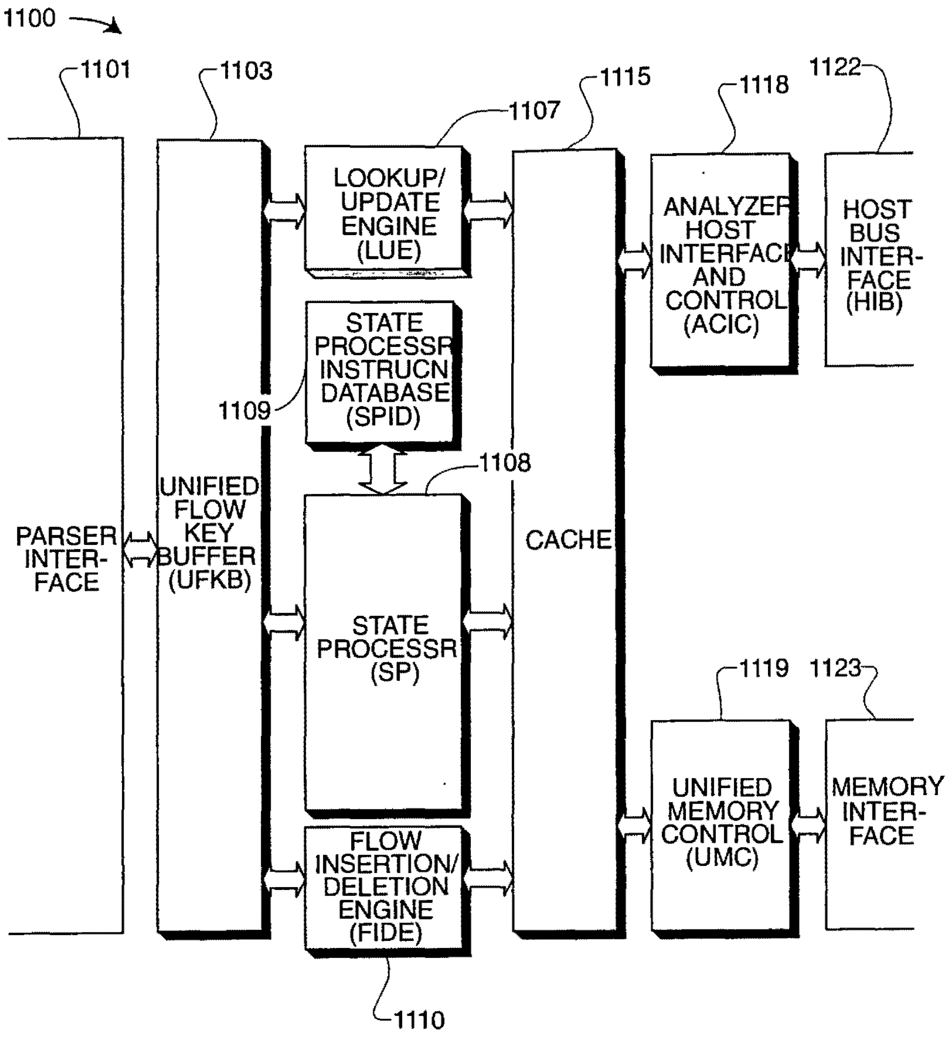

1100
1101
1103
1107
1115
1118
1122
LOOKUP/ UPDATE ENGINE (LUE)
ANALYZER HOST INTERFACE AND CONTROL (ACIC)
HOST BUS INTER-FACE (HIB)
STATE PROCESSR INSTRUCN DATABASE (SPID)
1109
1108
PARSER INTER-FACE
UNIFIED FLOW KEY BUFFER (UFKB)
CACHE
STATE PROCESSR (SP)
1119
1123
UNIFIED MEMORY CONTROL (UMC)
MEMORY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

FIG. 11

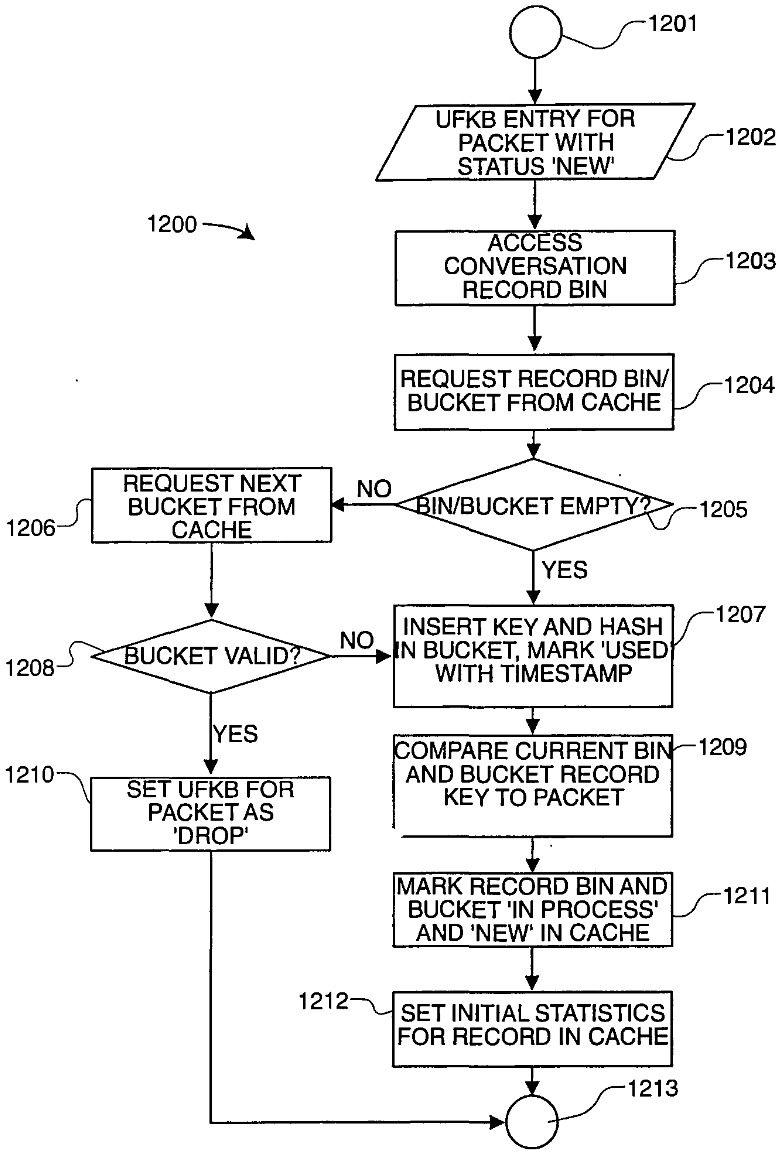

1201
UFKB ENTRY FOR PACKET WITH STATUS 'NEW'
1202
1200
ACCESS CONVERSATION RECORD BIN
1203
REQUEST RECORD BIN/ BUCKET FROM CACHE
1204
BIN/BUCKET EMPTY?
1205
NO
REQUEST NEXT BUCKET FROM CACHE
1206
YES
INSERT KEY AND HASH IN BUCKET, MARK 'USED' WITH TIMESTAMP
1207
BUCKET VALID?
1208
NO
YES
SET UFKB FOR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

FIG. 12

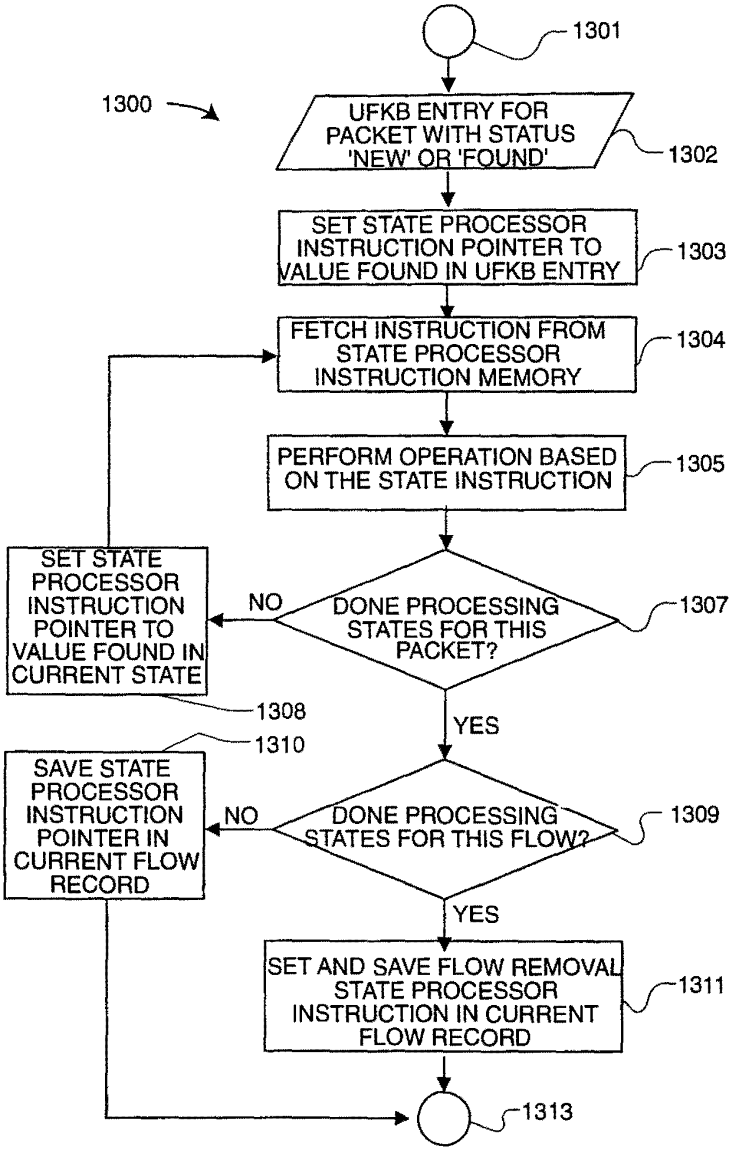

1301
1300
UFKB ENTRY FOR PACKET WITH STATUS 'NEW' OR 'FOUND'
1302
SET STATE PROCESSOR INSTRUCTION POINTER TO VALUE FOUND IN UFKB ENTRY
1303
FETCH INSTRUCTION FROM STATE PROCESSOR INSTRUCTION MEMORY
1304
PERFORM OPERATION BASED ON THE STATE INSTRUCTION
1305
SET STATE PROCESSOR INSTRUCTION POINTER TO VALUE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

FIG. 13

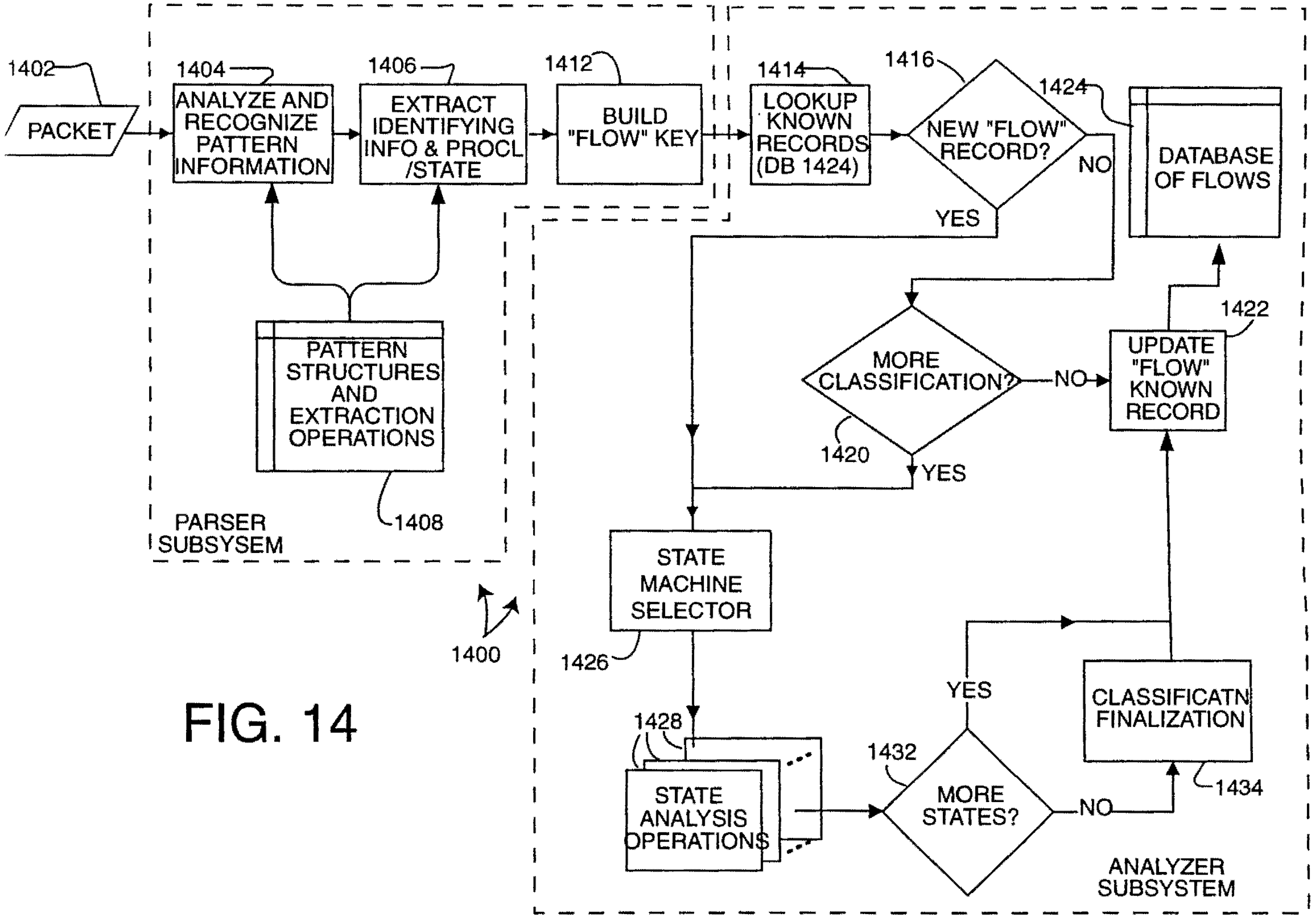

1402
PACKET
1404
ANALYZE AND RECOGNIZE PATTERN INFORMATION
1406
EXTRACT IDENTIFYING INFO & PROCL /STATE
1412
BUILD "FLOW" KEY
1414
LOOKUP KNOWN RECORDS (DB 1424)
1416
NEW "FLOW" RECORD?
1424
NO
YES
DATABASE OF FLOWS
PATTERN STRUCTURES AND EXTRACTION OPERATIONS
1408
PARSER SUBSYSEM
MORE CLASSIFICATION?
1420
NO
YES
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

14/18

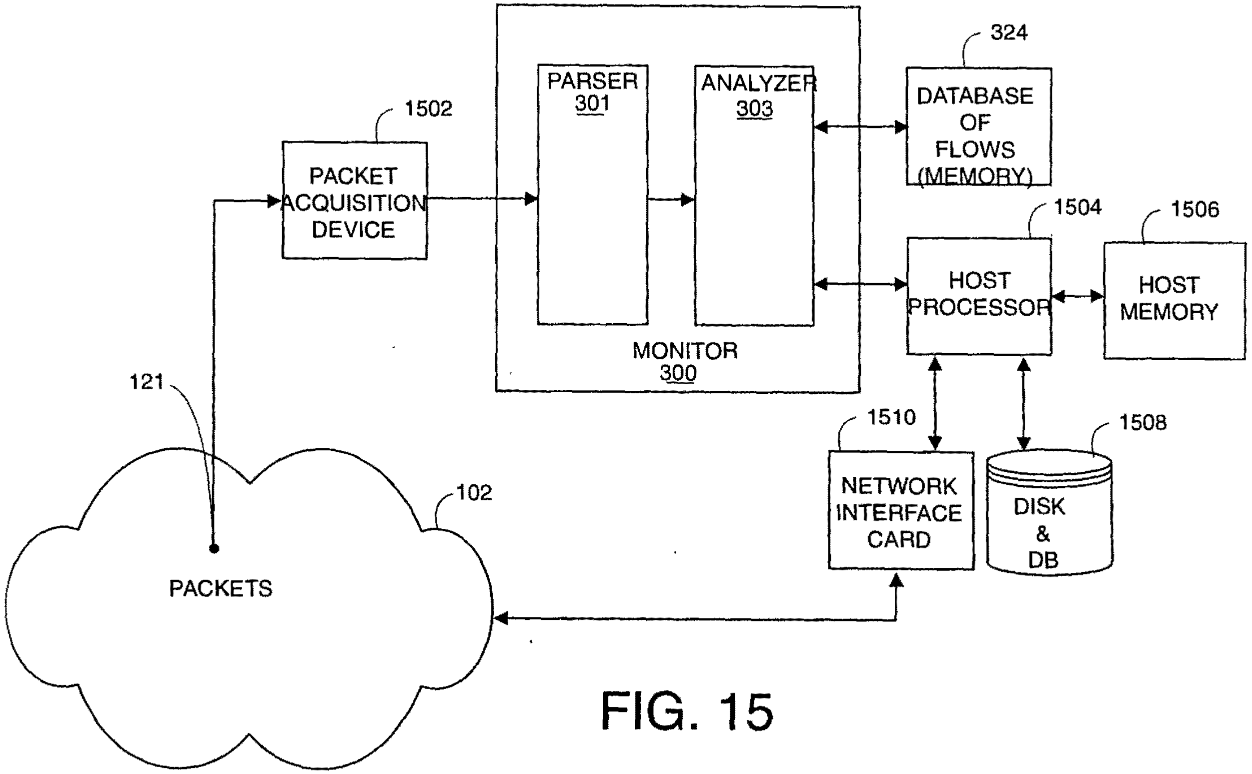

PARSER
301
ANALYZER
303
324
DATABASE OF FLOWS (MEMORY)
1502
PACKET ACQUISITION DEVICE
1504
1506
HOST PROCESSOR
HOST MEMORY
MONITOR
300
121
1510
1508
NETWORK INTERFACE CARD
DISK & DB
102
PACKETS

FIG. 15

-001-1

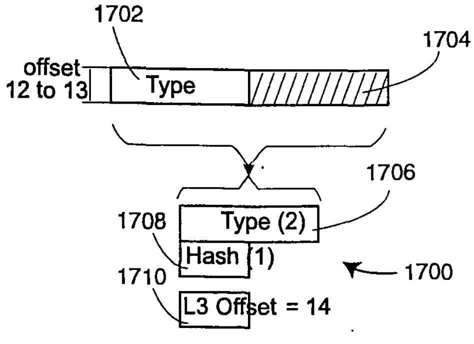


FIG. 17A

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

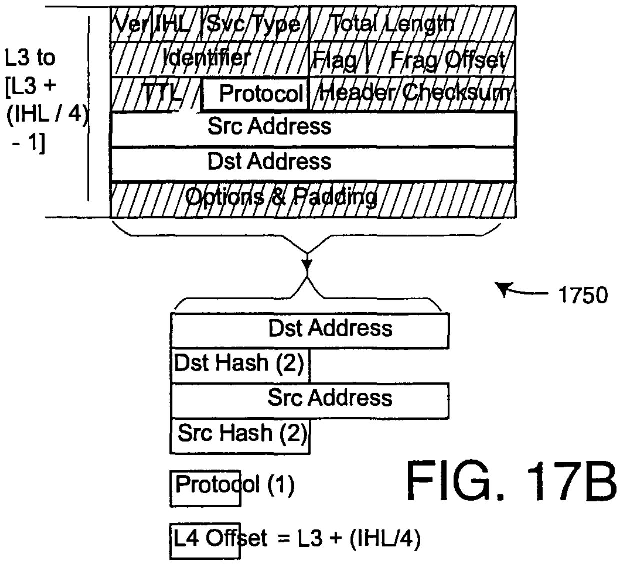


FIG. 17B

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

-001-1

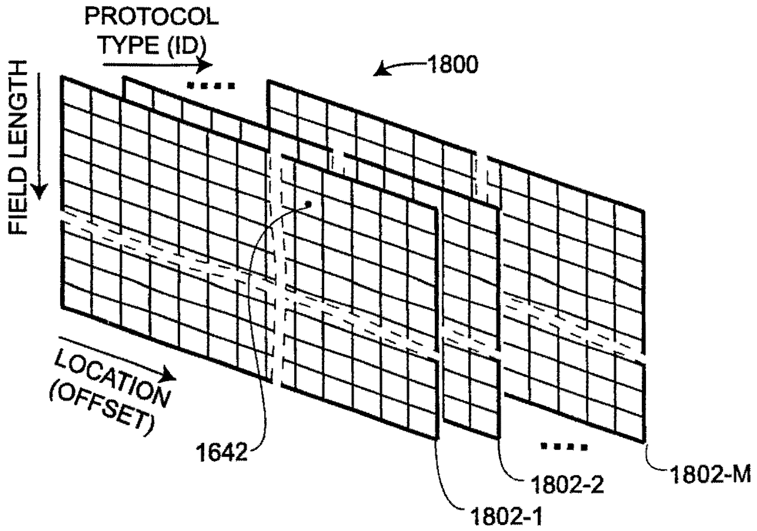


FIG. 18A

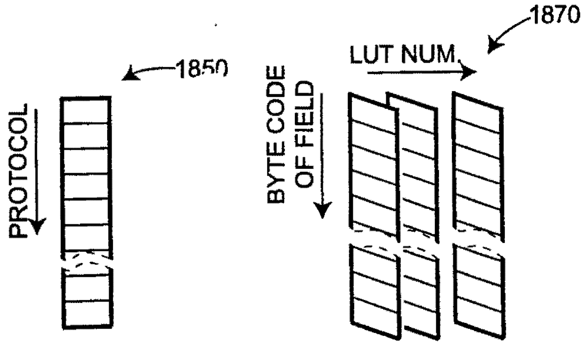


FIG. 18B

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	Total # Claims		Number Extra	X	Fee	Fee	=	Total
	Sm/Lg					Sm. Entity	Lg. Entity		
Basic Filing Fee	201/101						690	=	
Total Claims >20	203/103	59	- 20 =	34	X		702	=	
Independent Claims >3	202/102	4	- 3 =	1	X		78	=	
Mult. 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

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 MUST be returned with the reply.

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

PART 3 - OFFICE COPY

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

SECOND
#3

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)

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

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

#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:
Dov Rosenfeld
5507 College Avenue, Suite 2
Oakland, CA 94618

Direct Telephone Calls To:
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

10/3/00
Date

Declaration and Power of Attorney (Continued)
Case No; «Case CaseNumber»
APPT-001-1
Page 2

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

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**

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:
Dov Rosenfeld
5507 College Avenue, Suite 2
Oakland, CA 94618

Direct Telephone Calls To:
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**

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:
Dov Rosenfeld
5507 College Avenue, Suite 2
Oakland, CA 94618

Direct Telephone Calls To:
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

______________________ 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**

Declaration and Power of Attorney (Continued)
Case No; «Case CaseNumber»
APPT-001-1
Page 2

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**

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:
Dov Rosenfeld
5507 College Avenue, Suite 2
Oakland, CA 94618

Direct Telephone Calls To:
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**

Declaration and Power of Attorney (Continued)
Case No; «Case__CaseNumber»
Page 2 APPT-001-1

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**

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:
Dov Rosenfeld
5507 College Avenue, Suite 2
Oakland, CA 94618

Direct Telephone Calls To:
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**

Declaration and Power of Attorney (Continued)
Case No; «Case CaseNumber»
Page 2 APPT-001-1

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**

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:
Dov Rosenfeld
5507 College Avenue, Suite 2
Oakland, CA 94618

Direct Telephone Calls To:
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**

Declaration and Power of Attorney (Continued)
Case No; «Case CaseNumber»
Page 2 APPT-001-1

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 9/21/60

Inventor's Signature **Date**

#3

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 Kette
Customer Service Center
Initial Patent Examination Division (703) 308-1202
PART 2 - COPY TO BE RETURNED WITH RESPONSE

GP/~~2758~~ 2131 #4 4-16-0

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

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

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant(s): Dietz et al.	Group Art Unit: 2155
Serial No.: 09/608237	Examiner:
Filed: June 30, 2000	
Title: METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK	

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

FORM 1449 SHEET 1 OF 5.

INFORMATION DISCLOSURE STATEMENT *(Use several sheets if necessary)*	ATTY. DOCKET NO. APPT-001-1	SERIAL NO. 09/608237
	APPLICANT Dietz et *al.*	#4
	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*
	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	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.

INFORMATION DISCLOSURE STATEMENT *(Use several sheets if necessary)*	ATTY. DOCKET NO. APPT-001-1	SERIAL NO. 09/608237
	APPLICANT Dietz et *al.*	#4
	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*
	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
MM	*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 M. [illegible]	DATE CONSIDERED 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.

*Et al.*FORM - 1449 SHEET 3 OF 5.

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

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 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.

INFORMATION DISCLOSURE STATEMENT *(Use several sheets if necessary)*	ATTY. DOCKET NO. APPT-001-1	SERIAL NO. 09/60
	APPLICANT Dietz et *al.*	#4
	FILING DATE 6/30/2000	GROUP ~~2755~~ 2157

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 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 5 OF 5.

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

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 M. Mel	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.

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

Patent

#5

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	

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.FOF.. . ..   SHEET 1 OF 1.

INFORMATION DISCLOSURE STATEMENT (Use several sheets if necessary)	ATTY. DOCKET NO. APPT-001-1	SERIAL NO. 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						

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 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.

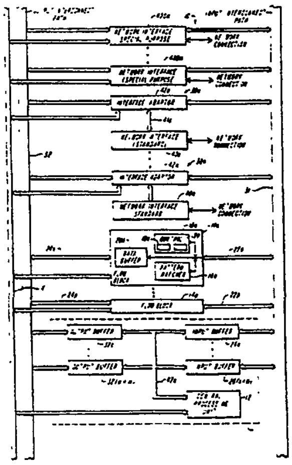
"Company Backgrounder Mar. 1988", ciscoSystems, Inc. Network Systems brochures.

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



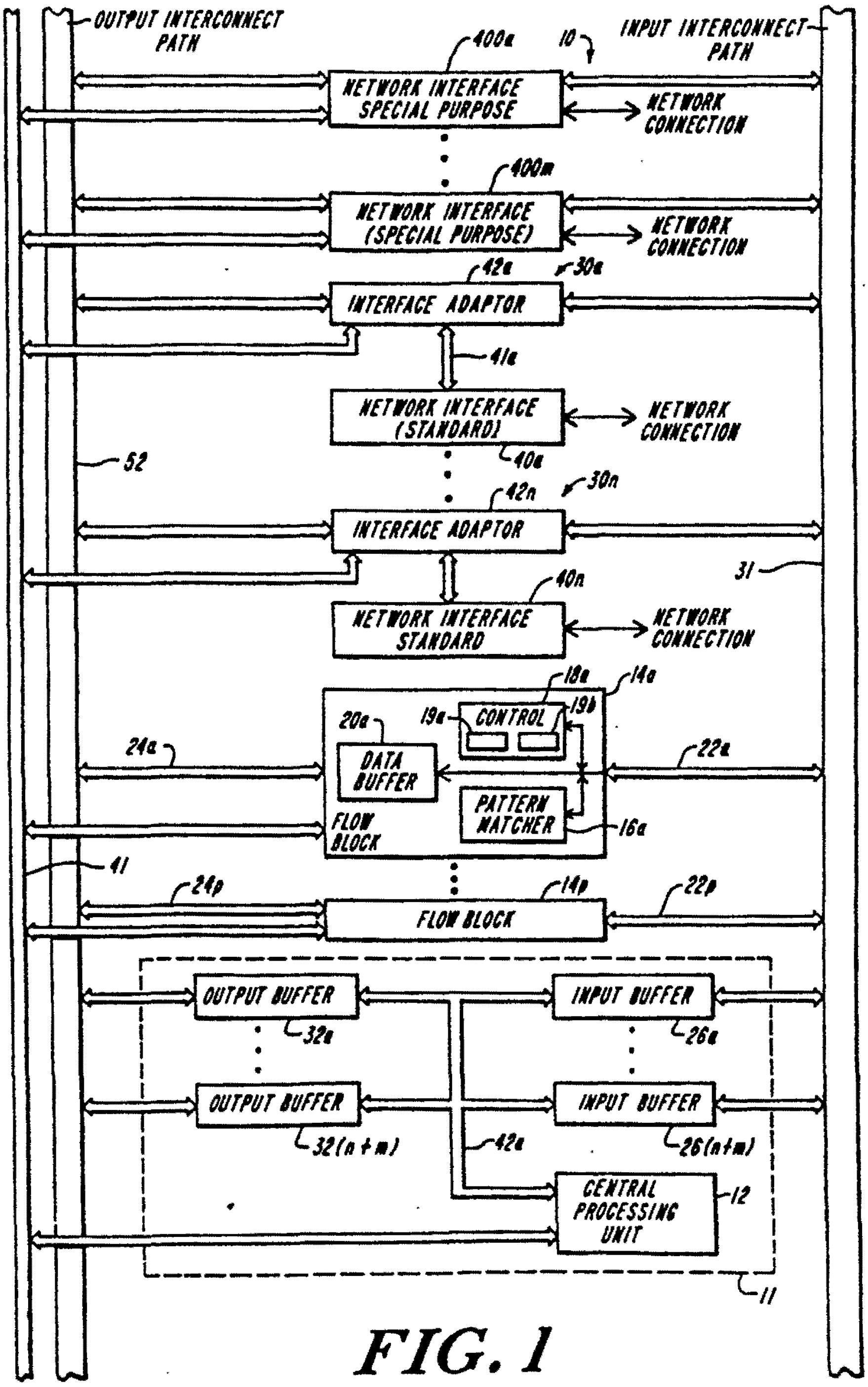

OUTPUT INTERCONNECT PATH
INPUT INTERCONNECT PATH
10
400a
NETWORK INTERFACE SPECIAL PURPOSE
NETWORK CONNECTION
400m
NETWORK INTERFACE (SPECIAL PURPOSE)
NETWORK CONNECTION
42a
30a
INTERFACE ADAPTOR
41a
NETWORK INTERFACE (STANDARD)
NETWORK CONNECTION
40a
52
42n
30n
INTERFACE ADAPTOR
31
40n
NETWORK INTERFACE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

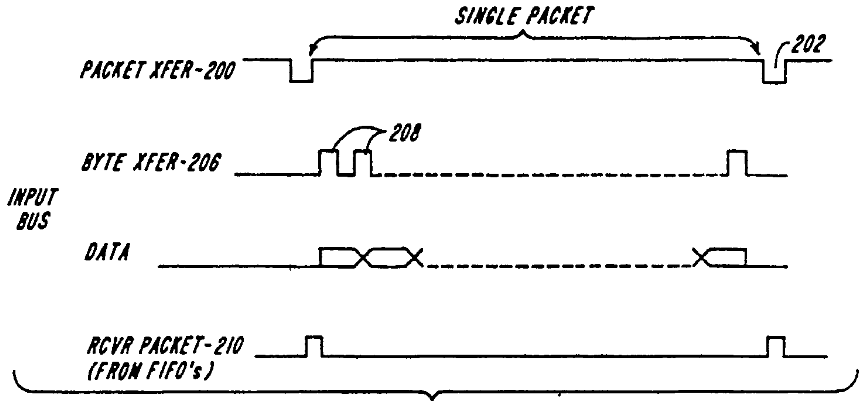

SINGLE PACKET
PACKET XFER-200
202
BYTE XFER-206
208
INPUT BUS
DATA
RCVR PACKET-210 (FROM FIFO's)

FIG. 2

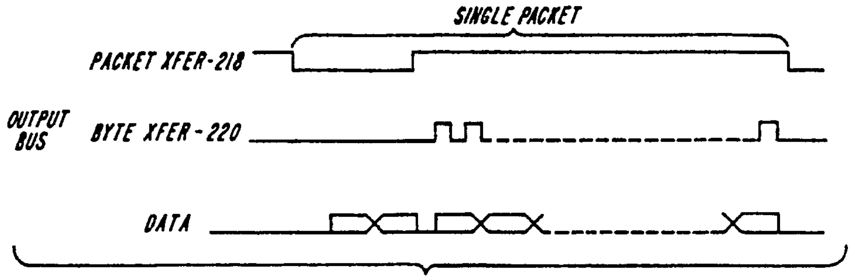

SINGLE PACKET
PACKET XFER-218
OUTPUT BUS
BYTE XFER-220
DATA

FIG. 3

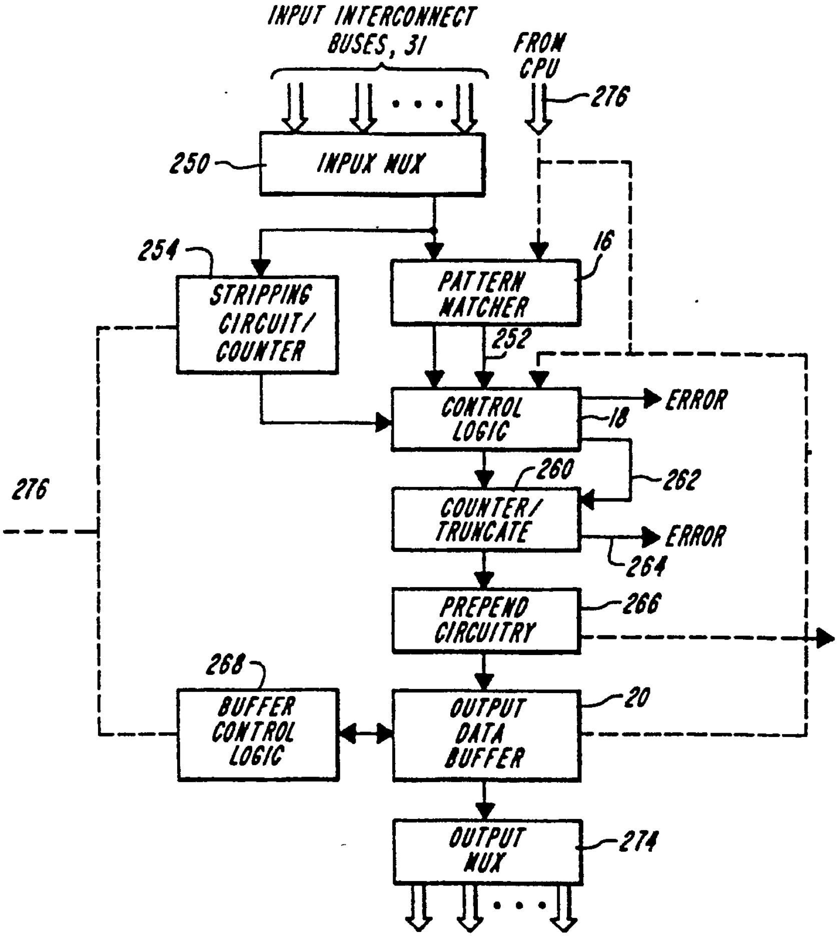

INPUT INTERCONNECT BUSES, 31
FROM CPU
276
250
INPUX MUX
254
STRIPPING CIRCUIT/ COUNTER
PATTERN MATCHER
16
252
CONTROL LOGIC
18
ERROR
276
260
262
COUNTER/ TRUNCATE
ERROR
264
PREPEND CIRCUITRY
266
268
BUFFER CONTROL LOGIC
OUTPUT DATA BUFFER
20
OUTPUT MUX
274

FIG. 4

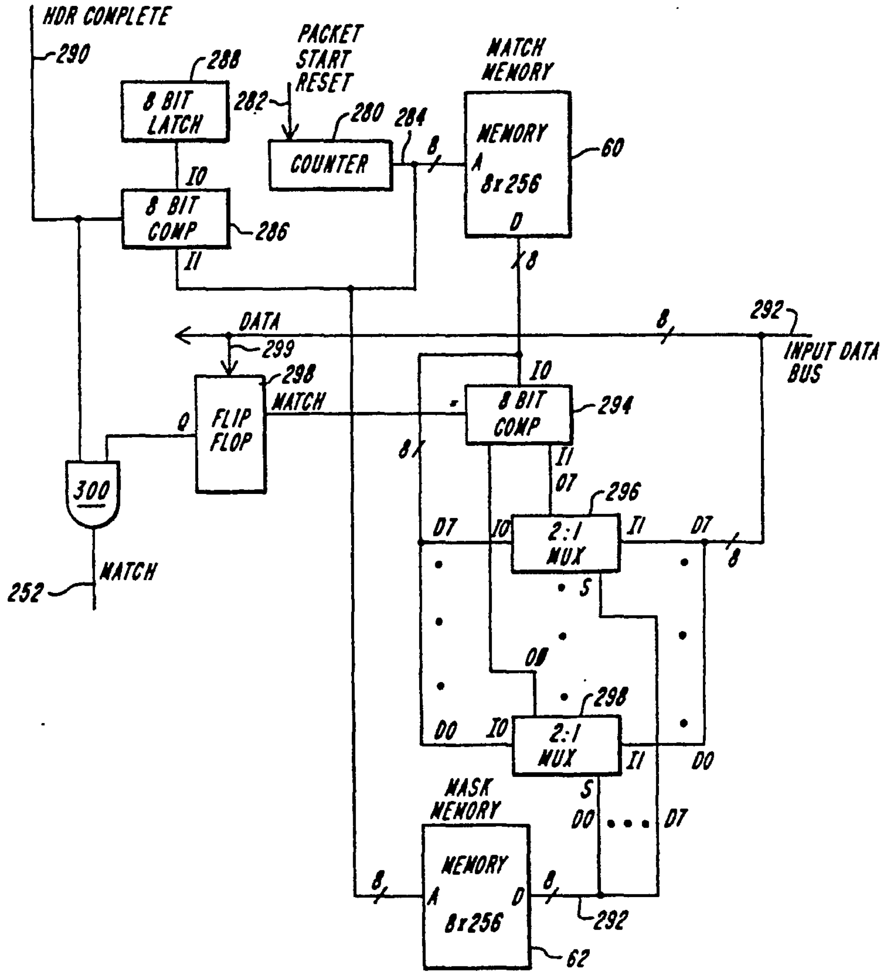

HDR COMPLETE
290
288
8 BIT LATCH
282
PACKET START RESET
280
284
COUNTER
8
MATCH MEMORY
MEMORY
A
8x256
D
60
I0
8 BIT COMP
286
I1
8
DATA
8
292
INPUT DATA BUS
299
298
Q
FLIP FLOP
MATCH
I0
8 BIT COMP
294
8
I1
07
296
300
D7
I0
2:1 MUX
I1
D7
8
S
MATCH
252
08
298
D0
I0
2:1 MUX
I1
D0
MASK MEMORY
S
D0
D7
MEMORY
8
A
D
8
8x256
292
62

FIG. 5

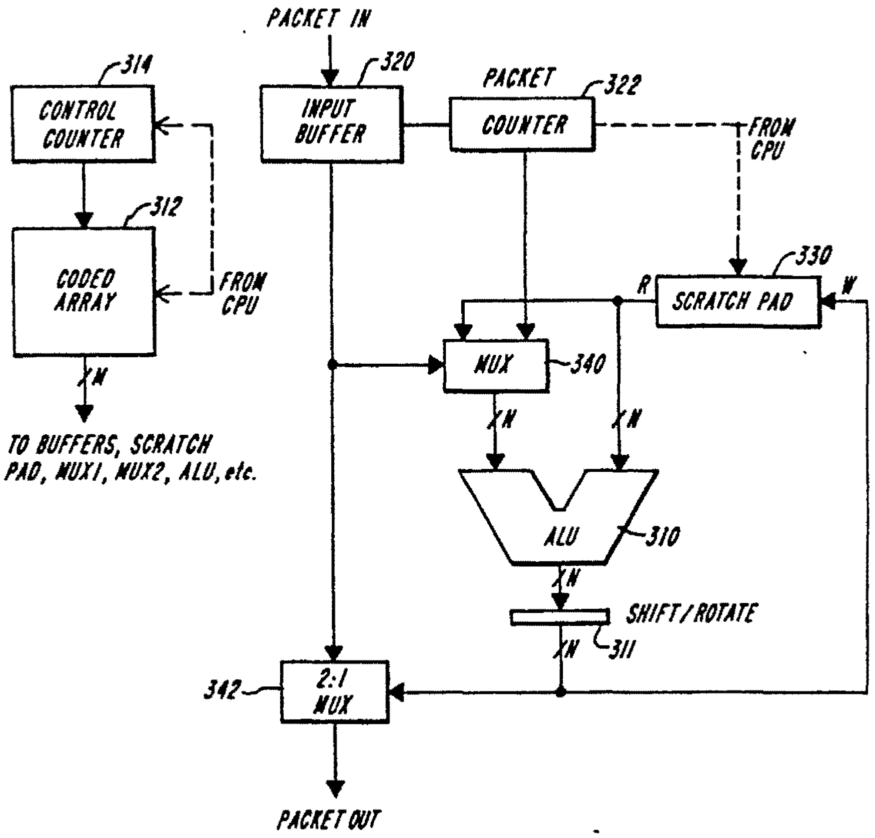

PACKET IN
314
CONTROL COUNTER
320
INPUT BUFFER
PACKET COUNTER
322
FROM CPU
312
CODED ARRAY
FROM CPU
330
R
SCRATCH PAD
W
MUX
340
XN
TO BUFFERS, SCRATCH PAD, MUX1, MUX2, ALU, etc.
XN
XN
ALU
310
XN
SHIFT/ROTATE
XN
311
342
2:1 MUX
PACKET OUT

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 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

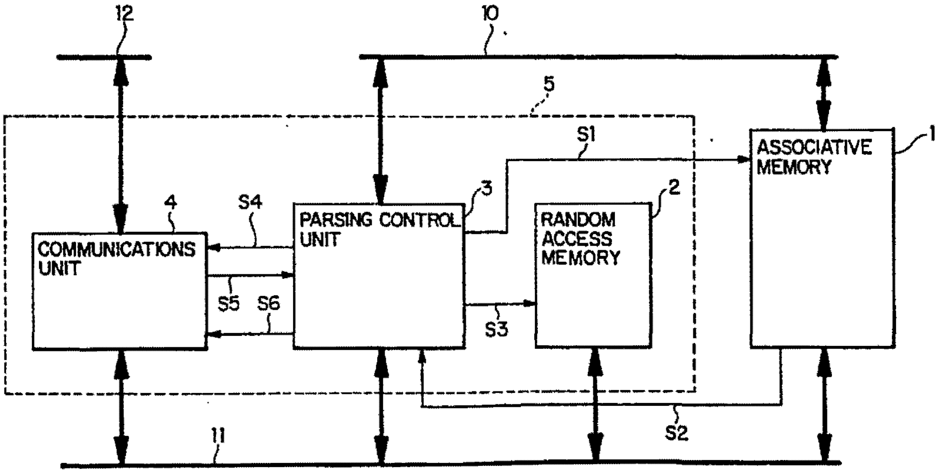
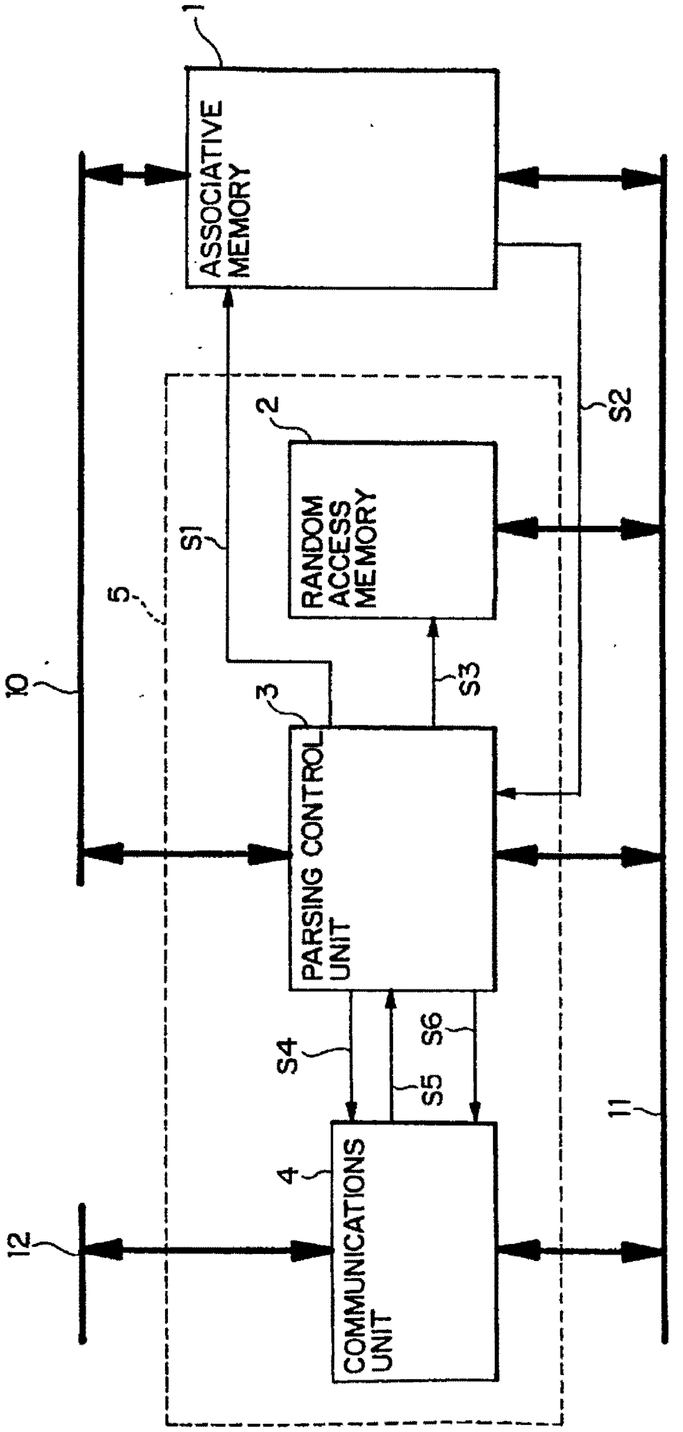


FIG. 1


1
ASSOCIATIVE MEMORY
2
RANDOM ACCESS MEMORY
3
PARSING CONTROL UNIT
4
COMMUNICATIONS UNIT
5
10
11
12
S1
S2
S3
S4
S5
S6

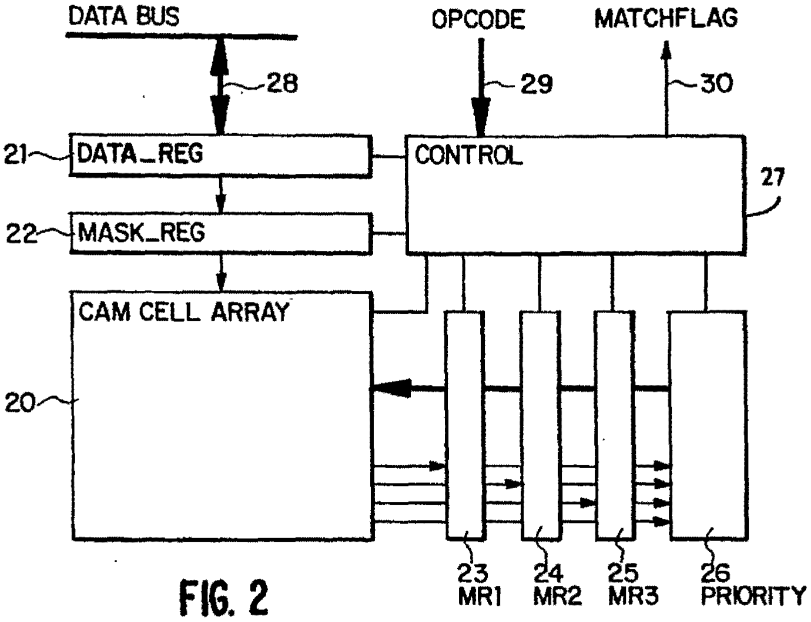

DATA BUS
OPCODE
MATCHFLAG
28
29
30
21
DATA_REG
CONTROL
27
22
MASK_REG
CAM CELL ARRAY
20
23
MR1
24
MR2
25
MR3
26
PRIORITY

FIG. 2

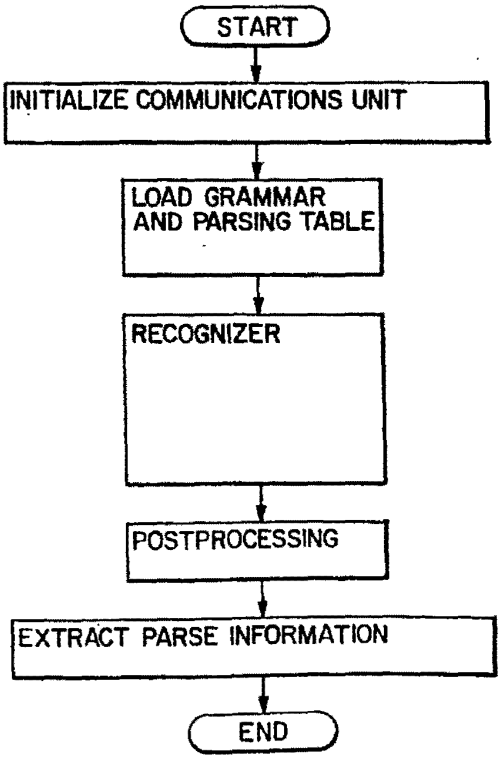

START
INITIALIZE COMMUNICATIONS UNIT
LOAD GRAMMAR AND PARSING TABLE
RECOGNIZER
POSTPROCESSING
EXTRACT PARSE INFORMATION
END

FIG. 6

FIG. 3

NUMBER	RULE
0	Z → S $
1	S → NP VP
2	NP → "JOHN"
3	VP → "THINKS"

```
"JOHN THINKS $"
↑     ↑     ↑↑
0     1     23
```

FIG. 5

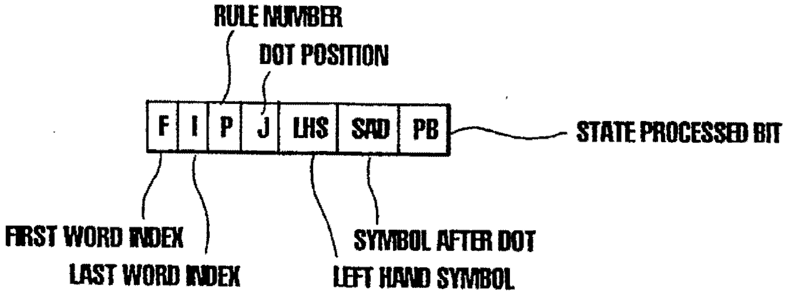


FIG. 7A

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

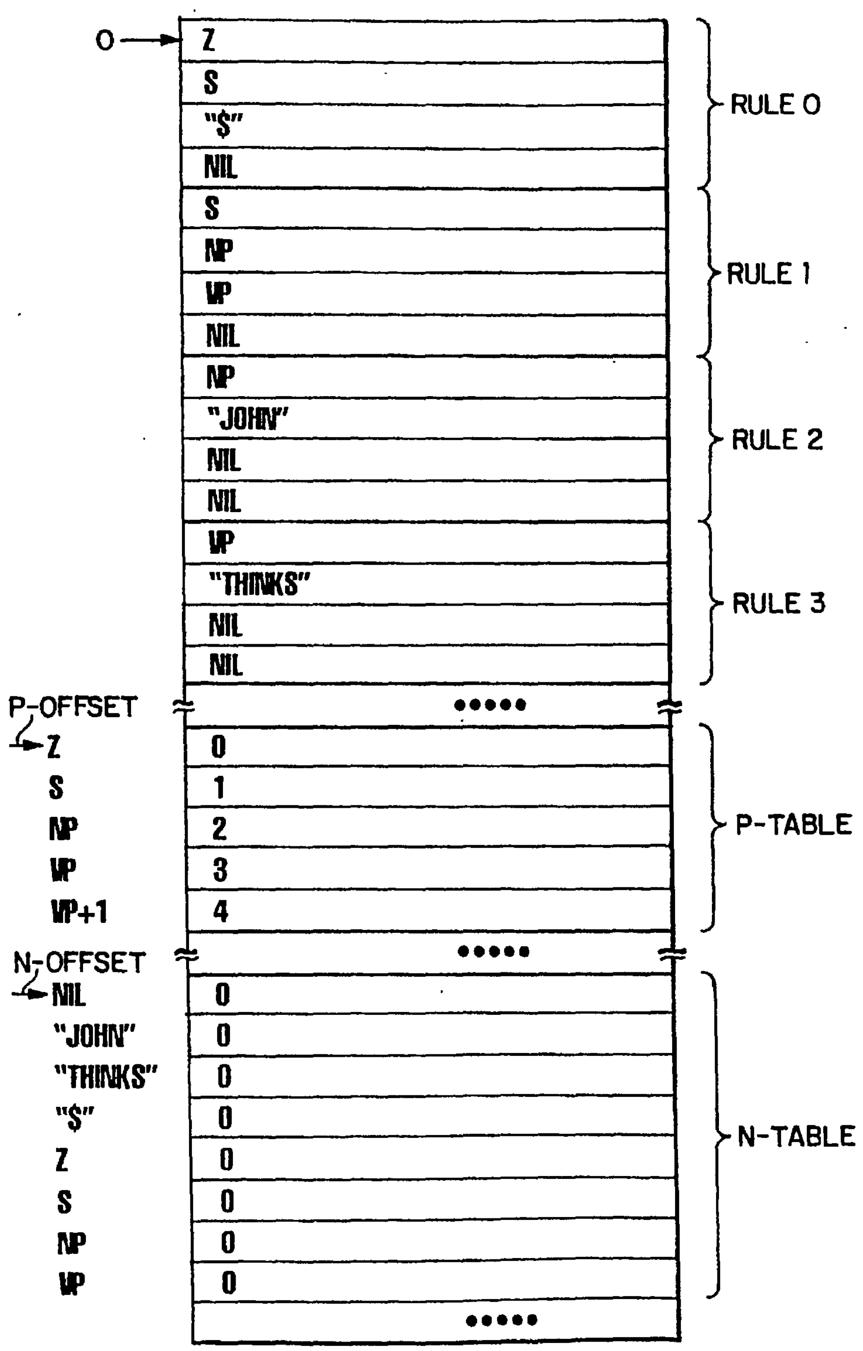

0
Z
S
"$"
NIL
RULE 0
S
NP
VP
NIL
RULE 1
NP
"JOHN"
NIL
NIL
RULE 2
VP
"THINKS"
NIL
NIL
RULE 3
P-OFFSET
•••••
Z 0
S 1
NP 2
VP 3
VP+1 4
P-TABLE
N-OFFSET
•••••
NIL 0
"JOHN" 0
"THINKS" 0
"$" 0
Z 0
S 0
NP 0
VP 0
N-TABLE
•••••

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 inefficiently 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 architecture 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 intercomnection 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;
        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

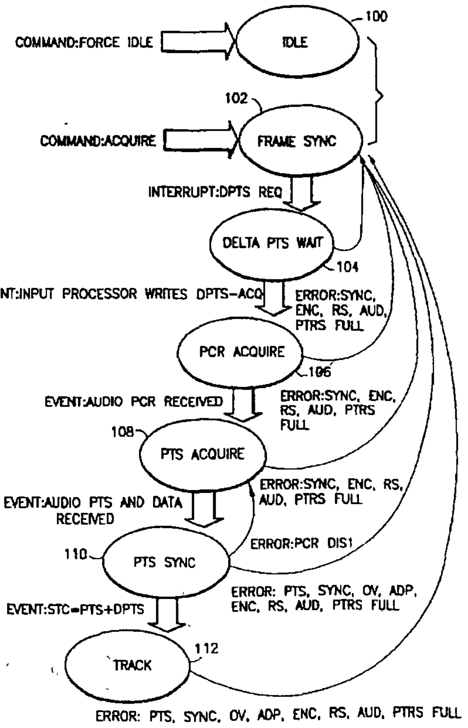
5,365,272	11/1994	Siracusa	348/461
5,376,969	12/1994	Zdepski	348/466
5,467,342	11/1995	Logston et al.	370/253
5,517,250	5/1996	Hoogenboom et al.	348/467
5,537,409	7/1996	Moriyama et al.	370/471

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


COMMAND:FORCE IDLE
IDLE
100
102
COMMAND:ACQUIRE
FRAME SYNC
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
EVENT:AUDIO PTS AND DATA RECEIVED
ERROR:SYNC, ENC, RS, AUD, PTRS FULL
ERROR:PCR DIS1
110
PTS SYNC
ERROR: PTS, SYNC, OV, ADP, ENC, RS, AUD, PTRS FULL
EVENT:STC=PTS+DPTS
TRACK
112
ERROR: PTS, SYNC, OV, ADP, ENC, RS, AUD, PTRS FULL

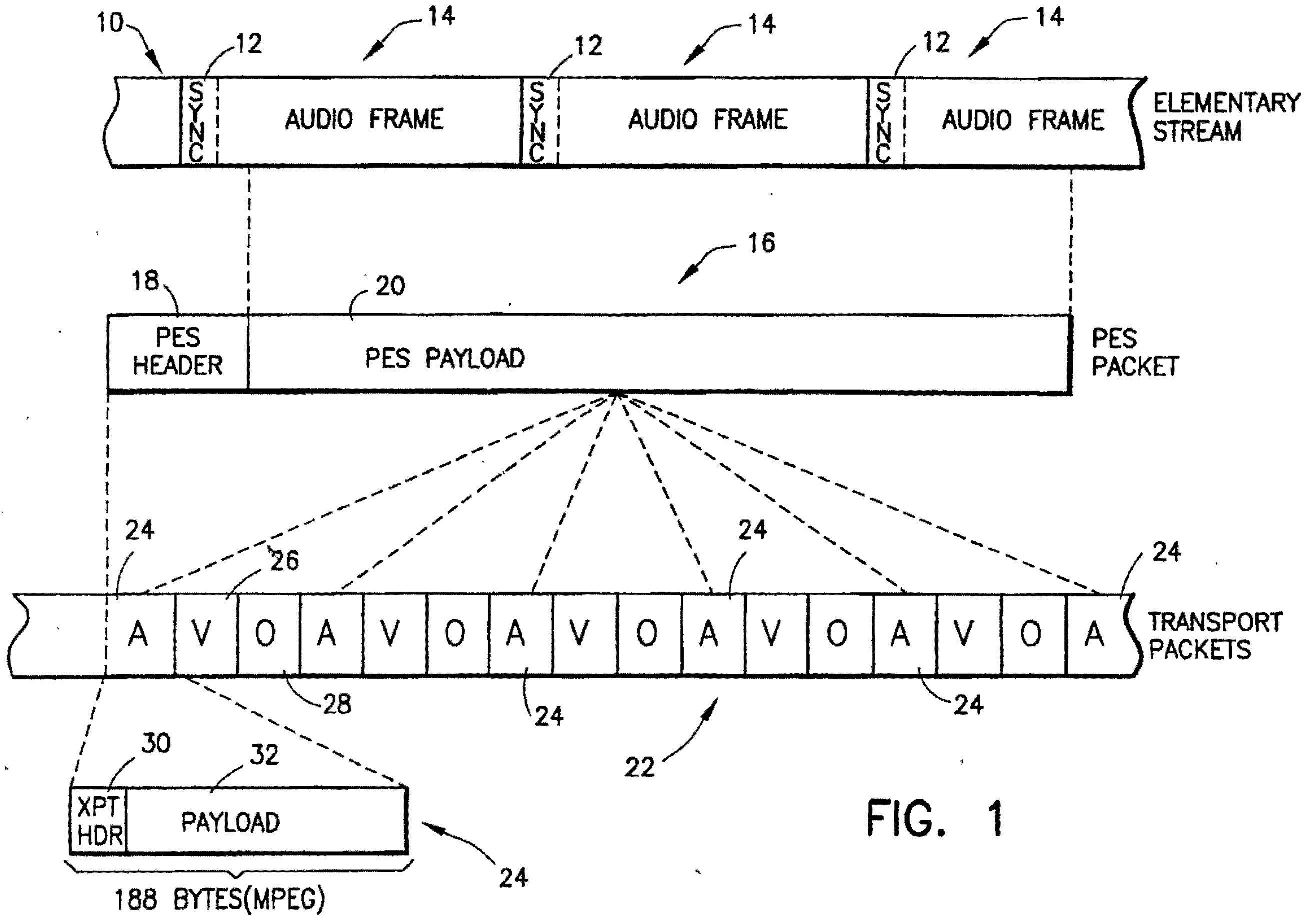

10
12
14
12
14
12
14
SYNC
AUDIO FRAME
SYNC
AUDIO FRAME
SYNC
AUDIO FRAME
ELEMENTARY STREAM
16
18
20
PES HEADER
PES PAYLOAD
PES PACKET
24
26
24
24
A
V
O
A
V
O
A
V
O
A
V
O
A
V
O
A
TRANSPORT PACKETS
28
24
24
22
30
32
XPT HDR
PAYLOAD
24
188 BYTES(MPEG)

FIG. 1

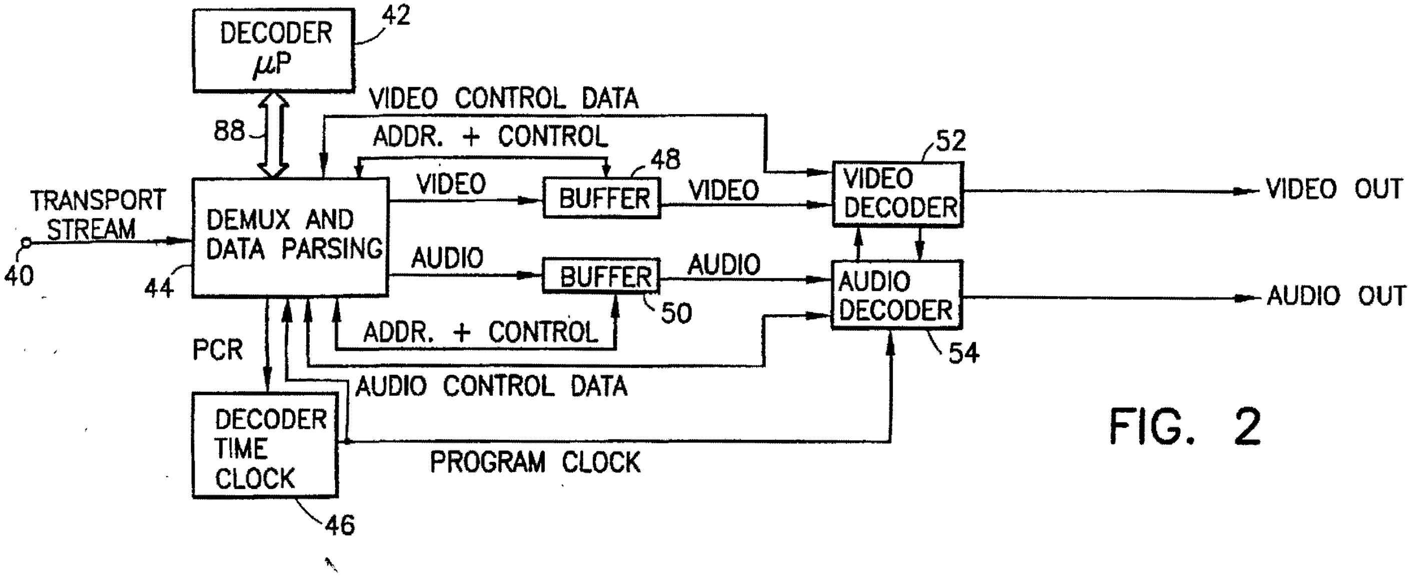

DECODER
μP
42
88
VIDEO CONTROL DATA
ADDR. + CONTROL
48
52
TRANSPORT STREAM
40
DEMUX AND DATA PARSING
44
VIDEO
BUFFER
VIDEO
VIDEO DECODER
VIDEO OUT
AUDIO
BUFFER
AUDIO
AUDIO DECODER
AUDIO OUT
50
54
PCR
ADDR. + CONTROL
AUDIO CONTROL DATA
DECODER TIME CLOCK
PROGRAM CLOCK
46

FIG. 2

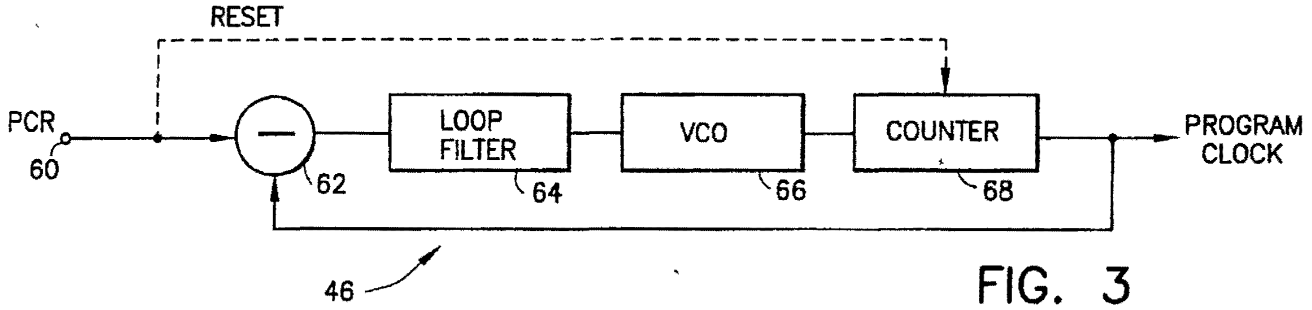
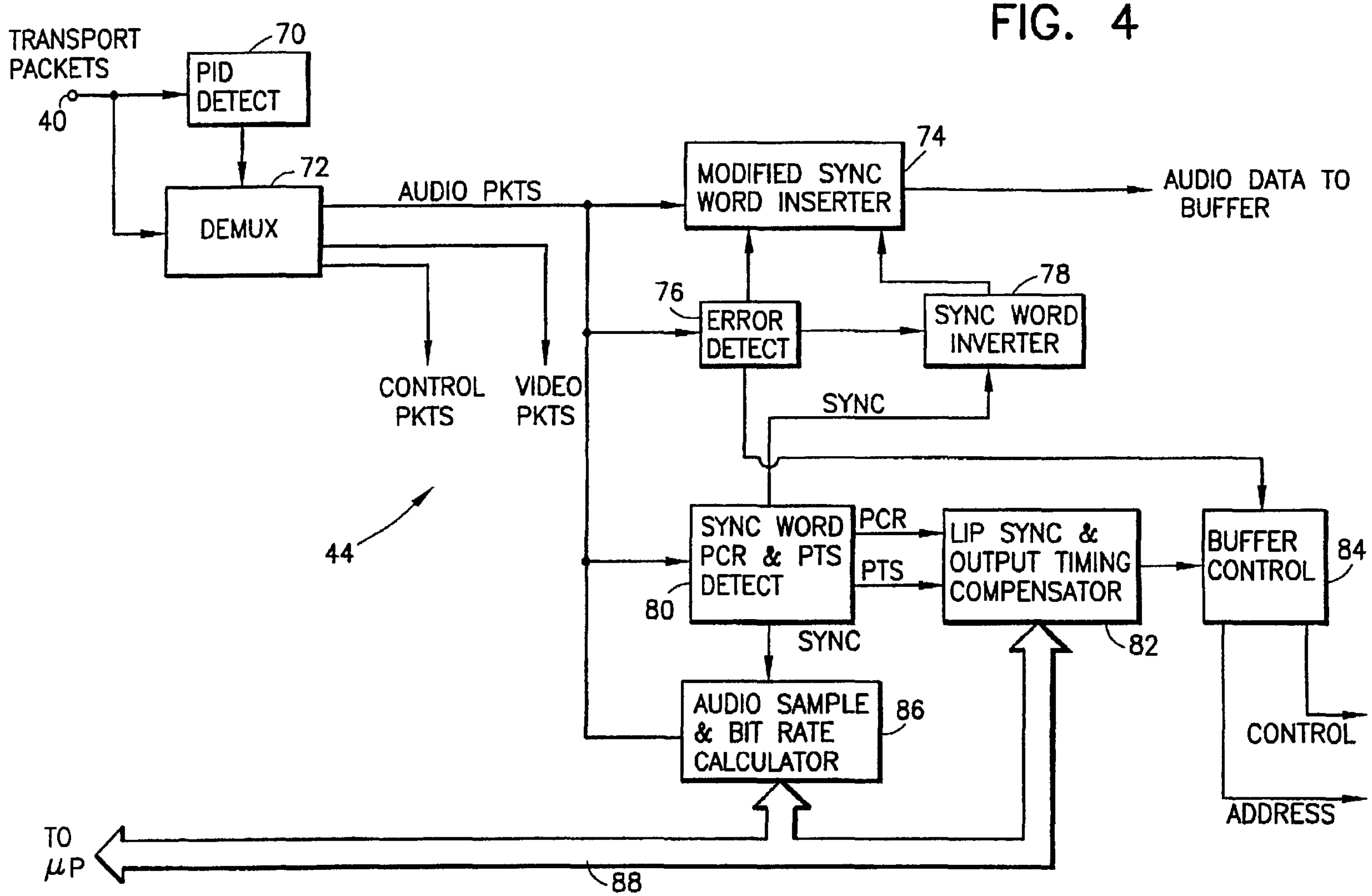

RESET
PCR
60
62
LOOP FILTER
64
VCO
66
COUNTER
68
PROGRAM CLOCK
46

FIG. 3

FIG. 4


TRANSPORT PACKETS
40
70
PID DETECT
72
DEMUX
AUDIO PKTS
CONTROL PKTS
VIDEO PKTS
74
MODIFIED SYNC WORD INSERTER
AUDIO DATA TO BUFFER
76
ERROR DETECT
78
SYNC WORD INVERTER
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

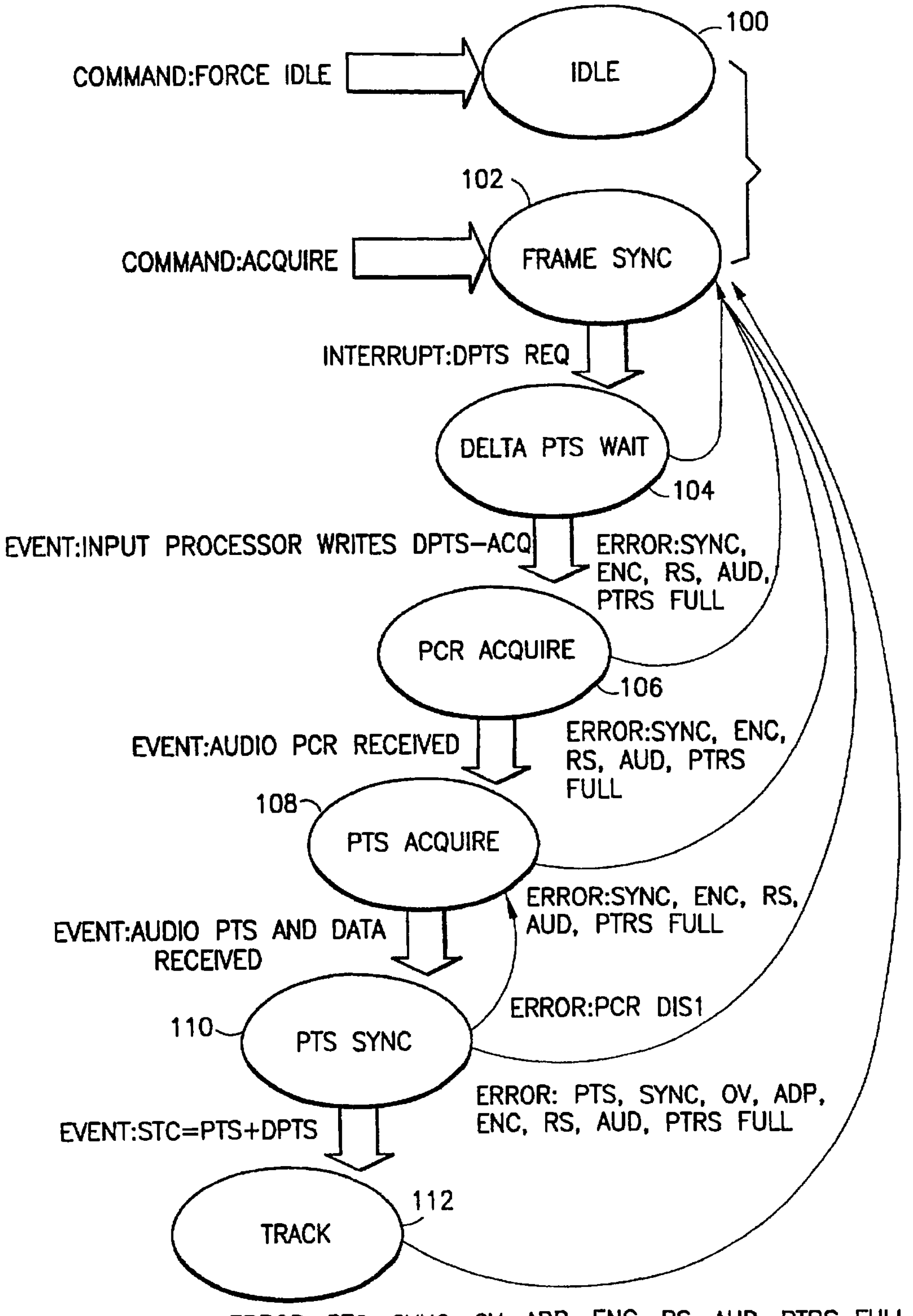

100
IDLE
COMMAND:FORCE IDLE
102
FRAME SYNC
COMMAND:ACQUIRE
INTERRUPT:DPTS REQ
DELTA PTS WAIT
104
EVENT:INPUT PROCESSOR WRITES DPTS–ACQ
ERROR:SYNC, ENC, RS, AUD, PTRS FULL
PCR ACQUIRE
106
EVENT:AUDIO PCR RECEIVED
ERROR:SYNC, ENC, RS, AUD, PTRS FULL
108
PTS ACQUIRE
ERROR:SYNC, ENC, RS, AUD, PTRS FULL
EVENT:AUDIO 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 audio 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., $\text{PCR}>\text{PTS}+\text{dPTS}_{acq}$, 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 $\text{PCR}=\text{PTS}+\text{dPTS}_{acq}$, 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 $\text{PCR}=\text{PTS}+\text{dPTS}_{acq}$, 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.
fs_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.
ptra_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.

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.)

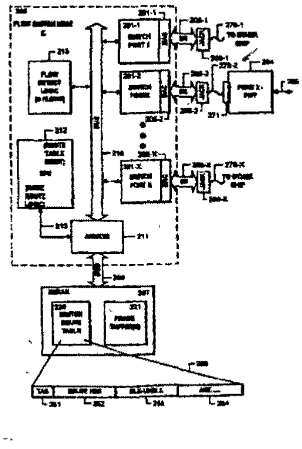
Primary Examiner—Ajit Patel

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.

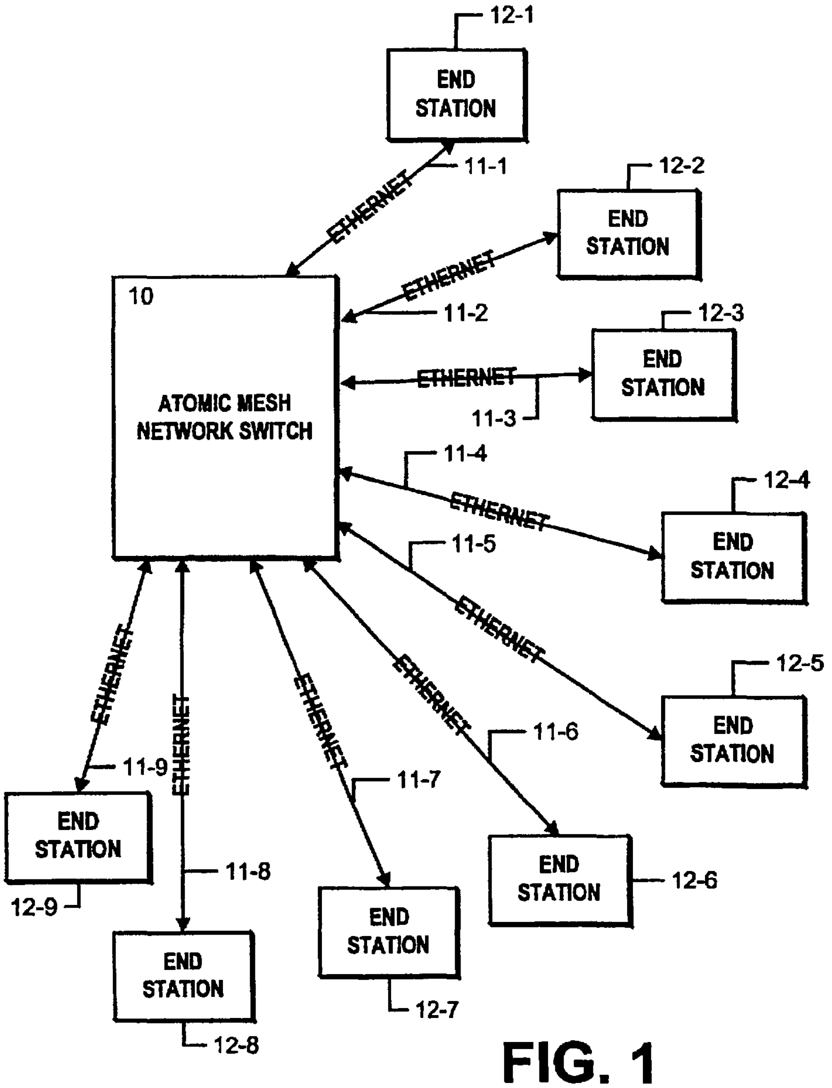

12-1
END STATION
11-1
ETHERNET
12-2
END STATION
ETHERNET
11-2
10
ATOMIC MESH NETWORK SWITCH
12-3
END STATION
ETHERNET
11-3
11-4
12-4
ETHERNET
END STATION
11-5
ETHERNET
12-5
END STATION
ETHERNET
11-6
ETHERNET
11-9
ETHERNET
END STATION
12-9
ETHERNET
11-7
11-8
END STATION
12-6
END STATION
END STATION
12-7
12-8

FIG. 1

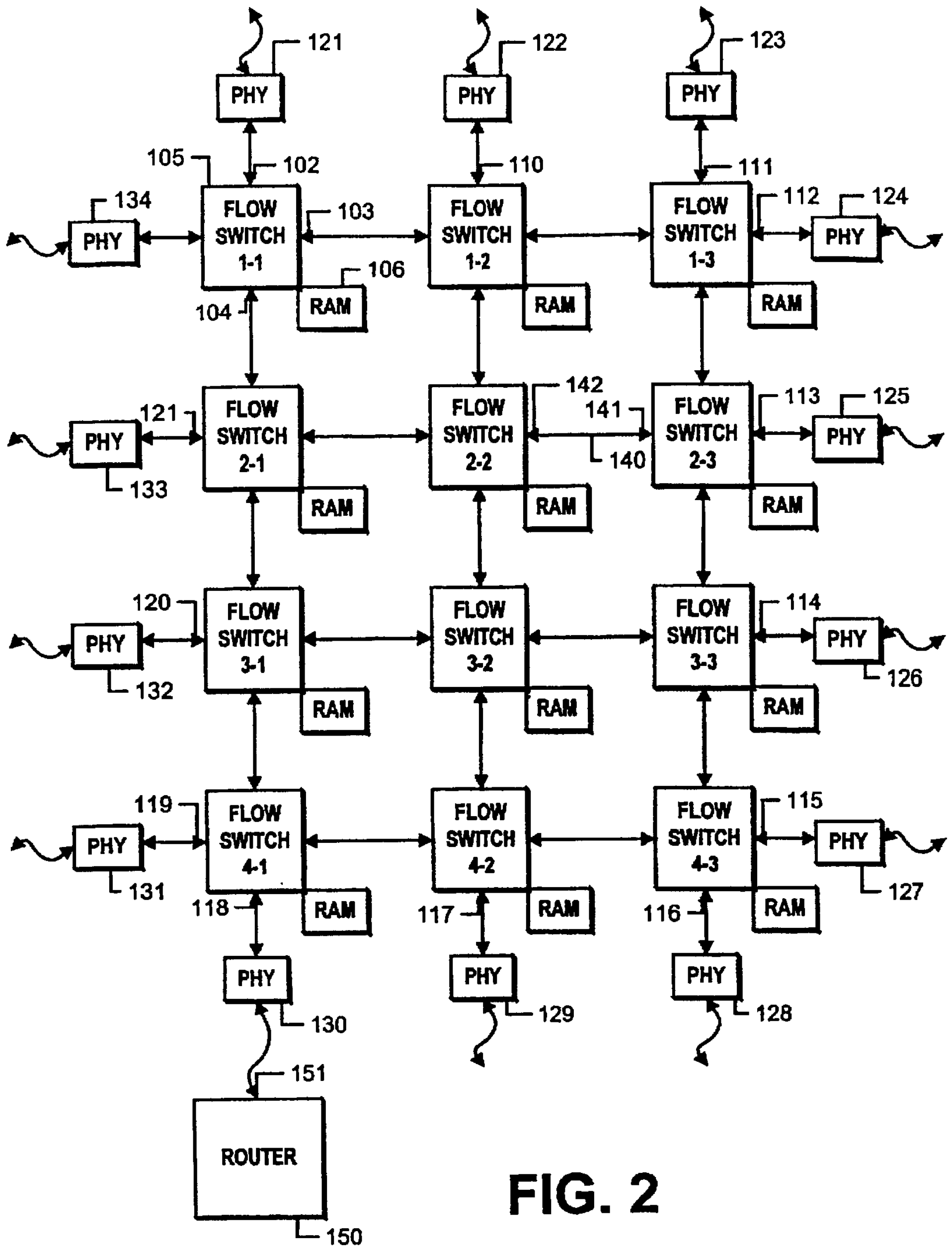

PHY
121
PHY
122
PHY
123
105
102
110
111
134
PHY
FLOW SWITCH 1-1
103
FLOW SWITCH 1-2
FLOW SWITCH 1-3
112
124
PHY
RAM
106
RAM
RAM
104
121
PHY
133
FLOW SWITCH 2-1
RAM
FLOW SWITCH 2-2
142
141
140
RAM
FLOW SWITCH 2-3
113
125
PHY
RAM
120
PHY
132
FLOW SWITCH 3-1
RAM
FLOW SWITCH 3-2
RAM
FLOW SWITCH 3-3
114
PHY
126
RAM
119
PHY
131
FLOW SWITCH 4-1
RAM
FLOW SWITCH 4-2
RAM
FLOW SWITCH 4-3
115
PHY
127
RAM
118
117
116
PHY
130
PHY
129
PHY
128
151
ROUTER
150

FIG. 2

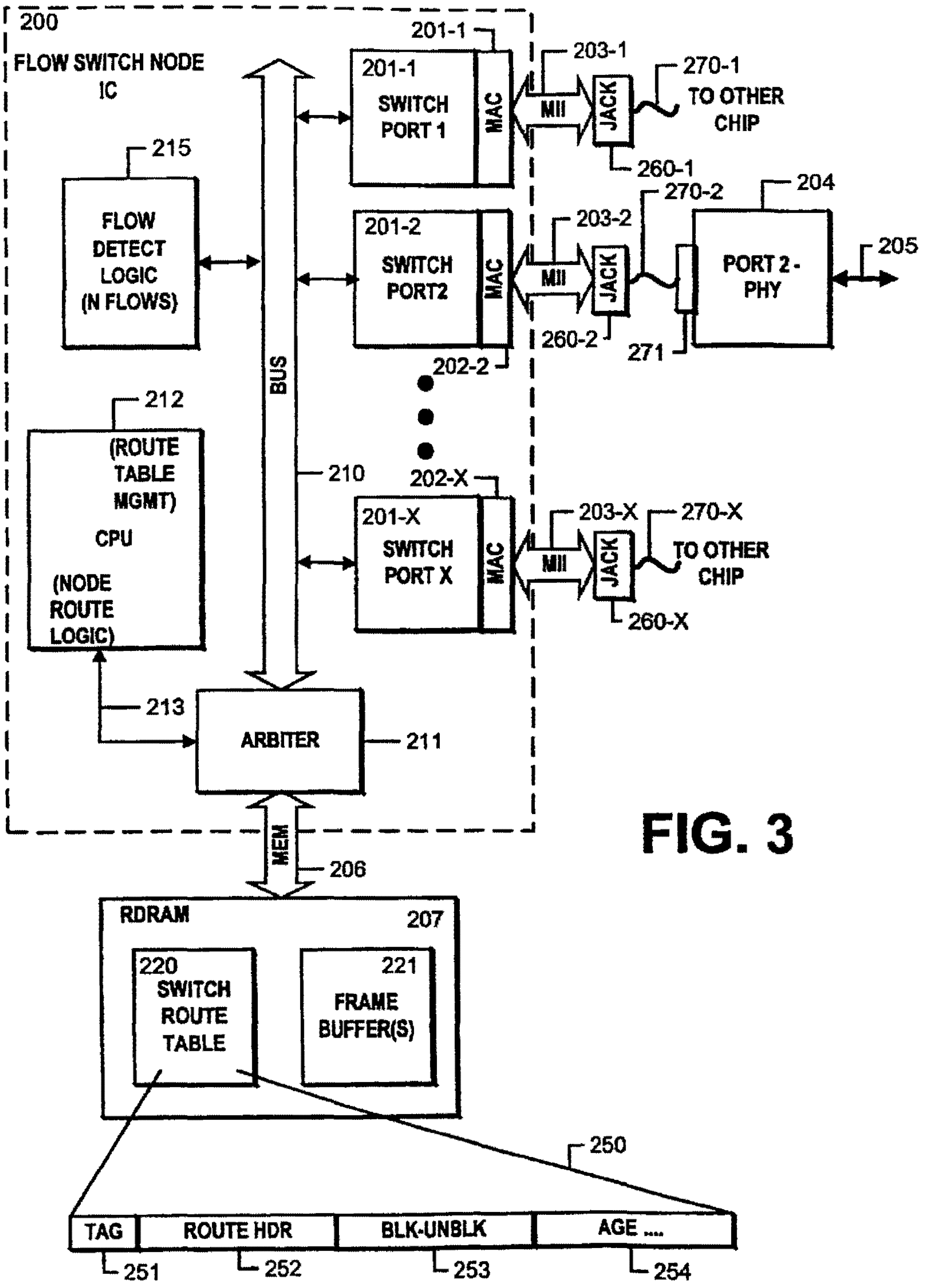
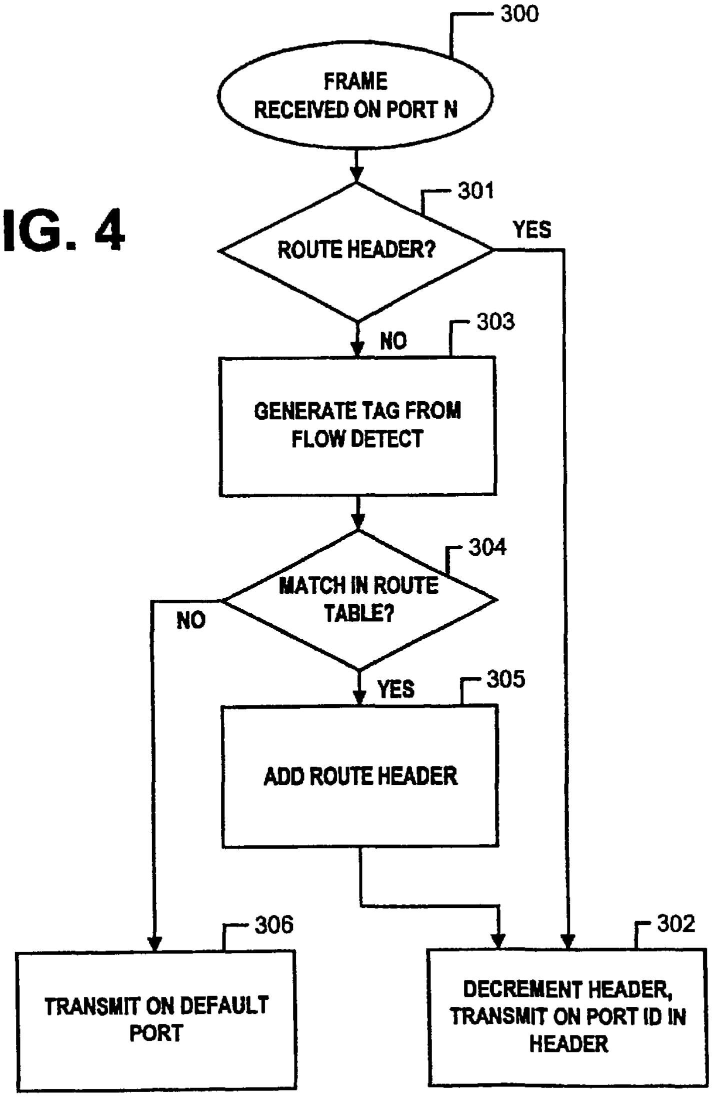

200
FLOW SWITCH NODE IC
201-1
201-1
SWITCH PORT 1
MAC
203-1
MII
JACK
270-1
TO OTHER CHIP
260-1
215
FLOW DETECT LOGIC (N FLOWS)
201-2
SWITCH PORT2
MAC
203-2
MII
JACK
270-2
204
PORT 2 - PHY
205
260-2
271
202-2
BUS
212
(ROUTE TABLE MGMT)
CPU
(NODE ROUTE LOGIC)
210
202-X
201-X
SWITCH PORT X
MAC
203-X
MII
JACK
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

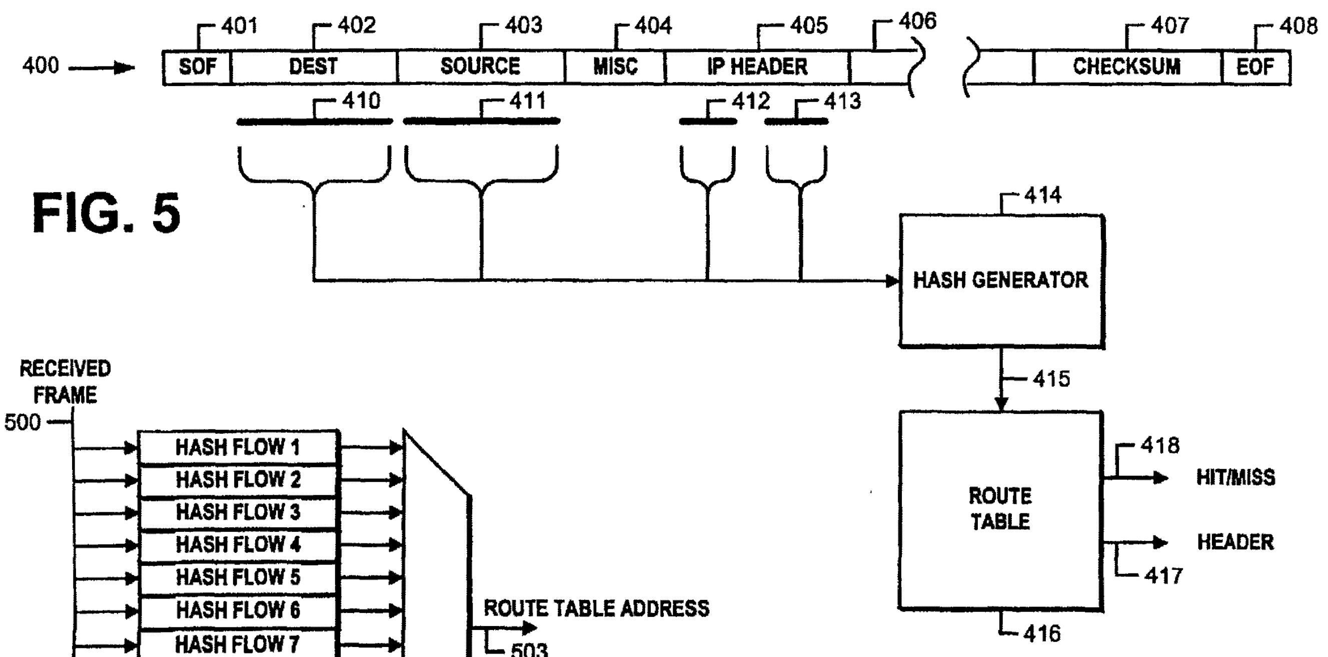

400
401
402
403
404
405
406
407
408
SOF
DEST
SOURCE
MISC
IP HEADER
CHECKSUM
EOF
410
411
412
413
414
HASH GENERATOR
415
ROUTE TABLE
418
HIT/MISS
HEADER
417
416

FIG. 5

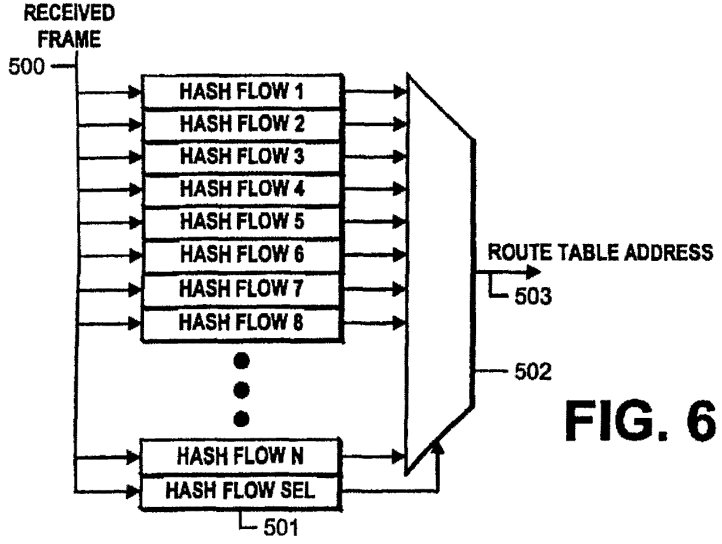

RECEIVED FRAME
500
HASH FLOW 1
HASH FLOW 2
HASH FLOW 3
HASH FLOW 4
HASH FLOW 5
HASH FLOW 6
HASH FLOW 7
HASH FLOW 8
HASH FLOW N
HASH FLOW SEL
501
502
ROUTE TABLE ADDRESS
503

FIG. 6

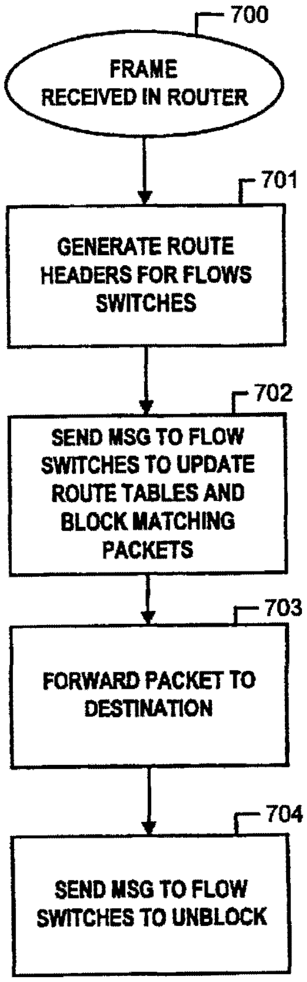

700
FRAME RECEIVED IN ROUTER
701
GENERATE ROUTE HEADERS FOR FLOWS SWITCHES
702
SEND MSG TO FLOW SWITCHES TO UPDATE ROUTE TABLES AND BLOCK MATCHING PACKETS
703
FORWARD PACKET TO DESTINATION
704
SEND MSG TO FLOW SWITCHES TO UNBLOCK

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 $(½)^{bits}$, 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 $(½)^{32}$, which is equal to 2.32×10^{-10}. The cumulative probability 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 $(1)*(1½^{32})*(1⅔^{32})*(1\text{-}3/2^{32})\ \ldots\ *(1-n/2^{32})$, 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 pseudorandom 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

Our Docket/Ref. No.: APPT-001

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:

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.

PTO-90C (Rev. 07-01)

Office Action Summary	Application No. 09/608,237	Applicant(s) DIETZ ET AL.	
	Examiner Moustafa M Meky	Art Unit 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: .

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

MOUSTAFA M. MEKY
PRIMARY EXAMINER

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.)

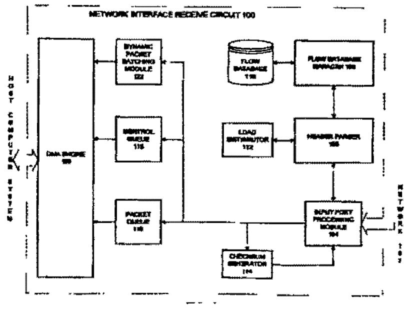
Primary Examiner—Wellington Chin
Assistant Examiner—William Schultz
(74) *Attorney, Agent, or Firm*—Park, Vaughan & Fleming LLP

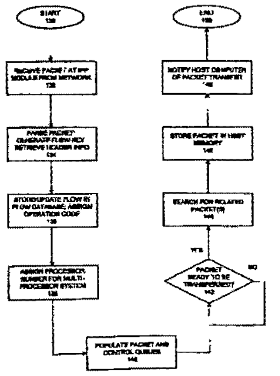
(57) **ABSTRACT**

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




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

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

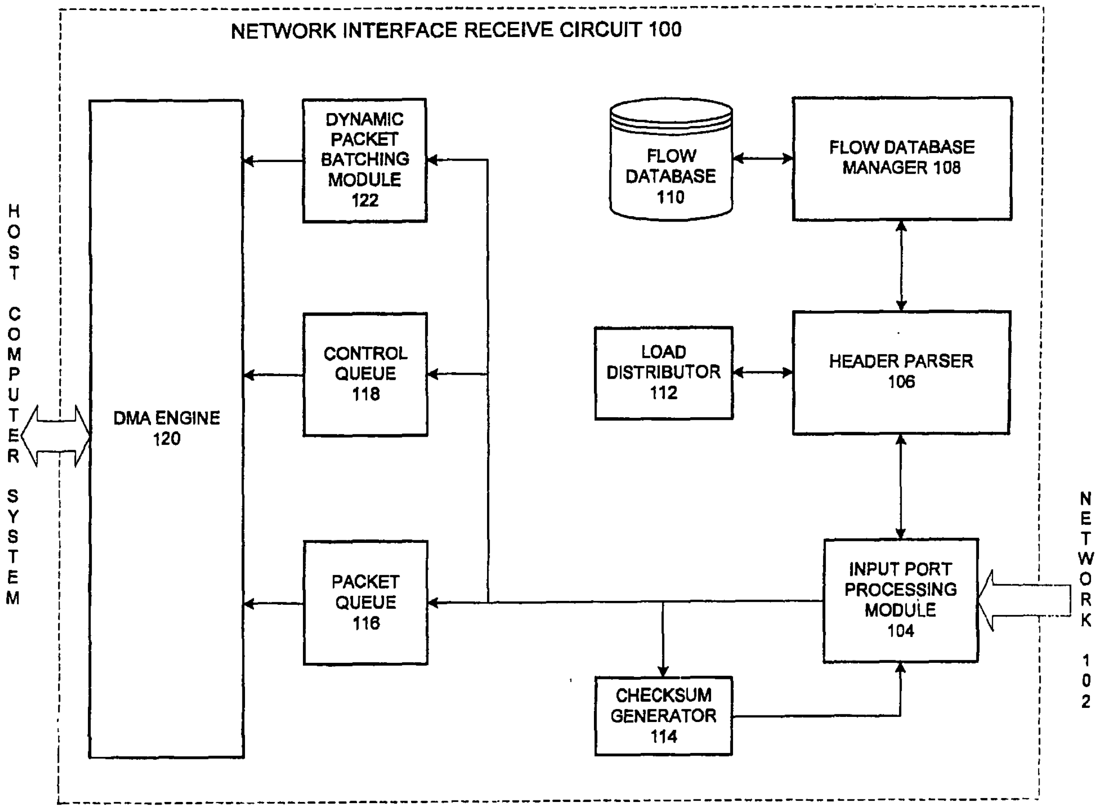

NETWORK INTERFACE RECEIVE CIRCUIT 100
HOST COMPUTER SYSTEM
DMA ENGINE 120
DYNAMIC PACKET BATCHING MODULE 122
CONTROL QUEUE 118
PACKET QUEUE 116
FLOW DATABASE 110
FLOW DATABASE MANAGER 108
LOAD DISTRIBUTOR 112
HEADER PARSER 106
INPUT PORT PROCESSING MODULE 104
CHECKSUM GENERATOR 114
NETWORK 102

FIG. 1A

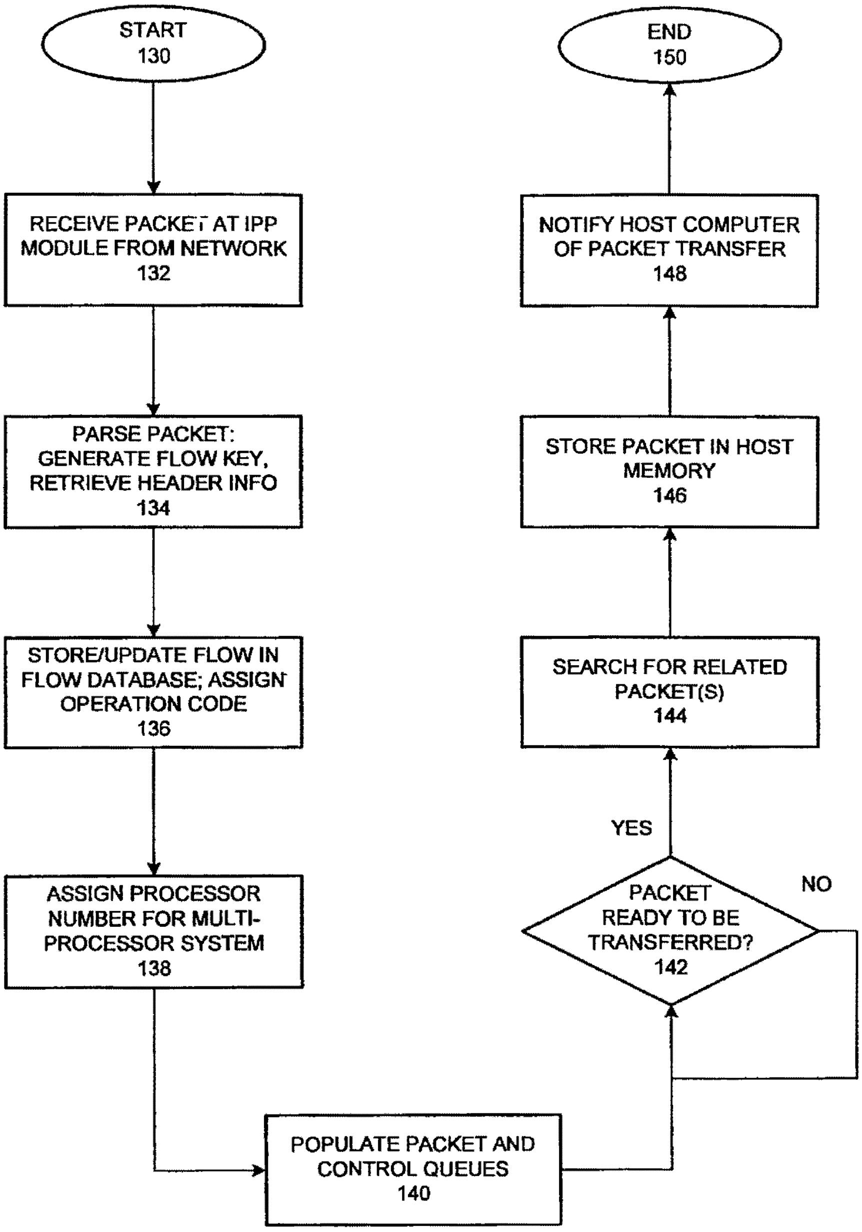

START
130
RECEIVE PACKET AT IPP MODULE FROM NETWORK
132
PARSE PACKET: GENERATE FLOW KEY, RETRIEVE HEADER INFO
134
STORE/UPDATE FLOW IN FLOW DATABASE; ASSIGN OPERATION CODE
136
ASSIGN PROCESSOR NUMBER FOR MULTI-PROCESSOR SYSTEM
138
POPULATE PACKET AND CONTROL QUEUES
140
PACKET READY TO BE TRANSFERRED?
142
NO
YES
SEARCH FOR RELATED PACKET(S)
144
STORE PACKET IN HOST MEMORY
146
NOTIFY HOST COMPUTER OF PACKET TRANSFER
148
END
150

FIG. 1B

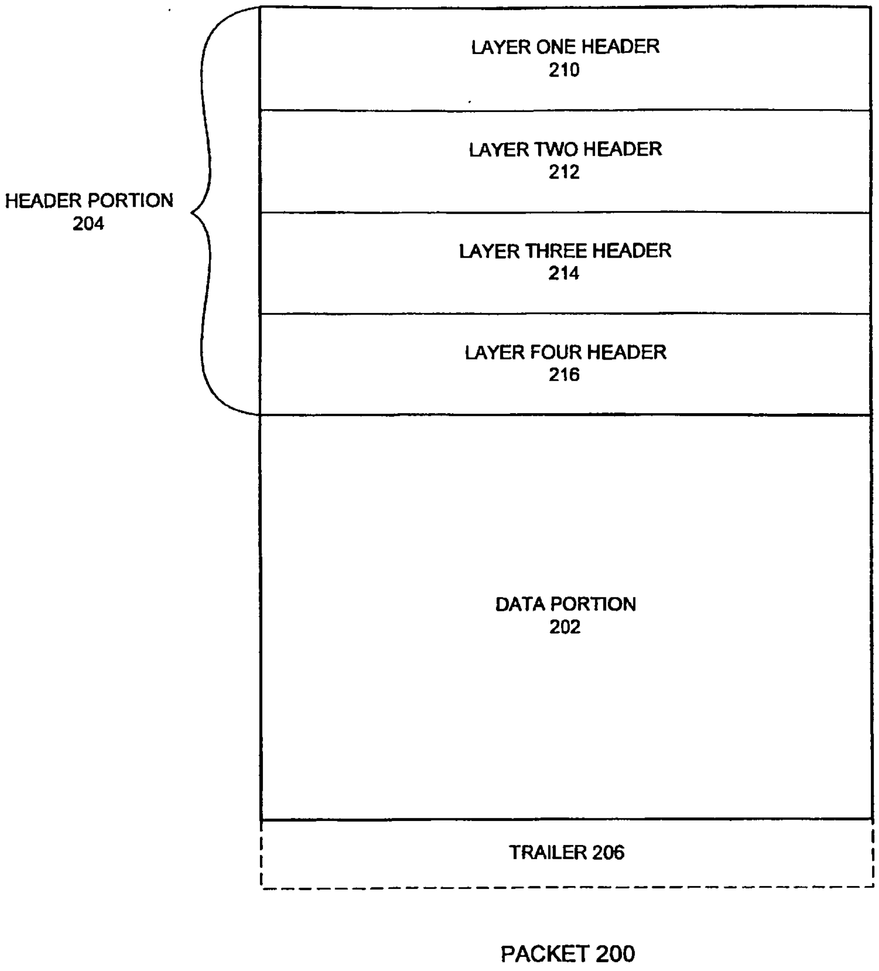

LAYER ONE HEADER
210
LAYER TWO HEADER
212
HEADER PORTION
204
LAYER THREE HEADER
214
LAYER FOUR HEADER
216
DATA PORTION
202
TRAILER 206
PACKET 200

FIG. 2

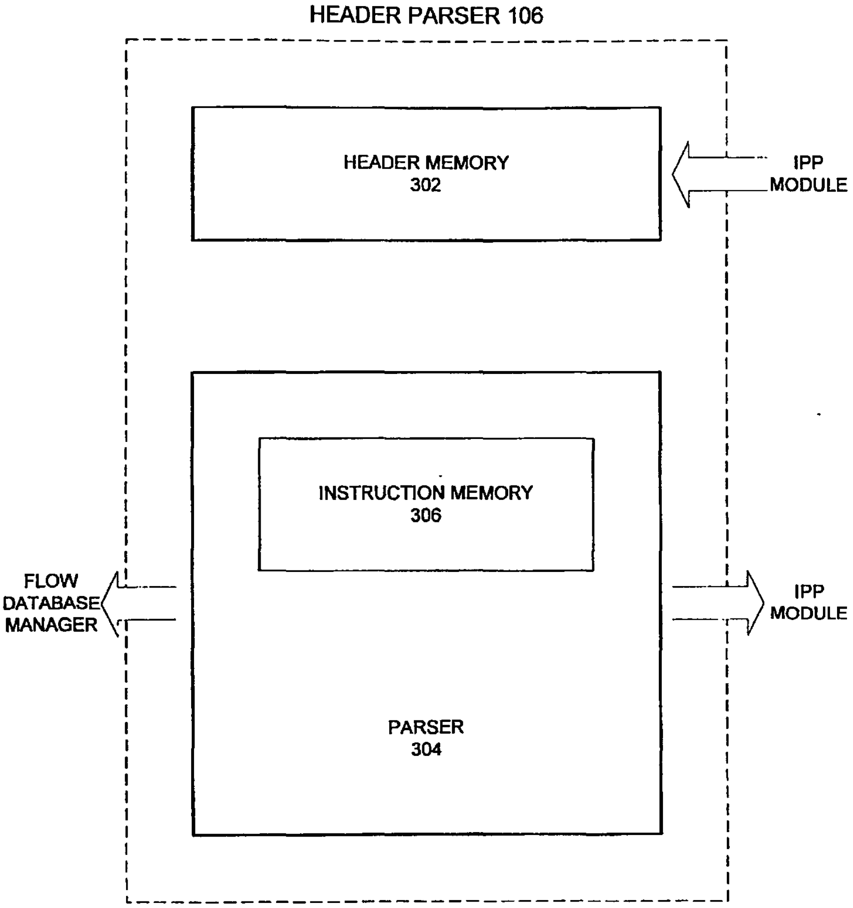

HEADER PARSER 106
HEADER MEMORY
302
IPP
MODULE
INSTRUCTION MEMORY
306
FLOW
DATABASE
MANAGER
IPP
MODULE
PARSER
304

FIG. 3

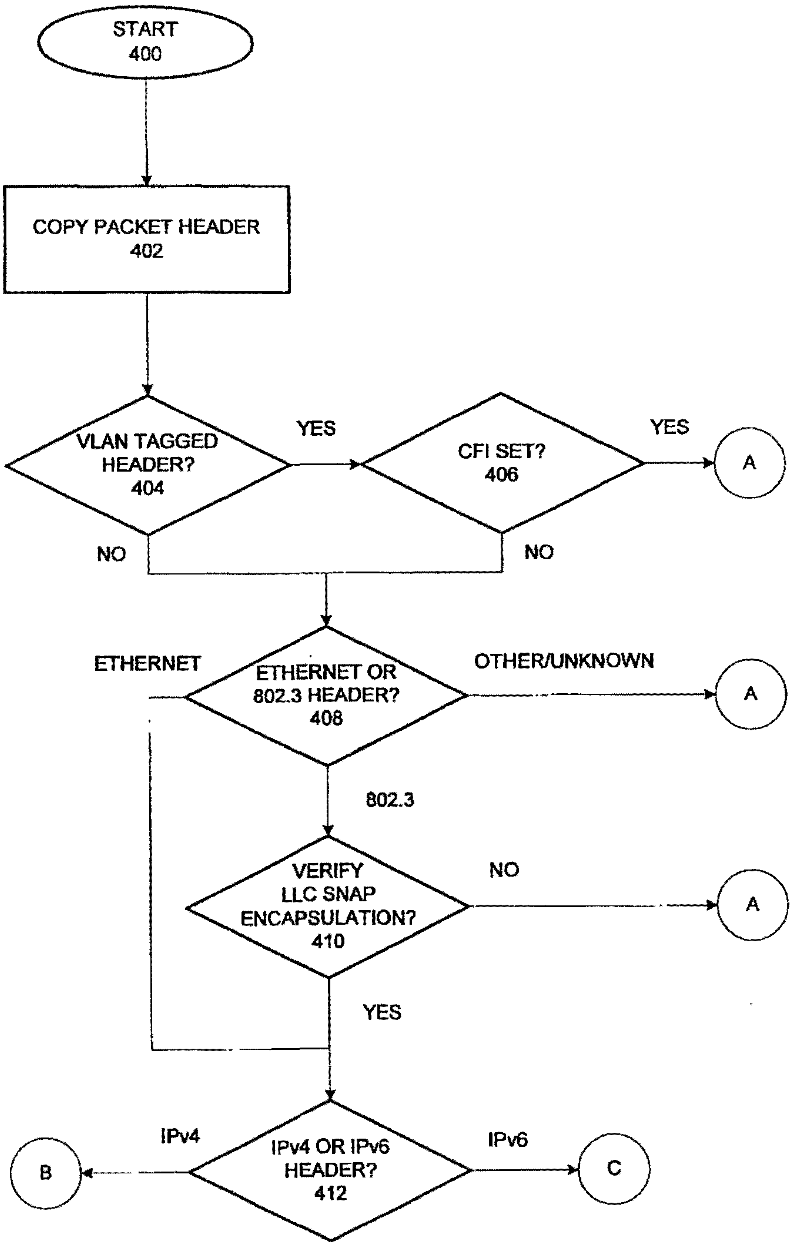

START
400
COPY PACKET HEADER
402
VLAN TAGGED
HEADER?
404
YES
CFI SET?
406
YES
A
NO
NO
ETHERNET
ETHERNET OR
802.3 HEADER?
408
OTHER/UNKNOWN
A
802.3
VERIFY
LLC SNAP
ENCAPSULATION?
410
NO
A
YES
IPv4
IPv4 OR IPv6
HEADER?
412
IPv6
B
C

FIG. 4A

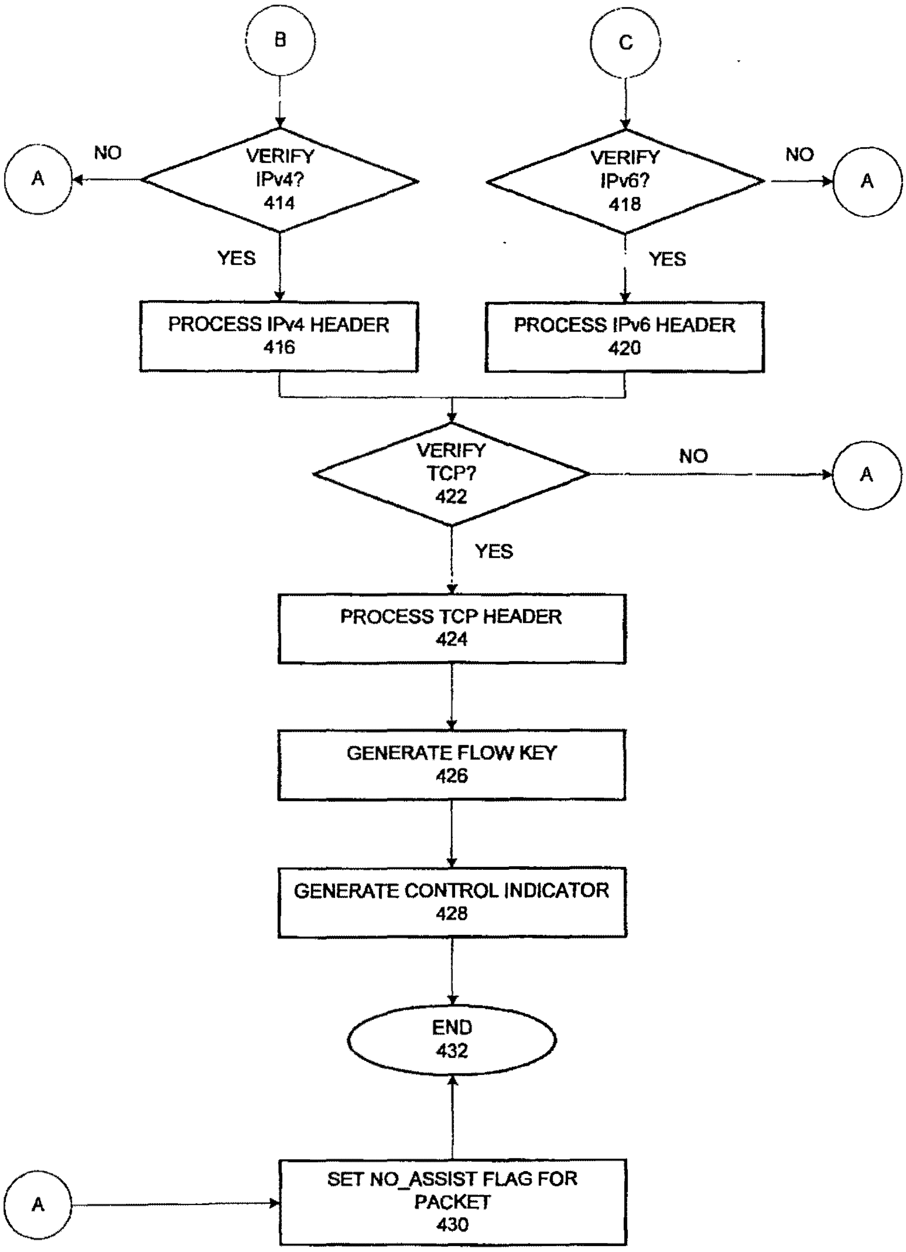

B
C
VERIFY IPv4? 414
NO
A
VERIFY IPv6? 418
NO
A
YES
YES
PROCESS IPv4 HEADER 416
PROCESS IPv6 HEADER 420
VERIFY TCP? 422
NO
A
YES
PROCESS TCP HEADER 424
GENERATE FLOW KEY 426
GENERATE CONTROL INDICATOR 428
END 432
A
SET NO_ASSIST FLAG FOR PACKET 430

FIG. 4B

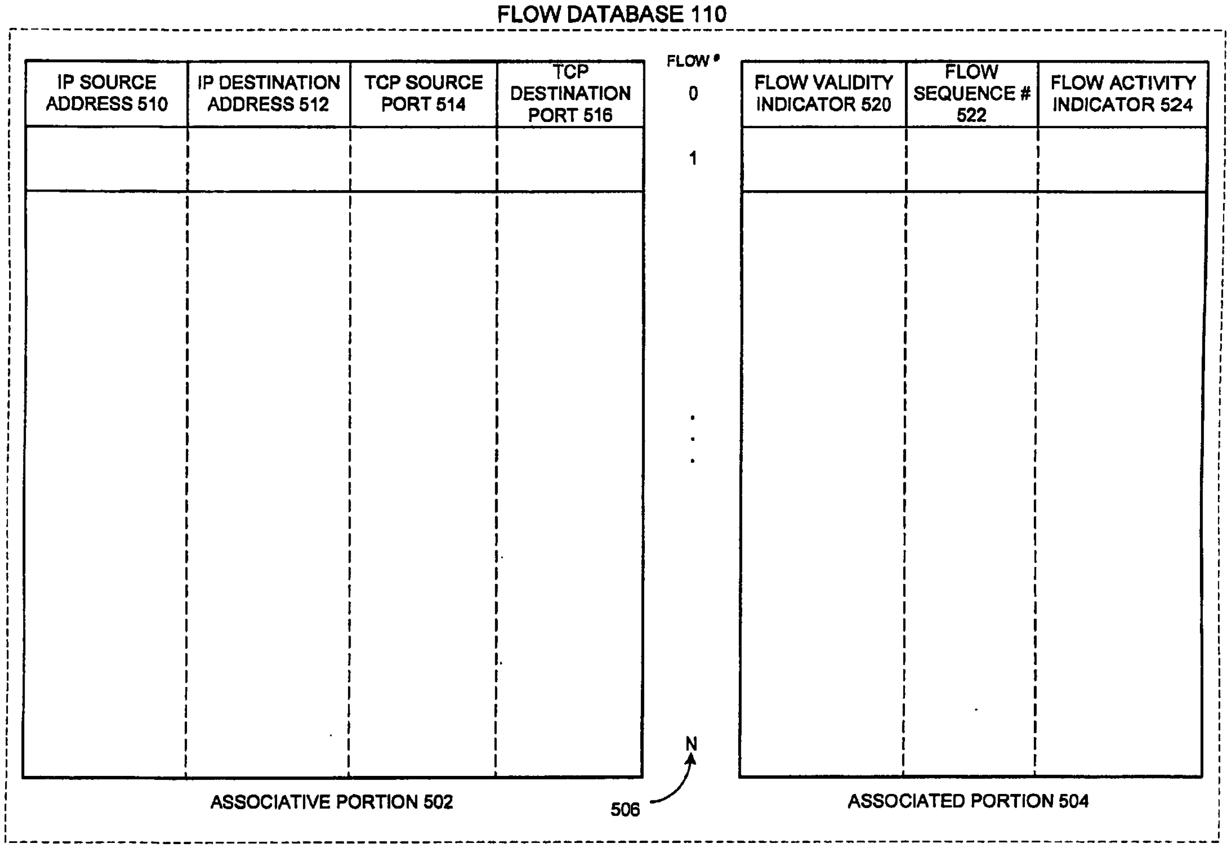

FLOW DATABASE 110
IP SOURCE ADDRESS 510
IP DESTINATION ADDRESS 512
TCP SOURCE PORT 514
TCP DESTINATION PORT 516
FLOW #
0
1
N
FLOW VALIDITY INDICATOR 520
FLOW SEQUENCE # 522
FLOW ACTIVITY INDICATOR 524
ASSOCIATIVE PORTION 502
506
ASSOCIATED PORTION 504

FIG. 5

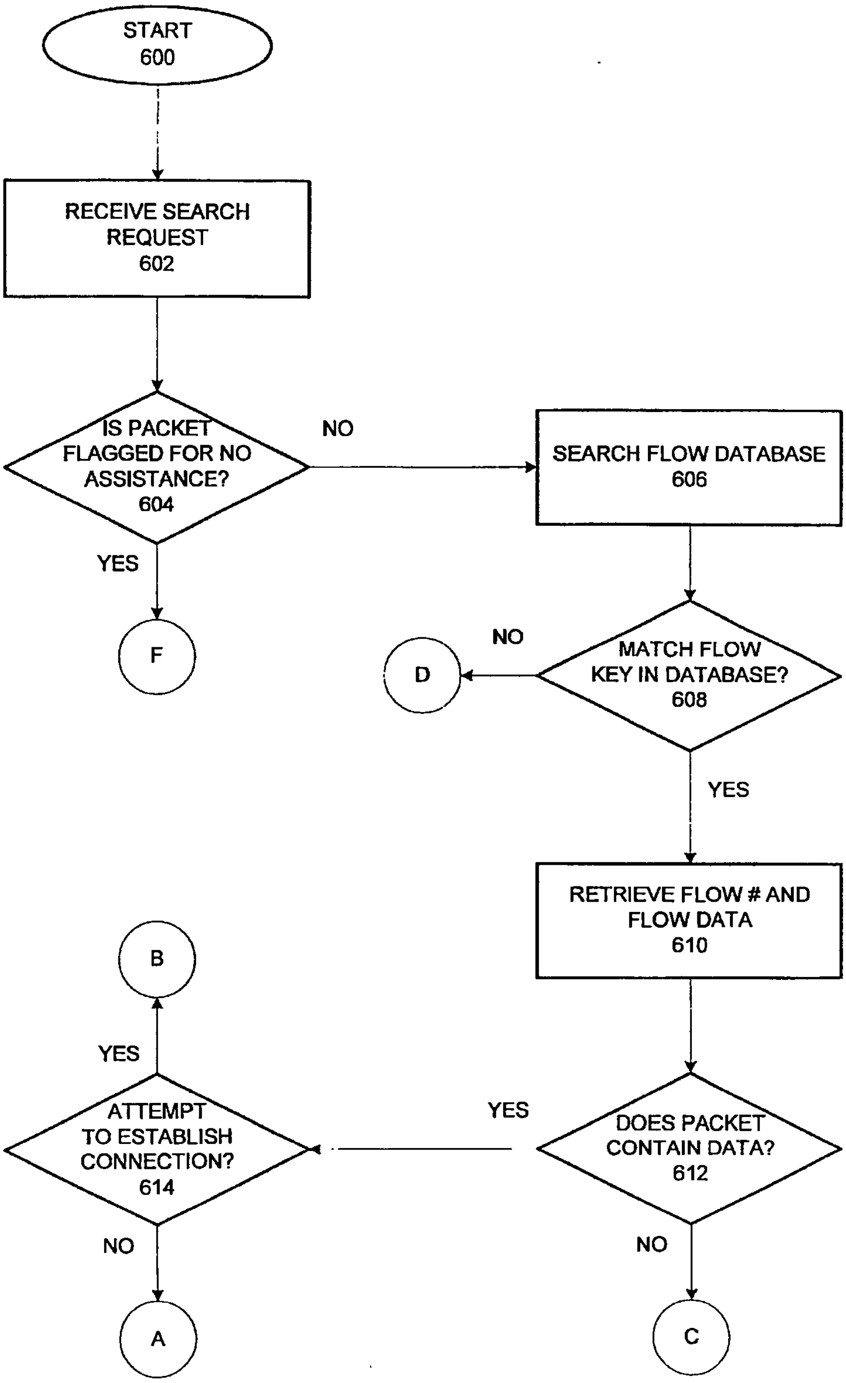

START
600
RECEIVE SEARCH REQUEST
602
IS PACKET FLAGGED FOR NO ASSISTANCE?
604
NO
YES
F
SEARCH FLOW DATABASE
606
MATCH FLOW KEY IN DATABASE?
608
NO
D
YES
RETRIEVE FLOW # AND FLOW DATA
610
B
YES
ATTEMPT TO ESTABLISH CONNECTION?
614
YES
DOES PACKET CONTAIN DATA?
612
NO
A
NO
C

FIG. 6A

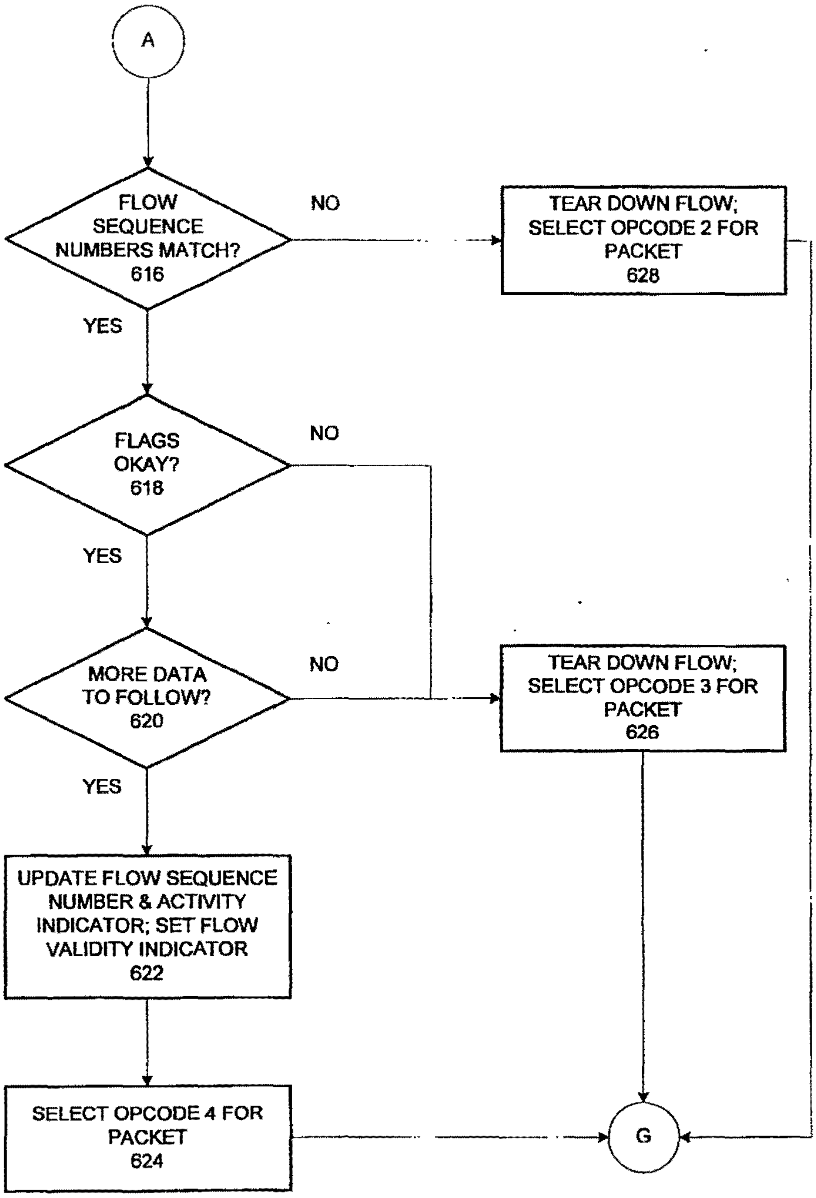

A
FLOW SEQUENCE NUMBERS MATCH? 616
NO
TEAR DOWN FLOW; SELECT OPCODE 2 FOR PACKET 628
YES
FLAGS OKAY? 618
NO
YES
MORE DATA TO FOLLOW? 620
NO
TEAR DOWN FLOW; SELECT OPCODE 3 FOR PACKET 626
YES
UPDATE FLOW SEQUENCE NUMBER & ACTIVITY INDICATOR; SET FLOW VALIDITY INDICATOR 622
SELECT OPCODE 4 FOR PACKET 624
G

FIG. 6B

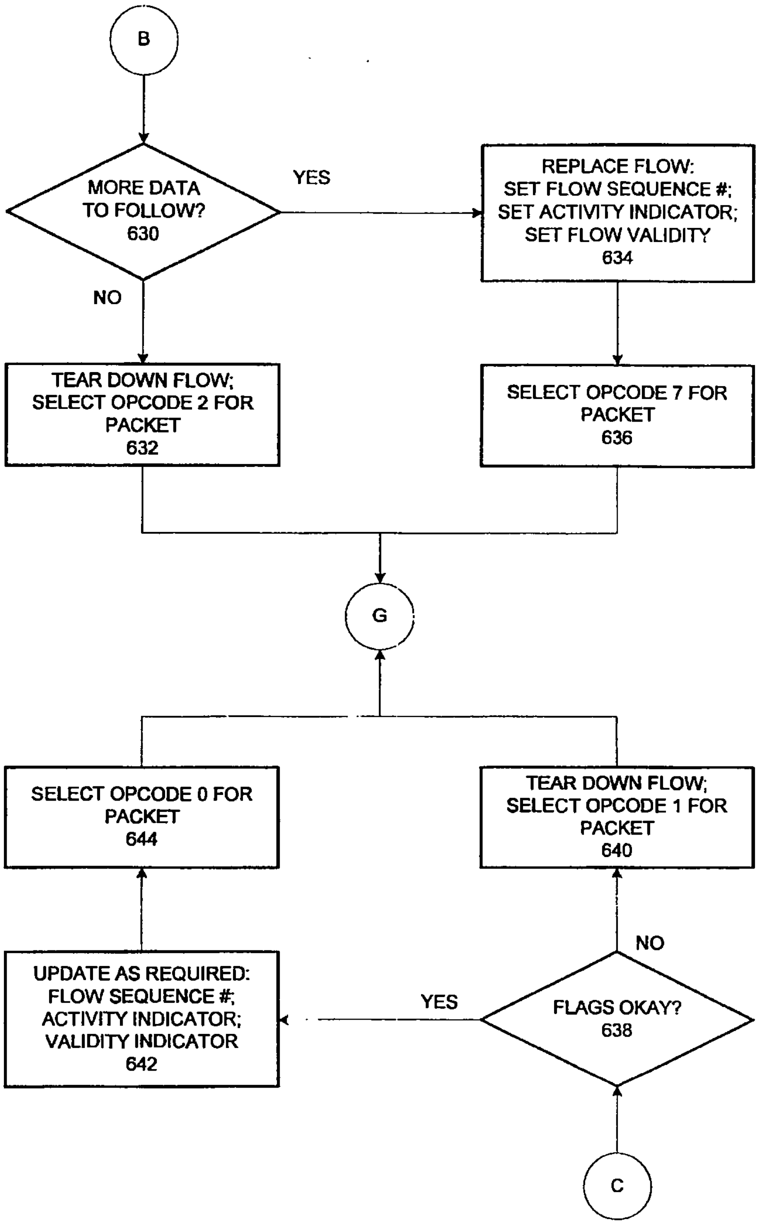

B
MORE DATA TO FOLLOW? 630
YES
NO
REPLACE FLOW: SET FLOW SEQUENCE #; SET ACTIVITY INDICATOR; SET FLOW VALIDITY 634
TEAR DOWN FLOW; SELECT OPCODE 2 FOR PACKET 632
SELECT OPCODE 7 FOR PACKET 636
G
SELECT OPCODE 0 FOR PACKET 644
TEAR DOWN FLOW; SELECT OPCODE 1 FOR PACKET 640
NO
UPDATE AS REQUIRED: FLOW SEQUENCE #; ACTIVITY INDICATOR; VALIDITY INDICATOR 642
YES
FLAGS OKAY? 638
C

FIG. 6C

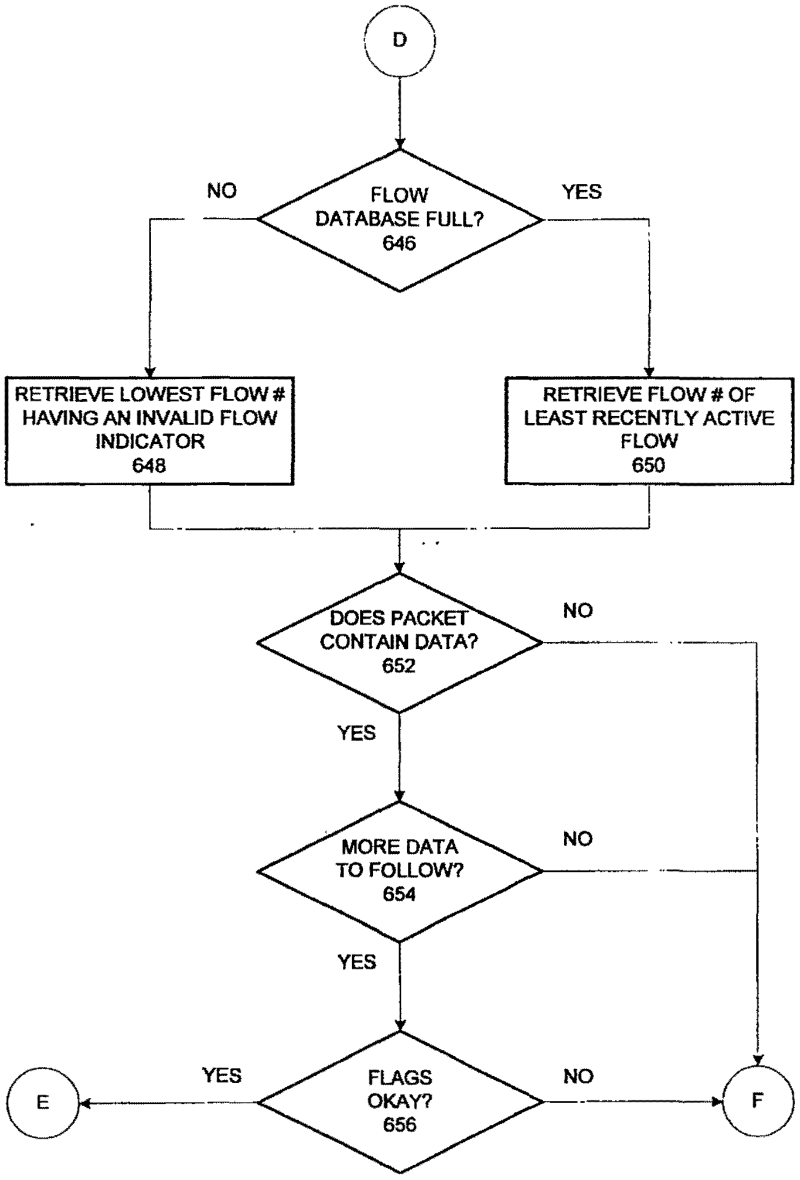

D
FLOW DATABASE FULL? 646
NO
YES
RETRIEVE LOWEST FLOW # HAVING AN INVALID FLOW INDICATOR 648
RETRIEVE FLOW # OF LEAST RECENTLY ACTIVE FLOW 650
DOES PACKET CONTAIN DATA? 652
NO
YES
MORE DATA TO FOLLOW? 654
NO
YES
FLAGS OKAY? 656
YES
NO
E
F

FIG. 6D

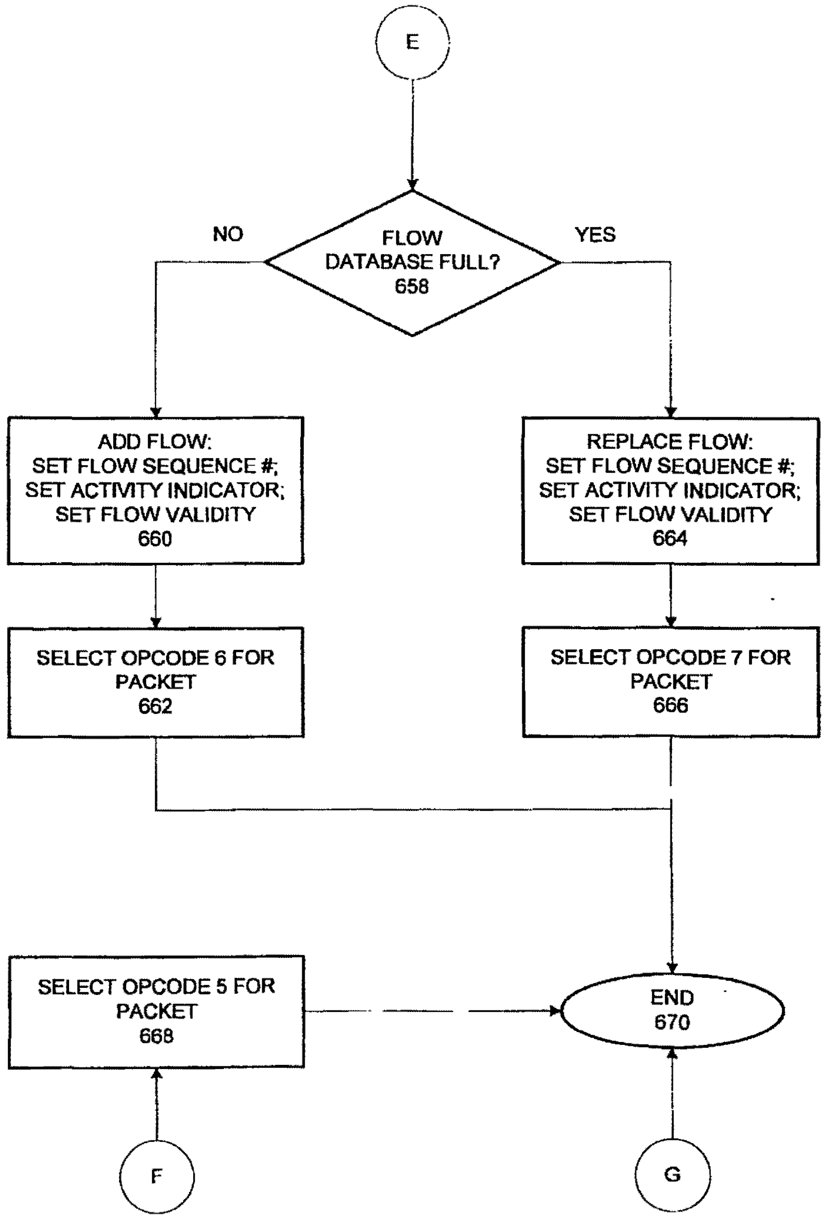

E
FLOW DATABASE FULL? 658
NO
YES
ADD FLOW: SET FLOW SEQUENCE #; SET ACTIVITY INDICATOR; SET FLOW VALIDITY 660
REPLACE FLOW: SET FLOW SEQUENCE #; SET ACTIVITY INDICATOR; SET FLOW VALIDITY 664
SELECT OPCODE 6 FOR PACKET 662
SELECT OPCODE 7 FOR PACKET 666
SELECT OPCODE 5 FOR PACKET 668
END 670
F
G

FIG. 6E

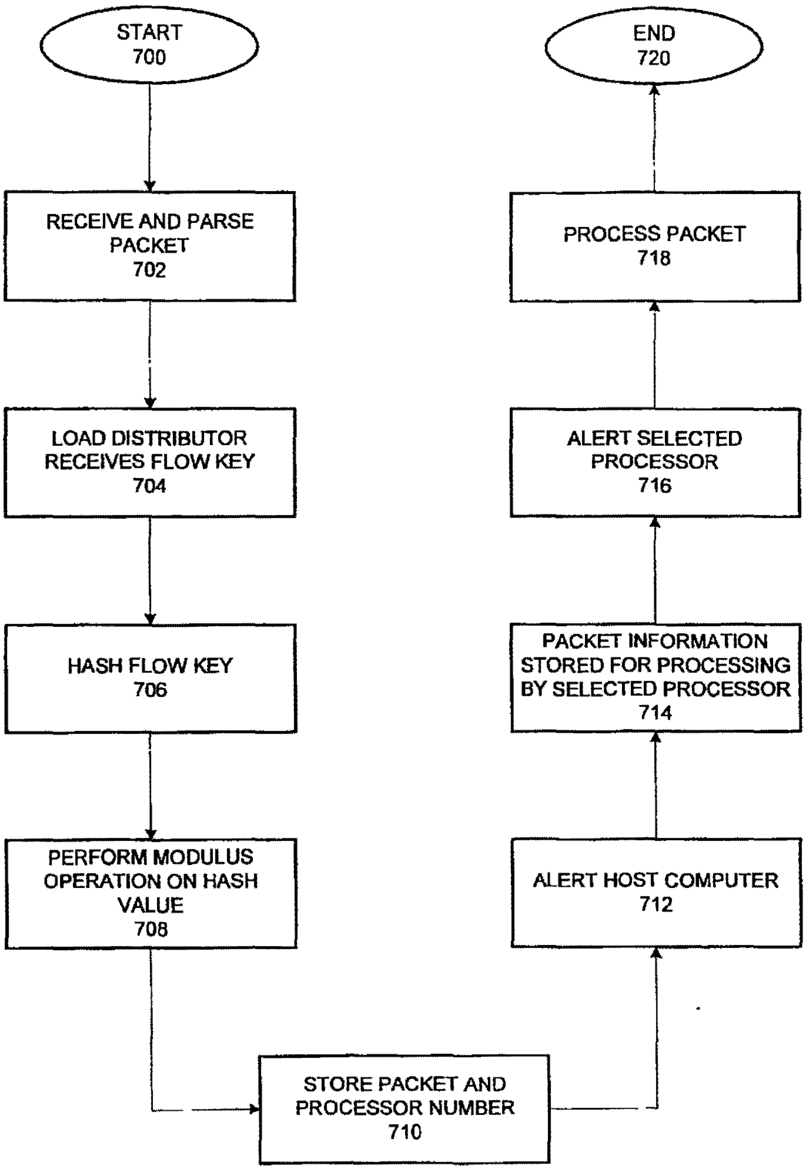

START
700
RECEIVE AND PARSE PACKET
702
LOAD DISTRIBUTOR RECEIVES FLOW KEY
704
HASH FLOW KEY
706
PERFORM MODULUS OPERATION ON HASH VALUE
708
STORE PACKET AND PROCESSOR NUMBER
710
ALERT HOST COMPUTER
712
PACKET INFORMATION STORED FOR PROCESSING BY SELECTED PROCESSOR
714
ALERT SELECTED PROCESSOR
716
PROCESS PACKET
718
END
720

FIG. 7

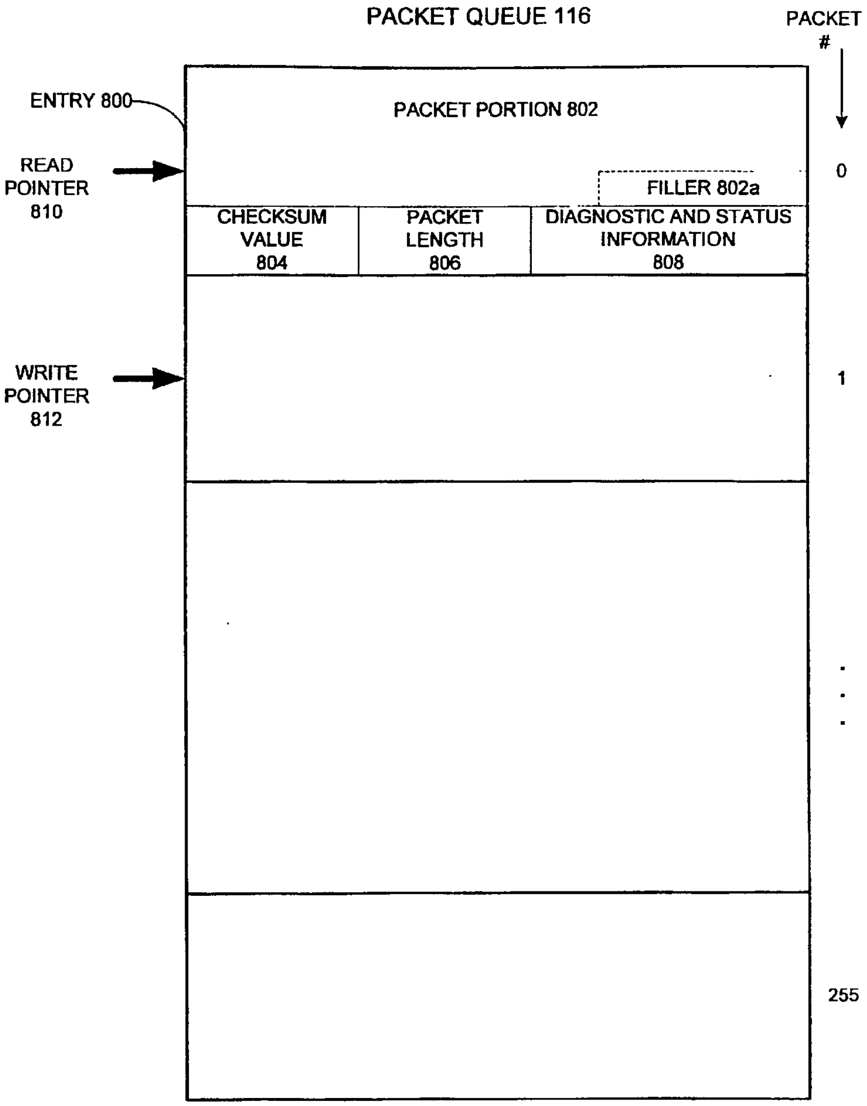

PACKET QUEUE 116
PACKET #
ENTRY 800
PACKET PORTION 802
READ POINTER 810
0
FILLER 802a
CHECKSUM VALUE 804
PACKET LENGTH 806
DIAGNOSTIC AND STATUS INFORMATION 808
WRITE POINTER 812
1
.
.
.
255

FIG. 8

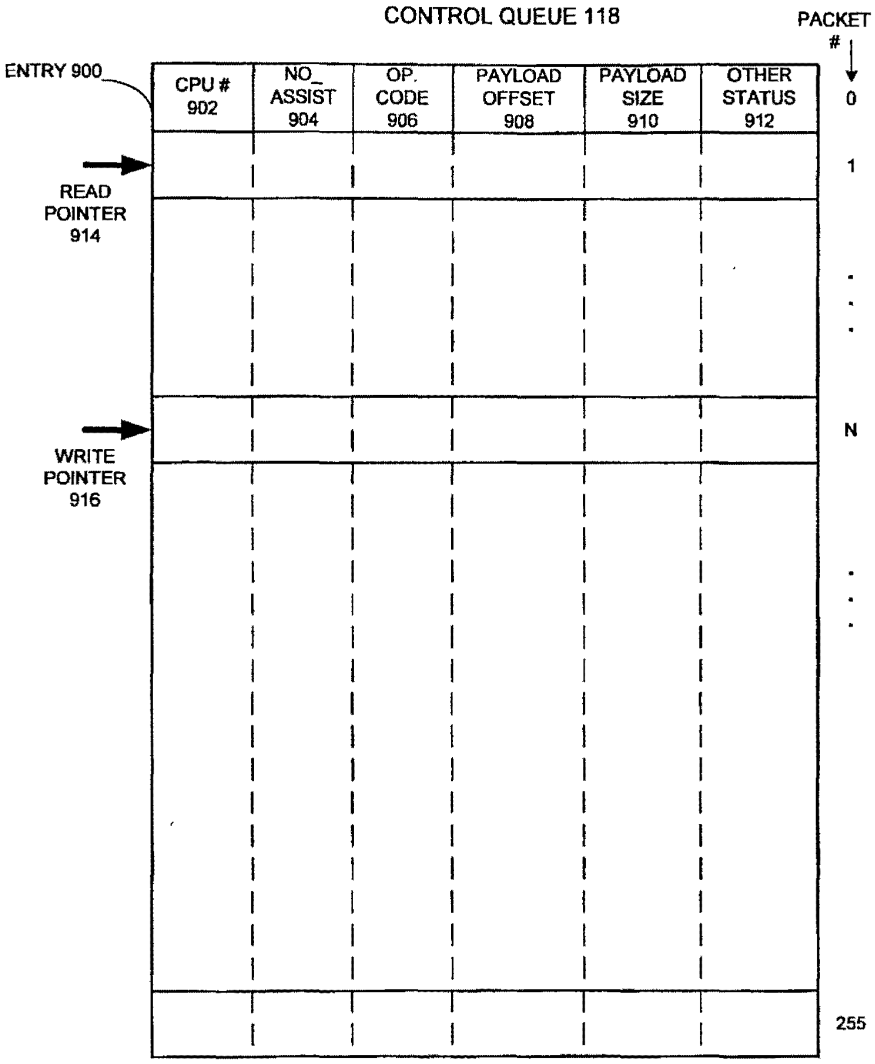

CONTROL QUEUE 118
PACKET #
ENTRY 900
CPU # 902
NO_ ASSIST 904
OP. CODE 906
PAYLOAD OFFSET 908
PAYLOAD SIZE 910
OTHER STATUS 912
0
1
READ POINTER 914
N
WRITE POINTER 916
255

FIG. 9

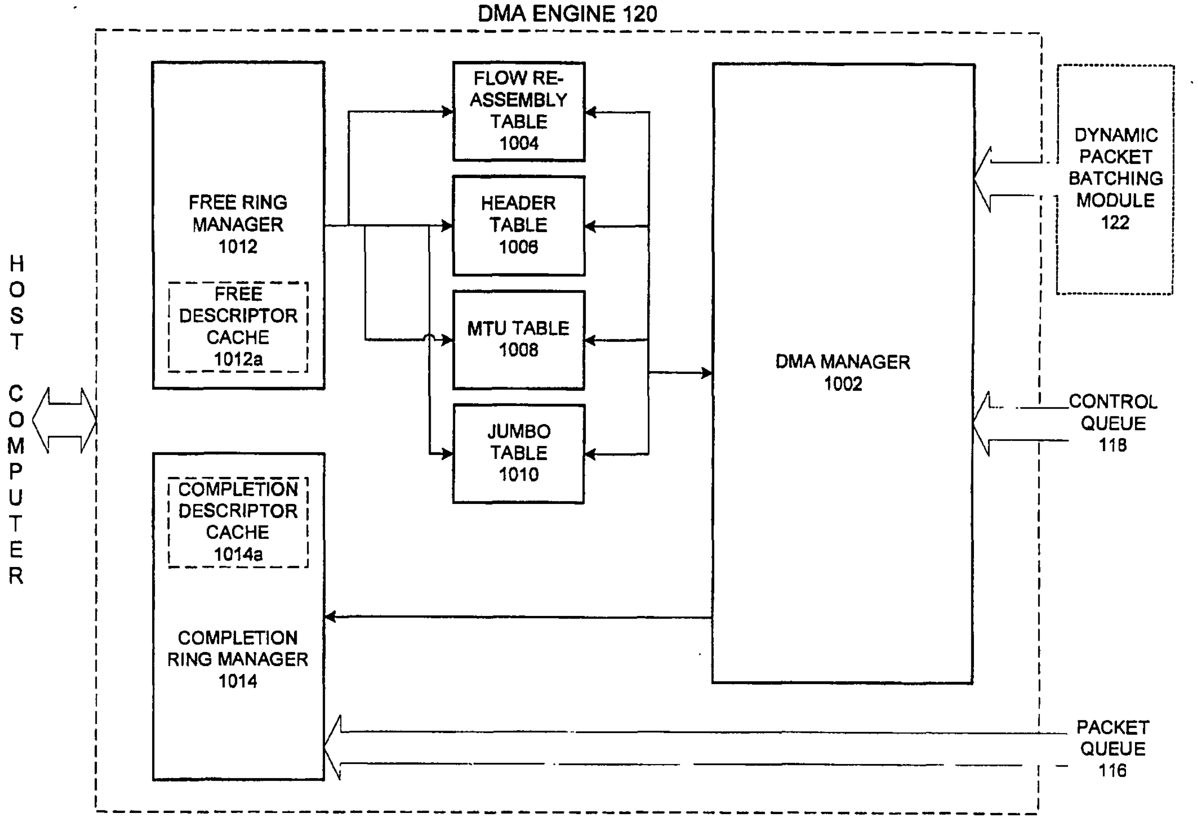

DMA ENGINE 120
HOST COMPUTER
FREE RING MANAGER 1012
FREE DESCRIPTOR CACHE 1012a
FLOW RE-ASSEMBLY TABLE 1004
HEADER TABLE 1006
MTU TABLE 1008
JUMBO TABLE 1010
DMA MANAGER 1002
DYNAMIC PACKET BATCHING MODULE 122
CONTROL QUEUE 118
COMPLETION DESCRIPTOR CACHE 1014a
COMPLETION RING MANAGER 1014
PACKET QUEUE 116

FIG. 10

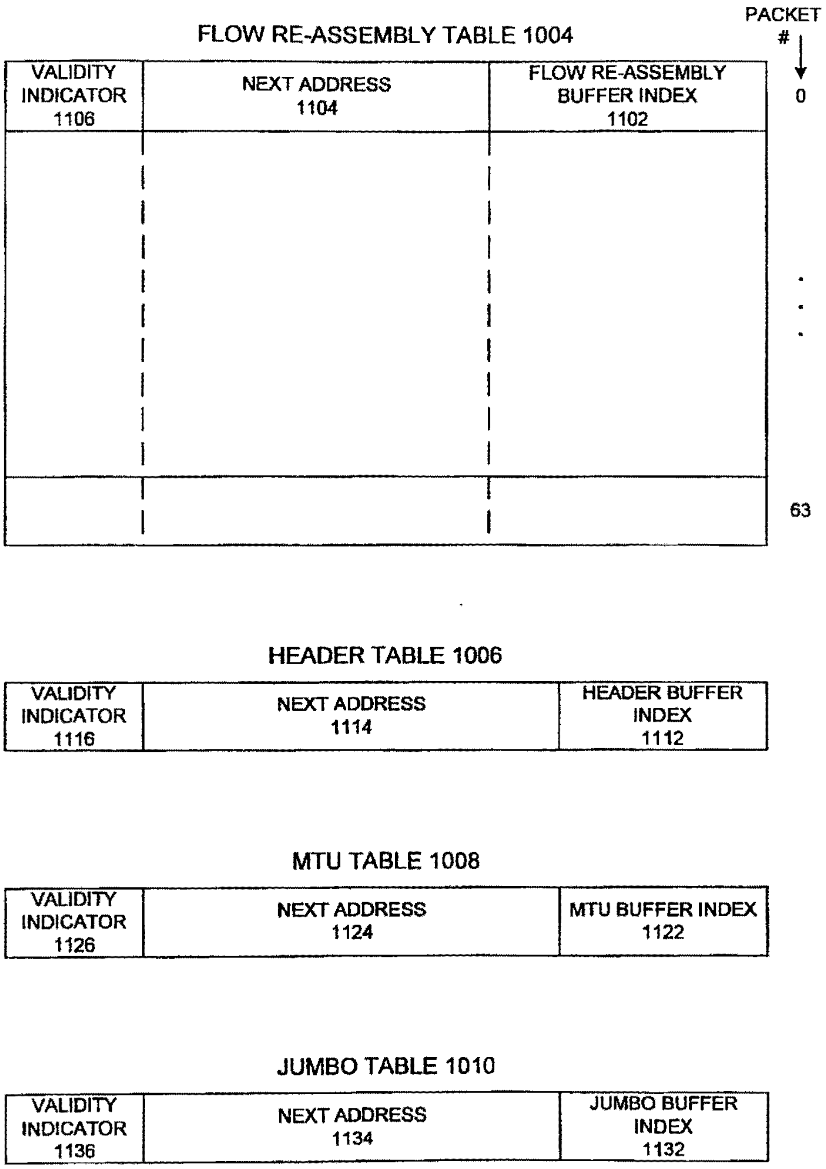

FLOW RE-ASSEMBLY TABLE 1004
PACKET #
0
VALIDITY INDICATOR 1106
NEXT ADDRESS 1104
FLOW RE-ASSEMBLY BUFFER INDEX 1102
63
HEADER TABLE 1006
VALIDITY INDICATOR 1116
NEXT ADDRESS 1114
HEADER BUFFER INDEX 1112
MTU TABLE 1008
VALIDITY INDICATOR 1126
NEXT ADDRESS 1124
MTU BUFFER INDEX 1122
JUMBO TABLE 1010
VALIDITY INDICATOR 1136
NEXT ADDRESS 1134
JUMBO BUFFER INDEX 1132

FIG. 11

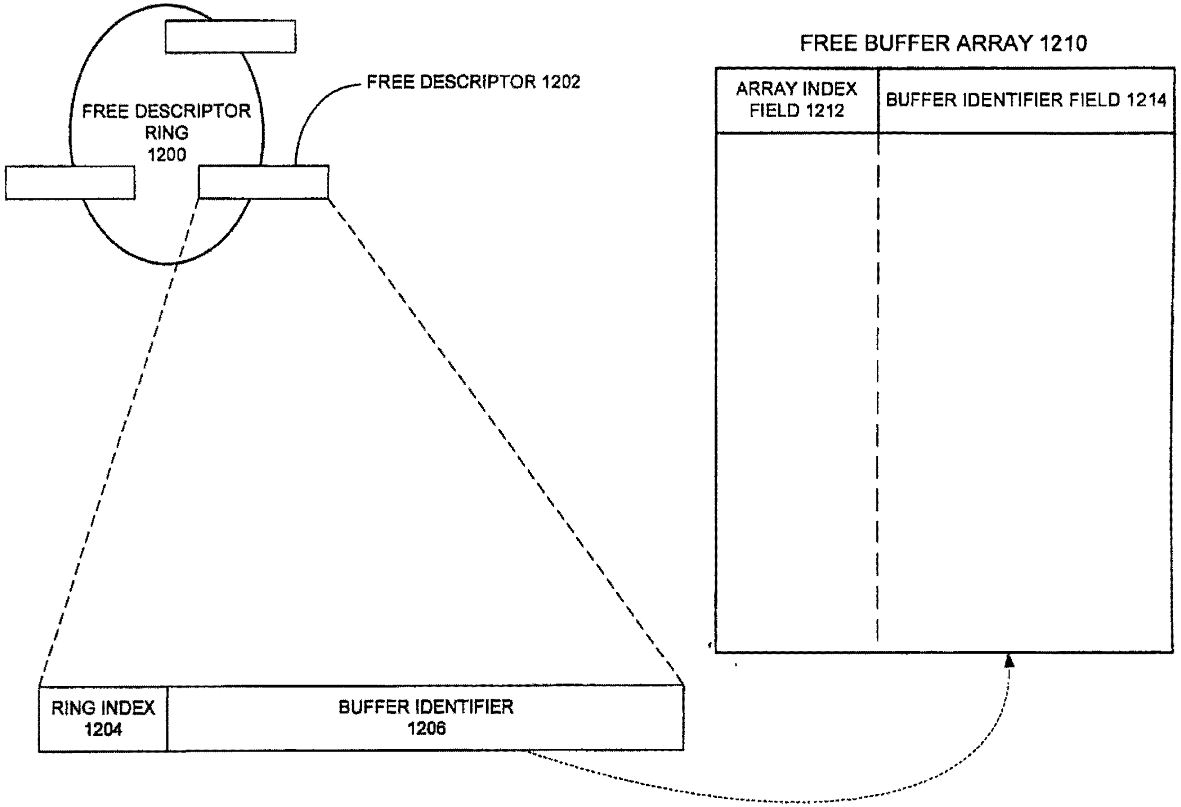

FREE DESCRIPTOR RING 1200
FREE DESCRIPTOR 1202
FREE BUFFER ARRAY 1210
ARRAY INDEX FIELD 1212
BUFFER IDENTIFIER FIELD 1214
RING INDEX 1204
BUFFER IDENTIFIER 1206

FIG. 12A

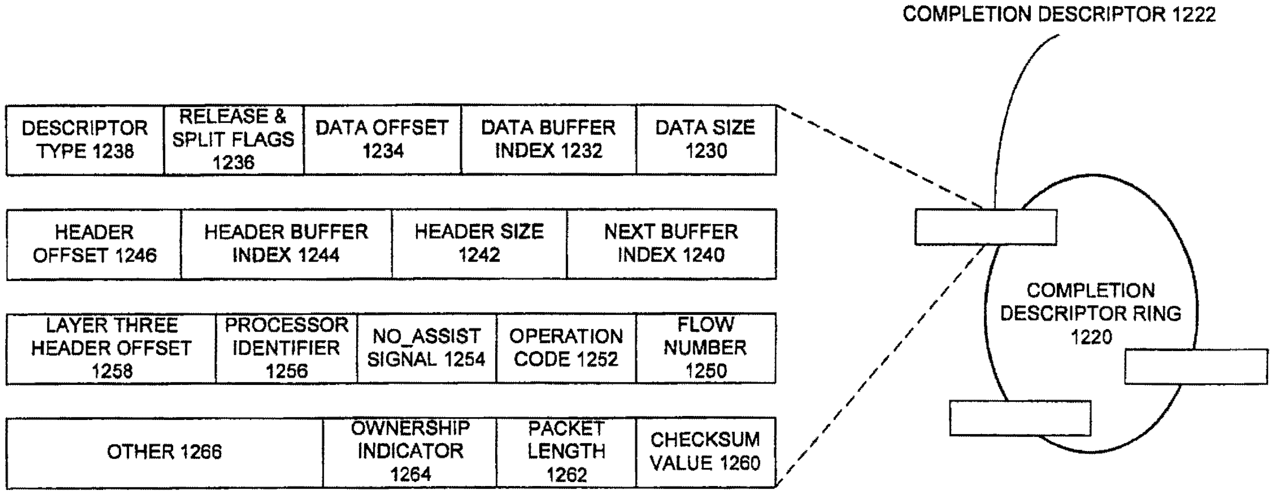

COMPLETION DESCRIPTOR 1222
DESCRIPTOR TYPE 1238
RELEASE & SPLIT FLAGS 1236
DATA OFFSET 1234
DATA BUFFER INDEX 1232
DATA SIZE 1230
HEADER OFFSET 1246
HEADER BUFFER INDEX 1244
HEADER SIZE 1242
NEXT BUFFER INDEX 1240
LAYER THREE HEADER OFFSET 1258
PROCESSOR IDENTIFIER 1256
NO_ASSIST SIGNAL 1254
OPERATION CODE 1252
FLOW NUMBER 1250
OTHER 1266
OWNERSHIP INDICATOR 1264
PACKET LENGTH 1262
CHECKSUM VALUE 1260
COMPLETION DESCRIPTOR RING 1220

FIG. 12B

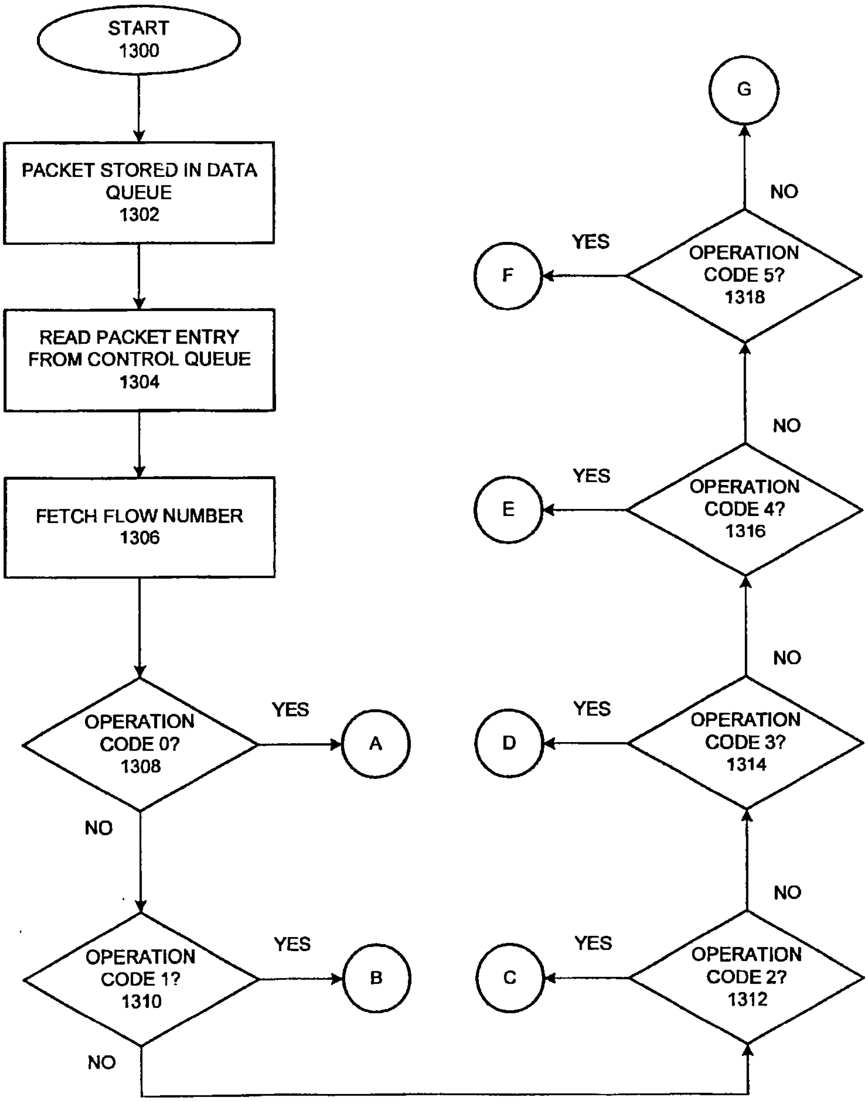

START
1300
PACKET STORED IN DATA QUEUE
1302
READ PACKET ENTRY FROM CONTROL QUEUE
1304
FETCH FLOW NUMBER
1306
OPERATION CODE 0?
1308
YES
A
NO
OPERATION CODE 1?
1310
YES
B
NO
OPERATION CODE 2?
1312
YES
C
NO
OPERATION CODE 3?
1314
YES
D
NO
OPERATION CODE 4?
1316
YES
E
NO
OPERATION CODE 5?
1318
YES
F
NO
G

FIG. 13

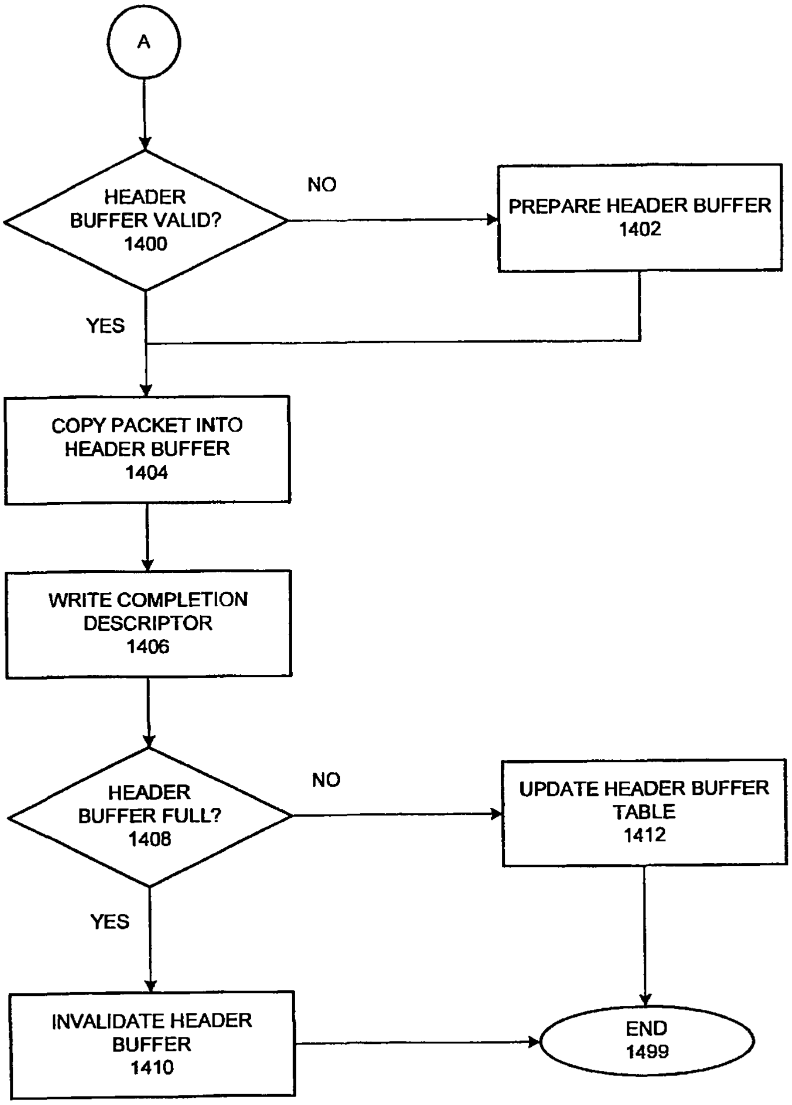

A
HEADER BUFFER VALID? 1400
NO
PREPARE HEADER BUFFER 1402
YES
COPY PACKET INTO HEADER BUFFER 1404
WRITE COMPLETION DESCRIPTOR 1406
HEADER BUFFER FULL? 1408
NO
UPDATE HEADER BUFFER TABLE 1412
YES
INVALIDATE HEADER BUFFER 1410
END 1499

FIG. 14

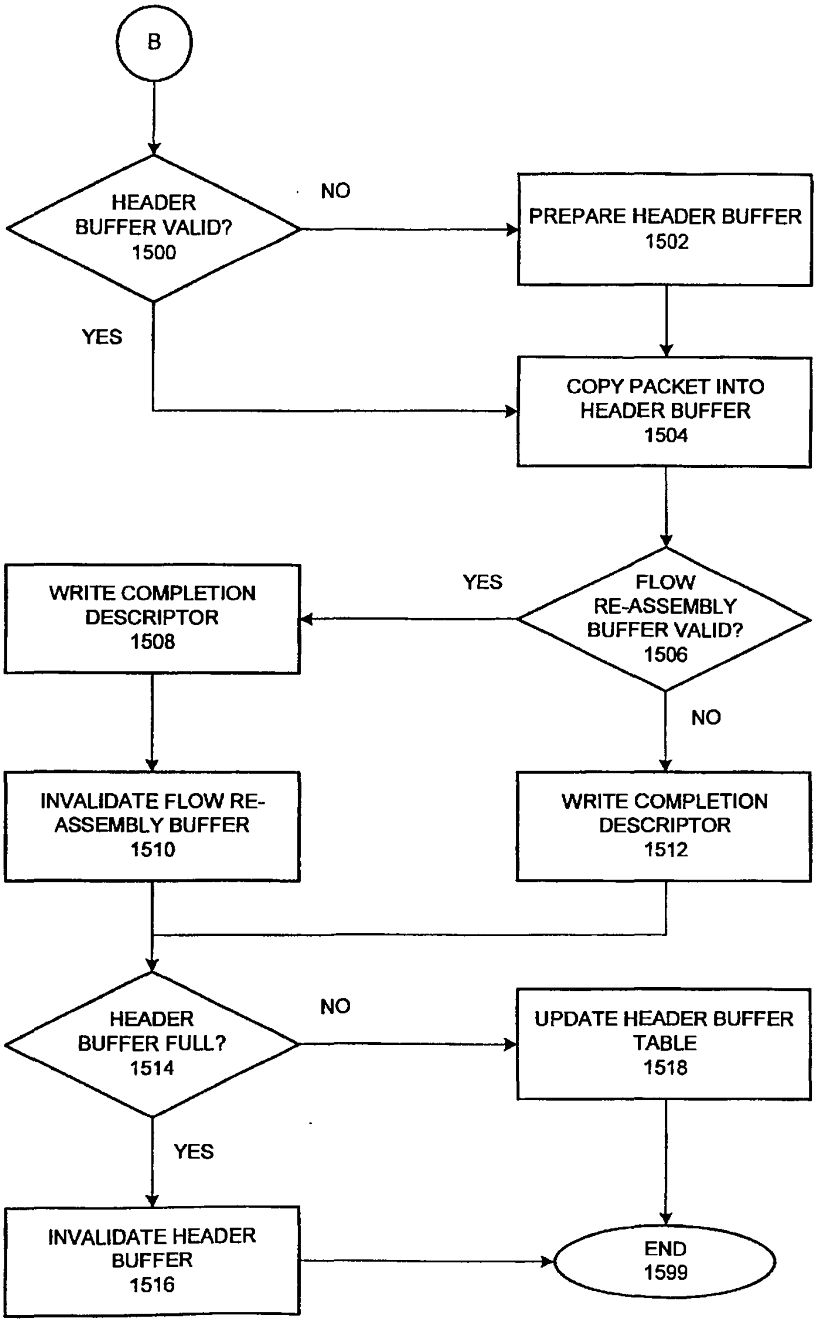

B
HEADER BUFFER VALID? 1500
NO
YES
PREPARE HEADER BUFFER 1502
COPY PACKET INTO HEADER BUFFER 1504
FLOW RE-ASSEMBLY BUFFER VALID? 1506
YES
NO
WRITE COMPLETION DESCRIPTOR 1508
INVALIDATE FLOW RE-ASSEMBLY BUFFER 1510
WRITE COMPLETION DESCRIPTOR 1512
HEADER BUFFER FULL? 1514
NO
YES
UPDATE HEADER BUFFER TABLE 1518
INVALIDATE HEADER BUFFER 1516
END 1599

FIG. 15

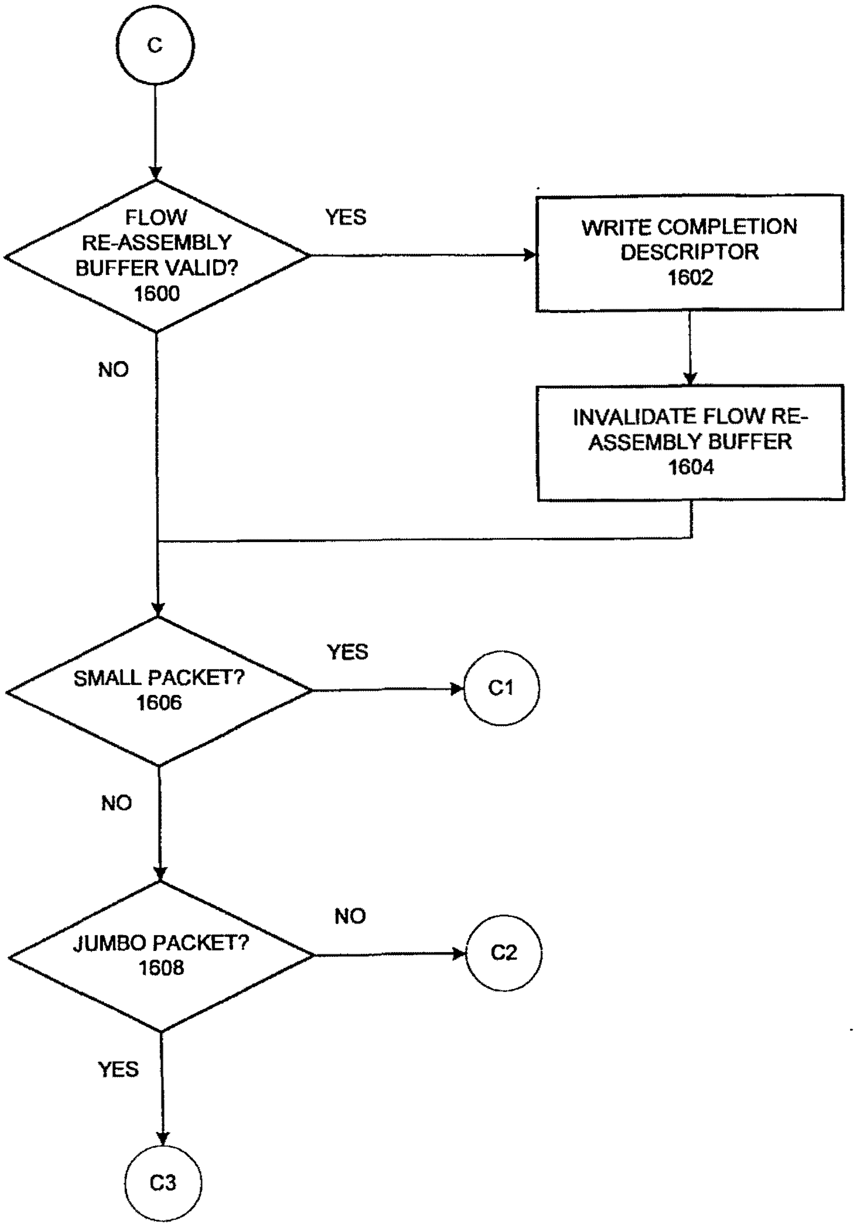

C
FLOW RE-ASSEMBLY BUFFER VALID? 1600
YES
WRITE COMPLETION DESCRIPTOR 1602
INVALIDATE FLOW RE-ASSEMBLY BUFFER 1604
NO
SMALL PACKET? 1606
YES
C1
NO
JUMBO PACKET? 1608
NO
C2
YES
C3

FIG. 16A

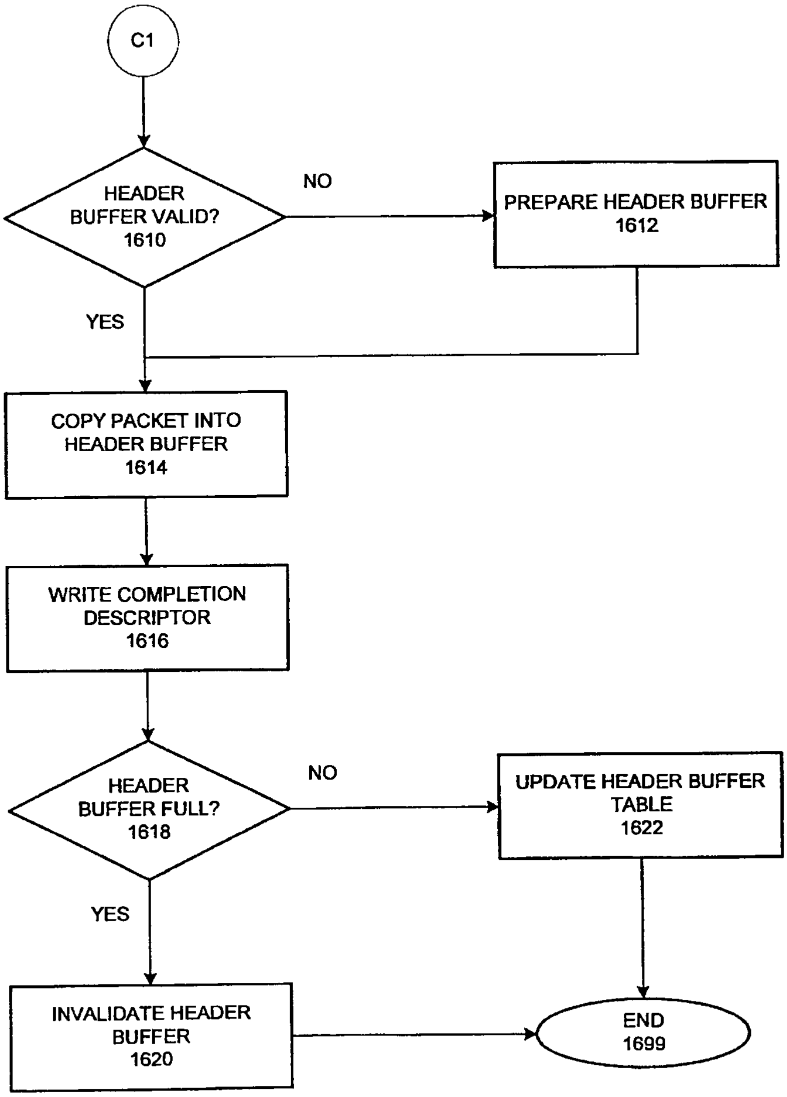

C1
HEADER BUFFER VALID? 1610
NO
PREPARE HEADER BUFFER 1612
YES
COPY PACKET INTO HEADER BUFFER 1614
WRITE COMPLETION DESCRIPTOR 1616
HEADER BUFFER FULL? 1618
NO
UPDATE HEADER BUFFER TABLE 1622
YES
INVALIDATE HEADER BUFFER 1620
END 1699

FIG. 16B

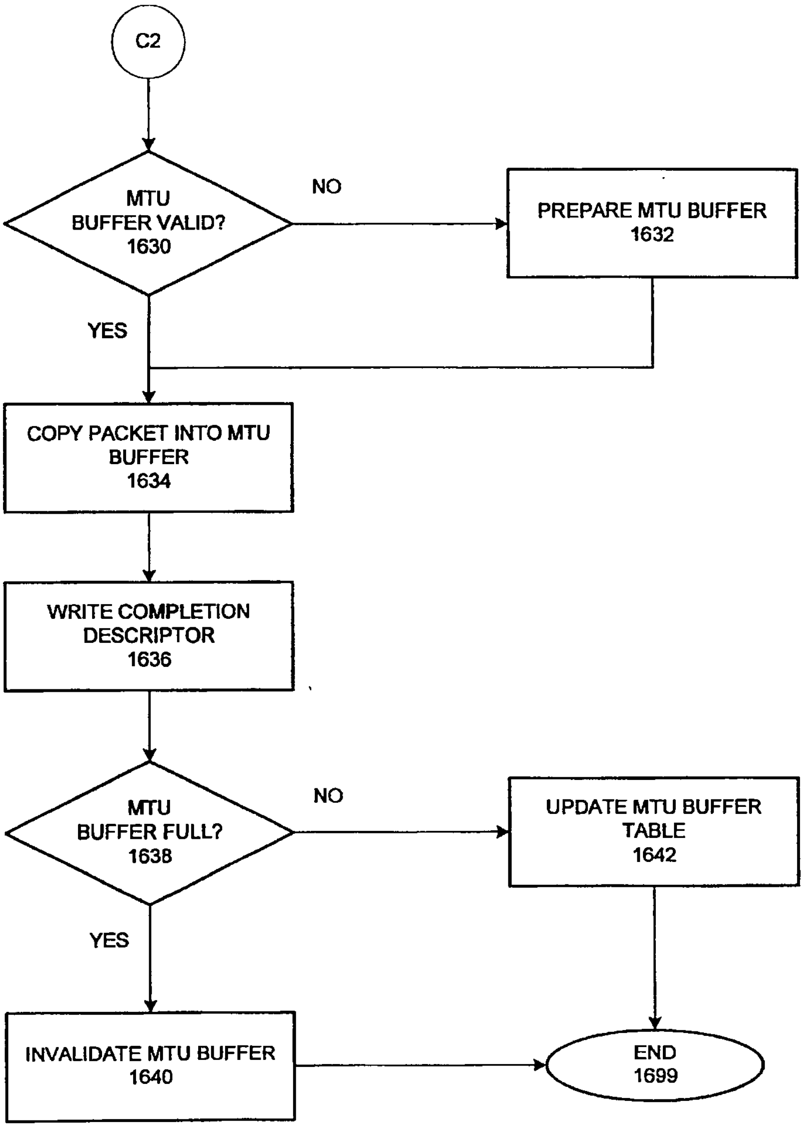

C2
MTU BUFFER VALID? 1630
NO
PREPARE MTU BUFFER 1632
YES
COPY PACKET INTO MTU BUFFER 1634
WRITE COMPLETION DESCRIPTOR 1636
MTU BUFFER FULL? 1638
NO
UPDATE MTU BUFFER TABLE 1642
YES
INVALIDATE MTU BUFFER 1640
END 1699

FIG. 16C

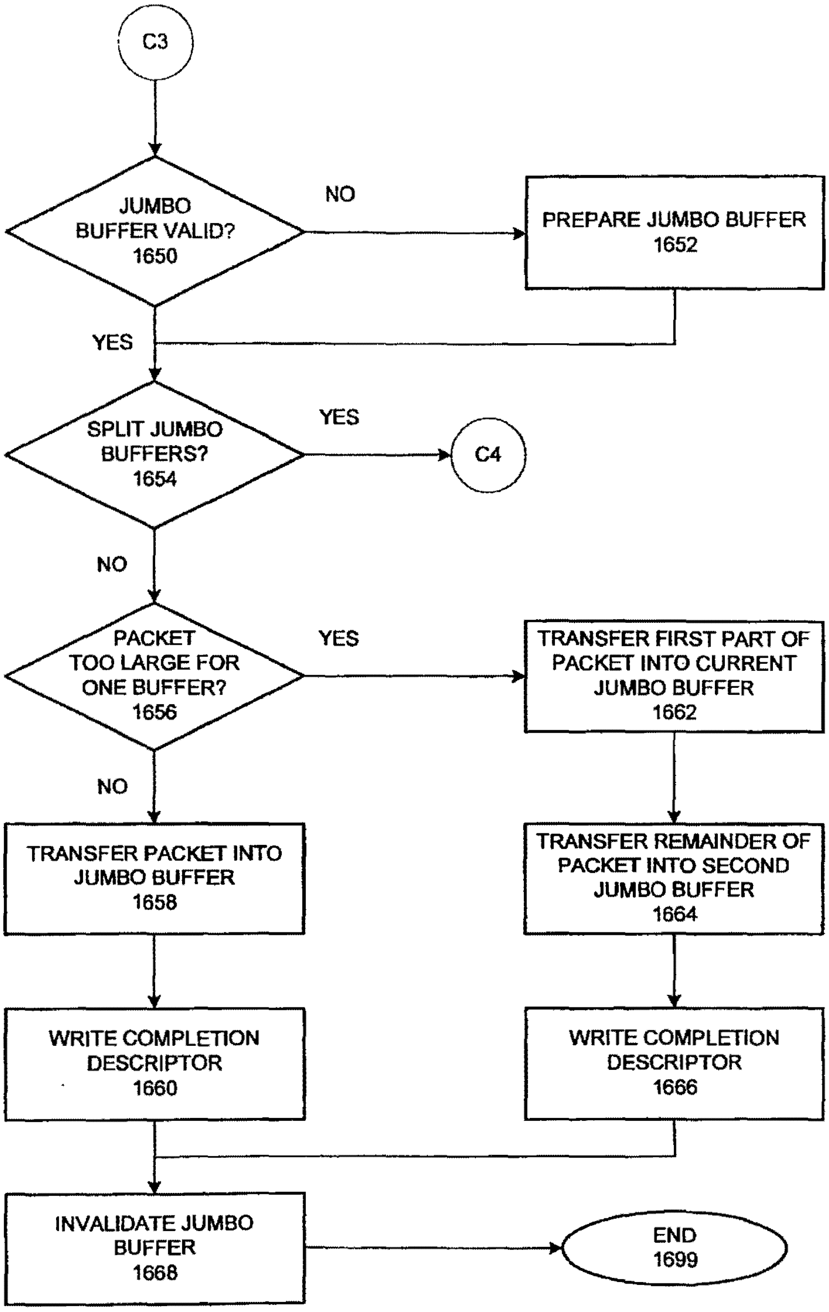

C3
JUMBO BUFFER VALID? 1650
NO
PREPARE JUMBO BUFFER 1652
YES
SPLIT JUMBO BUFFERS? 1654
YES
C4
NO
PACKET TOO LARGE FOR ONE BUFFER? 1656
YES
TRANSFER FIRST PART OF PACKET INTO CURRENT JUMBO BUFFER 1662
NO
TRANSFER PACKET INTO JUMBO BUFFER 1658
TRANSFER REMAINDER OF PACKET INTO SECOND JUMBO BUFFER 1664
WRITE COMPLETION DESCRIPTOR 1660
WRITE COMPLETION DESCRIPTOR 1666
INVALIDATE JUMBO BUFFER 1668
END 1699

FIG. 16D

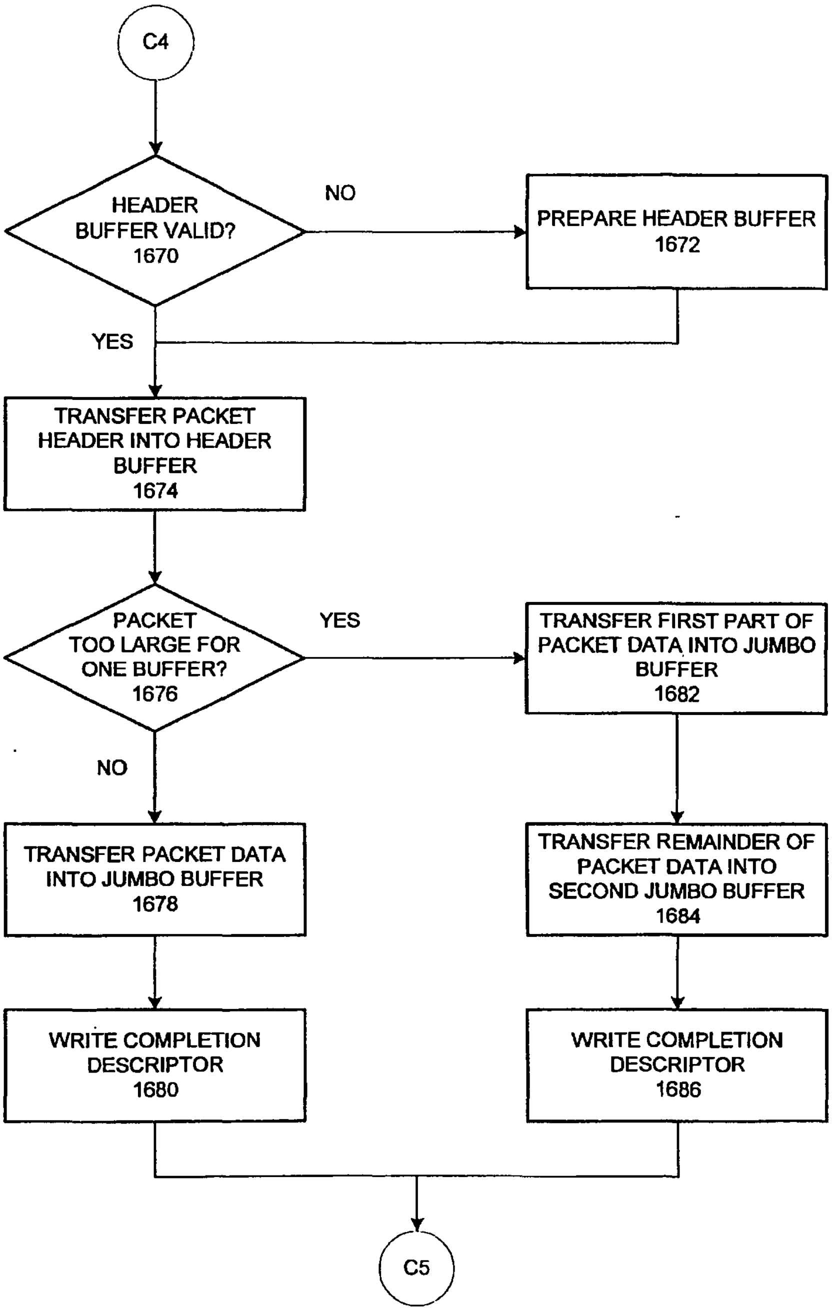

C4
HEADER BUFFER VALID? 1670
NO
PREPARE HEADER BUFFER 1672
YES
TRANSFER PACKET HEADER INTO HEADER BUFFER 1674
PACKET TOO LARGE FOR ONE BUFFER? 1676
YES
TRANSFER FIRST PART OF PACKET DATA INTO JUMBO BUFFER 1682
NO
TRANSFER PACKET DATA INTO JUMBO BUFFER 1678
TRANSFER REMAINDER OF PACKET DATA INTO SECOND JUMBO BUFFER 1684
WRITE COMPLETION DESCRIPTOR 1680
WRITE COMPLETION DESCRIPTOR 1686
C5

FIG. 16E

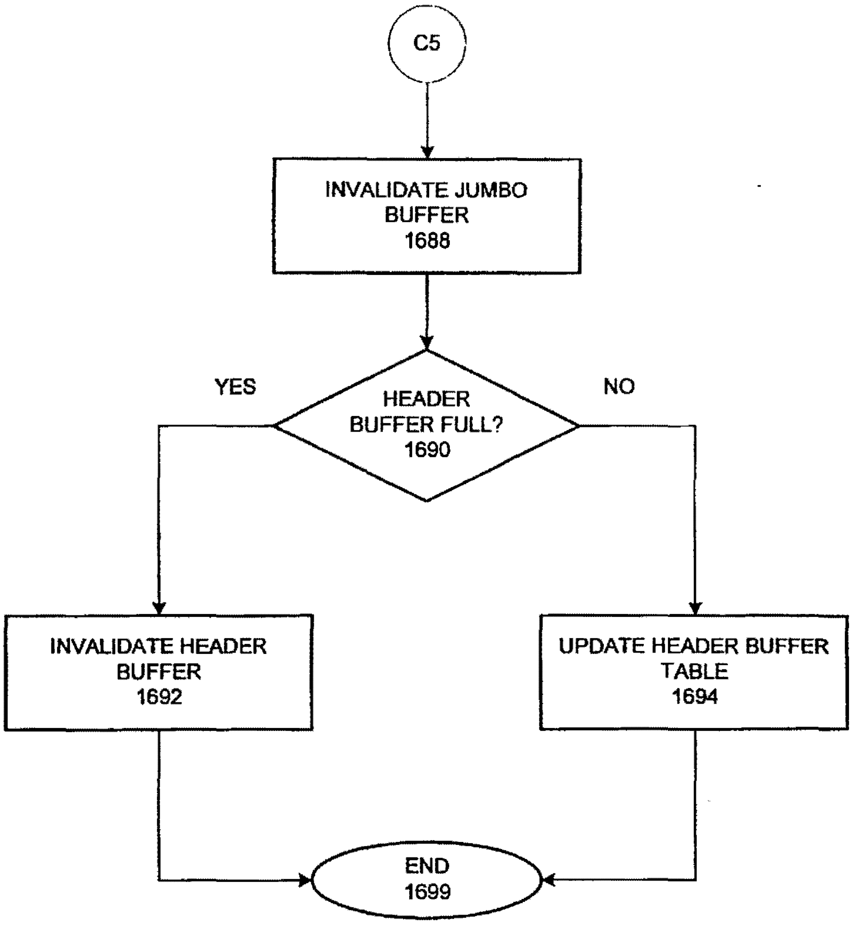

C5
INVALIDATE JUMBO BUFFER 1688
HEADER BUFFER FULL? 1690
YES
NO
INVALIDATE HEADER BUFFER 1692
UPDATE HEADER BUFFER TABLE 1694
END 1699

FIG. 16F

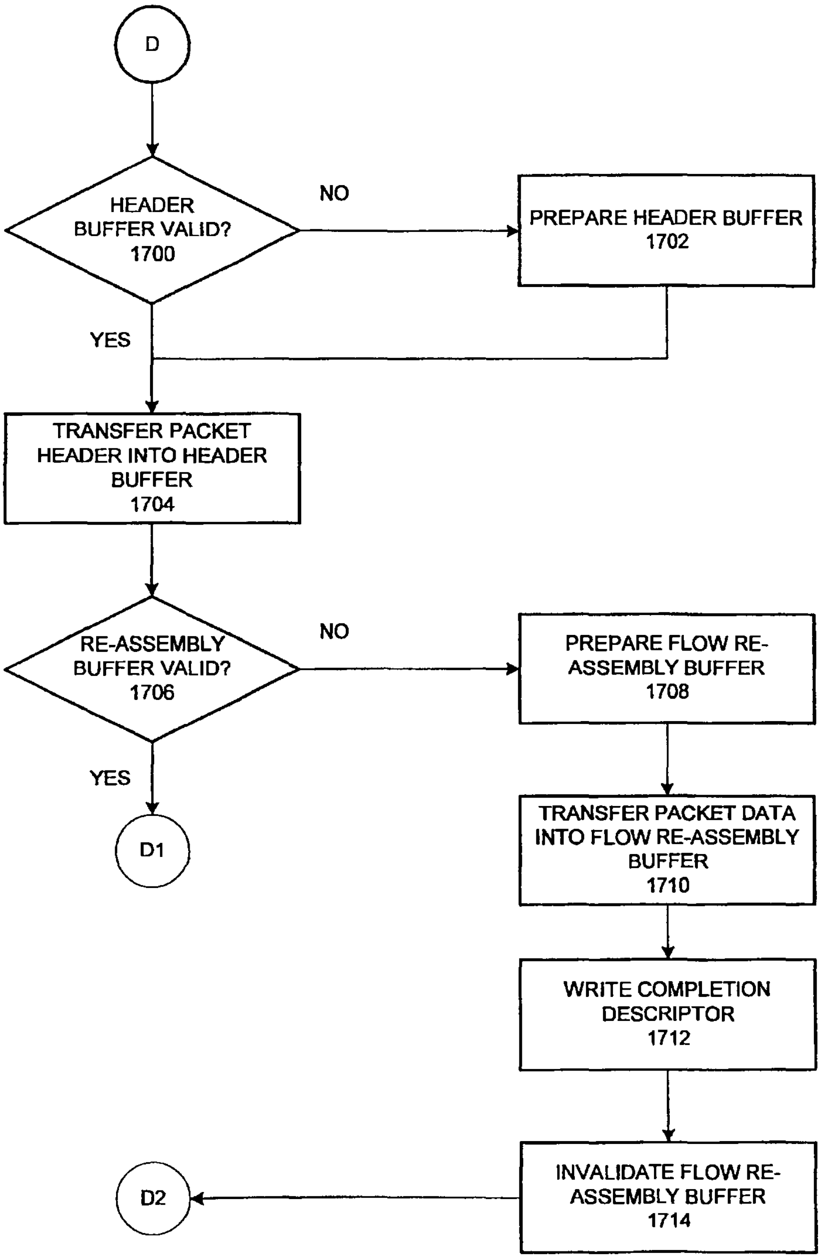

D
HEADER BUFFER VALID? 1700
NO
PREPARE HEADER BUFFER 1702
YES
TRANSFER PACKET HEADER INTO HEADER BUFFER 1704
RE-ASSEMBLY BUFFER VALID? 1706
NO
PREPARE FLOW RE-ASSEMBLY BUFFER 1708
YES
D1
TRANSFER PACKET DATA INTO FLOW RE-ASSEMBLY BUFFER 1710
WRITE COMPLETION DESCRIPTOR 1712
INVALIDATE FLOW RE-ASSEMBLY BUFFER 1714
D2

FIG. 17A

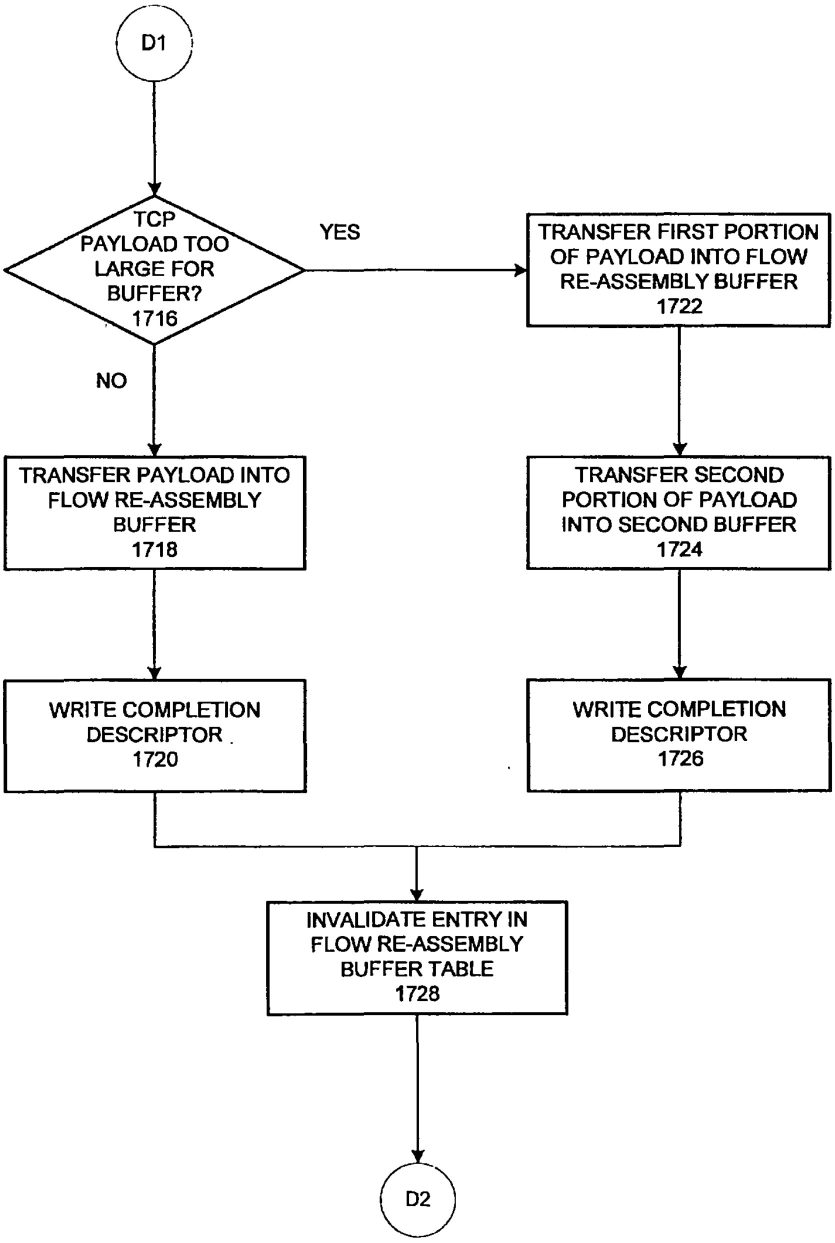

D1
TCP PAYLOAD TOO LARGE FOR BUFFER? 1716
YES
NO
TRANSFER FIRST PORTION OF PAYLOAD INTO FLOW RE-ASSEMBLY BUFFER 1722
TRANSFER PAYLOAD INTO FLOW RE-ASSEMBLY BUFFER 1718
TRANSFER SECOND PORTION OF PAYLOAD INTO SECOND BUFFER 1724
WRITE COMPLETION DESCRIPTOR 1720
WRITE COMPLETION DESCRIPTOR 1726
INVALIDATE ENTRY IN FLOW RE-ASSEMBLY BUFFER TABLE 1728
D2

FIG. 17B

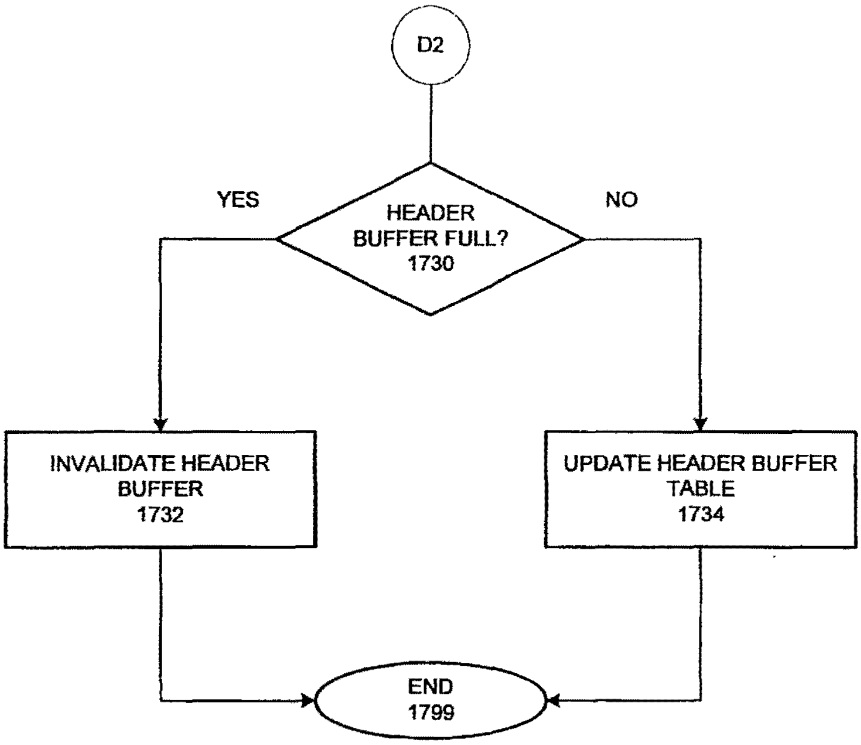

D2
YES
HEADER BUFFER FULL? 1730
NO
INVALIDATE HEADER BUFFER 1732
UPDATE HEADER BUFFER TABLE 1734
END 1799

FIG. 17C

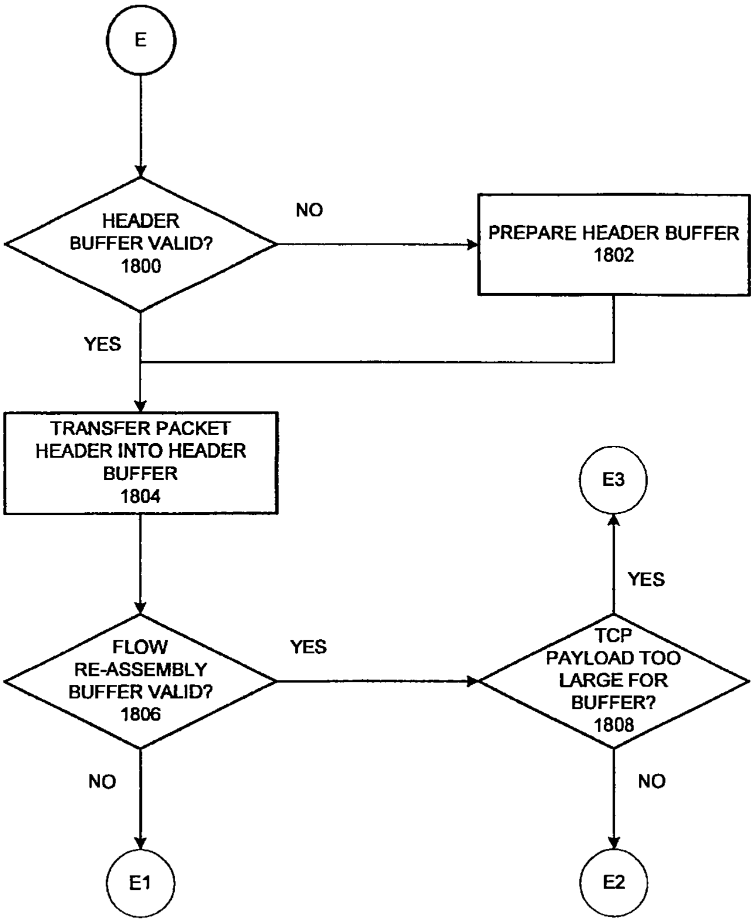

E
HEADER BUFFER VALID? 1800
NO
PREPARE HEADER BUFFER 1802
YES
TRANSFER PACKET HEADER INTO HEADER BUFFER 1804
E3
YES
FLOW RE-ASSEMBLY BUFFER VALID? 1806
YES
TCP PAYLOAD TOO LARGE FOR BUFFER? 1808
NO
NO
E1
E2

FIG. 18A

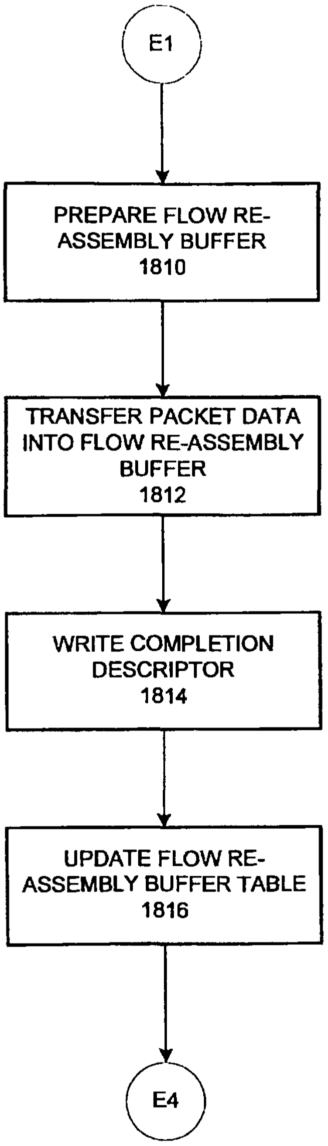

E1
PREPARE FLOW RE-ASSEMBLY BUFFER
1810
TRANSFER PACKET DATA INTO FLOW RE-ASSEMBLY BUFFER
1812
WRITE COMPLETION DESCRIPTOR
1814
UPDATE FLOW RE-ASSEMBLY BUFFER TABLE
1816
E4

FIG. 18B

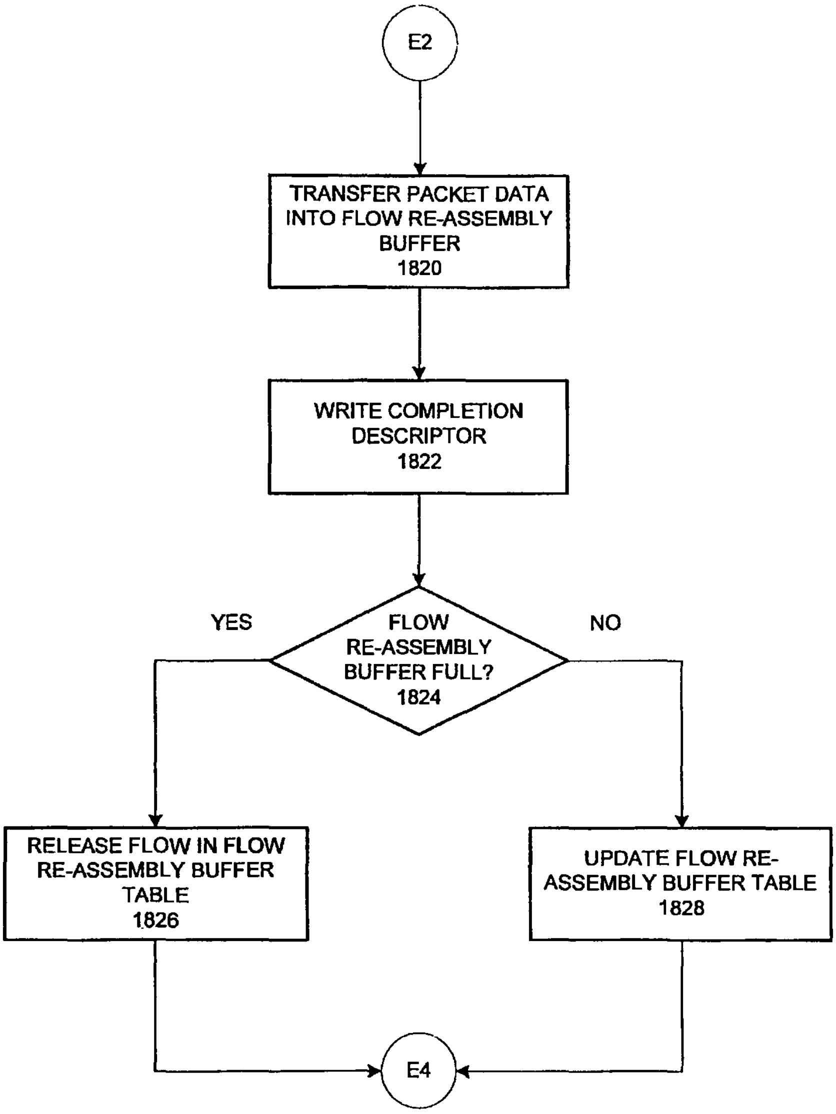

E2
TRANSFER PACKET DATA INTO FLOW RE-ASSEMBLY BUFFER 1820
WRITE COMPLETION DESCRIPTOR 1822
FLOW RE-ASSEMBLY BUFFER FULL? 1824
YES
NO
RELEASE FLOW IN FLOW RE-ASSEMBLY BUFFER TABLE 1826
UPDATE FLOW RE-ASSEMBLY BUFFER TABLE 1828
E4

FIG. 18C

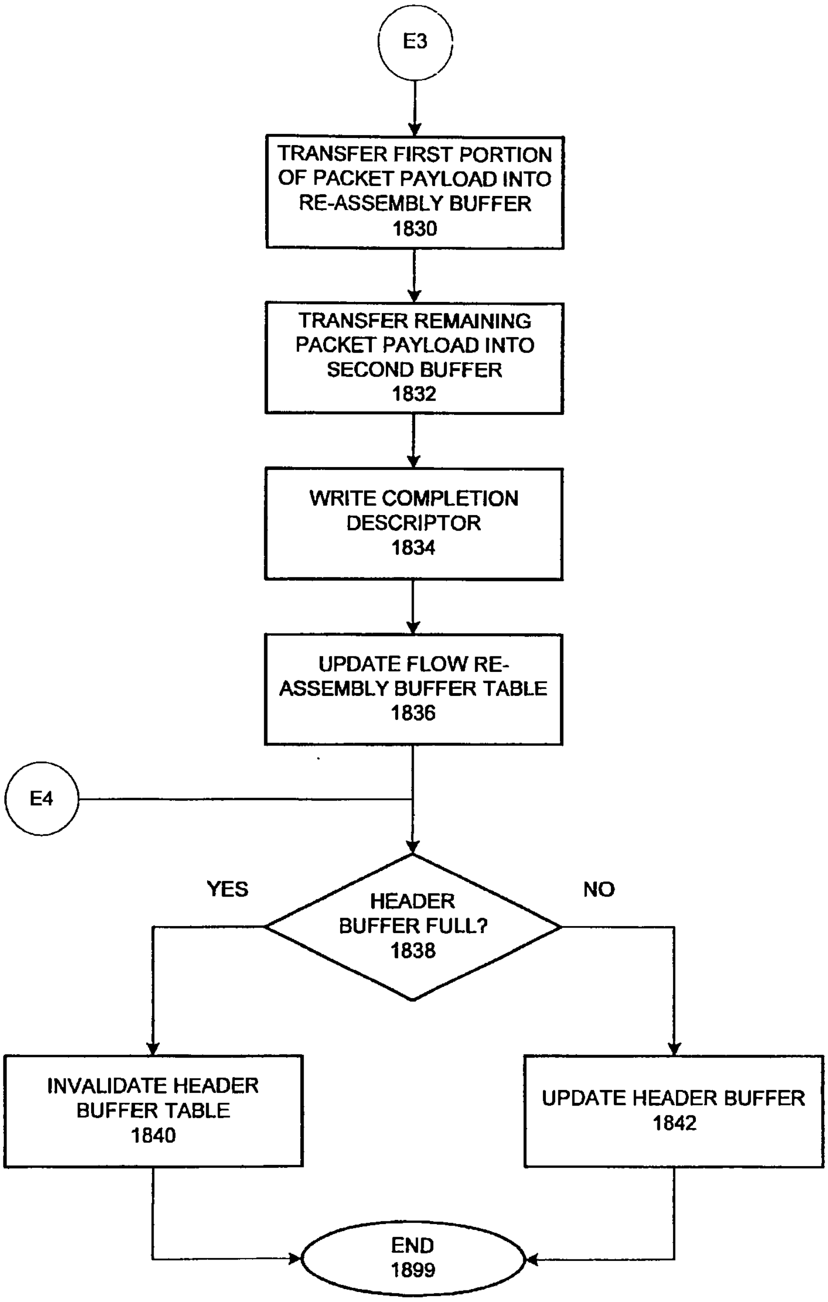

E3
TRANSFER FIRST PORTION OF PACKET PAYLOAD INTO RE-ASSEMBLY BUFFER 1830
TRANSFER REMAINING PACKET PAYLOAD INTO SECOND BUFFER 1832
WRITE COMPLETION DESCRIPTOR 1834
UPDATE FLOW RE-ASSEMBLY BUFFER TABLE 1836
E4
HEADER BUFFER FULL? 1838
YES
NO
INVALIDATE HEADER BUFFER TABLE 1840
UPDATE HEADER BUFFER 1842
END 1899

FIG. 18D

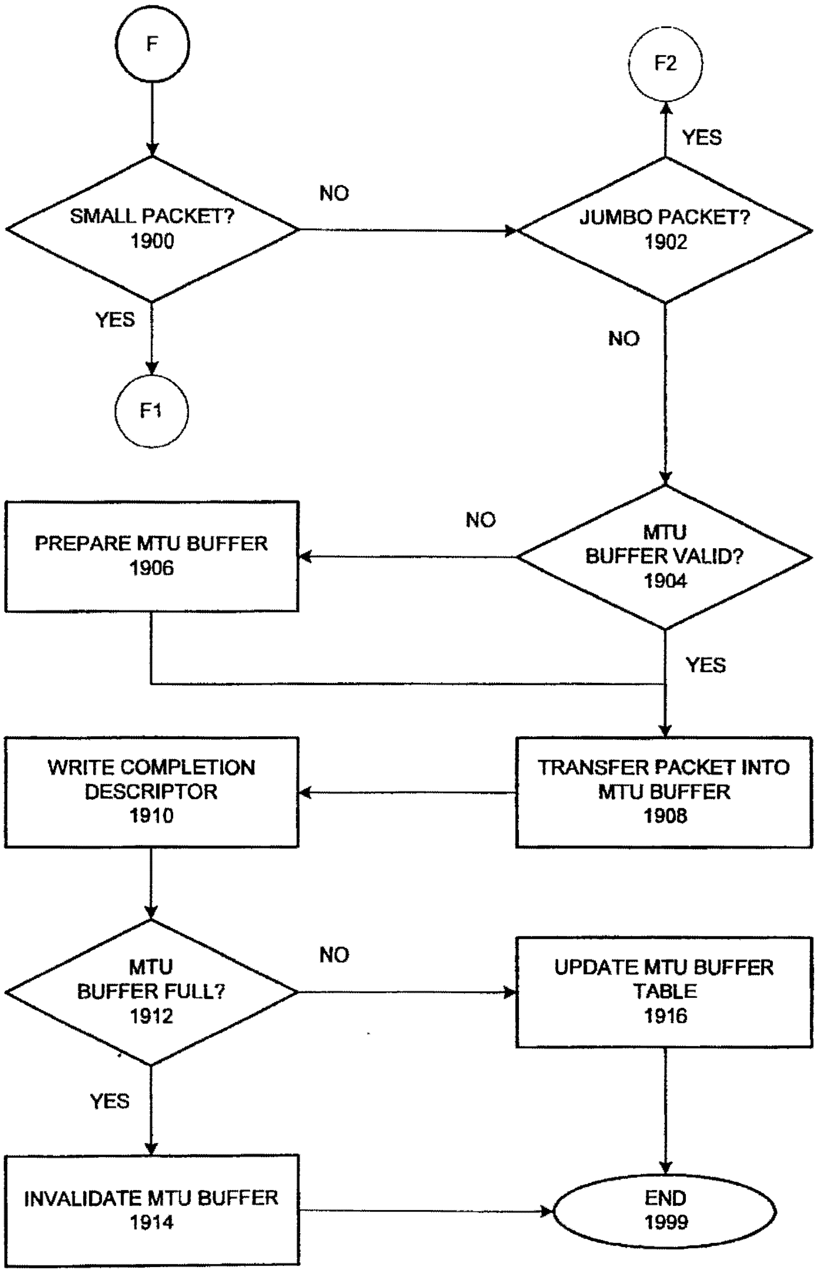

F
SMALL PACKET?
1900
NO
JUMBO PACKET?
1902
YES
F2
YES
F1
NO
PREPARE MTU BUFFER
1906
NO
MTU
BUFFER VALID?
1904
YES
WRITE COMPLETION
DESCRIPTOR
1910
TRANSFER PACKET INTO
MTU BUFFER
1908
MTU
BUFFER FULL?
1912
NO
UPDATE MTU BUFFER
TABLE
1916
YES
INVALIDATE MTU BUFFER
1914
END
1999

FIG. 19A

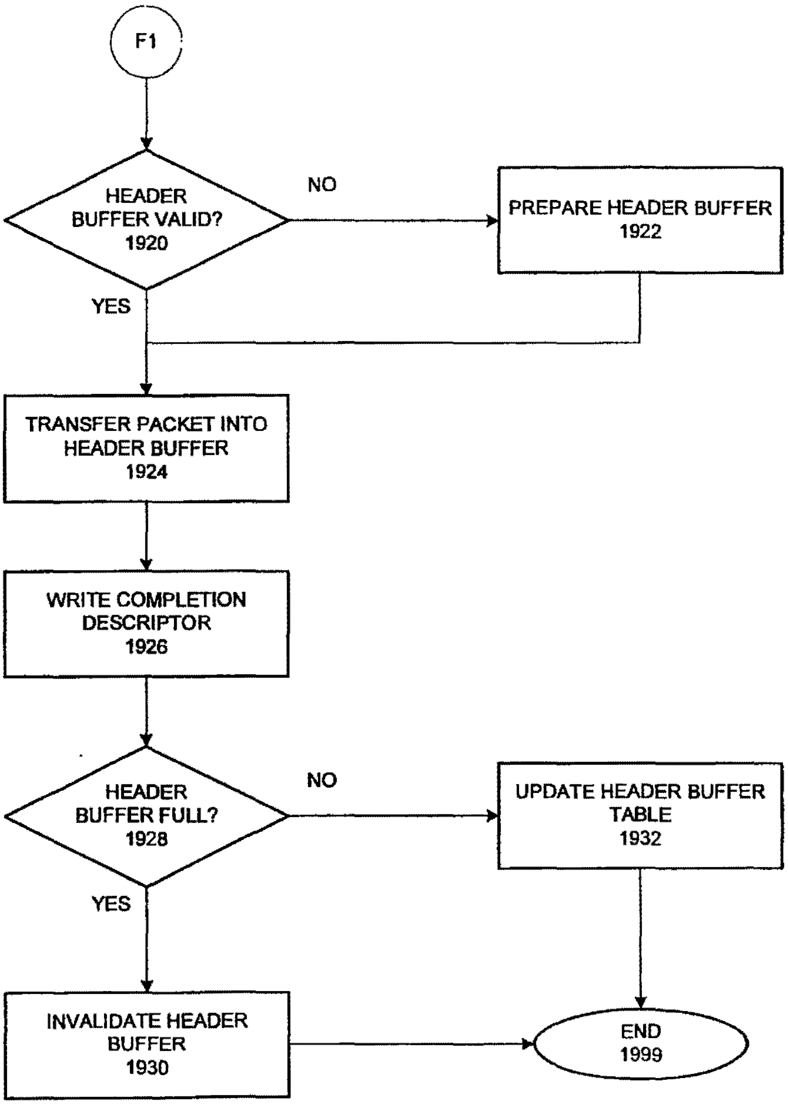

F1
HEADER BUFFER VALID? 1920
NO
PREPARE HEADER BUFFER 1922
YES
TRANSFER PACKET INTO HEADER BUFFER 1924
WRITE COMPLETION DESCRIPTOR 1926
HEADER BUFFER FULL? 1928
NO
UPDATE HEADER BUFFER TABLE 1932
YES
INVALIDATE HEADER BUFFER 1930
END 1999

FIG. 19B

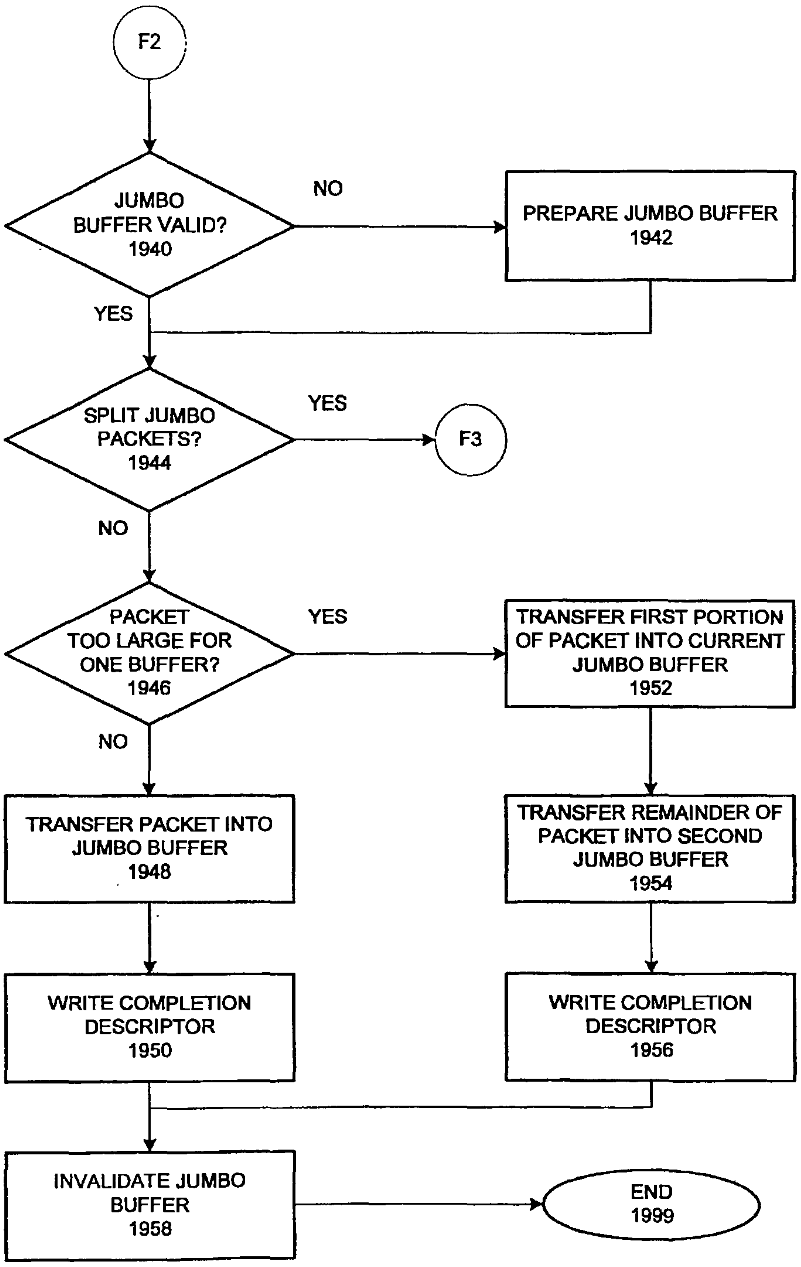

F2
JUMBO BUFFER VALID? 1940
NO
PREPARE JUMBO BUFFER 1942
YES
SPLIT JUMBO PACKETS? 1944
YES
F3
NO
PACKET TOO LARGE FOR ONE BUFFER? 1946
YES
TRANSFER FIRST PORTION OF PACKET INTO CURRENT JUMBO BUFFER 1952
NO
TRANSFER PACKET INTO JUMBO BUFFER 1948
TRANSFER REMAINDER OF PACKET INTO SECOND JUMBO BUFFER 1954
WRITE COMPLETION DESCRIPTOR 1950
WRITE COMPLETION DESCRIPTOR 1956
INVALIDATE JUMBO BUFFER 1958
END 1999

FIG. 19C

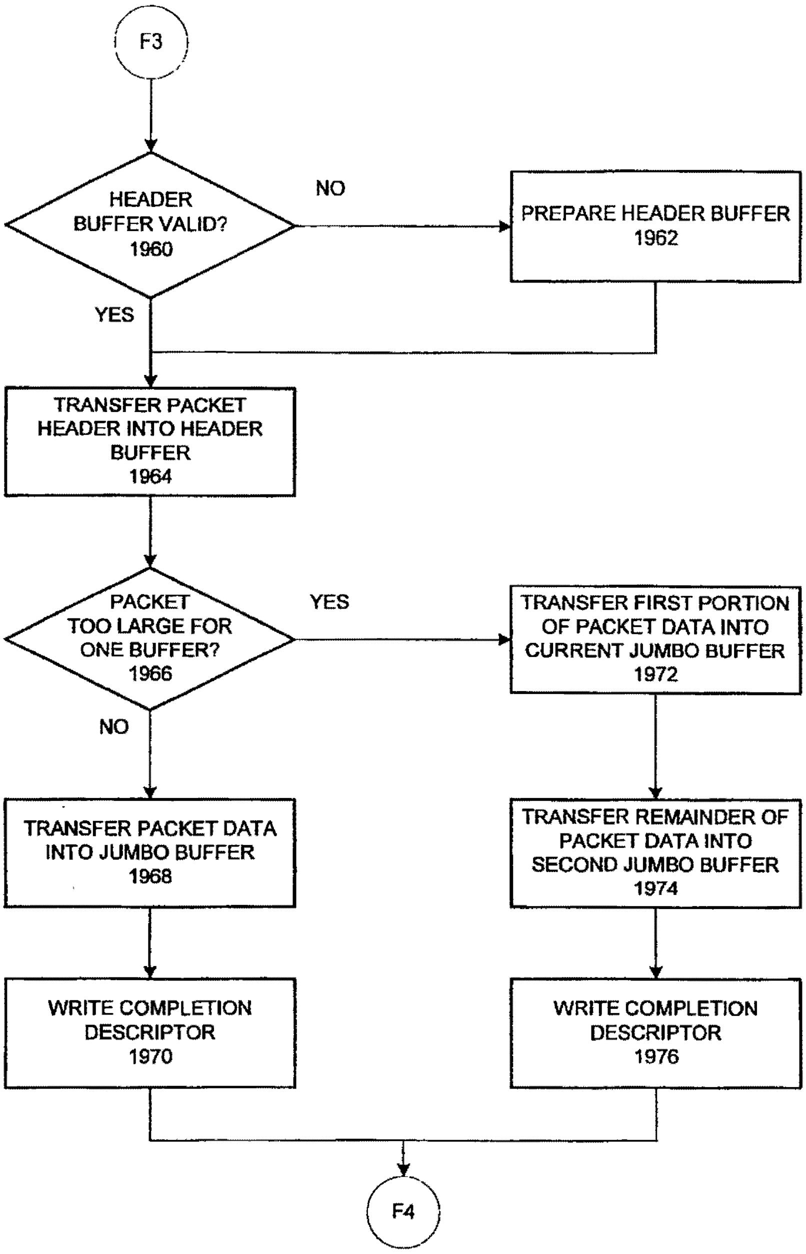

F3
HEADER BUFFER VALID? 1960
NO
PREPARE HEADER BUFFER 1962
YES
TRANSFER PACKET HEADER INTO HEADER BUFFER 1964
PACKET TOO LARGE FOR ONE BUFFER? 1966
YES
TRANSFER FIRST PORTION OF PACKET DATA INTO CURRENT JUMBO BUFFER 1972
NO
TRANSFER PACKET DATA INTO JUMBO BUFFER 1968
TRANSFER REMAINDER OF PACKET DATA INTO SECOND JUMBO BUFFER 1974
WRITE COMPLETION DESCRIPTOR 1970
WRITE COMPLETION DESCRIPTOR 1976
F4

FIG. 19D

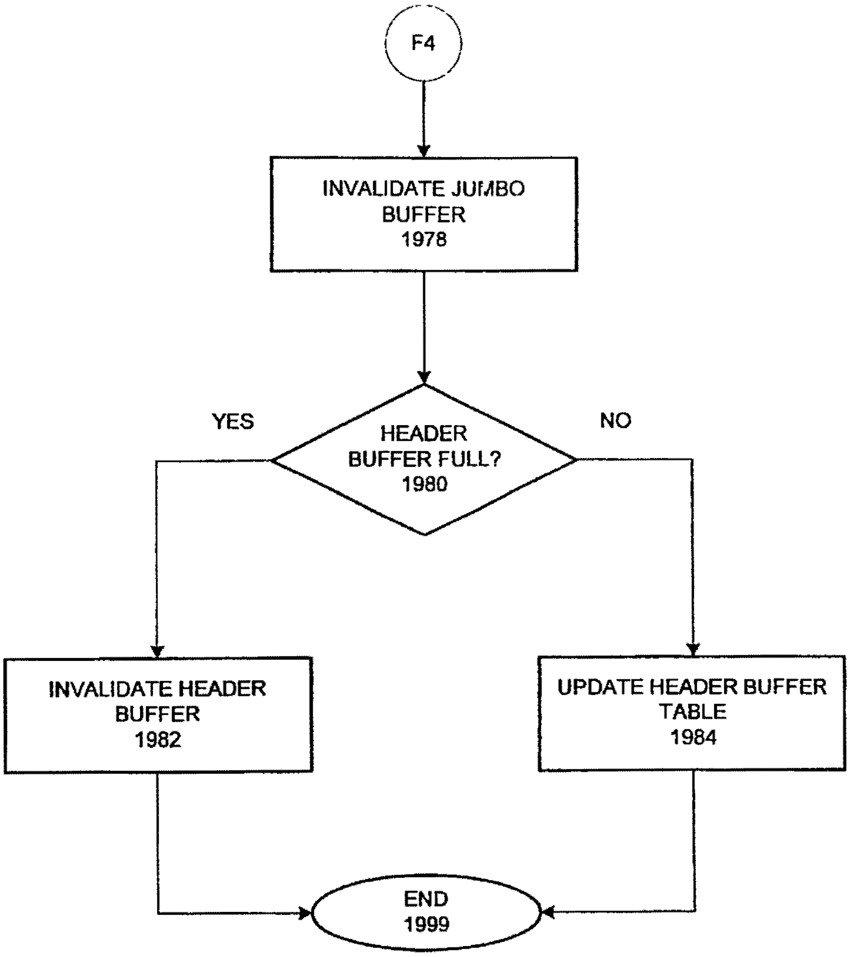

F4
INVALIDATE JUMBO BUFFER 1978
HEADER BUFFER FULL? 1980
YES
NO
INVALIDATE HEADER BUFFER 1982
UPDATE HEADER BUFFER TABLE 1984
END 1999

FIG. 19E

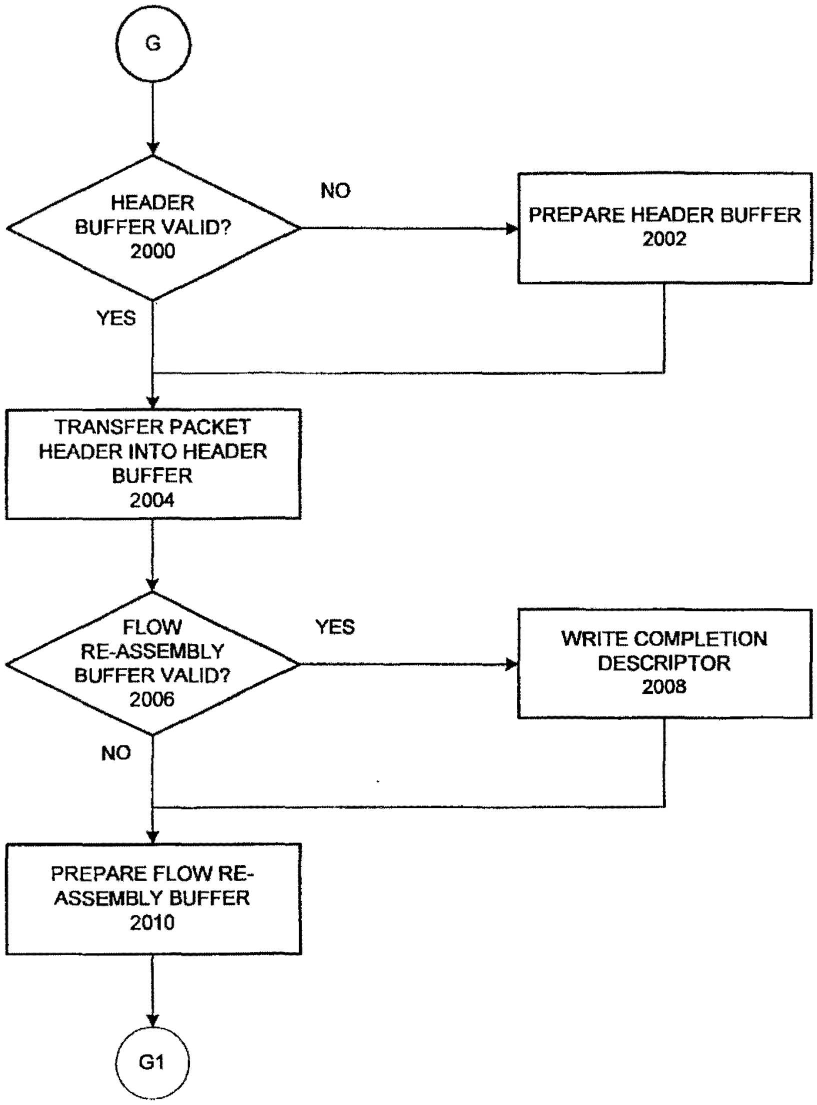

G
HEADER BUFFER VALID? 2000
NO
PREPARE HEADER BUFFER 2002
YES
TRANSFER PACKET HEADER INTO HEADER BUFFER 2004
FLOW RE-ASSEMBLY BUFFER VALID? 2006
YES
WRITE COMPLETION DESCRIPTOR 2008
NO
PREPARE FLOW RE-ASSEMBLY BUFFER 2010
G1

FIG. 20A

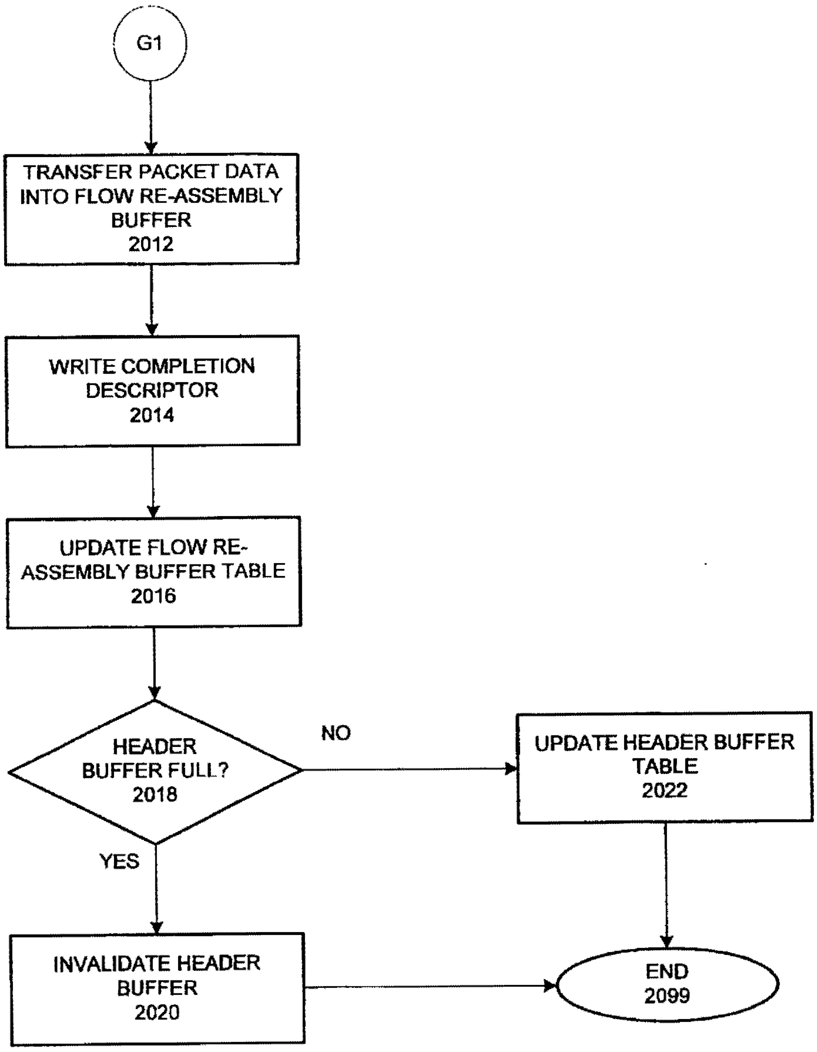

G1
TRANSFER PACKET DATA INTO FLOW RE-ASSEMBLY BUFFER
2012
WRITE COMPLETION DESCRIPTOR
2014
UPDATE FLOW RE-ASSEMBLY BUFFER TABLE
2016
HEADER BUFFER FULL?
2018
NO
UPDATE HEADER BUFFER TABLE
2022
YES
INVALIDATE HEADER BUFFER
2020
END
2099

FIG. 20B

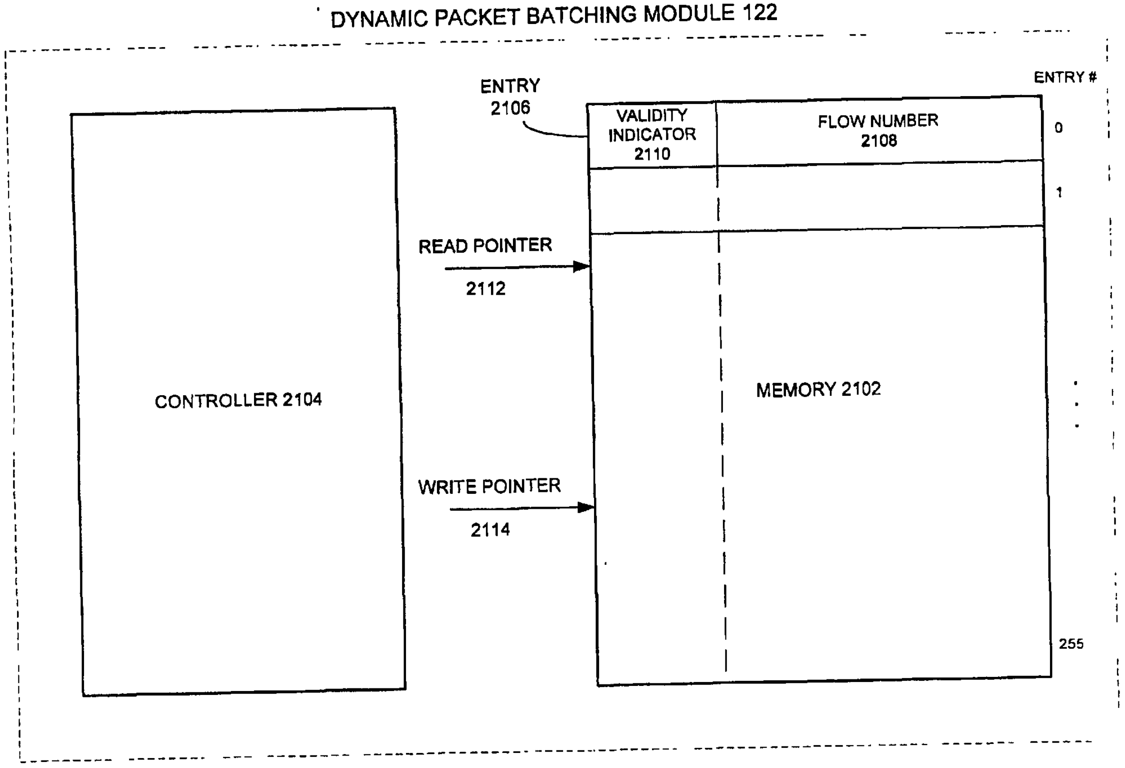

DYNAMIC PACKET BATCHING MODULE 122
ENTRY 2106
ENTRY #
VALIDITY INDICATOR 2110
FLOW NUMBER 2108
0
1
READ POINTER 2112
CONTROLLER 2104
MEMORY 2102
WRITE POINTER 2114
255

FIG. 21

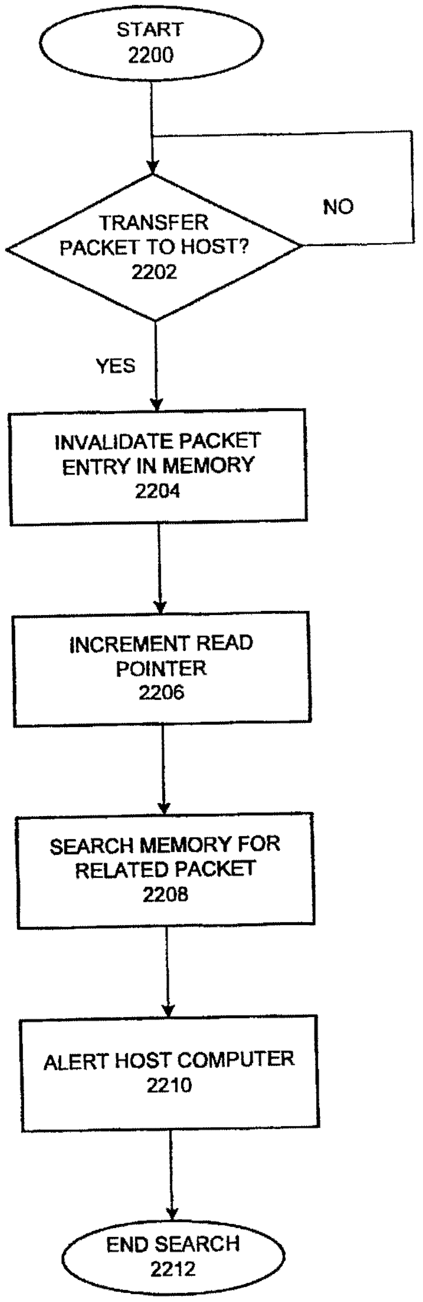

START
2200
TRANSFER
PACKET TO HOST?
2202
NO
YES
INVALIDATE PACKET
ENTRY IN MEMORY
2204
INCREMENT READ
POINTER
2206
SEARCH MEMORY FOR
RELATED PACKET
2208
ALERT HOST COMPUTER
2210
END SEARCH
2212

FIG. 22A

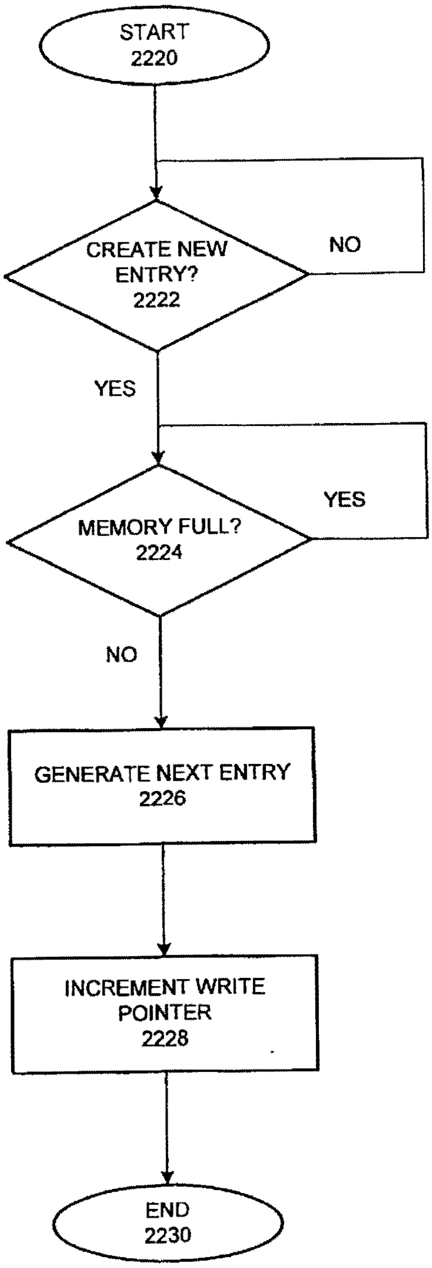

START
2220
CREATE NEW
ENTRY?
2222
NO
YES
MEMORY FULL?
2224
YES
NO
GENERATE NEXT ENTRY
2226
INCREMENT WRITE
POINTER
2228
END
2230

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

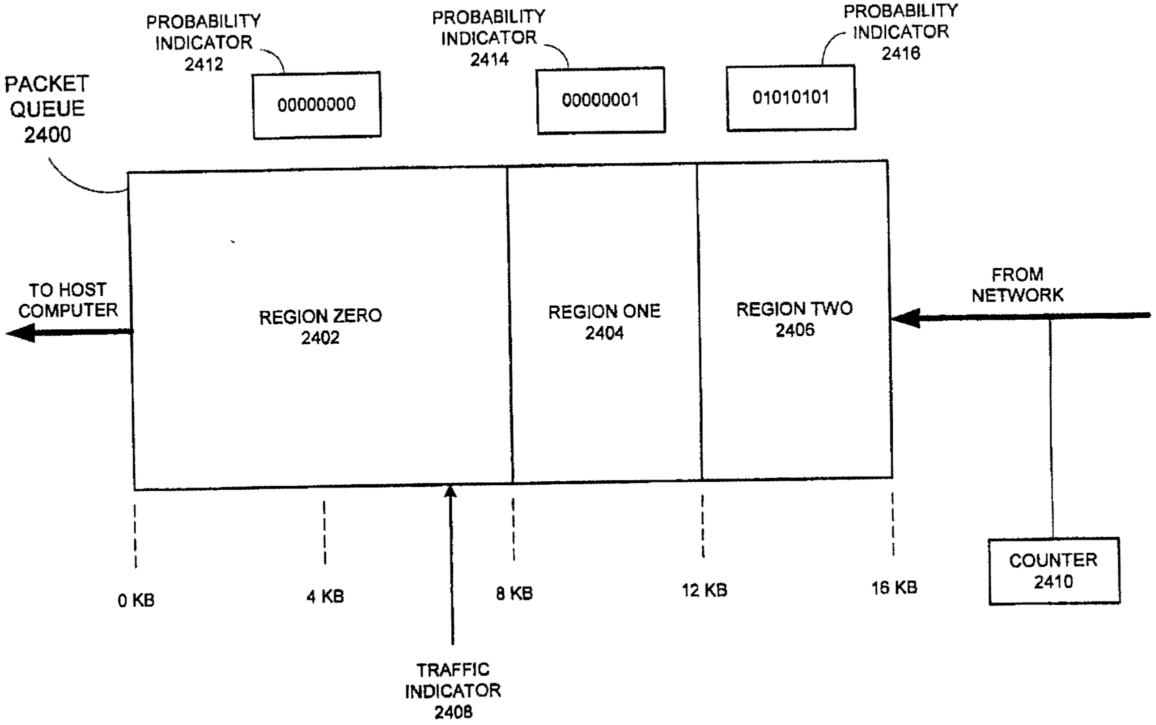

PROBABILITY INDICATOR 2412
00000000
PROBABILITY INDICATOR 2414
00000001
PROBABILITY INDICATOR 2416
01010101
PACKET QUEUE 2400
TO HOST COMPUTER
REGION ZERO 2402
REGION ONE 2404
REGION TWO 2406
FROM NETWORK
0 KB
4 KB
8 KB
12 KB
16 KB
COUNTER 2410
TRAFFIC INDICATOR 2408

FIG. 24

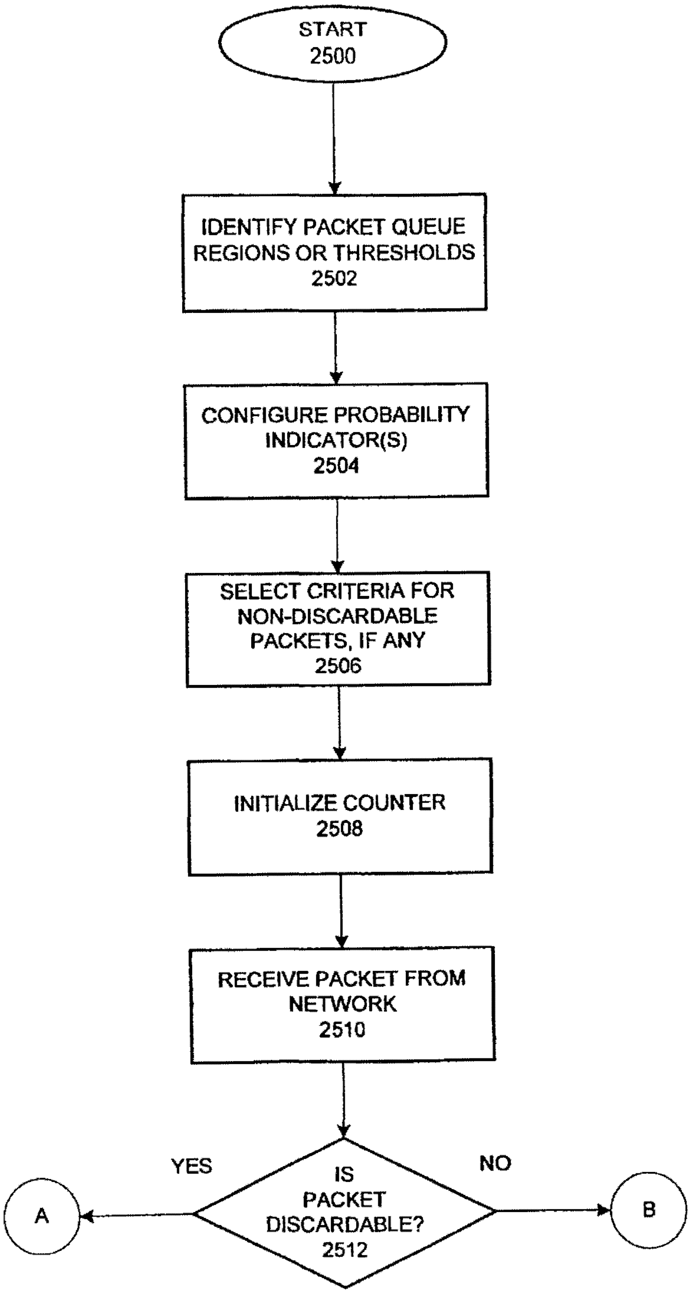

START
2500
IDENTIFY PACKET QUEUE REGIONS OR THRESHOLDS
2502
CONFIGURE PROBABILITY INDICATOR(S)
2504
SELECT CRITERIA FOR NON-DISCARDABLE PACKETS, IF ANY
2506
INITIALIZE COUNTER
2508
RECEIVE PACKET FROM NETWORK
2510
YES
IS PACKET DISCARDABLE?
2512
NO
A
B

FIG. 25A

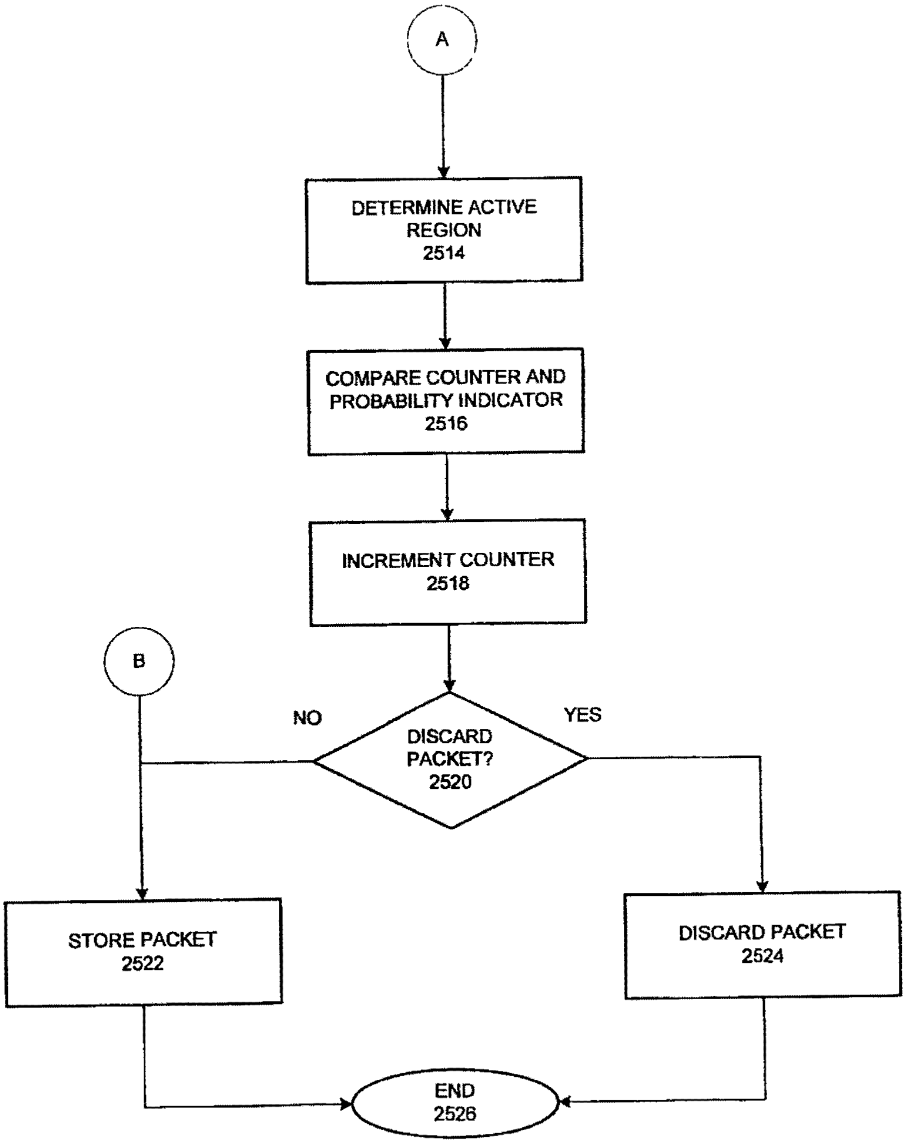

A
DETERMINE ACTIVE REGION 2514
COMPARE COUNTER AND PROBABILITY INDICATOR 2516
INCREMENT COUNTER 2518
B
NO
DISCARD PACKET? 2520
YES
STORE PACKET 2522
DISCARD PACKET 2524
END 2526

FIG. 25B

METHOD AND APPARATUS FOR DYNAMIC PACKET BATCHING WITH A HIGH PERFORMANCE NETWORK INTERFACE

TABLE OF CONTENTS

BACKGROUND

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.

flow of the packet
flow includes source
& destination of
the packet
(e)
flow database
(1)
if new flow regist
then reg the in the
flow database

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.

flow memory (database)
(1)

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.

NIC = acquistion device (1)

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.

NIC = acqu device (1)

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).

3 [illegible] source & destint packet [illegible] same [illegible] pattern

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.

acquisition [illegible] 2

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.

packet stored in 116 (mem) 4

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.

106 3
- examine
- extract
- fw - processing

flow [illegible] 110 1

destint = prog

FDBM 108 = lookup engine
5

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 [illegible] (1P)

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.

flow database 110

determ packet is part of new or existg flow

opert code

108 does (5)

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.

106 (3) process

processor

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).

(106) (6)

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 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.

(112) Select (1) proces

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.

100 (2)

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.

106 (3) identify flow pckt

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.

108 (5) flow database 110 (4) update

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**.

168

5

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.

benefits

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.

108 (5)

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 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**.

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 preselected 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.

TABLE 1-continued

Op. Code	Criteria for Selection	Result of Operation Code
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 **802a** to supplement the packet so that the packet portion ends at an appropriate boundary (e.g., byte, word, double word).

Filler **802a** 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 **802a** 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 preselected 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 multitasking 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-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 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 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 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 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 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.

* * * * *

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 (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.)

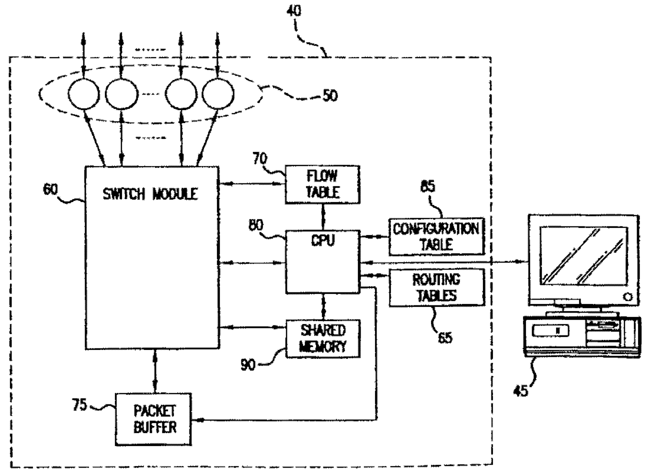
Primary Examiner—Douglas Olms
Assistant Examiner—Phirin Sam

(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

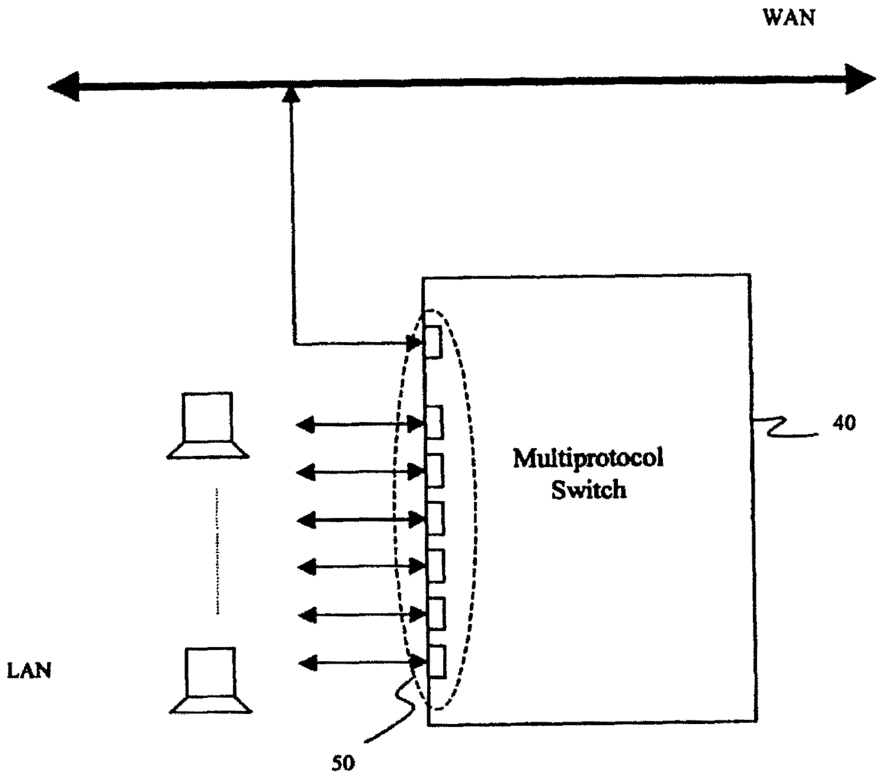


FIG. 1

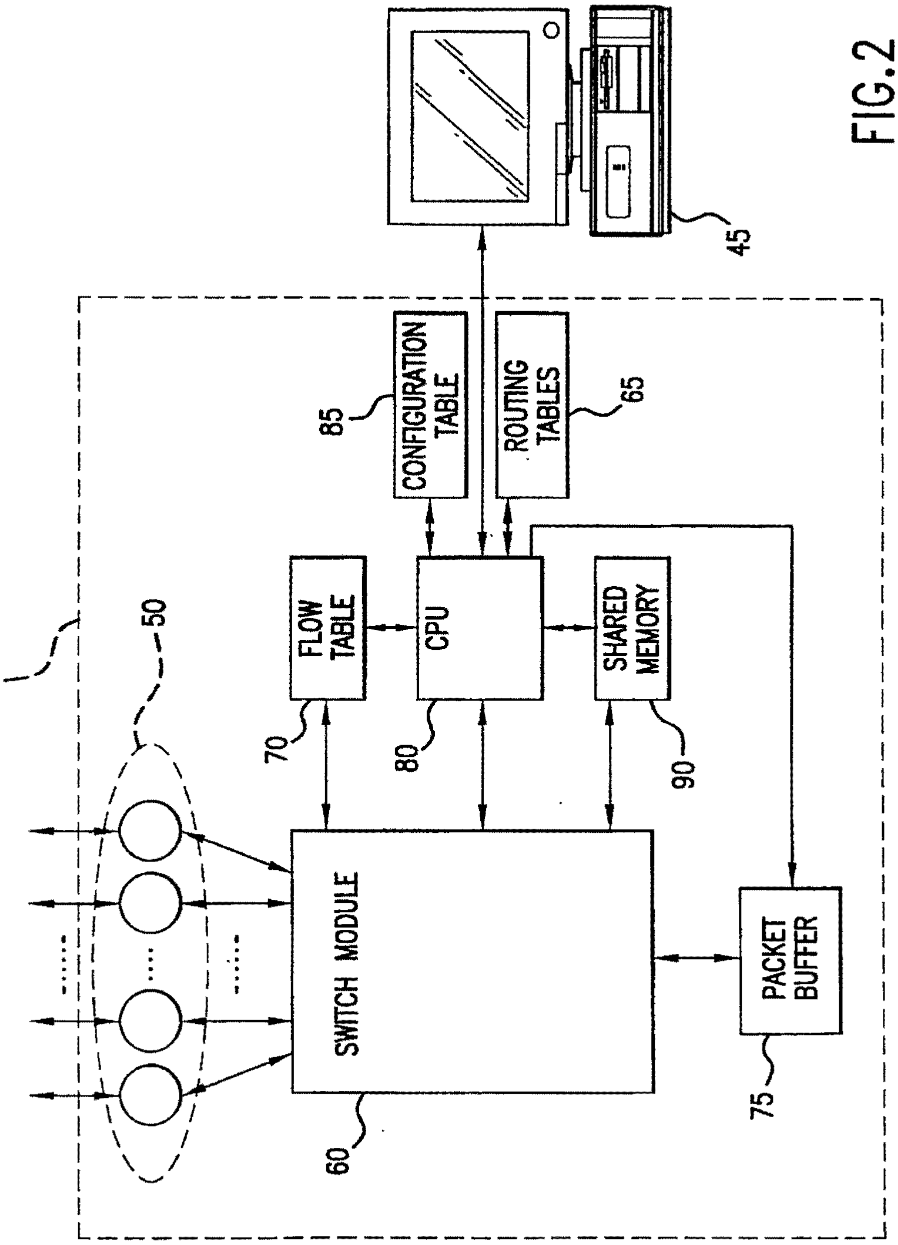

45
85
CONFIGURATION TABLE
ROUTING TABLES
65
40
50
FLOW TABLE
CPU
SHARED MEMORY
70
80
90
SWITCH MODULE
PACKET BUFFER
60
75

FIG.2

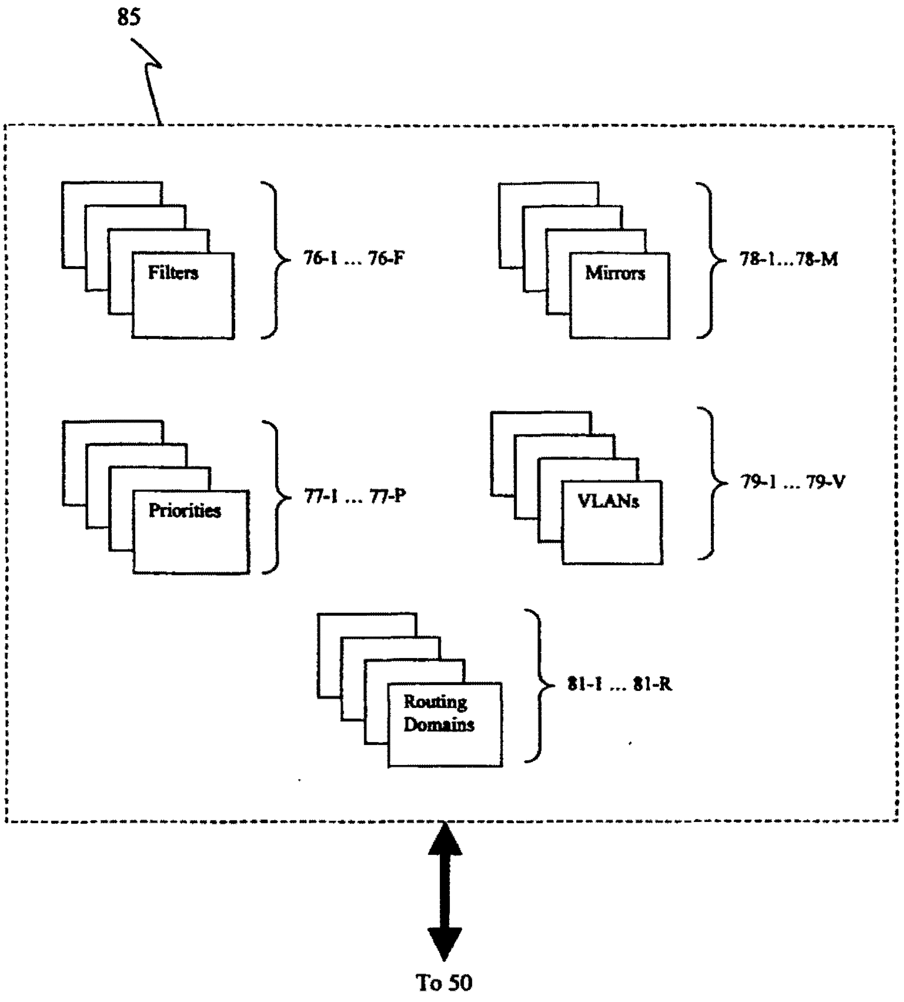

85
Filters
76-1 ... 76-F
Mirrors
78-1...78-M
Priorities
77-1 ... 77-P
VLANs
79-1 ... 79-V
Routing Domains
81-1 ... 81-R
To 50

FIG. 3

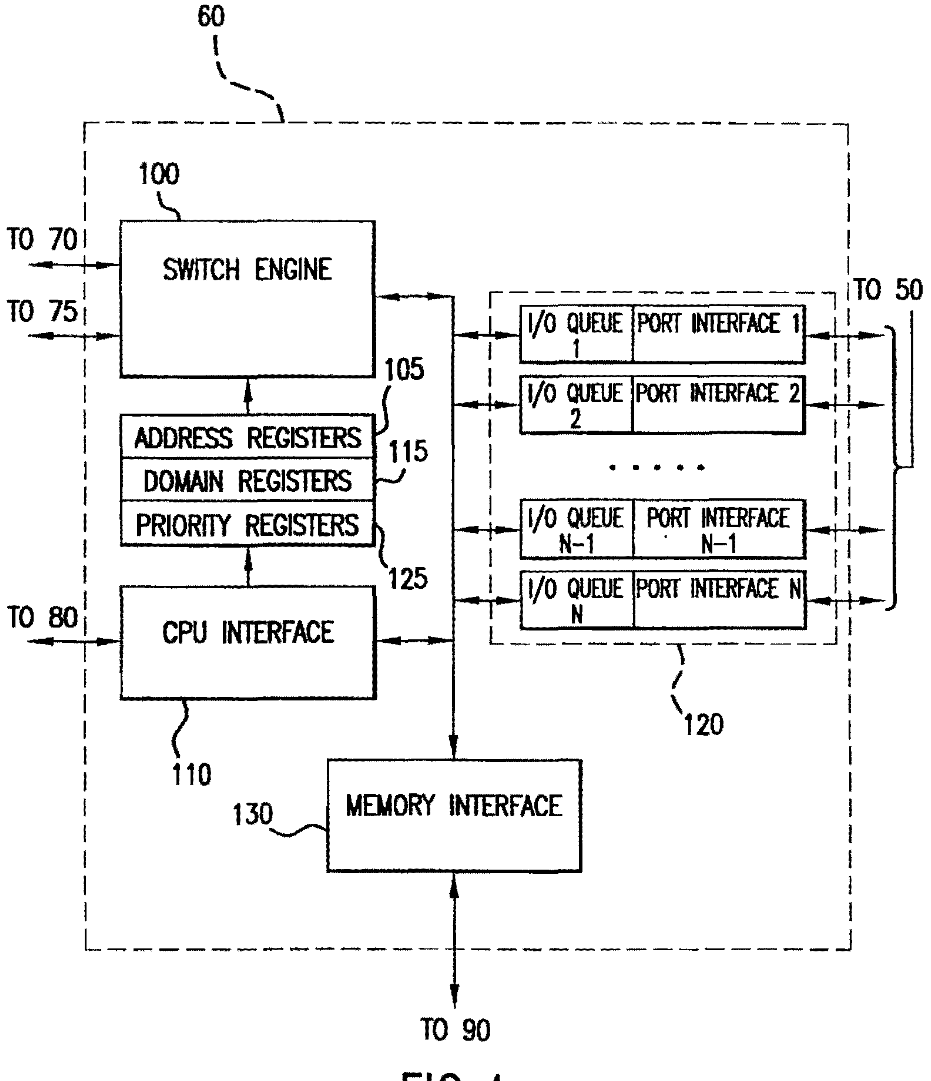

60
100
SWITCH ENGINE
TO 70
TO 75
105
ADDRESS REGISTERS
DOMAIN REGISTERS
PRIORITY REGISTERS
115
125
TO 80
CPU INTERFACE
110
I/O QUEUE 1
PORT INTERFACE 1
I/O QUEUE 2
PORT INTERFACE 2
I/O QUEUE N-1
PORT INTERFACE N-1
I/O QUEUE N
PORT INTERFACE N
TO 50
120
130
MEMORY INTERFACE
TO 90

FIG.4

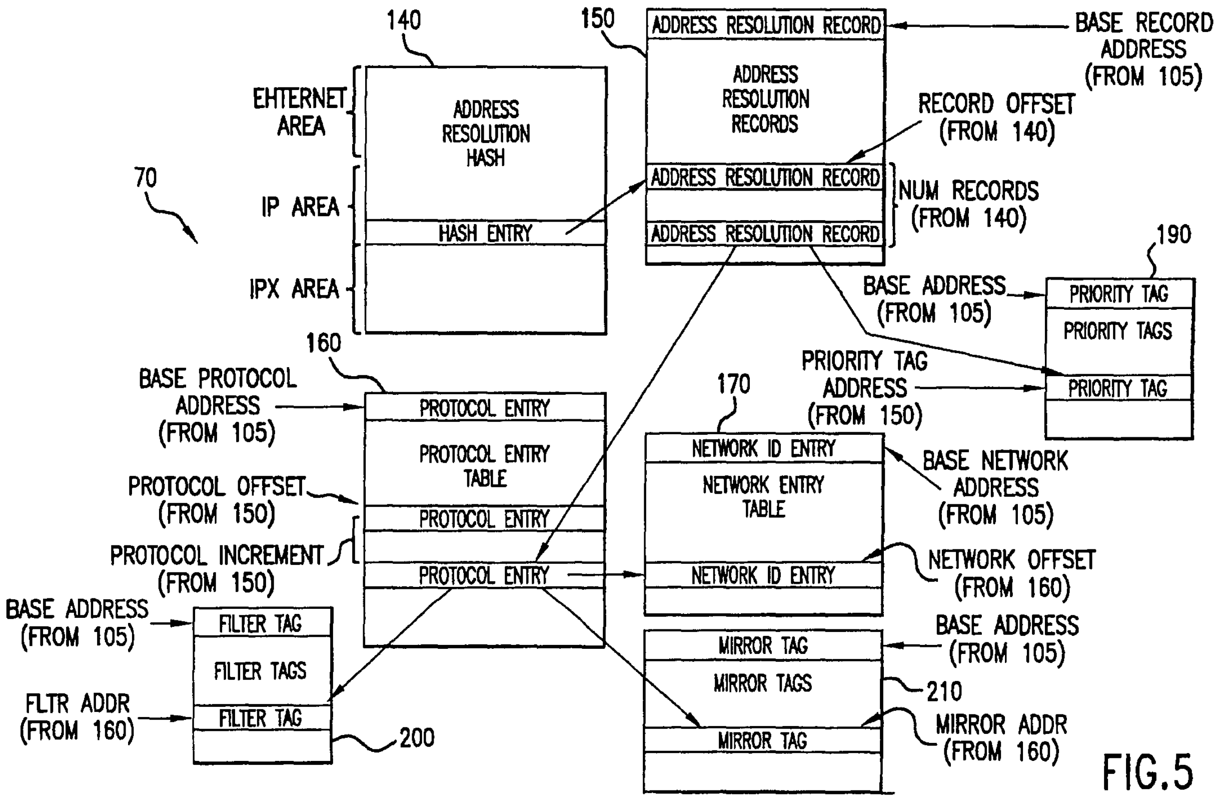

140
150
70
EHTERNET AREA
IP AREA
IPX AREA
ADDRESS RESOLUTION HASH
HASH ENTRY
ADDRESS RESOLUTION RECORD
ADDRESS RESOLUTION RECORDS
ADDRESS RESOLUTION RECORD
ADDRESS RESOLUTION RECORD
BASE RECORD ADDRESS (FROM 105)
RECORD OFFSET (FROM 140)
NUM RECORDS (FROM 140)
190
BASE ADDRESS (FROM 105)
PRIORITY TAG
PRIORITY TAGS
PRIORITY TAG
PRIORITY TAG ADDRESS (FROM 150)
160
BASE PROTOCOL ADDRESS (FROM 105)
PROTOCOL ENTRY
PROTOCOL ENTRY TABLE
PROTOCOL OFFSET (FROM 150)
PROTOCOL ENTRY
PROTOCOL INCREMENT (FROM 150)
PROTOCOL ENTRY
170
NETWORK ID ENTRY
NETWORK ENTRY TABLE
NETWORK ID ENTRY
BASE NETWORK ADDRESS (FROM 105)
NETWORK OFFSET (FROM 160)
BASE ADDRESS (FROM 105)
FILTER TAG
FILTER TAGS
FILTER TAG
FLTR ADDR (FROM 160)
200
MIRROR TAG
MIRROR TAGS
MIRROR TAG
BASE ADDRESS (FROM 105)
210
MIRROR ADDR (FROM 160)

FIG.5

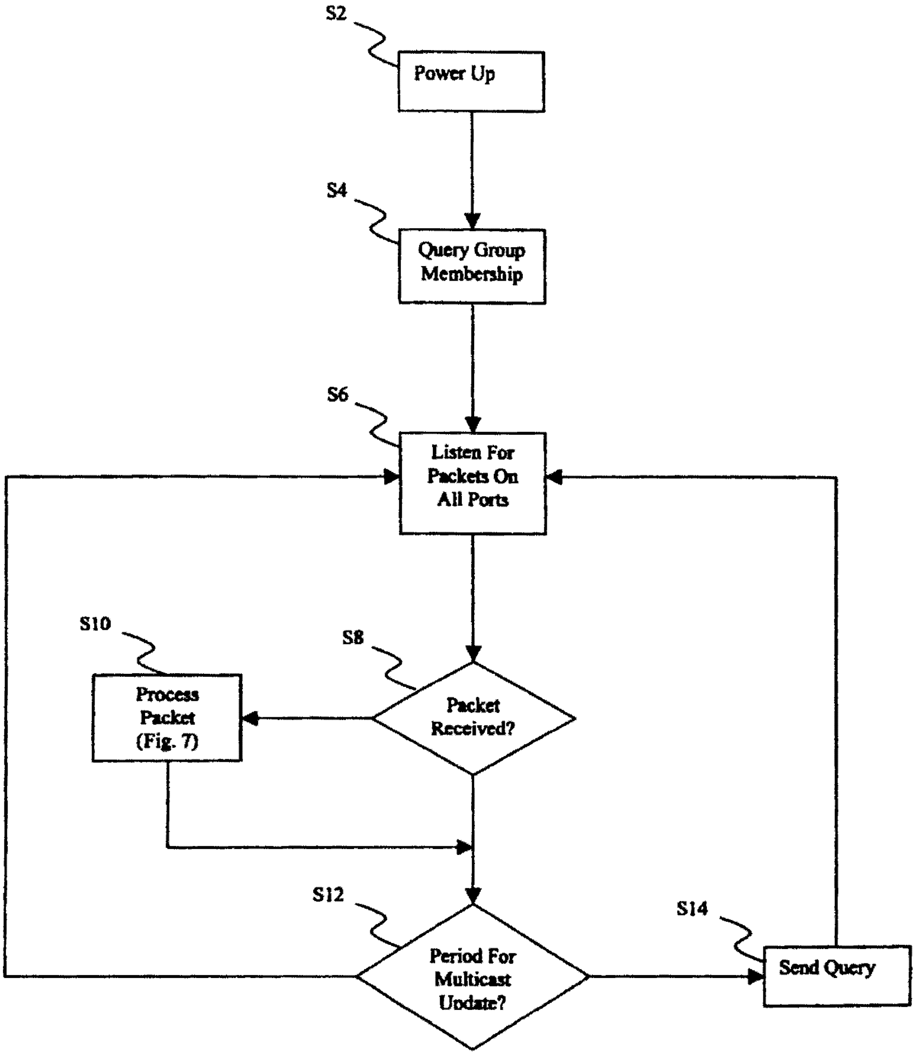

S2
Power Up
S4
Query Group Membership
S6
Listen For Packets On All Ports
S10
S8
Process Packet (Fig. 7)
Packet Received?
S12
S14
Period For Multicast Undate?
Send Query

FIG. 6

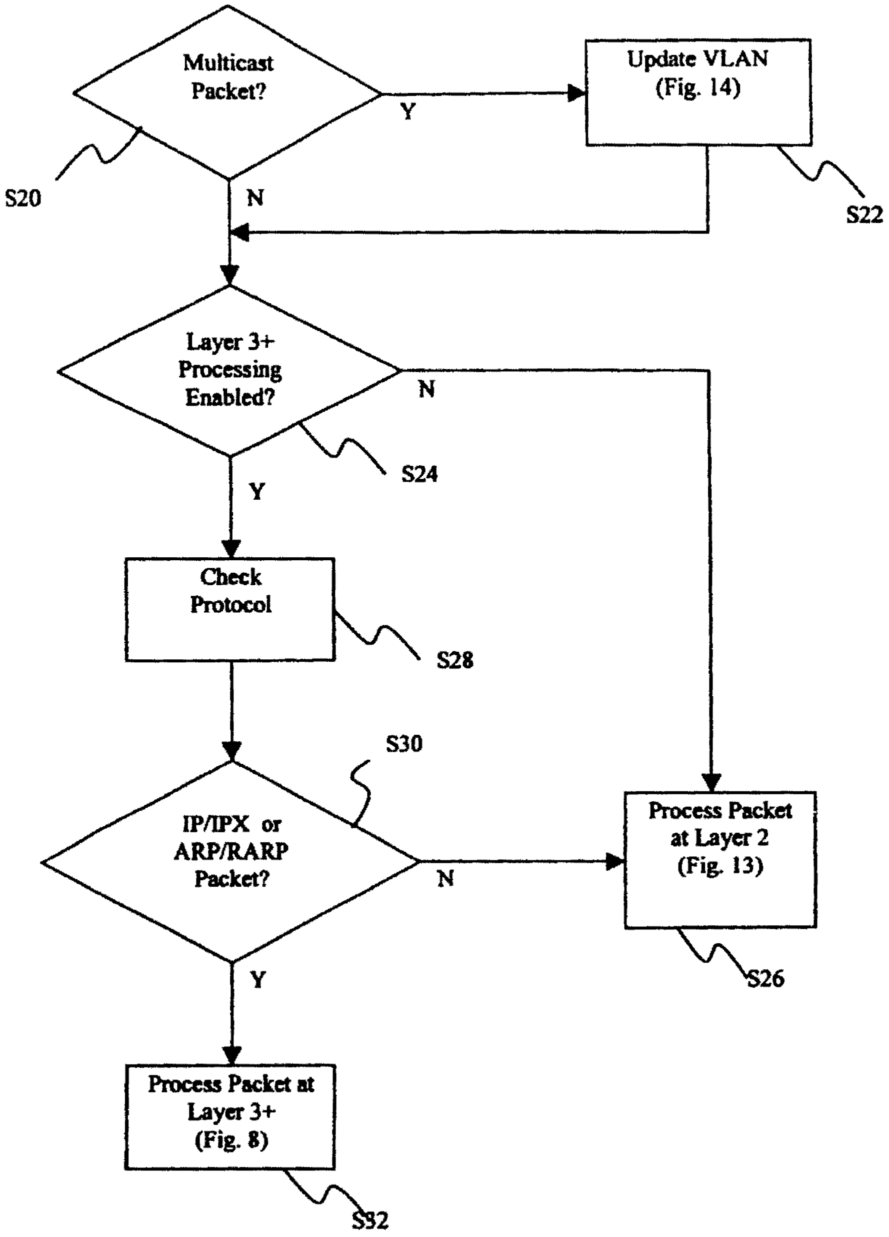

Multicast Packet?
Y
N
S20
Update VLAN (Fig. 14)
S22
Layer 3+ Processing Enabled?
N
Y
S24
Check Protocol
S28
S30
IP/IPX or ARP/RARP Packet?
N
Y
Process Packet at Layer 2 (Fig. 13)
S26
Process Packet at Layer 3+ (Fig. 8)
S32

FIG. 7

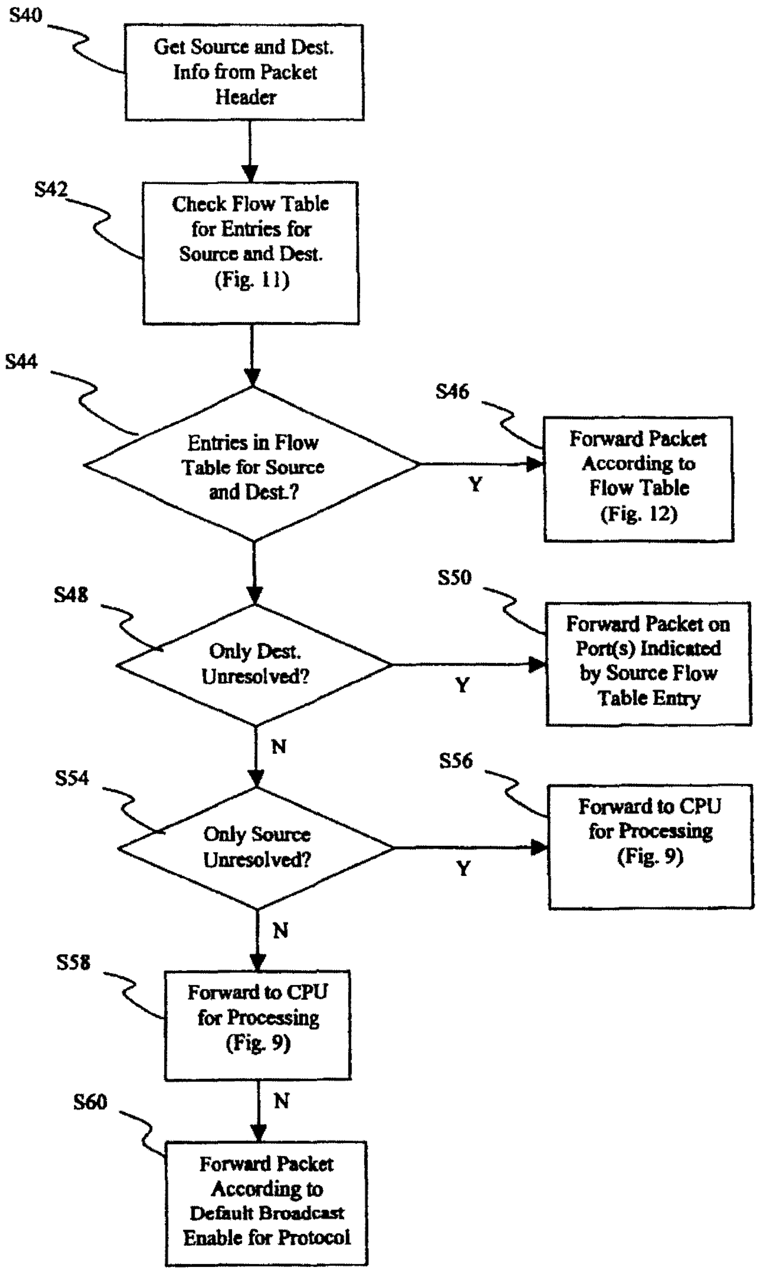

S40
Get Source and Dest. Info from Packet Header
S42
Check Flow Table for Entries for Source and Dest. (Fig. 11)
S44
Entries in Flow Table for Source and Dest.?
Y
S46
Forward Packet According to Flow Table (Fig. 12)
S48
Only Dest. Unresolved?
Y
S50
Forward Packet on Port(s) Indicated by Source Flow Table Entry
N
S54
Only Source Unresolved?
Y
S56
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

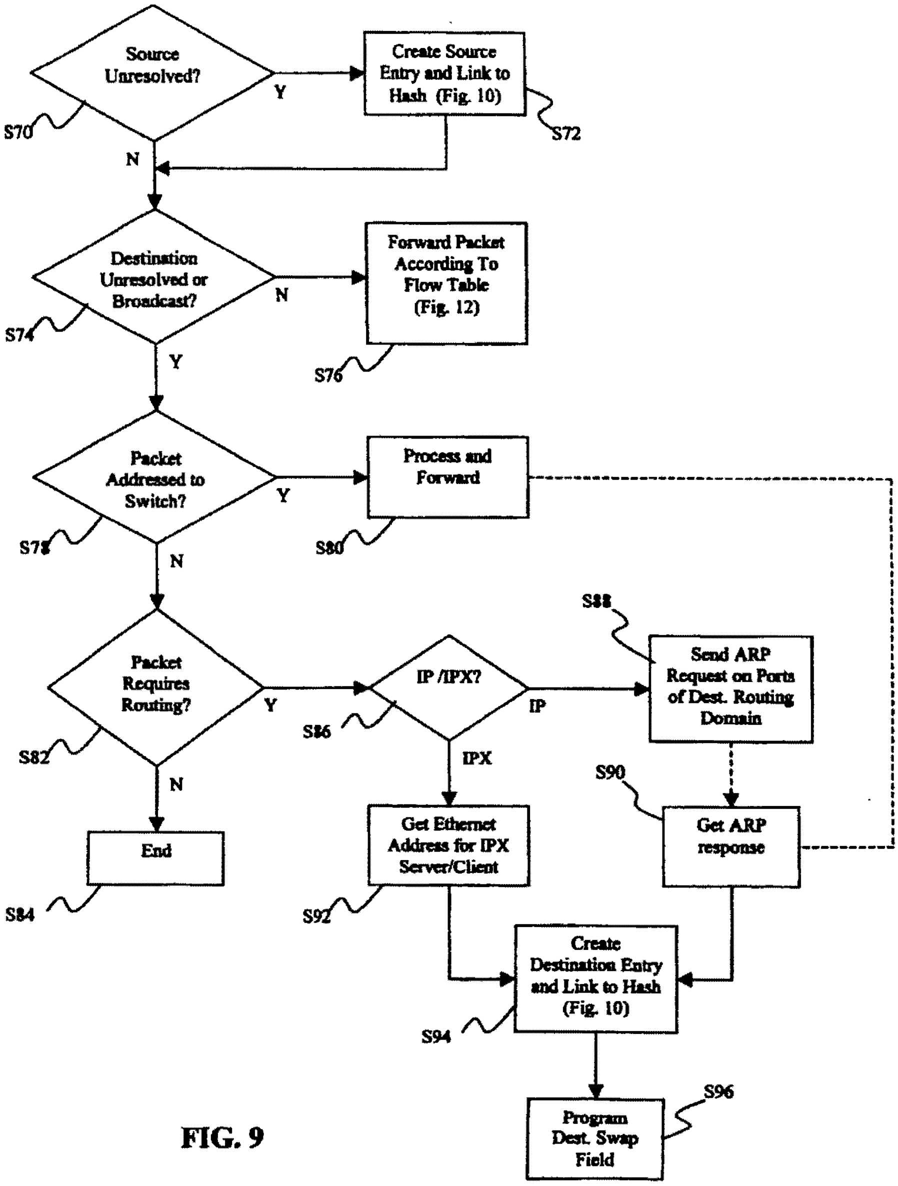

Source Unresolved?
S70
Y
N
Create Source Entry and Link to Hash (Fig. 10)
S72
Destination Unresolved or Broadcast?
S74
N
Y
Forward Packet According To Flow Table (Fig. 12)
S76
Packet Addressed to Switch?
S78
Y
N
Process and Forward
S80
S88
Packet Requires Routing?
S82
Y
N
IP /IPX?
S86
IP
IPX
Send ARP Request on Ports of Dest. Routing Domain
S90
Get ARP response
End
S84
Get Ethernet Address for IPX Server/Client
S92
Create Destination Entry and Link to Hash (Fig. 10)
S94
S96
Program Dest. Swap Field

FIG. 9

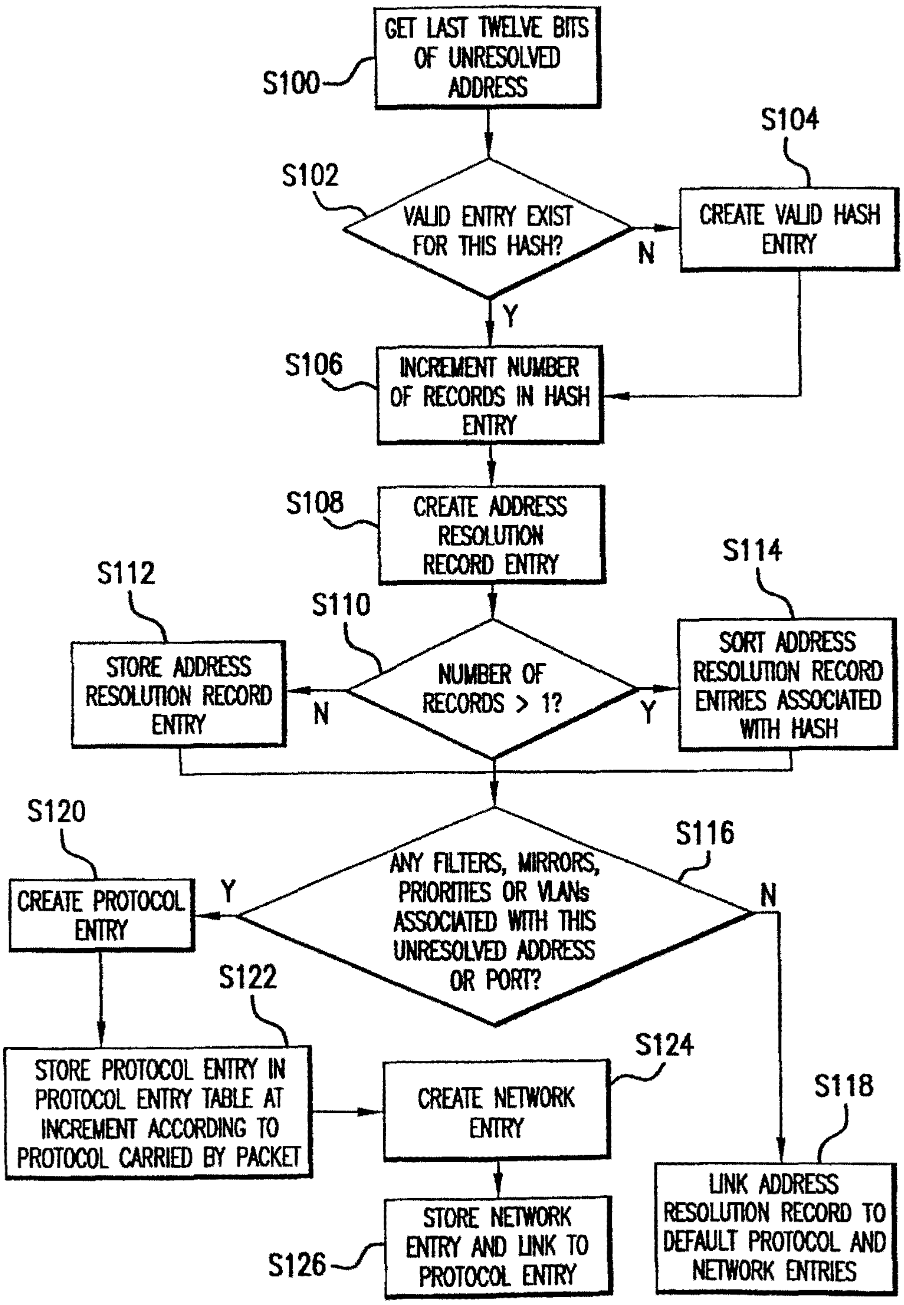

S100
GET LAST TWELVE BITS OF UNRESOLVED ADDRESS
S102
VALID ENTRY EXIST FOR THIS HASH?
N
S104
CREATE VALID HASH ENTRY
Y
S106
INCREMENT NUMBER OF RECORDS IN HASH ENTRY
S108
CREATE ADDRESS RESOLUTION RECORD ENTRY
S110
NUMBER OF RECORDS > 1?
N
S112
STORE ADDRESS RESOLUTION RECORD ENTRY
Y
S114
SORT ADDRESS RESOLUTION RECORD ENTRIES ASSOCIATED WITH HASH
S116
ANY FILTERS, MIRRORS, PRIORITIES OR VLANs ASSOCIATED WITH THIS UNRESOLVED ADDRESS OR PORT?
Y
S120
CREATE PROTOCOL ENTRY
N
S118
LINK ADDRESS RESOLUTION RECORD TO DEFAULT PROTOCOL AND NETWORK ENTRIES
S122
STORE PROTOCOL ENTRY IN PROTOCOL ENTRY TABLE AT INCREMENT ACCORDING TO PROTOCOL CARRIED BY PACKET
S124
CREATE NETWORK ENTRY
S126
STORE NETWORK ENTRY AND LINK TO PROTOCOL ENTRY

FIG.10

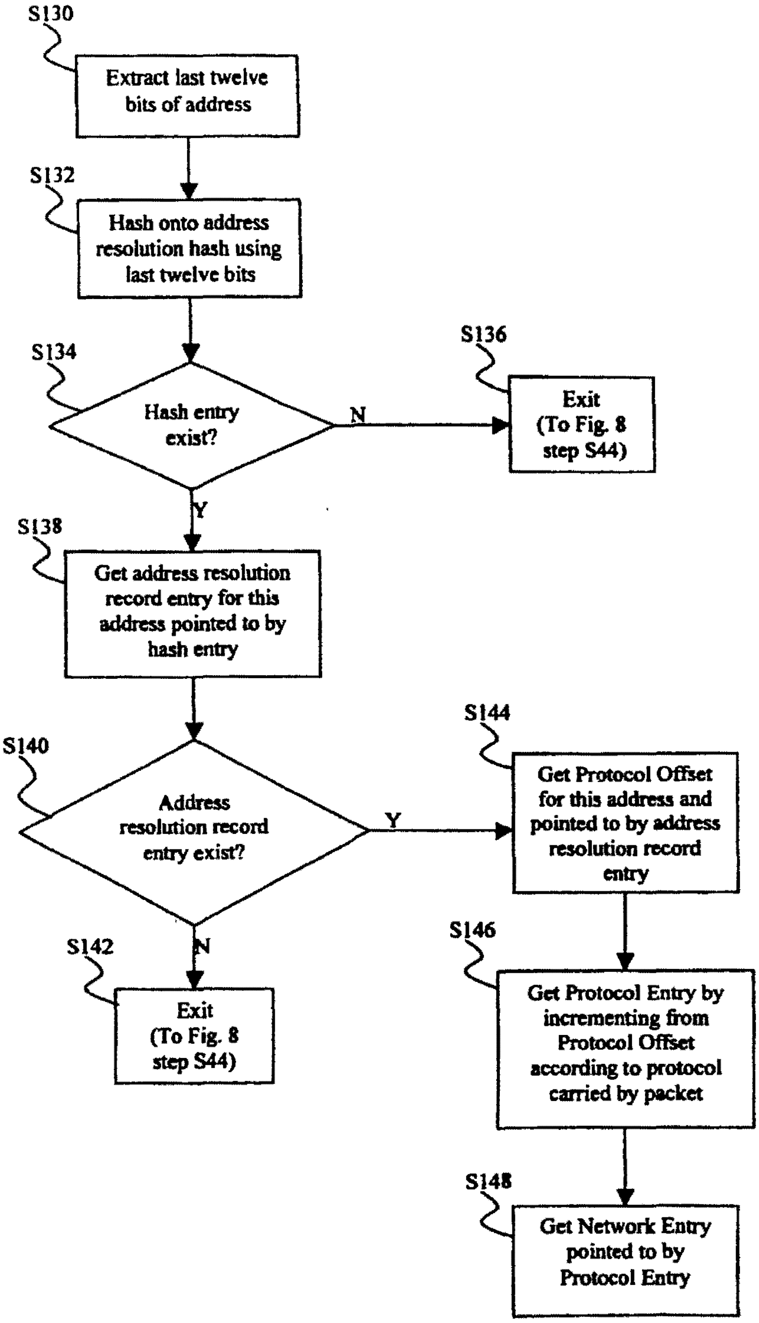
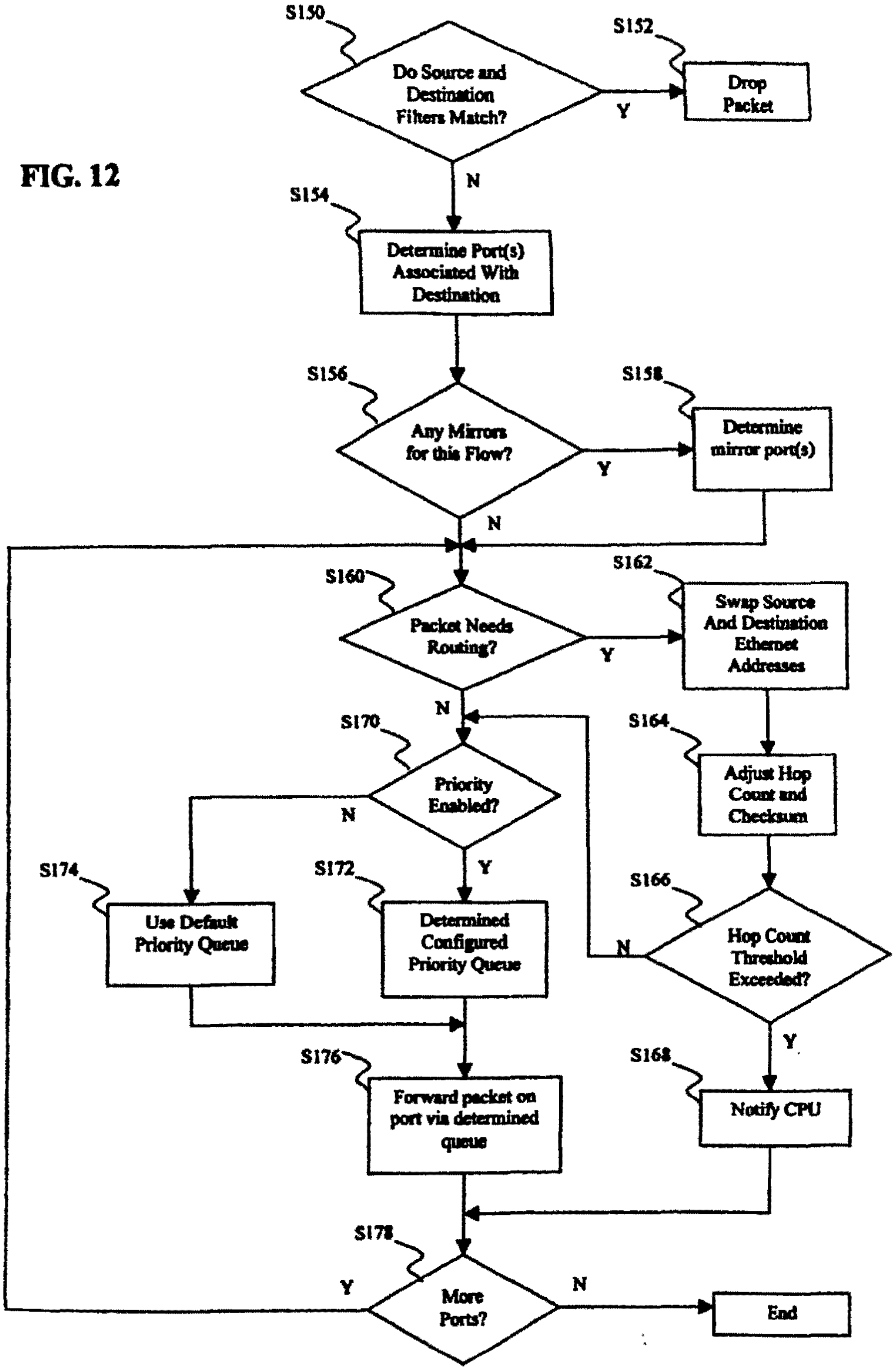

S130
Extract last twelve bits of address
S132
Hash onto address resolution hash using last twelve bits
S134
Hash entry exist?
N
S136
Exit (To Fig. 8 step S44)
Y
S138
Get address resolution record entry for this address pointed to by hash entry
S140
Address resolution record entry exist?
Y
S144
Get Protocol 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?
Y
S152
Drop Packet
N
S154
Determine Port(s) Associated With Destination
S156
Any Mirrors for this Flow?
Y
S158
Determine mirror port(s)
N
S160
Packet Needs Routing?
Y
S162
Swap Source And Destination Ethernet Addresses
N
S170
Priority Enabled?
N
Y
S164
Adjust Hop Count and Checksum
S166
Hop Count Threshold Exceeded?
N
Y
S174
Use Default Priority Queue
S172
Determined Configured Priority Queue
S176
Forward packet on port via determined queue
S168
Notify CPU
S178
More Ports?
Y
N
End

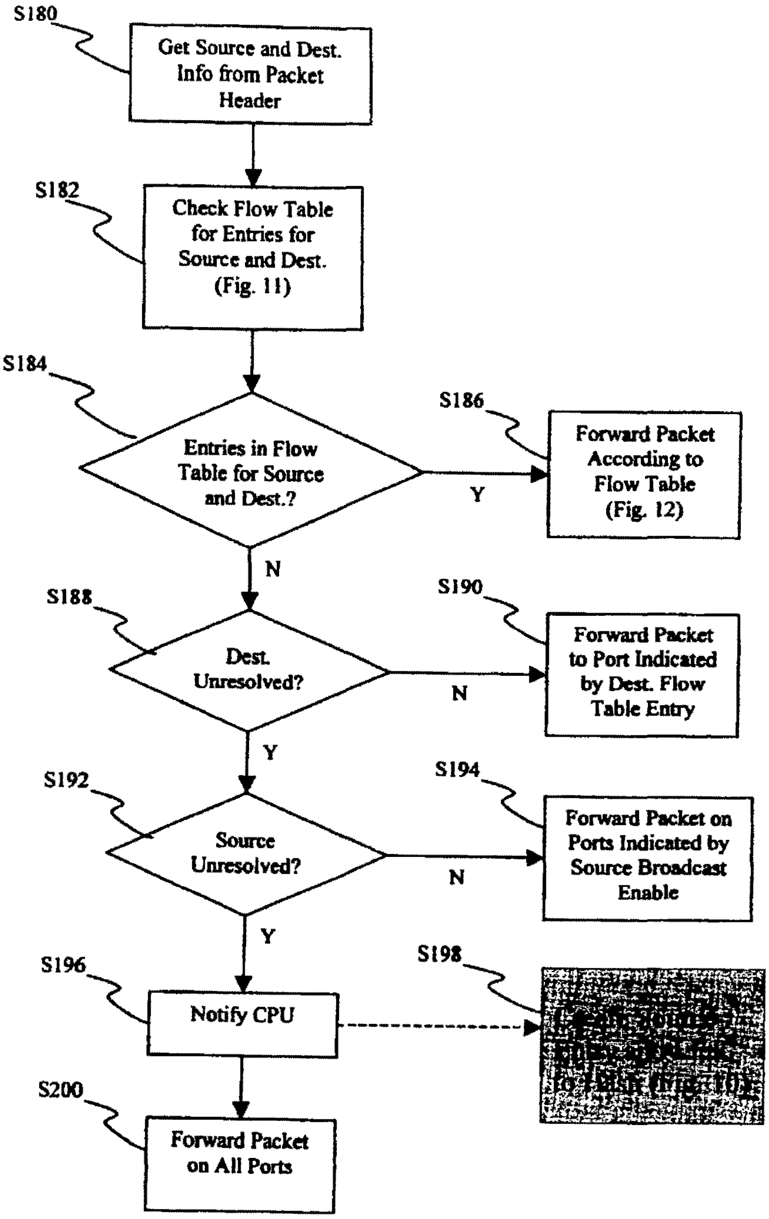

S180
Get Source and Dest. Info from Packet Header
S182
Check Flow Table for Entries for Source and Dest. (Fig. 11)
S184
Entries in Flow Table for Source and Dest.?
Y
S186
Forward Packet According to Flow Table (Fig. 12)
N
S188
Dest. Unresolved?
N
S190
Forward Packet to Port Indicated by Dest. Flow Table Entry
Y
S192
Source Unresolved?
N
S194
Forward Packet on Ports Indicated by Source Broadcast Enable
Y
S196
Notify CPU
S198
(Fig. 10)
S200
Forward Packet on All Ports

FIG. 13

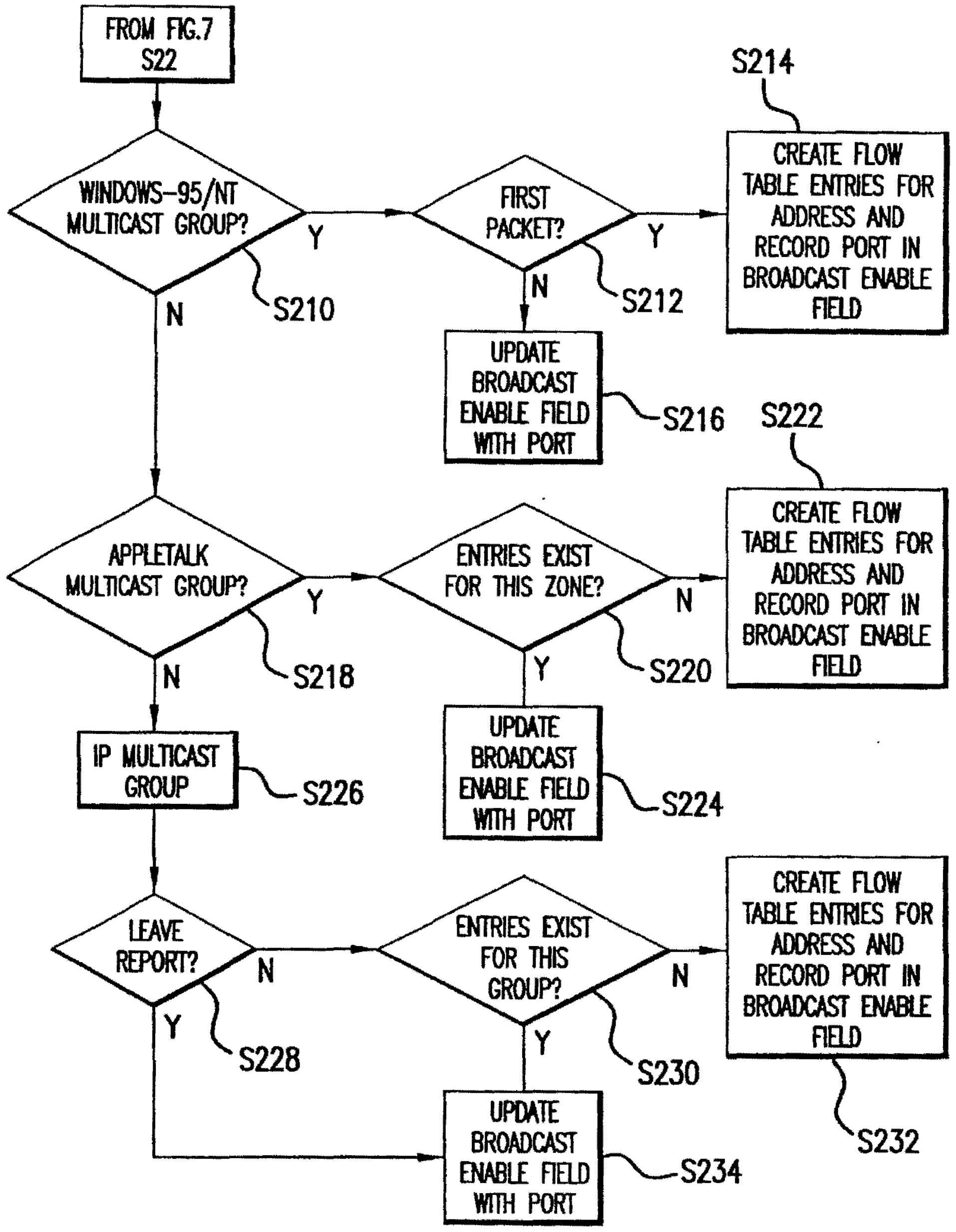

FROM FIG.7 S22
WINDOWS-95/NT MULTICAST GROUP?
S210
Y
N
FIRST PACKET?
S212
Y
N
S214
CREATE FLOW TABLE ENTRIES FOR ADDRESS AND RECORD PORT IN BROADCAST ENABLE FIELD
UPDATE BROADCAST ENABLE FIELD WITH PORT
S216
S222
APPLETALK MULTICAST GROUP?
S218
Y
N
ENTRIES EXIST FOR THIS ZONE?
S220
N
Y
CREATE FLOW TABLE ENTRIES FOR ADDRESS AND RECORD PORT IN BROADCAST ENABLE FIELD
IP MULTICAST GROUP
S226
UPDATE BROADCAST ENABLE FIELD WITH PORT
S224
LEAVE REPORT?
S228
N
Y
ENTRIES EXIST FOR THIS GROUP?
S230
N
Y
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-