H / PKW

OUR DOCKET NO. APPTIT-001

IN THE U.S. PATENT AND TRADEMARK OFFICE

Provisional Application Cover Sheet

ASSISTANT COMMISSIONER FOR PATENTS
Washington, D.C. 20231

Sir:

This is a request for filing a PROVISIONAL APPLICATION under 37 CFR 1.53 (b)(2).

INVENTOR(s)/APPLICANT(s)

Last Name	First Name, MI	Residence (City and Either State or Foreign Country)
Dietz	Russel S.	San Jose, CA
Maixner	Joseph R.	Santa Cruz, CA
Koppenhaver	Andrew A.	Vienna, 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

Dov Rosenfeld
5507 College Avenue
Suite 2
Oakland, CA 94618

ENCLOSED APPLICATION PARTS (check all that apply)

(X)	Specification	*Number of Pages* ..	242
(X)	Drawing(s)	*Number of Pages*	25
( )	Power of Attorney		
(X)	Additional inventors are being named on separately numbered sheets attached hereto.		

METHOD OF PAYMENT

A check in the amount of $ 150.00 to cover the filing fee is enclosed.

If the check is insufficient, please charge any missing fees to Deposit Account 50-0292 .

Respectfully submitted,

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

Date: 6/30/1999

Telephone No.: +1-510-547-3378

'Express Mail" label no. EE516848835US

Date of Deposit: 6/30/1999

I hereby certify that this is being deposited with the United States Postal Service 'Express Mail Post Office to Addressee' service under 37 CFR 1.10 on the date indicated above and is addressed to the Assistant Commissioner for Patents, Washington, D.C. 20231.

By ____________________
Typed Name: Dov Rosenfeld

Provisional 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, Tennessee
Sarkissian	Haig A.	Bexar County, Texas

Please type a plus sign (+) inside this box → +

PTO/SB/16 (2-98)
Approved for use through 01/31/2001. OMB 0651-0037
Patent and Trademark Office; U.S. DEPARTMENT OF COMMERCE

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

PROVISIONAL APPLICATION FOR PATENT COVER SHEET

This is a request for filing a PROVISIONAL APPLICATION FOR PATENT under 37 CFR 1.53 (c).

INVENTOR(S)

Given Name (first and middle [if any])	Family Name or Surname	Residence (City and either State or Foreign Country)
Russel S.	Dietz	San Jose, CA
Joseph R.	Maixner	Santa Cruz, CA
Andrew A.	Koppenhaver	Vienna, VA

☒ *Additional inventors are being named on the* 1 *separately numbered sheets attached hereto*

TITLE OF THE INVENTION (280 characters max)

METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK

CORRESPONDENCE ADDRESS

Direct all correspondence to:

☒ Customer Number: 21921
Type Customer Number here

Place Customer Number Bar Code Label here

OR

X Firm *or* Individual Name	Dov Rosenfeld				
Address	5507 College Avenue, Suite 2				
Address					
City	Oakland	State	CA	ZIP	94618
Country	USA	Telephone	+1-510-547-3378	Fax	+1-510-653-7992

ENCLOSED APPLICATION PARTS *(check all that apply)*

☒ Specification *Number of Pages* 242
☒ Drawing(s) *Number of Sheets* 25
□ Small Entity Statement
☒ Other (specify) check, postcard

METHOD OF PAYMENT OF FILING FEES FOR THIS PROVISIONAL APPLICATION FOR PATENT *(check one)*

☒ A check or money order is enclosed to cover the filing fees

☒ The Commissioner is hereby authorized to charge any mising fees or credit any overpayment to Deposit Account Number: 50-0292

FILING FEE AMOUNT ($): $150.

The invention was made by an agency of the United States Government or under a contract with an agency of the United States Government.

X No.
~~Yes~~, the name of the U.S. Government agency and the Government contract number are: ______

Respectfully submitted,

Date: June 30, 1999

SIGNATURE ______

REGISTRATION NO. *(if appropriate)*: 38,687

TYPED or PRINTED NAME Dov Rosenfeld

TELEPHONE +1-510-547-3378

Docket Number: APPTIT-001

USE ONLY FOR FILING A PROVISIONAL APPLICATION FOR PATENT

This collection of information is required by 37 CFR 1.51. The information is used by the public to file (and by the PTO to process) a provisional application. Confidentiality is governed by 35 U.S.C. 122 and 37 CFR 1.14. This collection is estimated to take 8 hours to complete, including gathering, preparing, and submitting the complete provisional application to the PTO. Time will vary depending upon the individual case. Any comments on the amount of time you require to complete this form and/or suggestions for reducing this burden, should be sent to the Chief Information Officer, U.S. Patent and Trademark Office, U.S. Department of Commerce, Washington, D.C., 20231. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. SEND TO: Box provisional Application, Assistant Commissioner for Patents, Washington, D.C., 20231.

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: EE516848835US in an envelope addressed to Box Provisional Application, Assistant Commissioner for Patents, Washington, D.C. 20231 on.

Date: June 30, 1999

Signed: ______

Name: Dov Rosenfeld, Reg. No. 38,687

PROVISIONAL APPLICATION COVER SHEET
Additional Page

PTO/SB/16 (2-98)
Approved for use through 01/31/2001. OMB 0651-0037
Patent and Trademark Office; U.S. DEPARTMENT OF COMMERCE

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

Docket Number	APPTITUDE-001	Type a plus sign(+) inside this box →	+

INVENTOR(S) /APPLICANT(S)

Given Name (first and middle[if any])	Family or Surname	Residence (City and either State or Foreign Country)
William H.	Bares	Germantown, Tennessee
Haig A.	Sarkissian	Bexar County, Texas

Number 1 of 1 additional pages

Our Ref./Docket No: **APPTIT-001**

METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK

Inventor(s):

DIETZ, Russel S.
San Jose, CA

MAIXNER, Joseph R.
Santa Cruz, CA

KOPPENHAVER, Andrew A.
Vienna, VA

BARES, William H.
Germantown, Tennessee

SARKISSIAN, Haig A.
Bexar County, Texas

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: EE516848835US in an envelope addressed to Box Provisional Application, Assistant Commissioner for Patents, Washington, D.C. 20231 on.

Date: June 30, 1999

Signed:

Name: Dov Rosenfeld, Reg. No. 38,687

METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK

FIELD OF INVENTION

The present invention relates to computer networks, and more specifically to the real-time elucidation of packets communicated within a data network, for example, between a client and a server, the elucidation including classification by protocol and application program.

COPYRIGHT NOTICE

A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document or the patent disclosure, as it appears in the Patent and Trademark Office patent file or records, but otherwise reserves all copyright rights whatsoever.

BACKGROUND TO THE PRESENT INVENTION

There has long been a need for network activity monitors. The popularity of networks used as a collection of clients obtaining services from one or more servers on the network, and especially the recent popularity of the Internet and other internets (an "internet" is a plurality of interconnected networks to form a larger single network) has made it increasingly important to be able to monitor the use of services offered on the network and rate those services accordingly. For example, objective information such as which services (*i.e.*, application programs) are being used, who is using them, how often they have been accessed, when they are being accessed, how long accesses have been, and so forth. Additionally, remote access by selected users to generate reports in real time on network use is needed. Finally, an network monitor which can provide alarms in real-time to notify selected users of network or site problems is needed.

Selected network activities may be retrospectively analyzed by reviewing log files. Log files 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. However, there exist several problems with this method. First, log file information does not provide any real-time usage map. Secondly, log file mining does not supply complete information. The method relies on logs maintained by numerous network devices and servers and the information in them must be subjected to refining and correlation. Sometimes, for example in the case of information about NetMeeting® (Microsoft Corporation, Redmond, Washington) sessions where two computers connect directly on the network and the data is never seen by a server or gateway, information is simply not available to any server or gateway, in order to make a log file entry. Creating log files requires data logging features of network elements to be enabled, placing a substantial load on the device performance, thus reducing network performance. Log-files also require a substantial amount of maintenance (there is no standard way of storing for log files), and grow rapidly.

NetFlow® (Cisco Systems, Inc., San Jose, California), RMON2, and other network monitor devices are available for the real-time monitoring of networks, but these lack visibility into application content and context and are therefore typically limited to providing network layer level information.

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

What is needed, therefore, is a network monitor that makes it possible to continuously analyze all user sessions on a heavily trafficked network, remotely and in a noninvasive manner. 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 all 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 level delivered, duration, time of day, data requested, and so forth). The network monitor also should not be reliant upon server resident information such as log files. It should thus allows a user such as a network administrator or an Internet service provider (ISP) the means to objectively measure and analyze network activity, customize the type of data that collected and analyzed, undertake real time analysis and receive timely notification of network problems.

Some prior art packet monitors classify packets into connection flows. The term connection flow is sometimes 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 example, 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.

Some conversational flows involve more than one connection, and some even involve more than one exchange of packets between a client and a server. This is a particularly true when using client/server protocols, such as RPC, DCOMP, and SAP, that enable a service to be set up or defined prior to any use of that service. For example, SAP (Service Advertising Protocol) is a NetWare (Novell Systems, Provo, Utah) protocol used to identify the services and addresses of servers attached to a network. In a first exchange, a client sends a SAP request to a server, for example, for print service. The server sends a SAP reply that identifies a particular address, for example, SAP #5, as the print service on that server. Such may be responses used to update a table, for example in a router, known as the Server Information Table. A client who has inadvertently seen this reply or who has access to the table (via the router that has the Server Information Table, for example) would know that SAP #5 for such this server is a print service. Therefore, in order to print data on the server, such a client does not need to make the request for a print service, but simply to send data to be printed specifying SAP #5. This sending of data to be printed again involves an exchange of data between a client and a server, disjoint from the previous exchange which was with a different client setting up that SAP #5 is a print service on this server is a second connection. It is desirable for a network packet monitor to be able to "virtually concatenate" the first exchange that defines SAP #5 as the print service on the server with the second exchange that uses the print service. The two packet exchanges would then be correctly identified as being part of the same flow if the clients were the same. They would even be recognized if the clients were not the same. One feature of the invention is to so correctly identify the second exchange as being associated with a print service on the server.

Other protocols that are similar in that they may lead to disjointed conversational flows include DCOM (Distributed Component Object Model), formerly called Network OLE (Microsoft Corporation, Redmond, Washington), which is Microsoft's technology for distributed objects, RPC (Remote Procedure Call), 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 defines the remote procedure call which allows those objects to be run remotely over the network. DCOM Microsoft's counterpart to CORBA, a standard from the Object Management Group (OMG) for communicating between distributed objects (objects are self-contained software modules). CORBA provides a way to execute programs (objects) written in different programming languages running on different platforms no matter where they reside in the network.

Prior art network monitors do not presently have the ability to recognize such 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." 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,"

SUMMARY

One aspect of the present invention is providing a network monitor that can recognize and classify at all protocol layer levels conversational flows that pass in either direction at a point in a network.

Another aspect of the present invention is providing a network monitor that can recognize and classify at all packets that are exchanges between a client and a server into respective client/server applications.

Another aspect of the present invention is providing a network monitor that can determine the connection and flow progress between clients and servers by the individual packets exchanged over a network.

Another aspect of the present invention is providing a network monitor that can determine the connection and flow progress between clients and servers by the individual packets exchanged over a network.

Another aspect of the present invention is providing a network monitor that can be used to help tune the performance of a network according to the current mix of client/server applications needing network resources.

A still further aspect of the present invention is providing a network monitor that can maintain statistics relevant to the mix of client/server applications using network resources.

Another aspect of the present invention is providing a network monitor that reports on the occurrences of specific sequences of packets used by particular applications for client/server network conversations.

Another aspect of an embodiment of the invention is properly analyzing each of the packets exchanged between a client and a server and maintain information relevant to the current state of each of these conversations.

Another aspect of an embodiment of the invention is a flexible processing system that can be tailored or adapted as new application entered the client/server market.

Another feature of an embodiment of the invention is maintaining statistics relevant to the conversations in a client/server network as their classified by an individual application.

Another feature of an embodiment of the invention is 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 conversation.

Additional features and advantages of the invention will be clear from the description which follows.

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

More aspects and advantages of the present invention are set forth in part in a description that follows, and in part are obvious from a description, or may be learned by practice of the present invention. The objects and advantages of the present invention may be realized by the elements and combinations particularly pointed out in the appended claims.

Embodiments of the present invention overcome the problems and disadvantages of prior art and achieves the objects of the present invention by analyzing each of the packets passing through any point in the network in either direction, extracting a signature for th conversation which may then be used for identifying the conversational flows. Another feature of the invention is forming and remembering the state of any conversational flow, which is determined by the relationship between individual packets of the conversational flow and the entire conversational flow over the network. By so remembering the state of a flow, a feature of the invention is to the determine the context of the conversational flow, including the application program it relates to and such parameters as the time, length of conversation, data rate, etc.

A monitor embodiments of the present invention determine the identities of any and all application programs executing on the network by evaluating each and every packet conversing between clients and servers. In one embodiment, the monitor comprises parser that includes a packet parsing module, and an identifying information extracting module to form a signature from a packet received by the parser. The monitor further comprises an analyzer which receives the signature from the parser and comprises a flow lookup/update engine, a flow insertion and deletion engine, a state processor, a cache and a unified memory controller. Each of these analyzer elements work in parallel to create and update flow recognition signatures. The monitor is scalable to handle more protocols and applications. As a flow signature is examined by the monitor, the lookup engine attempts to find the signature in a flow-entry database. If the first part of the flow matches an already identified signature that resides in the cache, the lookup engine retrieves the flow from the cache, else if the first part of the flow matches an already identified signature that is not in the cache, it retrieves the flow from a flow database. The flow entry for previously encountered flows preferably includes state information, and this state information is used in the state processor to execute any operations defined for the state, and to determine the next state. The flow entry is updated by adding values to counters in the flow-entry database entry. If a flow does not exist, the protocol is identified and the state processor starts executing whatever operations are defined for the initial state. The state processor sends a flow signature to the flow insertion and deletion engine that adds the flow to the database as a new item. The state processor updates the flow based on the current state and the flow-signature information. The state processor processes single and multi packet protocol recognition. It may have to search through a series of possible states to determine the flow's actual state. The result of this processing is a consolidated flow entry. This enables the monitor to correctly determine disjointed flows. For example, a PointCast session (PointCast, Inc., Cupertino, CA) will open multiple conversations packet-by-packet that might look like separate flows to prior art monitors. However, each of these connections is merely a sub-flow under the PointCast master flow, so a single flow that consolidates all of the information for the flow is desired. The analyzer is able to so consolidate individual connections since the state of the overall flow is maintained by the monitor. The unified memory controller can be setup to work with various memory device types and controls an SRAM tag memory for shadowing of flow entries. The cache is used to optimize memory bandwidth. On a typical network, the packets will have a certain amount of congruity so a cache architecture can have a relatively high hit rate.

Invention Overview

A real-time traffic classification system, which has the ability to derive the application or service being used over the data communications network, comprises the following modules found in Fig. 10 and Fig. 11. A pattern analysis and recognition engine 1006, a pattern extraction engine 1007, a unique signature generation engine (elements of 1007), a signature matching engine 1107, a protocol and layer identification engine (elements of 1107), the state oriented processing engine 1108, the derived set of rules 1109 and a set of active and in process signatures and records (Fig. 3, 319). The pattern analysis and recognition engine 1006 is used to derive and determine the type of network packets that exist on the network. Once a pattern match has occurred, the pattern is passed on to the pattern extraction engine 1007 for the generation of a signature. The pattern extraction engine extracts components from each of the packets required in the formation of unique signature. Once these elements have been extracted from the packets, the information is passed on to the unique signature generation engine 1007. The signature generation engine then sequences and formats the extracted information into a unique signature that will be used to identify other packets within the same conversation on the network. The contents of the unique signature are passed on to a matching engine 1107, which looks up the signature from the database of currently known conversations or flows. If the signature-matching engine determines an existing conversation, information is passed on to update the contents of the record in the database and processing is terminated for this packets 1112. If either no match is found or a match is found with remaining state or rules to be processed, the protocol layer identification engine 1107 is initiated to derive the layering involved in the packets. With the layering information interpreted and understood, the system begins the process of protocol application identification. This process is initiated by the state oriented processing engine 1108. This processing engine uses a set of derive states or rules to apply to each of the individual packets 1109 and signature these to determine the extent of the application used in the conversation. When the processing engine determines the application component of a conversation, that information is updated in the conversation record for this particular flow. In this way, multiple packets from a conversation can be used to derive the application component of a particular set of packets exchanged between nodes in a network. In addition to maintaining the actual application information relative to conversation in a network, the system maintains real-time statistics relevant to these applications.

Packet Parsing Sub-System

The packet parsing system consists of two main sub engines. These engines are the pattern analysis and recognition engine (PAR) and the field extraction engine (FEE).

The pattern analysis and recognition engine interprets each packet that is seen entering the system. As individual fields from each packet enter the system the field contents are analyzed for specific patterns. As more fields under the system fewer pattern to remain to be analyzed and through the process of elimination particular pattern for packet is found.

The patterns for this engine are stored in a special pattern database. The pattern database contains a sparsely populated three-dimensional array of patterns and links to additional those beyond the patterns that are being currently analyzed. Because this is a sparsely populated three-dimensional array, as patterns enter the system the depth of nodes is eliminated rapidly. Once a node does not contain a link to a deeper level, the pattern matching is complete. At that point, the field extraction engine instruction found at that node in the array is sent to the field extraction engine with this packet.

The field extraction engine takes the packet contents and the extraction instructions from the pattern analysis and recognition engine to continue processing the packet. Each of the elements found within the instructions of the field extraction engine component are removed from the packet and inserted into a buffer for signature generation. Once all the operations requested of the field extraction engine are completed for this packet, the signature is set as complete, and a hash key is generated to identify this signature.

Packet Analysis Sub-System.

When the parsing system has successfully completed the task of deriving, determining and extracting the required information, the remaining pieces of the packet and the generated signature for the packet are passed to the packet analysis system.

All of these elements from the packets are formulated into a flow signature and stored in the unified flow key buffer of the packet analysis system. This buffer is designed to maintain and hold multiple flow signature is from the packets being analyzed in a client/server network. While the flow signature of a packet exists in the unified flow key buffer, several operations are performed to further derive the application content of the packet involved in the client/server conversation.

The first step in the process of packet analysis is to look up the instance in the current database of known flow signature ease for packets. The look up/update engine accomplishes this task. This engine uses the hash key and remaining fields of the flow signature from the packet to determine if this packet is flow record exists in the flow database of the packet analysis system. Once the look up processing has been completed the flag stating whether it was found or is new, will be set within the unified flow key buffer structure for this packet flow signature.

After the packet flow signature has been looked up and contents of the current flow signature database tree, the state processor will begin analyzing the packet payload to further derive the application component of this packet. The exact operation of the state processor and functions performed by at will very depending on the current packet seek once in the stream of a conversation. The state processor will performed the next logical operation that was stored from the previous packet seen with this same flow signature. If any processing is required on this packet, the state processor will execute state processor instructions from the state processor instruction database until they're either are no more left for this packet or the instruction signifies and processing for this packet.

Since the seek once love packet exchanges between client and server is crucial in deriving the application component of a conversation, the state processor functions are required to be variable and program. Each new application that exists on the network may have different characteristics for identifying the components within packets. The state processor functions take into consideration this variable method of communicating in a client/server network. The actual operations performed by the state processor are described in the section under state processor instruction database operations.

If during the look up process for this particular packet flow signature, the flow is required to be inserted into the active database, the flow insertion and deletion engine is initiated. This engine operates independently from the other two engines within the analysis system. The look up update engine will determine whether the flow insertion and deletion engine is required to operate for particular packet flow signature.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention is more fully understood from the detailed preferred embodiments of the present invention, and should not be taken to limit the present invention to any specific embodiment because such are provided only for explanation and better understanding. 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 that 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 some types of packet formats of the packets that might be exchanged in exchanged in starting an illustrative example 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 are also illustrated and represent the one or many flow signatures that can be generated and used in the process of analyzing packets and 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 parsing system process embodiment of the present invention that can form part of the parser in the inventive packet monitor;

Fig. 6 is a flowchart of a packet element extraction process embodiment of the present invention that can form part of the parser in the inventive packet monitor;

Fig. 7 is a flowchart of a flow-signature building process embodiment of the present invention that can form part of the parser in the inventive packet monitor;

Fig. 8 is a flowchart of a monitor lookup and update process embodiment of the present invention that can form part of the analyzer in 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 sub-system including the pattern recognizer and extractor that can form part of the parser module in the inventive packet monitor;

Fig. 11 is a functional block diagram of a hardware analyzer including a state processor can form part 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 the inventive packet monitor;

Fig. 13 is a flowchart of a state processor embodiment of the present invention that can form part of the analyzer in 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 host processor.;

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

Fig. 17 is an example of the header 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. 18 is an example of the IP header of in the Ethernet packet shown in Fig. 17 and some of the elements that may be extracted to form a signature according to one aspect of the invention;

Fig. 19 is functional block diagram of the Unified Flow Key Buffer component of the Analyzer sub-system of Fig. 11;

Fig. 20 is top level block diagram of the state processor component of the Analyzer sub-system of Fig. 11;

Fig. 21 is data flow block diagram of the state processor component of the Analyzer sub-system of Fig. 11;

Fig. 22 is top level block diagram of the search engine component of the Analyzer sub-system of Fig. 11;

Fig. 23 is data flow block diagram of the search engine component of the Analyzer sub-system of Fig. 11;

Fig. 24 is a flow chart of the process of compiling high level language files according to an aspect of the invention; and

Fig. 25 shows various PDL file modules to be compiled together by the compiling process illustrated in Fig. 24 as an example, in accordance with a compiling aspect 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 the signal names are not needed to understand the operation and practice of the invention. Also, the term MeterFlow is to be understood to mean the preferred embodiment of the invention.

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 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 by its connection point 121 and, according to one aspect of the invention, can elucidate what application programs are associated with each packet passing by connection point 121.

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 network 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, and 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, called a conversational flow, 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 as a protocol stack. The ISO (International Standardization Organization) has defined a general model which provides a framework for design of communication protocol layers. This model 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 which is similar to but does not exactly conform to the ISO mode. A protocol in a certain layer may not be aware 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).

Every packet passing the connection point 121 is looked at by the monitor 108 for analysis. But not every packet carries the same information useful for recognizing all levels of the protocol, up to level 7, recognizing its associated application program. 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. But if the particular application program will always follow this up with the sending of a type-B packet and the other application programs do not, then in order to recognize packets of that application's conversational flow, the monitor can engage itself in a search for packets that match the type-B packet to associate with the type-A packet. If such is spotted, 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 conversations 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 if 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. In such case, 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 must therefore mask out all the unimportant parts of each packet that will not contribute to its classification. But 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 type of packet and eventually of the associated application programs by the packets that their executions produce is a multistep process within the monitor 108. At a first level, several application programs will all produce a first kind of packet, for example. A first "signature" is produced that will allow monitor 108 to efficiently identify any packets that belong to the same conversational 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. They 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 follow-on packet of a second packet type but which may belong to the conversational flow is recognized, using the signature, then at such a second level, 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 application classification, all application programs that are not in the set of those that lead to such a second packet type following the first packet type may be excluded in the process of classifying the conversational flow that includes these two packets. A signature is produced that allows recognition of any future packets that may follow on in the conversational flow according to the known patterns for the protocol and for the possible applications. It may be that the application is now recognized, or recognition may need to proceed to a third level of analysis of those packets that are selected using the second level signature. Therefore, for each packet, the monitor parses the packet, generates a signature from the packet to determine if this signature identified a previously encountered conversational flow or shall be used for to recognize future packets belonging to the same conversational flow, and in real time, the packet is further analyzed in the context of the sequence of packets so far encountered (the state) and the possible future sequences such a past sequence may generate in conversational flows associated with different applications until the applications are identified. The 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 particular application programs executing 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 recognition of the application program has been accomplished by the monitor 108, eavesdropping can commence.

Fig. 3 represents 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. A parser sub-system 301 examines the packets using pattern recognition process 304 and recognizes pattern information in the packet 302. A process 306 in parser sub-system 301 extracts signature information from the packet 302. Both the pattern models for parsing and the related extraction masks are supplied from a parsing pattern structures and extraction operations database 308 filled by a compiler and optimizer 310. In an alternate embodiment, the contents of database 308 may be otherwise generated. A process 312 in parser sub-system 301 builds a unique conversational flow signature from the extracted information. The signature is then analyzed in analyzer sub-system 303. In analyzer sub-system 303, a process 314 uses the newly built conversational flow signature in a lookup of preexisting conversational flow signatures in which the associated state of any previously encountered flow is stored. Note that the lookup may be from a flow signature buffer (called flow key buffer) or from a cache or from the externally kept database of known flows 324. A process 316 steers control to a process 318 if the conversational flow signature is a new one, after which the protocol is determined from the extracted information and with reference to a database 326 of state patterns and processes. Otherwise, a process 320 determines, from the state in the looked-up conversational flow signature information, if more classification by state processing of the conversational flow signature is necessary. If no, a process 322 updates a flow-entry database 324 with the new conversational flow signature, and otherwise, and also in the case of the new flow signature, a state processing process 328 is commenced.

The flow-entry database 324 stores flow entries with a 128-byte pattern for each, which includes for updated flows the unique flow-signature, state information and extracted information from the packet. Each entry completely describes a flow. Such database is organized into buckets that each contain a number, denoted N, of flow entries, with N being 4 in the preferred embodiment. Buckets are accessed via a hash data value created by the parser subsystem, *i.e.,* by the parser/extraction engine/key builder part of the system based on information in a packet. Such hash spreads the flows across the database and is preferably based on a hashing algorithm. Such technique allows 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, for 128K flow entries, 16M bytes are required. Using a 16-bit hash gives two flow entries per bucket. Such has been empirically shown to be more than adequate for the vast majority of cases.

Still in the analyzer sub-system 303, the process 318 identifies the protocol in use that produced the original packet from the data extracted by the identifying information extractor 306 and using a collection of state patterns and processes 326.

In both the case that the flow is a new one, and that it is one previously encountered requiring further analysis, a 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 network traffic monitor 300 provides for single packet protocol recognition of flows, and, by maintaining the state of the flows and also by knowing that for some types of flows, new flows may be set up using the information from previously encountered flows that were it not for the system knowing that this at first apparently unassociated flows may be used to characterize other flows, the network traffic monitor 300 allows for multiple packet protocol recognition of flows even with disjointed sub-flows that occur in serve announcement type flows. In the case of a new flow, it further provides for the new flow as identified by a flow signature to be sent to the flow insertion and deletion engine to add the flow to the database as a new item. Again in the case of a new flow, for some types of protocols, the new flow is associated with previously encountered flows. The analyzer sub-system 303 determines current state of a flow, and further providing for a consolidated flow entry in the flow-entry database. 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. A state transition rule is a rule typically containing a test followed by the next-state to proceed to if the test is true, and 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 until the test is true, or there are no more tests to perform. The state processor starts the process by using the last protocol recognized by the parser as an offset into a jump table. The jump table finds the state processor instructions to use for that protocol in the state patterns and processes database 328. Most instructions test something in a unified flow signature buffer, or the flow entry in the database of known flows 324 if it exists. The state processor may have to test bits, do comparisons, add or subtract to perform the test.

In the preferred embodiment, a cache is used which intercepts all requests to the flow database 324 to speed access.

In state processing, a process 330 decides if more operations are needed in order to find a unique flow signature for the application. If not, a process 332 decides if there are 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, a process 334 finalizes the classification of the conversational flow.

In the preferred embodiment, the database of parsing patterns and extraction operations 308 and the database of state patterns and state processes are generated by an optimizing compiler 310 from protocol description language 336 and a selection of packet layers 338.

Referring again to the compiler 310 (Fig. 3), the compilation process includes creating the parsing patterns and operations needed in pattern recognition process 304 and the extraction operations needed in process 306, and information for the analyzer subsystem 303 on identifying protocols and what state transitions and processes to carry out when a packet is determined to be in a particular state, in order maintain state to enable analyzing flows beyond single-packet-type conversational flows. The compiler and optimizer 310 uses as input files (336 and 338) that describe each of the particular areas of a packet that require decoding for each protocol and application. Files 336 are the protocol description files written in a high-level protocol description language (PDL) by a user who is familiar with protocols and packet structure. File 338 includes commands to indicate which of the protocols are to be converted into databases 308 and 326.

By maintaining the datagram layer selections and the protocols in a high level language, the user, for example the network administrator or the Internet service provider, can include new protocols or new application programs as they become known, or modify existing protocols and applications as their specifications are modified, thus enabling network traffic monitor 300 to classify flows involving such protocols or applications. The compiler implementation of the invention thus provides for ease of maintenance.

For example, Ethernet packets can use several different information formats, but one basic format recurs constantly. A starting Ethernet file of datagram layer selections includes what patterns to look for and identifies what elements need to be parsed or extracted. The parsing job includes decoding the frame, extracting the source and destination address, and then determining the particular protocol from one of the fields.

The contents of the protocol field can cause one of several processing branches to be taken. One branch is for an Ethernet version 2 packet. Another branch can cause the protocol field to be looked at for IEEE-type Ethernet packet decoding. An Ethernet type-2 packet branch directs the parser to check to see if the protocol type is within a certain range.

Such parser checking instructions are described in a protocol language included in embodiments of the present invention. All the possible daughter packets for a parent packet are produced at a compiler output. Such daughter packets define the meaning of specific values in the protocol type field, *e.g.,* for a next node to be decoded. A next node, or daughter packet, for an Ethernet type-2 packet may be an IP-type protocol. The data value extracted is hexadecimal 0800, and such would cause the parsing system to decode IP.

Various included files are used to guide IP-packet decoding. The locations of particular elements that need to be extracted from the packet header are predefined, including the network layer addresses, protocol type, etc. Such files also include models of the possible daughter packets of the IP-protocol, according to the values found in the protocol field for the IP-header.

If the particular IP-protocol is TCP, various file elements can be extracted that will tell the compiler and organizer 310 that a connection is about to occur. For TCP/IP, the files have the port values in the connection identifier areas. The compiler and organizer 310 then can evaluate the data transferred to those ports.

As an example, a particular daughter packet including a port data value is used in Microsoft Exchange DCOMP/DCE-RPC. When a data value is found that is known to correspond to DCE-RPC, for example, the compiler and organizer 310 will thereafter be able to evaluate the file as a Microsoft Exchange DCE-RPC file. During the exchange of packets in a flow, the states that occur will follow a familiar sequence and are recorded in a file earmarked for Microsoft Exchange DCOMP/DCE-RPC.

To continue with the illustrative example, a first state record can be used to determine whether or not a particular flow is connecting. If it is, a sequence of operations can run on the incoming packets to determine if the application is running on top of DCOMP. If one of the later daughter packets reaches a state that is described in a DCOMP file, an application identifier is attached to the flow by the state processor. Once an application identifier is attached, a removal or tear-down state is included for the particular flow in case such flow may disappear, as is the case when a flow lives across multiple connections. If such flow is able to disappear, information is provided as to what states will occur, and in what sequence the packets occur to tear down the flow.

In the example of Microsoft Exchange DCOMP, flows can live beyond multiple connections or multiple sessions. So once a flow has been learned, it is saved in a flow record. The flow signature is used to point to one of the three applications that can run on top of Microsoft Exchange. These are described in a DCOMP file accessible by the compiler and organizer 310, and the information content of which is included in the database 305 of pattern structures and operations.

The parser sub-system 301 extracts flow signature information at each level in a flow hierarchy tree climb. Specific flow-signature elements at specific levels can be used to build a flow signature that is compact and efficient. At the base level, the packet acquisition device includes information on the type on network. For example, if it was indicated to the parsing sub-system that an Ethernet frame has been received, then in the parsing sub-system, in accordance with the contents of database 305, the source and destination addresses, locations, and sizes, are commanded to be extracted. Ethernet frames have end-point addresses that are useful in building flow signatures.

If on the other hand, the packets were frame-relay type packets, as indicated by the packet acquisition device and recognized by the parser, then, from information in database 305, the parser knows that for a frame-relay base layer, there are no specific end-point addresses that can help identify a flow. So for those types of packets, the database 305 instructs the parser sub-system 301 not to try to extract any end-point addresses.

In the base layer, the parser needs some identifier where the values for the next nodes are located. Ethernet has a protocol-type field. Frame relay has a protocol-like field in a control header. Identifying fields are used to determine what the next layer will be. For Ethernet, there can be a network layer, some type of encapsulation, *e.g.,* LLC 802.2, IEEE 802.3, V2-Ethernet, or even IP (a network layer).

The parsing sub-system, *e.g.,* pattern recognition process 304, needs to be told where to get the IP-destination and source addresses end-point data for a network layer to build an improved flow signature. The size and location of such elements are specified in an IP-file. At the network layer, information is specified as to where to look for a next possible node, and could be an end-point node as in ICMP.

The TCP connection protocol uses "connection identifiers" in every packet in a flow, but not necessarily in the same location in every packet. Packets can therefore be identified as being a part of a particular flow. And whether or not to apply states that deal with connections or disconnections that exist in the next layer up to these particular packets. It tells what those connection identifiers are, where they are and what their length is. In the TCP/IP-example, these are port numbers. It also tells us whether or not states that apply to connections and disconnections apply to this particular packet. Also it tells us what the possible daughter packets are. So at each of these levels, we are learning what there is in the packets that we can reapply over and over again to packets of this particular flow.

The compiler and organizer 310 will take all of the information that it gets from the individual descriptions of all possible protocols and all possible levels, and it will generate a series of elements, or instructions or operations that are stored in database 305 and that the parser then performs on every packet that it receives. Alternatively, a user can build the pattern structures and extraction operations for database 305 directly. However, the non-compiler version clearly is not as flexible as the version that includes the compiler.

Those operations, instructions or elements in database 305 not only tell the parser subsystem 301 what to look for and what it is that it's looking at, but also tells the parser subsystem 301 whether there is specific information at an appropriate layer that needs to be extracted to build up the flow signature. In addition, it also will let the parser subsystem 301 know what the next element is that needs to come out and where the offsets (pointers to location in the packet) of those elements are. It also will help the parser set up the location of those elements in the signature in a way that is interpretable by the analyzer subsystem 303 even when the locations change from packet to packet in the packets of the flow, depending, for example, on direction; or how they are evaluated, and what it is that they look like; and how they are formatted. In the preferred embodiment, for example, a source and destination address are always set up in the signature with the lower value address appearing first. The location to look for elements related to these addresses are then changed in a consistent manner. In this way, the signature for an exchange of packets between a server and a client will have the addresses appearing in the same order so that the signature identifies the same or related flows no matter what the direction of packets. Upon the parsing process 308 determining that a particular element need to be extracted, it is then passed on to the extracting engine, together with the packet. The extracting engine then extracts all the elements to determine the flow signature.

The locations and the information extracted from packets are adaptively determined for particular packet types. There is no fixed definition of what to look for and where in order to form a flow signature. In prior art systems, fixed locations are specified for particular types of packets. For example, in one prior art system, if a DECnet packet appeared, six specific fields at six locations in the packet are looked at in order to identify the packet session. If an IP-packet appeared, six formatted into differently located fields were specified for an IP-packet. In the present invention, the number of levels is variable for any protocol. The number of layers is variable and is whatever number is sufficient to uniquely identify as high up a level in the system as we wish to go, all the way to the application level in the so-called ISO-model.

With the proliferation of all the new network protocols that is occurring in the world, it becomes more difficult to specify all the possible places to look at in a packet to determine a session type. In embodiments of the present invention, a high-level language is used for specifying new protocols and new levels, including new applications. The compiler and organizer 310 describes at a machine level what information is relevant in packets that should be decoded. The parser and extraction systems (parser subsystem 301) use such in their instructions and operations, so they can adapt, and be adapted to a variety of different kinds of headers, layers, and components and need to be extracted or evaluated, for example, in order to build up a unique flow signature.

While the process steps shown in Fig. 3 can be implemented in software or hardware, the preferred embodiment is a hardware embodiment shown in Figs. 10 and 11. An alternate simpler implementation is shown in Fig. 14, and this implementation is easily implemented in software.

A type of prior art activity analysis is described by McCreery, *et al.* in United States Patent 5,787,253. A packet analyzer "324" is diagrammed in Fig. 4C of that Patent. All packets coming off a wire are run through the analyzer "324". It begins by decoding the IP-packet, and then passes the results through a set of known filters "344". A select number of packets then trickle into a set of buffers "338". An application protocol translator "346" takes an accumulation of all of the data buffers with all the packets collected, and then reconstructs them with a decoded packet recompiler "341". The decoded elements are attributed to transactions, and packets that are similar to one another are buffered up in data buffer "348". Packet analyzer "324" takes the transactions themselves that they again buffer up, and then use data sorter "340" to say "this is an application of some kind for this transaction."

Embodiments of the present invention look at sequences of packets over time, and learn about the protocols and applications and maintain state, which results in simple criteria that then can be applied in real time with adequate processing time. The McCreery, *et al.* method by buffering a number of patents provides only for "after the fact" analysis technique. A certain number of packets must be available before any analysis of those packets is possible. If the number of such packets is $N_{packets}$, then there is always a delay of $N_{packets}$ before any results are possible. Also, a buffer of $N_{packets}$ must be maintained. When a new packet arrives, that packet and all the previous $(N_{packets}-1)$ packets must be re-analyzed.

The present invention automatically maintains flow records, which in one aspect includes storing states, the invention also and generates sets of patterns that can be used recognize flows and then determining state in order to carry out state transition analysis in real time for each different protocol and application so the incoming packet information can be analyzed packet-by-packet. In a complex analysis, as more and more packets are examined, state transitions are traversed. The parsing system, and state operations build flow signatures that can easily be recognized and allow future packets that are part of the same flow to have their state analysis continued from the state easily recognized from their flow signature. That is, these flow signatures are then used to recognize various processing states in the flows examined by matching them to previously constructed flow signatures. When enough packets related to an application of interest have been processed, a final recognition state is ultimately reached. A simple flow signature can then be constructed for rapidly determining packet associations from a single packet belonging to that flow.

Several state analyzers are preferably run in parallel so a large number of protocols and application 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.

For every flow that has already been encountered, as indicated by a flow entry being present in the flow database, there are various criteria for recognizing a packet's particular state level. When each new flow starts, signatures that recognize the flow are automatically generated on-the-fly, and as further packets in the 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, these states associated with a set of state transitions for the 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. One of the great advantages of the present invention is that 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 can be generated that will key on every new incoming packet that relates to the conversational flow. A simple match can be made that saves much processing overhead, and allows high packet rates to be successfully monitored on the network.

In contrast, the prior art described in United States Patent 5,787,253, always has to start up by decoding the IP-packets and then go through all the steps for every new packet. Such system always has to go through every operation for each packet, and therefore uses the processing overhead for recognized flows and not-yet-recognized flows.

Example of Packet Elucidation

One of the two major subsystems is the Analyzer sub-system 303. This component is responsible for creating and maintaining classified traffic flows, processing statistics for packets and flows, managing the traffic flow database and cache, and performing state-based analysis of traffic flows.

The processes required for recognizing and maintaining state information for traffic flows are now described.

In order for the Analyzer 303 to successfully classify network traffic by application, there are several data elements required from each packet to be analyzed. Prior to sending a packet of information to the Analyzer, all additional information must be formatted and sent along with the appropriate packet content.

The Analyzer 303 must specifically receive each packets in a conversation in the order which they are exchange between the client and the server for proper state based classification.

Many conversational flows on a network that are associated with application have several states which must be remembered and maintained for proper traffic analysis and for traffic to be classified by protocols and application.

In the embodiments of the invention, there are several different methods in place for the creation of states in client/server network traffic. Even though there are several different methods for the creation of state. It is possible to isolate these different approaches into two basic categories.

The first category is commonly referred to as “server announcement”. In the server announcement mode there are messages which are put out onto the network, in either a broadcast or multicast approach which, all stations in the network receive and decode to derive the appropriate connection point for communicating for that particular application, with the particular server. There are several examples for this type of server announcement implementation with state based protocols. 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 second category is referred to as “in-stream analysis”. This 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 applications. 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 for 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.

One of the primary processes for tracking applications in the stream of the client/server packet exchange, is through session tracking. 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 process of session tracking, a client will make the request of 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 server will never be used again during this data exchange.

One of the best examples 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 conversation. 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 a network monitor must analyze the initial request from the client. Also, the reply from the server with the key Port information must be analyzed and used to create a signature for monitoring the remainder of this data exchange.

Another important capability for a network monitor in session tracking is the understanding of the current state for particular connections in the network. Many of the application protocols, which can be monitored, are transported via protocols that have built-in state information. An example of such a transport protocol is the common TCP, a 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. Such a sequence of events is called a connection-oriented data exchange. Tracking the state is necessary to correctly analyze connection-oriented exchanges, and the states relate to the various types of connection and maintenance messages.

Server announcement is a process used to ease communications between a server with multiple applications that are all able to 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 server announcements, messages which broadcast or otherwise sent out (*e.g.*, muticast) on the network, and all stations in the network receive and decode such messages to derive the appropriate connection point for communicating for 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 individual methods of server announcement protocols vary. However, the basic underlying process remains similar.

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 in a remote machine. Sun-RPC is used as an example of server announcement oriented communications processes.

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

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), the successor to IANA (Internet Assigned Numbers Authority), which manages Internet addresses, domain names and 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 IANA, Port number 111 is associated with Sun RPC.

As an example, consider a client (*e.g.,* CLIENT 3 shown as 106 in Fig. 1) makes 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 the 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 program 'program'. Therefore, any network monitor should include a mechanism for remembering the exchange so that future packets that use the Port number 'port' 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 get to 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'.

The working of the present invention with some Sun RPC procedures is now illustrated with the help of Fig. 9. Fig. 9 represents a dataflow 900 that occurs in the system of Fig. 3 for Sun Remote Procedure Call. Referring now to a data flow 900 in Fig. 9, assume a client 106 (CLIENT 3 in Fig. 1) is communicating via its interface to the network 118 to a server 110 (SERVER 2) via the server's interface to the network 116. Further assume that Remote Procedure Call is used to communicate with the server 110. The data flow 900 starts with a step 910 that a Remote Procedure Call bind lookup request is issued by client 106. 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 proceeds as follows:

Process for Sun RPC Analysis

1. Decode Sun RPC by TCP or UDP Port 111
2. Check RPC type field for Id
3. If value is PortMapper, save paired socket (i.e. dest for dest, src for src)
4. Decode ports and mapping, save ports with socket/addr key
5. There may be more than one pairing per mapper packet
6. Saving is complete

Note that 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 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.

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

Fig. 2 shows how a signature and flow states are built by the monitor 300 in the example of Sun RPC. 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 generates a pair of flow signatures, "signature-1" 210 and "signature-2" 212, from information found in the packets 206 and 207 which correspond in the example to the Sun to a Sun RPC Bind Lookup request and reply, respectively.

Consider first the Sun RPC Bind Lookup request. Packet 206 corresponds to such a request sent from CLIENT 3 to SERVER 2. This packet contains important information that is sued 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) also will 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 06 (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 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. 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 contains which contains the cross-hatched destination source Port number pattern from the field 218 of the packet 206 that 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. I the case of the Sun RPC example, the cross hatch represents a set of port numbers of UDS for p1 that will be used to recognize this flow (*e.g.*, Port 111). Port 111 indicates this is Sun RPC. Some applications are directly determinable ("known") at the parser level, and the Sun RPC Bind Lookups are such applications. So in this case, the signature KEY1 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 record.

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 record is obtained, and in this flow record 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 a next-state the state processor should proceed to for more complex recognition jobs, *e.g.*, a state "st^1". In the 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, the use the particular application service requested in the original Bind Lookup Request. 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 a source address fields with server "S_1" followed by client "C_1". Such values are automatically filled in at the time that the addresses are first created in a particular flow signature. Large collections of flow signatures are preferably kept in a lookup table in a least-to-highest order for the best spread of flow signatures and hashes.

The client and server thereafter 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 that 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, *e.g.*, 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.

Announcement Based Flows

There are two different types of specific operations that are required to be performed by the state processor. The first sequence of operations is known as a learning sequence. In the learning sequence, each packet that is exchanged for single flow contains all of the information required to interpret the final state, and therefore the final application for that flow.

An example of a learning sequence in the state processor can be commonly seen in the HTTP protocol. In this protocol, a sequence of packets is exchanged between the client and the server. In each of the packets, the header information and additional packet payload combined to provide traffic signature. In specific frames or packets, there are key elements of information, which are used to derive the actual application involved in the flow. In other words, during the exchange of information between the client and server key elements of data are extracted by the parser system and evaluated as payload by the analyzer system. This evaluation process occurs within the state processor. As packets are exchanged between the client and server, specific key elements clause the flow involved to move deeper into the set of states to protocol.

A well-known example of this type of exchange over HTTP can be found by evaluating the exchange of a GIF image. During the initial exchange, the flow signature is derived from the specific packet headers. After the connection identifiers have been determined, the payload of each HTTP message sent client to the server is evaluated for a specific string. The first string that is attempted to be located key "Content – Type". Once this key string is isolated in a message from the client to the server, and additional search is initiated. This next search is going to start after the location of the previous search. A string search for the word "image", along with other strings starting at the same location, is initiated. In this example, the word "image" is found after our last string.

The system has now isolated the specific content-type in the exchange between the client and server. One last search needs to be accomplished in order to derive the specific image type for this content. The system will begin searching for "gif", along with other strings at the same location. Once in this string has been located from our example, the flow signature and record for this set of exchanges between the client and server is updated. This updated flow record contains the application of a GIF image for this particular traffic flow.

The second type of traffic flow analysis that is accomplished the analyzer functions, is an association. In association, we typically find the use of a server announcement message. 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 records in the system. Because the announcement is sent to one or more stations, the client involved in a future conversation to 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.

When a server announcement message is received by the traffic monitor, the normal parsing operations and building of a flow signature and hash key are accomplished as with any other packet. In addition, payload information from the packet is sent along with signature and hash key for this flow. The flow will be recognized as described below.

A specific example of such an announcement is a bind server announcement message found in the Sun RPC protocol. An RPC server will make an announcement to a group of stations (clients) on a network. This announcement contains all of the different program identification numbers and the associated port numbers for both the TCP and UDP transport services. The monitor saves these linkages found in the payload of this packet to generate future flow signatures and records that can be used for packets exchange between clients of this server and the server.

Referring to 9, When the RPC bind announcement message enters the parsing section of our system, all of the header elements are processed as normal and key information is extracted to form the flow signature and key. In addition, the payload section of the RPC message is formatted and sent along with the signature and key to the analyzer. Because the protocol has been identified as RPC, and the child of this protocol has been identified as a bind announcement, the bind announcement state will be initiated by the state processor. This will occur in addition to the normal processing of this flow signature and record.

The state processor will be instructed to remove and review each of the individual program identifiers found in the payload of this packet. For each of the individual program identifiers, the state processor will locate the specific protocol used with the ports mapped to that protocol. To enable future flows that utilize mappings to be properly classified, the state processor will generate a special flow record and inserted in the flow database. This special flow record has specific flags set in the key match flags field. This enables the protocol identification process of the analyzer to locate the application from a subset of the normal information used during the lookup process.

At some point after this additional announcement message has been received and processed, message is utilizing these mappings will be correctly processed by the inventive monitor system. When a message or packet enters the system it is processed normally. The significant difference will be found in that the transport port information will not been known by the pattern recognition portions of the parser. This will cause the flow signature sent to the analyzer to classify the packet for the specific transport involved and include the data ports involved.

The lookup engine in the analyzer will attempt to isolate this flow record information to a specific flow found in the memory of our system. When a specific flow cannot be found, the lookup engine will attempt and other lookup and remove the clients address from the signature when key is generated. This will cause a match on multiple buckets within the flow information stored in the system memory. The system then validates this announcement by reviewing the key match flags field. If this field has a flag stating that the source address, or the client in this case, may be, then this signature will be blended with the signature stored in the analyzer and the application identifier will be set to the one found in the record.

At this point, this new flow is fully classified to the proper application. They are other protocols that use similar types of announcement methods (Novell SAP, *etc.*) . Therefore, the monitor system performs special limited key analysis in order to handle the maintenance of announcement oriented flow keys. Without this feature, flows that utilize server announcements would not be able to be properly classified for the application or service involved in the conversation.

The Overall Flow (Fig. 3)

Fig. 3 is a is a description of the overall flow of the invention and is now described in detail. The flow starts off at a number 301. There are two aspects of the invention shown here. First is the flow that describes how to generate to operations that occur on packets. The second aspect is the processing of the packets.

A flow is a stream of packets being exchanged between any two addresses in the network. Thus, 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 needed to identify the flow. There are other fields not important for identifying the flow, such as checksums, and those parts are discarded.

The PDL files describe what the system will be looking for in packets and what the sets of states and state transitions are for a sequence of packets that will determine the application or service content of the packets at the particular location in the communication network. 336 is the protocol description language files, and block 338 is the set of packet layering selections. That is the specific selections of layers and patterns of the set in 336 that the system will be evaluating.

Block 302 is a packet input into some buffer for analysis by the system. The protocol description language files 336 describes both patterns and states to identify applications and services, while the packet layer selections database 338 deals with the layering involved in those patterns and states, so 336 and 338 combined describe how one would decode, analyze and understand the information in packets, and how the information is layered. This information is input into compiler and optimizer 310. When 310 executes, it generates two sets of internal data structures. The first is block 308, the set of pattern structures and extraction operations. The pattern structures are what will be recognized in the packets and the extraction operations are what elements of a packet are to be extracted from the packets based on the patterns that get matched, the extracted elements then being combined to build up a conversation flow signature that is used for recognizing other relevant packets. The other internal data structure that is build by compiler 310 consists of the state patterns and processes, and shown as block 326. The state patterns and processes are state operations that have to occur and the patterns that have to be analyzed within states and processes that need to be performed upon moving from one state to the next, that movement and related processes depending on the packet is being analyzed in sequence. The elements in 326 are used in part of the state processing.

Once the compiling is complete, the system has all of the information that it needs to begin processing packets. In 302 a packet or series of packets enter the system, and the a step that the packets go through is being analyzed for pattern recognition in pattern analysis and recognition (PAR) engine 304. Once a pattern or a set of patterns has been identified, the pattern(s) will be associated with a set of extraction operations, and these extraction operations enter the extracting and information identifying (EII) engine 306 which has access to the packet and where the identifying information that is required to recognize this packet as part of a flow is extracted from the packet, and put into a particular sequence. The information in sequence (as well as the packet data) will next be processed by block 312 in which a unique flow signature for this conversation is built. For this purpose, a conversation signature typically includes the client and server address pairs that will be used recognize further packets that are or may be part of this conversation. The informational and the packet will then pass onto lookup engine 314 which looks in an internal data store of records of known flows that the system already has encountered, and decides whether or not this particular packets flow record a "match" with a known flow. A record is associated with each flow. The lookup is in an internal flow buffer and also includes a cache. If the flow is not in the buffer or cache, it may be in an external memory 324 (the database of flows). Block 316 determines whether this is a new record or a record that already exists. If this is a new record, then the data moves to protocol identification block 318, where the system further determines from the patterns that were analyzed and from where in the packet's state sequence one is, and from the type of protocol, whether there are any particular states and state operations that need to be executed on this packet or on any future packets that come in for this sequence of conversations.

It is important understand that the process of pattern analysis and extracting of identifying information is used to reduce the amount of information needs to be analyzed to derive what flow any particular packet belongs. Once identification is complete, there is only a small amount of information that is required to identify whether or not a particular packet is part of a flow or conversation. That information is extracted in block

306 based on the pattern that is recognized in block 304, and only the information that's needed to identify the packet as being part of a particular flow is extracted. Extraneous information including the actual data blocks in the application protocol, the protocol data unit ("PDU") or checksums, or routing fields, are all discarded because they are not required to uniquely identify the packet as part of a flow conversation.

In protocol identification block 318, it is decided what the states are, if any, that need to be applied to this packet or to future packets belonging to this conversation or flow. After this, in block 328, the first state operation for this particular flow record or pattern or protocol is applied to the packet, and this is continued until there are no more operations left (block 330). Thus the systems continues looping between block 330 and 328 applying additional operations to this particular packet until all those operations are completed, that is, there are no more operations for this packet. At this point, it is determined in 332 whether nor not there are more state analysis is required in this state sequence, no meaning we have reached a final identifying state. If we are in a final state, the process moves on to 334 for a classification finalization where we finish the process of classifying the set of packets.

Protocol identification is used to create an identification of the actual application or service that is involved by applying both state and patterns that have been derived from this particular packet and those states and patterns are then used to decide what the final set of states are for this packet or following packets before we know exactly what this is to be classified to a particular application.

A state operation may be one operation on one packet or it may be multiple operations on a packet, and carrying out the operation or operations may leave one in a state that causes exiting the system without really knowing everything about the conversation yet, but maybe knowing more about a state that is needed to execute next.

In 332 in the classification, the analyzer decides whether we are at an end state. If not at an end state, the record is updated for this (now known) record in block 322. Since this was a new record, we in fact record the record for the first time. If we reach a final state, then after finalization in 334, we also update the record for this known flow, in this case, a new but now known flow. The updating (or, if new, recording) in block 322 includes updating the states information for the known record, and carry out any statistical operations for that record. We then move to block 324 where we keep the record of this conversation together with the records of other conversations that we have maintained in the conversation database 324. 324 the set of all the conversations that have occurred. Information about all the packets that may have occurred for any conversation is included in a reduced form, such as a single element that includes what service or application the packets were associated with, and a set of statistics representing data that was exchanged, how that data was exchanged, the performance of the exchange, for all of those packets that were part of the conversation. In the preferred hardware implementation, database 324 is an external memory.

Note that in block 322, there is one record associated with each flow, but typically there may be multiple flows associated with each conversation. That is, several new flows records will be created before the final conversation is determined, and all the packets that created those flows are then associated with the same conversation. Each conversation record in 324 will therefore point to one or more flow records. For example, in the Sun RPC example used throughout this description, one could be mounting a disk (NDISK) and have multiple files open on that disk, but in reality, all the packets represent an NDISK set of transactions for that particular client and server.

If in block 316 the flow is determined not to be a new flow, but an existing flow, then in 316 is determined whether more classification is required (in the form of state operations). If yes, the system performs the required state analysis in the loop consisting of blocks 328 and 330. If we have reached a final state or there are further packets to be analyzed (determination in block 320), we ultimately update the flow records of known flows (in 322).

Note that the information created by block 306, *i.e.,* the extracted information, as well as the actual packet payload move over into 312. Block 312 then builds a unique signature from the extracted packet information, and the unique signature of and the actual packet payload move into 314 where the information is looked up. That information (the signature) *and* the packet payload flow through 314, 316, and into 318. In 318 that information is used to determine the protocol.

One signature feature of the invention is the automatic generation of patterns to search for and signatures for searching for such patterns from analyzing packets according to parsing rules, and also generating a states transitions to search for. Another feature is recognizing applications (or protocols at any level) by carrying out state analysis on a sequence of packets to recognize one or more applications. For example, the DEC patent recognizes well known patterns in single packets.

The compilation process

The compilation process includes creating the parsing patterns and operations, extraction of identifying information, and the states that are required to analyze beyond single packets.

The compiler starts off with a series of files which describe each of the particular areas of a frame that require decoding. Example, an Ethernet frame. Ethernet packets can consist of several different formats of information, with a basic format that remains substantially the same. The system therefore starts with a file. That basic Ethernet file tells the system what to look for as far as a patterns is concerned, and where elements need to be parsed or extracted. In the Ethernet case, the parsing will be: decode the frame extracting the source and destination address, and then evaluate a field for a particular protocol. The protocol field is extracted and then evaluated. The contents of the protocol field will cause one of several things to happen. Either there's a value there that says this is an Ethernet version packet, or there's a value there which sends it off to find the protocol field for IEEE type Ethernet packet decoding. As an example, consider the Ethernet type 2 packet. First there's a check to see if the protocol type is within a certain range, the check being done by the parser (the compiler simply states it). Once the check is described in the language, a listing of children are found in the language (compiler output), and the children listing contains specific values of the protocol type field, and what those values mean for the next node to be decoded. For example, a next node (child) for an Ethernet type 2 packet may be an IP type (*i.e., Internet)* protocol, and the value that would be found is HEX 0800. That value would cause the parsing system to want to decode IP. There is another file which describes the decoding of an IP packet. That file describes what elements are to be extracted from an IP packet header, including the network layer addresses that are used and other information, such as protocol type. In that particular file are also the children of the IP protocol, and depending on the values found in the protocol field for the IP header. What values are found causes one of another set of operations to be performed on the packet, and those are described in another set of files. Consider TCP as the particular IP protocol. Now within the TCP file, there are described what and where are elements of information that, for example, tell the compiler that this particular node gives us some information about the possibility of a connection happening. Because of this, the connection identifier or identifiers need to be filled in, depending on the protocol, and those connection identifiers are described in the language: where they are, how to evaluate them, and if there are any possibility of children for those values. For TCP/IP, the example would be the Port values. The compiler output instructs evaluating those Port number values. One of the possible children of the Port number value, for example, may be Microsoft Exchange's DCOMP or DCERPC. When the value corresponding to DCERPC is found, that will cause the compiler to evaluate the Microsoft Exchange's DCOMP/ DCERPC file. Within that file, is a set of information describing the states from the actual packets that will occur during the exchange of packets for a flow. The first state, of course, will be to determine whether or not this particular flow is in a connecting state, and if it is in a connecting state, what sequence of operations are performed on which packets to determine if it truly is an application running on top of DCOMP. If one of the children states which is described in one or more operations in the DCOMP file causes a match, than that particular application identifier will be loaded into the record for this particular flow by the state processor. Once this application identifier is loaded in, there is a state for removal or tearing down of this particular flow in the case that this flow had the ability also to disappear. If the flow does not have the ability to disappear – that is, there are no tear-downs and it lives across multiple connections and disconnections. If it has the ability to disappear, then we are given information as to what states will occur in what sequence of packets to tear this particular flow down and to relearn it. In the example of Microsoft Exchange DCOMP, those particular flows live beyond multiple connections or multiple sessions, so once the flow has been learned, it will be saved in the flow record, and that flow signature will be used to always point to one of the three applications known to run on top of Microsoft Exchange which are described in the DCOMP file of the compiler as either the mail transport adapter, the information store or the directory look-up.

The parsing system at each level extracts key information for building the signature. There are specific key elements at specific levels that are used to help build a flow signature that will more precisely identify the specific flow for a set of packets. At the base level, for example, if we were to determine in the parsing system that we are dealing with an Ethernet frame, Ethernet frames have end-point addresses that are useful in building a better flow signature, so the system is told to extract the source and destination addresses, including where the locations and sizes of those addresses are. In a frame-relay base layer, for example, there are no specific end point addresses that help identify the flow better, so for those type of packets, the compiler instructs the parser not to extract the end-point addresses. Once we get into a base layer, there needs to be some identifier that tells the parser where the children are – that is, where the next nodes potentially are. For Ethernet, there's the type (protocol type) field. For frame relay, there's a protocol- like field in the control header. We use those identifying fields to determine what the next layer is. In the Ethernet example, there can be a network layer, or some other type of encapsulation of Ethernet, for example LLC 802.2 or IEEE 802.3 or it could be V2 Ethernet going right into IP. In the example of IP, that's another special layer, where we now have a network layer. The parsing system needs to be told where to get end-point data for the network layer to build a better flow signature. This is the IP destination and source addresses, which are in every IP packet. The size and location of those would be specified in the IP file. At the IP (or other network layer), there is information specified as to where to look for any possible next nodes. The next node could be an end point node. For example, ICMP. That is, we know it's ICMP, and we're done (that the last node), or it could be TCP, and if it's TCP, the TCP file will contain information about that level.

TCP is an example of a protocol that can tell us about the connections. Whenever we get to a protocol that tells about the possibility of a connection, connection identifiers are needed. That is, something that is going to exist in every packet, perhaps not in the same location in every packet, that the system can identify that this particular packet is part of this particular flow, and whether or not to apply states that deal with connections or disconnections that exist in the next layer up to these particular packets. It tells are what those connection identifiers are, where they are and what their length is. In th TCP/IP example, these are port numbers. It also tells us whether or not states that apply to connections and disconnections apply to this particular packets, Also it tells us what the possible children are. So at each of these levels, we are learning what there is in the packets that we can reapply over and over again to packets of this particular flow.

The compiler will take all of the information that it gets from the individual descriptions of all possible protocols and all possible levels, and it will generate a series of elements of elements or instructions or operations that the parser then performs on every packet that it receives. Those operations or instructions or elements not only tell the parser what to look for and what it is that it's looking at, but it also tell the parser whether there's specific information at an appropriate layer that needs to be extracted to build up the flow signature. In addition, it also will let the parsing system know what the next element is that needs to come out and where the offsets (pointers to location in the packet) of those elements are, and also will help the parser understand how the location of those elements may change from packet to packet, depending, for example, on direction. How they are evaluated, and what it is that they look like, and how they are formatted. Upon the parser determining that a particular element (or elements) need to be extracted, is then passed on to the extracting engine, together with the packet. The extracting engine then extracts all the elements to determine the flow signature.

What is unique here is that locations and the information extracted from any packet is adaptively determined for the particular type of packet. There is no fixed definition of what to look for where in order to form the flow signature. In prior art systems, such as Chiu's DEC patent there were fixed locations specified for particular types of packets. For example, if a DECnet packet appeared, the system looked at six specific fields (at 6 locations) in the packet in order to identify the session of the packet. If one the other hand, an IP packet appeared, six different locations were specified for an IP packet. The system was only able to recognize sessions. The physical layer, going onto the network layer, than the protocol layer. There were fixed locations for each of these. In the present invention, the number of levels is variable for any protocol. The number of layers is variable and is whatever number is sufficient to uniquely identify as high up the level system as we wish to go, all the way to the application level (in the OSI model). Clearly, with the proliferation of protocols, the specifying of all the possible places to look at to determine the session becomes more and more difficult. Adding a new protocol or application likewise is difficult. In the present invention, a language exists for specifying new protocols and new levels, including new applications. The compiler is used to describe what information is relevant to packets and packets that need to be decoded, and the parser and extraction systems function using those instructions and operations. So, they can adapt, and be adapted to a variety of different kinds of headers, layers, and components and need to be extracted or evaluated, for example, in order to build up a unique signature. The only thing that is fixed is that when you build the language, you want to try to fill in the components that have end point addresses at the lowest layer, components that have end point addresses that identify the actual workstations that are involved, and also something that identifies where the layers that manage the connections or disconnections of particular communications occur. You want to fill in those general areas, and where to look for the next layer. So the system can adapt to new protocols. The prior art is very specific to specific types of packets that you want to parse.

One feature of the analyzer 300 is the parsing and extracting system comprising processes 304, 306 and database 308, together with the compiler 310 that generates the pattern structures and patterns and extraction operations. This parsing system is designed to be flexible in its implementation. The compiler system 310 uses as input descriptions of protocols and applications written in a protocol description language (PDL), these PDL commands describe the patterns and extraction operations that will be required in a manner that is independent of the different types of packets that will be used to carry the information. The compiler creates from these a set of specific patterns in database 308 to be analyzed by the parsing system (block 304) and then a set of extraction operations (also in 308) that are dependent on the patterns that are analyzed. The elements that are parsed by the parser will cause specific elements to be extracted. As an illustrative example of how this aspect of the invention provides for flexibility in extracting information from any type of packet, it is known that the headers of Ethernet packets are different from headers of frame relay network packets. By describing the structure of the packets in the PDL, both header types can be accommodated by the system, and the parser will know how to parse each of these headers and how to extract identifying information t build a unique flow signature at this level. The particular information above the base layer also will differ depending on the network layers are involved in the conversational flow in our particular implementation the invention is set up such that the network layer is not important in how things are extracted from the actual packet cannot specifically defined in prior art such as the deck of patent the specific examples given our eight DECnet packet with a DECnet transport headers with a DECnet session layer headers and there's even the specific example describing how to extract DECnet specific elements out of the DECnet session layer patter and turn it into a DECnet specific hash for key in the eight in the invention that we have described the elements better used to make the signature are independent from the actual type of network layer transport layer session layer and application involved out in the exchange of packets on the network the only requirements that we have our that at specific base layer such as Ethernet if you have access to and point addresses they should be included in elements to be extracted by the extraction engine to build the unique flow cake also information that tells you about the children or protocols beyond the Ethernet layer need to be told love so the parsing system can recognize them in the extraction in extract those components to be used in the flow signature to however from that point on the specific information is only relevant to the type of layer you're apt once you're at the IP layer or the network layer for example we also work last that information be extracted which shows the end points of the workstations involved now that information does reside in the header of the IP frame however the way that we request the extraction is independent of IP or DECnet or Novell or any other type of network layer protocol once we get to a particular a network layer the next section that we're interested in of course is how the packet is being transported to if the transport layer or whatever transport protocol is in place has information that tells about the connection end points involved in the conversation or flow and also has information relative to telling us whether this layer is causing a connection to occur or not.

In some communication arts, the term “frame” generally refers to encapsulated data at 0SI 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” also is used. In the present application, 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.

In order for an analyzer to be able to analyze different packet or frame formats, the analyzer is required to perform a parsing to understand the data encapsulated in the different fields. As the number of possible packet formats or types increases, the amount of logic required to parse these different packet formats also increases.

A network analyzer preferably can analyze many different protocols. At a base level, there are a number of standards used in digital telecommunications, including Ethernet, HDLC, ISDN, Lap B, ATM, X.25, Frame Relay, Digital Data Service, FDDI (Fiber Distributed Data Interface), and T1, among others. Many of these standards employ different packet and/or frame formats. For example, data is transmitted in ATM and frame-relay systems in the form of fixed length packets (called “cells”) that are 53 octets (*i.e.,* bytes) long and several such cells may be needed to make up the information that might be included in the packet employed by some other protocol for the same payload information, for example, for example in a conversational flow that uses the frame-relay standard or in a conversational flow that uses the Ethernet protocol. Fig. 16 shows the header 1600 (base level 1) of a complete frame of information and includes information on the destination media access control (Dst MAC 1602) and the source media access control (Src MAC 1604). Also shown in Fig. 16 is some (but not all) of the information specified in the parsing structures and extraction operations database 308 to be extracted at this level by extractor 306 with which to build the information used for further analysis. This includes all of the header information at this level in for form of 6 bytes of Dst MAC information 1606 and 6 bytes of Src MAC information 1610. In addition, the hash key to be made from the Dst MAC (2 byte Dst Hash 1608) and from the Src MAC (2 byte Src Hash 1612) which are part of the conversational flow key built in block 312 for ease of recognition. Finally, information is included on where to find the next level’s information. Fig. 17 now shows one of the possible levels-2 format, that of an Ethernet packet 1700. The Ethernet packet 1700 includes a two-byte type field 1702 for the type of protocol used for the next level and the remaining information 1704, shown hatched because it is masked out by extractor 306 according to information in pattern structures and extraction information database 308. Also shown is some of the extracted part. That is, the extracted part 1702 is shown also as data 1706 which is part of the extracted information. Also included is the 1-byte Hash 1710 for this information used in building the flow signature. Finally, an offset field 1710 which provides the offset to use to obtain level 3 information is included, and for the Ethernet packet, this is 14 bytes for the start of the frame.

Other packet types also may be analyzed. For example, in an ATM system, each ATM packet comprises a five octet "header" segment followed by a forty-eight octet "payload" segment. The header segment of an ATM cell contains information relating to the routing of the data contained in the payload segment. The header segment also contains traffic control information. Eight or twelve bits of the header segment contain the Virtual Path Identifier (VPI), and sixteen bits of the header segment contain the Virtual Channel Identifier (VCI). Each ATM exchange translates the abstract routing information represented by the VPI and VCI bits into the addresses of physical or logical network links and routes each ATM cell appropriately.

At the next layer, there similarly are many different formats. There is the well known IP (internet protocol), SNA, VINES VIP, APPLETALK, *etc.*

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 instructions database 328, and this part can occur off-line or from a central location. A convenient high-level compiling language is input by a user. High-level commands that describe the network applications and protocols to be used are interpreted during initialization for use by the parsing subsystem system 301. In addition the state instruction database is generated. A starting point 401 inputs new "source-code" information into a high-level compiler description file 402. A compiler 403 generates a program code 404 for packet parse-and-extract operations, and a program code 405 for packet state instructions and operations. The program code 404 for packet parse-and-extract operations is organized into a pattern, parse, and extraction database 406. The program code 405 for packet state instructions and operations is organized into a state-processor instruction database 407. 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, and also the extraction process 600 description, and Fig. 6.). 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 herein, and Fig. 11.) In a step 410, the analyzer has been initialized and is ready to perform recognition.

The PDL Compiler is used to convert a set of PDL source files into a layered set of specific protocol identifiers, patterns, extraction operations and states. The PDL compiler uses the PDL source files and layer selections as the primary input for pattern analysis, extraction operation, flow key generation and state operation details.

The compiling process is illustrated in Fig. 24. First the compiler must load all of the PDL source files listed at execution into a scratch pad memory (2403). Next the compiler review the files for the correct syntax (2405). Once completed, the compiler creates a set of patterns in the form for CPL (2407). CPL is the intermediary file form that the PDL Compiler outputs to the CPL system to perform the final optimization.

After the patterns have been created, the compiler creates the extraction operations in CPL that are required at each level for each PDL module. This creates a set of operations to perform for the building of the flow key and for links between layers (2409).

With the flow key operations complete, the PDL compiler creates the operations required to extract the payload elements from each PDL module. These payload elements are used by states in other PDL modules at higher layers in the processing (2411).

The last pass is to create the State CPL operations required by each PDL module. The State operations are complied and CPL is created for later use (2413).

CPL stands for Compiled Protocol Language. This is the ‘assembly code’ form for the Traffic Classification System. The PDL Compiler is designed to evaluate each PDL module, form the operations required and walk the tree of layers. The last operation performed by the PDL Compiler is to output the CPL instructions.

These CPL instructions have a fix layer format, they include all of the patterns, extractions and states required for each layer and for the entire tree for a layer. This CPL file is then run by the Optimizer to create the final output binary memory structures that will be used by the Traffic Classification system.

Fig. 33 shows the PDL files for a sample operation of the system.

Detailed operation

Fig. 5 shows a flowchart of how the actual parsing system functions. Starting at 501, the packet is input to the packet buffer in step 502 and set at the first packet component. Step 503 loads the next (initially the first) packet component from the actual packet. The packet components are extracted from each packet one element at a time. Then, in 504 a check is made if the load packet component operation completed successfully. If not, this indicates no more packet components, and the system builds the packet signature in step 512. If the operation succeeded as determined in step 504,the in 505 are fetched the node and processes from the pattern database 406 according to the node pattern in the packet. This gives us a set of patterns and processes defined for that node to apply to that particular element in the packet. The system checks in 506 if the fetch pattern node operation completed, indicating there was a pattern node that loaded in 505. If yes, then the node and process are applied in 507 to the component extracted in 503. If a pattern match is obtained in 507, as indicated by the test in 508, that means the system has found a node in the parsing elements, and the system proceed to step 509 to extract the elements. Step 509 if described in detail in a separate flow diagram (Fig. 6). If applying the node process to the component does not produce a match, then from step 508, the system requests the next pattern from the pattern database (called "folding the pattern database") and the system returns to step 505 to apply the next node and process and extract it and check, and thus the loop between 508 and 505, called the applying pattern loop. Once the system either completes all the patterns and has either matched or not, the system moves to step 511, which is the next packet component. This step tells the system to move or ratchet itself to the next element of the packet that was input in 502. Then again one loads the first packet component. The system then reapplies the pattern process and runs the 505 to 508 loop.

Once all the packet components have been the extracted from the input packet in 502, then in 504 the load more packet component operation is determined not to have completed, and the system moves to build a packet signature which is described in Fig. 6

Fig. 6 describes in the form a flowchart the step 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 the system has a completed packet component and a pattern node that was received from the pattern engine and available loaded in a buffer at 602. The first step is to load the packet component that the system received from the pattern analysis process of Fig. 5. Again, the system checks to see if the load completed, that is, if there's more packet components. The first-time through there is, so at 605 the system now takes the extraction and process elements that the system received from the pattern node component in 602, and the system fetches those. The system checks in 606 if the fetch was successful, indicating that there are extraction element that can be used, and the first time through the answers is yes, so then the system applies that extraction process to the packet component based on the instruction received from a pattern node. That process removal of the element from the packet component and cause that element to be saved. In step 608, the system checks if there is more to extract out of this packet component, and if the answer is no, the system moves back to 603 to load a new packet component. If the answer is yes, then the system moves to the next packet component ratchet and move beyond the packet component that is at hand. That new packet component is then loaded in step 603. As the system moved through the loop between 608 and 603, extra extraction processes are applied either to the same packet component if there's a more to extract, or to a different packet component if there is no more to extract.

The extraction process builds the signature. Once we cannot load a packet component in 603, indicated by failure in the load successful test 604, all the components have been extracted. The signature that has been built is loaded into the signature buffer and the system proceeds to Fig. 7

Fig. 7 completes the signature building process. The system starts in 702 with the signature buffer and the pattern node elements that the system received on exiting the extraction process of Fig. 6 and the system loads the pattern node element out of the element database, the system the checks 704 if the load was successful, *i.e.,* if there are more nodes. The first-time through there are more, so the system in 705 hashes the signature buffer element based on the hash elements that are found in the pattern node that in element database. That is, in 705, for each individual pattern node element there is a sequence of instructions of how to build a signature, and these are followed. In 706 the resulting signature and the hash are packed. In 707 the system moves on to the next packet component which is loaded in 703.

The 703 to 707 loop continues until there are no more pattern no elements left. Once all the pattern of elements have been hashed, then at 708 reached from 704, the system generates the output of Fig. 7's process for the analyzer and move to Fig. 8

Fig. 8 is a flow diagram describing the operation of the lookup/update engine. The process starts at 801 from Fig. 7 with a signature including the hash and the key elements of the packet. In 802 those elements have been loaded into what is called herein the unified flow key buffer (UFKB). An entire UFKB entry is removed from the buffer in 803 and then the system computes a "record bin number" from the hash. That is we apply any simple hashing model to the information that was extracted and that results in a record bin number. In 804, the system requests that a bucket from that bin be loaded into a cache. A bin may have one or more "buckets". The cache is described in more detail elsewhere herein. In 805 the system checks to see if operation 804 returned with a bucket from the bin number, a yes indicating that there are more buckets in the bin. If this is the first bucket for the requested bin, then this is the first-time through and the bucket request 804 is successful and the system moves to 807 where it compares the current bin and bucket record signature to the packet. That is, the system examines to see if this is the right packet, now that the hash has gotten the process this far. In 8 08 the system checks to see if there is a match, and if so, 810 marks that record bin and bucket as "in process" in the cache and a timestamp is put in the cache to indicate to the system that this record bin bucket this time through. Step 811 sets the unified flow key buffer element that the system extracted in 802 for this particular packet that is being processed as "found." The "found" indication allows the (other) state machines in the system to begin processing this UFKB element. Then in 812 the system updates the statistics for the record in the cache based on the statistical and operations that received when the system entered process at 801. The process exits at 813.

Regarding updating step 812, the system the system is designed so that when it sees a packet, it goes through the process described above which collapses this packet into a flow which consists of multiple packets that went between the client and server for this particular application. Hence, for every packet that the system sees, the system performs a set of statistical operations those operations, which may be counting the packets, obtaining a statistics on the size of the packet, or it could be counting differences difference between this packet and one that was received in the opposite direction via the time stamp, so the system can display the frequency as to which packets are being exchanged. The statistics might be an operation that takes this time stamp in relation to ship to a packet going in the same direction so the system can see the proximity of one packet to another flowing in the same direction. All of these statistics may be used in combination with each other to analyze many different aspects of the date communication network's ability to transfer information for this application. This analysis might include measuring the quality of service of a conversation, measuring how well an application is performing in the network, measuring how much a an application is consuming of the network resources, and so forth, and all such analyses come this operation 812 of applying simple statistical calculations to each packet and rolling them up into these so-called flows that are being generated.

If at 808 the signature match does not succeed then in 809 the system requests the next bucket for this bin can goes back to 804 to request that the cache make ready the next bucket. If this operation 804 is not successful, indicating that there are no more buckets in the bin. So the system goes through requesting bucket until either there is a match in 808 or 805 states there are no more buckets in the bin. If eventually no match was obtained and there are no more buckets in the bin, then the system needs to set up a new flow, since this flow has not previously been encountered, and in 806 the system marks the flow in the unified flow key buffer for this packet as "new", and in 812, the same statistical operations are performed for this packet in the cache, that the statistics for this packet of a new flow are captured. The operation exits at 813.

The hardware system

Using Figs. 10 and 11, each of the individual hardware elements that the data flows through in the system are now described. Note that while we are describing a particular hardware implementation of the invention described in Fig. 3, it would be cleared to one in the art that the flow of Fig. 3 may be implemented alternatively, on a general-purpose computer, or only partly implemented in hardware. Fig. 14 shows such an implementation. The hardware embodiment easily meets the speed of over a million packets per second, which the software system of Fig. 14 may be suitable for slower networks. In the future as processors become faster, more and more of the system may be implemented in software has would be cleared to one in the art.

Fig. 10 is a description of the parsing and extracting system. The PAR system includes the following items required to get the system started. Memory 1001 is the pattern recognition database memory. This is where the patterns that are going to be analyzed are stored. Memory 1002 is the extraction operation database memory and this is where the extraction instructions are stored. Both 1001 and 1002 correspond to internal data structure 308 of Fig. 3. The system operation typically starts by an initialization during which these database memories are loaded a through host interface multiplexor and control registers 1005. The two memories are loaded through the internal buses 1003 and 1004. Note that the elements in 1001 and 1002 are re compiled in operation 310 of Fig. 3 externally to the system shown in Fig. 10.

A packet enters the parsing system via 1012 into a parser input buffer memory 1008 using control signals 1021 and 1023 controlling an input buffer interface controller 1022. Interface is easily generated by a standard control logic as is well-known in the art. The interface is to a packet acquisition device. How to generate the packet starts and next packet signals 1021 and 1023 also is known in the art to control the data flow into parser input buffer memory 1008. Once a packet starts to load into parser input buffer memory 1008, pattern recognition engine 1006 carries out the operations on the input buffer memory described in block 304 of Fig. 3. Once a pattern is recognized, the pattern operation identifiers are sent to an extraction engine1007. The operations of the extraction engine are those carried out in blocks 306 and 312 of Fig. 3. Operation identifiers rather than data is transferred allowing the extraction engine 1007 to perform extraction operation on data in input buffer 1008, say located in at 1009, while more packet information is being pattern analyzed simultaneously by the pattern recognition engine 1006. That is a pipeline is used to provide sufficient processing speed to accommodate the high-speed of the packets passing in the network. The operation identifiers are in the form of extraction instruction pointers to tell the extraction engine aware where to a find the instructions in the extraction operations database memory 1002 for extracting an element of the packet in the input buffer memory.

The extraction engine 1007 performs the extraction operations on the parser input buffer memory 1008/9 and outputs the extracted elements in the form of a flow signature into a parser output buffer memory 1010. Any additional payload from the packet that is required for further analysis also in included. Once information that is in the parser output buffer memory 1010, the information is then pushed out (at 1013) into the unified flow key buffer shown as item 1103 on Fig. 11 describing the analyzer. An analyzer interface controller 1011 is used to manage the flow of data into the analyzer (Fig. 11), including to the unified flow key buffer 1103. The analyzer interface control 1011 tells the unified flow key buffer section of the analyzer via 1025 when data is ready to be sent by into the unified flow key buffer, and the analyzer is responsible to keep a ready signal 1027 high (or low, depending on implementation) when the analyzer can except the data of from the parser output buffer memory 1010.

Fig. 11 shows the hardware components and dataflow for the analyzer subsystem. Prior to the system starting, the information that is generated by the compiler is inserted into a database memory for the state processing, called state processor instruction database (SPID) memory 1109. The loading of SPID occurs through host bus interface 1122 which has direct access to analyzer host interface controller 1118 which in turn has access to cache system 1115 and 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 into over the host bus interface 1122.

Once the state processor instruction databases memory 1109 is loaded, the system is ready for receiving packet flow signatures and payload that come from the parser (Fig 10), in particular units 1010 and 1011 via the parser interface 1101. The unified flow key buffer (UFKB) 1103 is a specially designed memory or sequence of memories that is set up to maintain and hold flow signatures is to be processed or that are in process. The flow key buffer 1103 also holds the payloads of those packets from which the flow signatures were determined. The contents of unified flow key buffer 1103 include several state (or status) identification to allow different processes to run concurrently. There are three finite state machines (FSMs) that can concurrently run: the lookup/update engine 1107, the state processor 1108, and the flow insertion and deletion engine 1110. Each processes data from the UFKB 1103, the data used by state processor 1108 having first been processed by lookup update engine 1107, and the data used by the flow insertion/deletion engine 1110 having first been processed by the state processor 1108. 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. Each entry may not need to be processed by all three engines. The three finite state machine engines run concurrently to allow lookups to occur while the state processor may be processing states for another item while yet another item is being inserted in engine 1110. Some entries may need to be processed more than once by a particular engine. There is bi-directional access between each of the finite state machines and the unified flow key buffer 1103. Once an element exists in the flow key buffer 11013, the first engine to use the data is the lookup/update engine 1107 which takes the flow signature that was generated by the parsing and extracting process and begins a lookup request to the cache system interface 1115. The lookup/update engine's operation is that of blocks 314 and 316 on Fig. 3. The caching system 1115 is described below. Once an element has been looked up then updated, or has not been found, the appropriate status for that element is updated in the unified flow key buffer entry for that particular flow signature and packet. If there are any state operations to be executed, control is passed over to state processor 1108 for that particular flow key buffer entry. State processor 1108 extracts from unified flow key buffer 1103 the information that was updated by lookup engine 1107. One of the elements stored in the flow key buffer as updated by the lookup/update engine 1107 is a state processor instruction to be executed, in the form of a number, and state processor 1108 extracts this element from the unified flow key buffer entry that is ready for the state processor, sets the processor 1108's instruction system to run the program counter based on the number stored in the flow key buffer entry by the lookup/update engine.

That instruction causes a sequence of one or more state operations to be executed in state processor 1108 to further analyze the payload that is in the flow key buffer entry for this particular flow signature and packet. Once the final state operation for that particular packet has been executed on the data in the unified flow key buffer 1103, that information is updated both in the cache system 1115 and in the unified flow key buffer 1103. Control is then passed on to the flow insertion/deletion engine 1110 for that flow signature and packet entry if the flow needs to be inserted or deleted from a database of flows. The flow insertion and deletion engine 1110 is responsible for creating new flows in the flow database, and deleting flows from the database so that they can be reused. This is carried out in a process of bucketing and binning described hereinbelow with the aid of Fig. 12 and carries out the operations of block 318 of Fig. 3. The flow insertion/deletion engine recognizes that needs to be processing information based on a status field in the unified flow key buffer 1103.

The cache and caching engine 1115 is designed to have information flowing in and out of it from five different points within the system. The lookup/update engine 1007 is able to request the cache system to 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 they are looked up through 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 system 1115 is intelligent enough to access to the flow database and to understand the data structures that exists on the other side of memory interface 1123. The cache can retrieve information from the memory through the member interface 1123 the unified memory controller 1119, and can also update information in the memory through the memory controller 1119. The cache system can also be maintained, change and managed by the analyzer host interface and control 1118, which, for example, allows for the direct insertion into the cache of specific flow records and other elements from the flow database via the host bus interface 1122.

Once a set of operations is performed on a unified flow key buffer entry by all of the state machines 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.

There are several interfaces to components of the systems external to the module of Fig. 11 for the particular hardware implementation. These include host bus interface 1122, designed as a generic interface which can operate with any kind of external processing system such as a microprocessor or a multiplexor (MUX) system so that one can hook the overall a 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. Another generic interface is memory interface 1123 designed to interface to any of many types memory systems that one may want to use to store the flow records. The unified memory controller 1119 deals with managing how memory is accessed and maintained. Member interface 1123 is "generic" so 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 out and into host interface multiplexor an control registers 1005. This enables the parsing system to be managed by an external system, for example a general purpose processor 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 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 integrated into another, say general-purpose system that is being a 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. Currently each of the block diagrams shown in Figs. 10 and 11 are implemented in a set of six field programmable logic arrays (FPGAs). The boundaries these FPGAs are as follows: Fig. 10 is implemented as two FPGAs. The parsing system that deals with pattern recognition in one FPGA, and this includes in the input side blocks 1006, 1008 and 1012, and also parts of 1005, and memory 1001. The extraction system is in another FPGA which includes 1002, parts of 1005, 1007, 1013, and 1011. The memories in Fig. 10 are included in the FPGAs as are the generic interfaces. Referring to Fig. 11, the unified looking buffer 1103 is in a single FPGA. The fourth FPGA includes state processor 1108 and the state processor instruction database memory 1109. In addition, portions of the state processor instruction database memory 1109 are maintained in external SRAMs. The fifth FPGAs includes finite state machine engines 1107 (the lookup/update engine) and 1110 (the flow insertion/deletion engine). The sixth FPGA includes the cache system 1115, the unified memory tour 1119 and the analyzer host interface and control 1118.

Note that rather than as a set of application specific integrated circuits (ASICs) such as FPGAs, one can implement the system as one or more VLSI devices. In the future, it is anticipated that device densities will continue to increase, so that the complete system may one day form a subunit (a "core") of a larger single chip unit.

The operation of the flow insertion and deletion engine 1110 is now described with the aid of Fig. 12. The engine is entered at 1201 upon existence of a unified flow key buffer entry for packet having the status of "new". With the status being "new" in the entry 1202, the next step if 1203, accessing a conversation record bin. This information is already maintained in the unified flow key buffer 1103 for this flow signature from a previous lookup that occurred using the lookup engine 1107. In 1204 the system requests that the record bin/bucket be maintained in the cache system 1115. As long as the cache system 1115 says that the bin/bucket is empty in 1205, step 1207 inserts the flow signature (with the hash) into the bucket and the bucket is marked "used" in the cache engine using a time stamp that is maintained throughout the process. Then in 1209 and the system 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" do in the cache system. This allows the caching engine to understand do that it needs to actually push the record out through the unified memory controller 1119, into off chip memory. Finally in 1212, the initial statistics for the record are set in the cache system so that they are either cleared or, whatever set procedures the particular statistical operations require the system to do for the first packet that is seen for a particular flow.

Back in step 1205, if the bucket is not empty, the system requests the next bucket for this particular bin in the cache system. If that particular bucket is not valid than control passes to 1207, repeating the processes of 1207, 1209, 1211 and 1212. If at 1208, the bucket is seen to be in a valid state, the set 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 operation of the state processor 1108 is now described. The state processor is entered at 1301 with a unified flow key buffer entry to be processed which is marked with status "new" or "found". This entry is retried from unified flow key buffer 1103 in 1301. In 1303, the state processor's instruction pointer the value found in the unified flow key buffer entry, and this instruction is fetched in 1304 from the state processor instruction database memory. In 1305 the operation or set of operations fetched is carried out by the state processor. The typical instructions include parsing operations to look up and possible analyze a pattern from the packet in the unified flow key buffer 1103, evaluate an offset in the payload of the packet, etc. The single state processor instructions are very primitive (*e.g.*, moves, compares), therefore many such instructions need to be performed on each unified flow key buffer entry. In 1307, a check is made to determine if there are no more instructions to be performed. Each instruction performed results in either another instruction that needs to be performed, or no more operations. Therefore, if at 1307 it is determined that there are more instructions, then in 1308 the system sets the state processor instruction pointer to the value found as the next instruction in the current state and the process moves to step 1304 where the next instruction is fetched for execution. This loop between 1304 and 1307 continues until there are no more instructions to be performed. In 1309, a check is made in 1309 if the processing on this particular packet has resulted in a final state. That is, the system is done processing not only for this particular packet, for the whole flow that the packet is part of. That is, at the end of processing this packet, there either is another state that another packet is needed for, or a final state has been reached. If there are no more states to process, then in 1311 the processor sets and saves the “flow removal state” as a state processor instruction in the current flow record that determines whether or not an operation to remove this flow is set in place. Some final states may need to put a state in place which tells the system to remove a flow for example if a connection goes away from a lower level connection identifier. In 1311 a flow removal state is set and saved in the flow record. The flow removal state may be a NOP (no -op) instruction which means there are no removal instructions, or it may be a set of operations that are needed to be performed to evaluate whether or not this flow is going to get reset back to a state where it needs to be re-evaluated to make sure that it still the flow or end-result flow that the system has temporarily determined that it is.

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 1316 the system saves the state processor instruction pointer in the current flow record in the current flow record. That will be next operation that will be performed the next time the lookup engine 1107 finds a match for this flow. Again, the processor now exits processing this particular unified flow key buffer entry at 1313.

The Parser Subsystem in More Detail

The preferred hardware implementation of the parser subsystem is now described in more detail.

Highlights

The following are the highlights of the preferred hardware implementation of the parser subsystem which is shown in Fig. 10.

- Synthesizable modules written in both the Verilog and VHDL
- Scalable architecture for any size switch or probe
- Can recognize many (e.g., > 2000) different protocols

- Extensible to new protocols
- Recognizes encapsulations
- Builds signature and payload data structure for analyzer (the flow signature)
- Scaleable protocol pattern recognition engine
- At 62.5 MegaHertz can process up to 1.5 MegaPackets per second
- Accepts protocol database output from the compiler

Architectural Overview

The overall architecture is shown in Fig. 10. The parser module consist of two main sub-modules. These are the pattern recognition engine (PRE) and the extractor. The PRE analyzes the packet and the extractor builds the flow signature from the packet and instructions from the pattern recognition engine .The parser has been split into two parts for several reasons. First and foremost, the split correctly partitions the functions to allow maximum reuse of silicon across the over two thousand protocols that can be supported. Another advantage of the split architecture is that the compiler can analyze the three dimensional space occupied by the offset, level, and pattern data of the specified protocols and compact the databases used in the parser module. The set of specified protocols defines a tree of linked nodes. Each protocol is either a parent node or a terminal node. A protocol is a parent node if it links to other protocols that can be contained in it. For example IP is a parent to UDP. Protocols can be the children of several parents. If a unique node was generated for each of the possible parent/child trees, the database would explode exponentially. Instead, child nodes are shared among multiple parents thus compacting the database.. Finally the PRE can be used on it's own when only protocol recognition is required.

The parser module pouches the network data through the DataPort interface. The data is first processed by the pattern recognition engine. This engine consists of a comparison engine and a database. 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 the second stage. This is the only protocol level that is not programmable. This is because the detection of the protocol at this level is simple and well defined. It is implemented with partial CAMs that return a node identifier if hit. This second stage has two full sixteen bit CAMs defined for future protocol additions. After this detection is completed the engine initializes Current Offset Pointer (COP) to the next part of the packet that needs to be checked. The node identifier from the previous stage and the data pointed to by the COP are used by the PRE to lookup an entry in the database. As each protocol is recognized, the pattern recognition engine emits a unique protocol identifier. It also emits a process code that the extractor uses to build the flow signature. This process is repeated until the node identifier's Terminal bit is set. At that point the PRE has completely recognized the protocols in the packet and readies itself for the next packet.

The extractor extracts information from the packet to build the flow signature. For example, it will extract the source and destination addresses from the packet and pack them into the flow signature data structure. It may also process certain parts of the packet to speed up flow processing performed by the analyzer. It will build a hash value from certain parts of the packet to speed looking up the flow in the analyzers' database. The extractor transfers data from it's input Buffer to it's output Buffer based on the sequence of instructions in it's instruction database. When the PRE 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. Instructions consist 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 extractor to copy n bytes data unmodified from the input Buffer to the output Buffer. 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 to the HASH generator. The result from the HASH generator is always written into the first two bytes of the flow signature. It is used to accelerate the lookup of the flow in the analyzers flow database. Once the flow signature is completed, the extractor transfers it to the analyzer for further processing.

The parser module databases can reside in ROM or RAM. If the databases are in a RAM the parser can be programmed to recognize new protocols or a different set of protocols.

Bandwidth requirements

The target throughput for the traffic monitor hardware running at 62.5 Megahertz is 1.5 million packets per second (PPS). This is the sustained maximum throughput of a single Gigabit channel. At this rate the parser module has 41.6 cycles to process each packet. In order to reduce the need for front end buffering external to the parser module, the architecture has been designed to complete the protocol recognition generation in no more than 36 cycles. Since there could be up to 12 different protocols in each to be processed, the parser module has been designed to average three cycles per protocol. This is the very worst case because a packet that has twelve levels of protocols in it will most likely be much larger than the minimum packet size. This can be used as to advantage again in the reduction of external buffering. The extractor must also complete the flow signature generation within 36 cycles to keep the system in balance and unstalled. This however can be extended if the payload copying instructions run to there maximum values.

The average packet will have between 4 and 5 levels of protocol with no encapsulations. At three cycles per protocol the PRE will use only 15 cycles to complete a packet. This means that the PRE has a typical sustained throughput of over three million packets per second.

Pattern Recognition Engine Sub-module – PRE

Highlights

The following are the highlights of the preferred implementation of the PRE:

- Scaleable protocol pattern recognition engine
- Supports from 1 to 2048 simultaneous unique protocol patterns
- At 62.5 MegaHertz can process up to 1.5 MegaPackets per second
- Accepts protocol database, the database produced by the compiler

Description

The Pattern Recognition Engine module searches it's database and the packet in order to recognize the protocols the packet contains. The database consists of 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 the **BaseOffset** from the control register to start the comparison. It loads this value into the Current Offset Pointer (COP). It then reads the byte at **BaseOffset** 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 Extractor. Finally it returns the value to add to the COP.

Database Word Definition	
Bit	**Description**
1:0	Opcode 00 Terminal Node found 01 Intermediate Node 10 Ending Terminal Node found
*	Next Lookup table * uses PAR_PRE_LU_WIDTH
*	Extractor Command * uses PAR_PRE_COM_WIDTH
*	Mask * uses PAR_PRE_MASK_WIDTH

Extractor Sub-module

Description

The Extractor cuts up (slices) the packet to build the flow signature. The Extractor module accepts commands from the Pattern Recognition Engine. Based on the command received, the Extractor either transfers data from the Parser Input Buffer to the

Parser Output Buffer or it transfers data from the Parser Input Buffer to it's internal hash generator. It contains a buffer that FIFO's up the commands. When the Pattern Recognition Engine asserts **PREDone** the Extractor completes any pending commands, transfers the hash to the Parser Output Buffer and asserts **SIDone**.

Instruction Word Definition

Bit	Description
1:0	Opcode 00 Nop 01 Move 10 Hash 11 Done
*	Source Address * uses PAR_PIB_AWIDTH
*	Destination Address * uses PAR_POB_AWIDTH
*	Length * uses PAR_SL_LEN_WIDTH

Implementation Information

The Extractor contains a byte wise barrel shifter that is used to pack data into the flow signature. A Moore finite state machine controls the execution of commands. The command comes into the Extractor and is shifted to provide an address. The Extractor uses this address to read the Extractor Instruction Database.

Extractor Instruction Database Sub-module -SID

Highlights

- Scaleable implementation
- Wraps either RAM or ROM instantiation or can be synthesized latches

Description

The Extractor Instruction Database module is a wrapper for the storage medium used to hold the pattern recognition database. Only the CPU can write this memory.

Implementation Information

The module can be synthesized or a RAM or ROM cell can be instantiated into the wrapper.

CPU Interface MUX and Control Register Sub-module - CMC

Description

The CPU Interface MUX and Control Register module controls the communication between the external CPU and the Parser. The CMC contains a MUX for the CPU read back. It also contains the control register for the Parser.

Parser Input Buffer Sub-module – PIB

Highlights

- Scaleable implementation
- Asynchronous three ported RAM
- Can be build from three separate single port RAM cells
- Wraps either RAM instantiation or can be synthesized latches
- Separate dual read and a single write interfaces

Description

The Parser Input Buffer is a wrapper for the buffer that is used to store the start of the packet. It is three ported with separate dual read and a single write interfaces. The data from the DataPort interface is stored in one of three logical or physical buffers through the write port. The Pattern Recognition Engine uses one of the read ports and the Extractor uses the other. The three interfaces never access the same third of the buffer at the same time. Each of the interfaces looks like a single buffer to the attached modules. The Parser Input Buffer controls which of the three buffers the module is controlling.

When the first packet comes in the DataPort Interface Control module writes the data into one of the three buffers. It then increments a modulo three counter to point to the next buffer. The Pattern Recognition Engine will then begin processing the packet. Finally after the Pattern Recognition Engine is finished the Extractor will get access to the buffer. In this way each of the three processes have access to a buffer and each get access to the packet in turn.

Implementation Information

The module can be synthesized or RAM cells can be instantiated into the wrapper. The instantiated RAM can be either a single three ported cell or three separate RAM cells. The Parser Input Buffer can be three separate RAM cells because the control logic will never try to read and write the same third of the buffer at the same time.

Parser Output Buffer Sub-module - POB

Highlights

- Scaleable implementation
- Asynchronous dual ported RAM
- Can be build from two separate single port RAM cells
- Wraps either RAM instantiation or can be synthesized latches
- Separate read and write interfaces

Description

The Parser Output Buffer is a wrapper for the buffer that is used to store the output of the Extractor. It is dual ported with separate read and write interfaces. The write interface is controlled by the Extractor. The read interface is controlled by the Analyzer Interface Control logic. The Parser Output Buffer maintains a pointer to the two buffers such that one buffer is controlled by the Extractor and one is controlled by the Analyzer Interface Control logic.

Implementation Information

The module can be synthesized or RAM cells can be instantiated into the wrapper. The instantiated RAM can be either a single dual ported cell or two separate RAM cells. The Parser Output Buffer can be two separate RAM cells because the control logic will never try to read and write the same half of the buffer at the same time.

DataPort Interface Control Sub-module - DPIC

Description

The DataPort Interface Control module handshakes with the external source of packets. The external device starts sending the packet to the DataPort Interface Control module by asserting **DPPacketDelim**. The transfer of data is coordinated by the **DPDataStb_N/DPReady_N** pair. If the external device decides to about the packet it can assert **DPKillPkt_N**.

Implementation Information

The Analyzer Interface Control module is implemented as a Moore type finite state machine. Each of the outputs of the state machine are registered to assure maximum setup time for the external device.

Analyzer Interface Control Sub-module -AIC

Description

The Analyzer Interface Control module handshakes with the Analyzer in order to transfer the flow signature for further processing. The Analyzer Interface Control module starts a transfer to the Analyzer by asserting **ParserKeyDelim**. It then transfers the data via the **AnalyzerReady/ParserDataAvail** handshake pair. The Analyzer Interface Control module also sends the address of the data to be sent to the Parser Output Buffer.

Implementation Information

The Analyzer Interface Control module is implemented as a Moore type finite state machine. Each of the outputs of the state machine are registered to assure maximum setup time for the Analyzer interface.

The Analyzer Module in Detail

The preferred embodiment hardware analyzer module illustrated in Fig. 11 is now described in more detail.

Highlights

Highlights of the preferred embodiment include:

- Flexible rule-based traffic classification;
- State-based tracking of traffic;
- ***Multiple*** packets for layer processing;
- Internal cache and memory controller;
- Direct high bandwidth (64 bit) memory interface;
- SGRAM/SDRAM support;
- Programmable rules/state processor;
- Selectable protocols in flows;
- Future protocols support; and
- Scalable system design.

Architectural Overview

The analyzer module preferred embodiment includes five major sub-modules with several supporting sub-modules. The major sub-modules as shown in Fig. 11 are the flow lookup/update engine, the flow insertion and deletion engine, the state processor, the cache, and the unified memory controller. Each of these sub-modules works in parallel to create and update flows.

As a flow signature enters the analyzer, the lookup engine attempts to find it in the flow database. If the flow exists, the lookup engine retrieves the flow from the cache. It then makes a decision based on the state information included in the flow entry to either send it to the state processor or not. In either case it updates the flow entry. This updating consists of adding values to counters in the flow database entry. If a flow does not exist, the state processor sends the flow signature to the flow insertion and deletion engine, which adds the flow to the database.

The state processor updates the flow based on the current state and the flow signature information. The state processor processes single and multi packet protocol recognition. It may have to search through a series of possible states to determine the flow's actual state. The result of the state processor's processing is a consolidated flow entry. For example, a PointCast session will open multiple conversations that on a packet by packet basis look like separate flows. Since each conversation is merely a sub-flow under the PointCast master flow, a single flow that consolidates all of the information for the flow is desired.

The unified memory controller can be setup to work with various configurations of SDRAM or SGRAM. It also controls the SRAM tag memory for shadowing of flow entries.

The cache is used to optimize memory bandwidth. On a typical network the packets will have a certain amount of congruity. This means that the cache can have a high hit rate.

Flow Entry Database

The Flow Entry Database consists of a series of 128 byte entries. Each entry completely describes a flow. The format and information contained in the flow is described in the PDL files. The database is organized into buckets. Each bucket contains N flow entries. N is determined by the designer. Buckets are accessed via a hash value created by the Parser based on information in the packet. This hash spreads the flows across the database and is preferably based on a hashing algorithm that has the spreading properties. This method allows fast look up of an entry while allowing for shallower buckets. The designer selects the bucket depth based on the amount of memory attached to the analyzer and the number of bits of the hash value used. For example, for 128k flow entries 16 Megabytes are required. Using a 16-bit hash gives two entries per bucket. This has been empirically shown to be more than adequate for the vast majority of cases.

Unified Flow Key Buffer - UFKB

Highlights

- Scaleable implementation

- Can be build from four separate dual port RAM cells
- Wraps either RAM instantiation or can be synthesized latches
- Separate read and write interfaces

Description

The Unified Flow Key Buffer is a wrapper for the buffers that are used to store the flow signatures from the Parser and the modified flow signatures from the Lookup and Update Engine and the State Processor. It is four ported with separate read and write interfaces. The four connections are to the Parser Interface Control, the Lookup and Update Engine, the State Processor and the Flow Insertion and Deletion Engine. In the Unified Flow Key Buffer logic hides from the interface which of the buffers is being accessed.

When the first word of the flow signature arrives from the Parser, the Lookup and Update Engine is notified. The Lookup and Update Engine places the first address it wants on the **LUEnUFKBAdd** bus and asserts **LUEnUFKBRdReq**. If the address requested is in the buffer the Unified Flow Key Buffer asserts **UFKBuLUERdy**. If not it waits for either the data to arrive or the transfer is terminated. Once the Lookup and Update Engine finishes processing the flow signature it asserts **LUEDone**. At the same time it will assert **LUEHoldBuf**. **LUEHoldBuf** tells the system that the buffer is to be sent to the State Processor.

The State Processor and Flow Insertion and Deletion Engine have similar interfaces except that the data is assumed to be already in the buffer so no ready is returned. Also Flow Insertion and Deletion Engine has no need to hold the buffer for another process so that once **FIDEDone** is asserted the buffer is freed.

Implementation Information

The module can be synthesized or RAM cells can be instantiated into the wrapper. The instantiated RAM should be four separate dual ported RAM cells.

The RAM must complete a write or read in a single cycle with simultaneous read and write to SEPARATE locations.

A block diagram of the UFKB is shown in Fig. 19.

Lookup and Update Engine - LUE

Highlights of the LUE

- Looks up flow entries
- Compares flow signature from parser to flow entries
- Updates packet count and byte count tables
- 64 bit byte count adder with early out
- Checks flow state to see if processing by the state processor is required

Description

The Lookup and Update Engine begins processing as soon as a flow signature arrives from the parser. The first transfer from the parser contains a hash value that is used as an offset into the flow entry database. The LUE checks the entry to see if it matches the flow signature by comparing the unique identification for that flow. If there is a match, the LUE updates the counters for the flow entry. The LUE also check the entry's flow state to see if the flow signature needs to be sent to the state processor.

The Lookup and Update Engine also outputs on a special data bus, two 16 bit values. One value is a word from the flow signature that can be a packet identifier or any thing else the design wants. The other is the protocol identifier for the flow. This can be programmed to output this data on every packet or only for packets that the corresponding flow is in the IDENTIFIED state.

Analyzer CPU Interface and Control - ACIC

Description

The Analyzer CPU Interface Control module controls the communication between the external CPU and the Analyzer. The ACIC contains MUX's for the CPU read back path. It also contains the control register for the Analyzer.

Flow Insertion and Deletion Engine - FIDE

Highlights

- Maintains flow entry database
- Deletes and inserts flows based on a LRU algorithm
- Builds flows from flow signature and State Processor instructions

Description

The Flow Insertion and Deletion Engine maintains the flow entry database. Flows are grouped into buckets by hash value. When a new flow needs to be inserted first the FIDE sees which of the entries

in the corresponding bucket is the oldest. It then builds the flow entry from the flow signature and State Processor instructions. Finally it places the entry in the database.

State Processor Instruction Database - SPID

Highlights

- Scaleable implementation
- Wraps either RAM or ROM instantiation or can be synthesized latches

Description

The State Processor Instruction Database module is a wrapper for the storage medium used to hold the State Processor Instruction database. Only the CPU can write this memory. The CPU interface is active if **AnalyzerEn** is active.

Implementation Information

The module can be synthesized or a RAM or ROM cell can be instantiated into the wrapper.

Unified Memory Controller - UMC

Highlights

- Supports Both SDRAM and SGRAM
- Maintains RAM refresh

Description

The Unified Memory Controller module controls the caches' access to the flow database contained in external RAM. Synchronous DRAM is controlled through a series of instructions feed to the RAM through the control pins. Synchronous DRAM requires at startup a specific series of commands for initialization. The Unified Memory Controller handles both processes thorough a state machine. Since the nature of the flow database requires random access, there is little use in attempting to keep multiple banks open. Auto-refresh is continuous when memory is not being accessed by the cache.

Implementation Information

The Unified Memory Controller module is implemented as a Moore type finite state machine. Each of the outputs of the state machine are registered to assure maximum setup time for the external device.

The Cache

Symbol

Highlights

- Fully associative
- True least recently used cache updating
- Simultaneous one write and two reads.

Description

The Cache module contains a fully associative, true LRU cache memory. Full associatively is achieved through the use of a content addressable memory (CAM). The need for a fully associative cache arises from the fact that the hash uses to generate the initial look up into the flow entry database spreads the entries pseudo randomly throughout the memory. Each hash value corresponds to a bucket containing N flow entries. N is set by the designer (see above).

The Cache can service two read transfers at one time. If there are more than two read requests active at one time the Cache services them in the order required (See Priority below).

The CAM contains the hash value associated with the corresponding bucket in the cache memory. When there is a cache hit, the CAM produces the most significant bits of the address in cache memory where the bucket is stored. The cache then accesses the cache memory at the address indicated concatenating the lower address bits provided by the requesting module. The cache then remembers that the requesting module had a cache hit and the memory location returned. This allows a cache lookup for a requesting module to occur only once per request. When the requesting module requires a different bucket, it drops then again raises its request and another CAM cycle is initiated.

The least recently used algorithm requires the CAM to also be a stack. When there is a cache hit the CAM location that produced the hit is put on the top of the stack. The other locations above the hit location are shifted down to fill in the gap. If there is a miss, the bottom location is read to determine the address in the cache memory to put the new bucket. All the locations shifted down as normally. Finally the new hash value and cache memory address are put at the top of the stack.

Priority

The Cache processes requests from the attached modules in the following order:

1 - LRU dirty write back. The Cache writes back the least recently used bucket if it is dirty so that there will always be a space for the fetching of cache misses.

2 – Lookup and Update Engine.

3 – State Processor.

4 – Flow Insertion and Deletion Engine.

5 – Analyzer CPU Interface and Control

6 – Dirty write back from LRU –1 to MRU. When there is nothing else pending the Cache writes dirty entries back to memory.

State Processor - SP

Highlights

- Flexible Rule-based Traffic Classification
- State-based Tracking of Traffic
- ***Multiple*** Packets for Layer Processing
- Programmable Rules/State Processor
- Selectable Protocols in Flows
- Future Protocols Support

Description

The State Processor module analyzes both new and existing flows in order to classify them by application. It does this by proceeding from state to state based on rules defined by the engineer. A rule is a test followed by the next state to proceed to if the test is true. The State Processor goes through each rule until the test is true or there are no more tests to perform. The State Processor starts the process by using the last protocol recognized by the Parser 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 may have to test bits, do comparisons, add or subtract to perform the test.

The State Processor Module in Detail

State Processor Top - Block Diagram

The overall top level view of the State processor is shown in Fig. 20.

Architecture

The State Processor executes its instructions from the State Processor Instruction Database (SPID) which if filled by the host CPU. The SP contains several sub blocks including a Program Counter (SPPC) a Control Block (SPCB), an ALU (SPALU), address generators and data bus Muxes to enable the movement of data from various sources to various destinations.

The two address generators, are:

a) The SP Flow Key (*i.e.,* flow signature) Address Generator that points to the UFKB and

b) The SP Flow Entry Address Generator that points to the Cache.

In addition, the State Processor incorporates four Data Muxes as follows:

a) SP ALU Data Mux A

b) SP ALU Data Mux B

c) SP UFKB Data Mux

d) SP Cache Data Mux

These muxes facilitate the movement of data within the various blocks of the State Processor and to/from the UFKB and the Cache.

Since various sub-modules of the State Processor contain memory elements such as the address generator ROMs and the Reference Memory RAM, the host must be given read and write access to these memory blocks.

Architecture (Data Flow) Block Diagram

Fig. 21 illustrates the data flow paths between the various State Processor sub modules. Data flows based on the size of the source and destination.

The internal sub-modules of the State Processor are now described.

State Processor Control Block - SPCB

The SP Control Block decodes instructions coming out of the SPID and separates them into various fields to control the State Processor. The main function of the SPCB is instruction decoding and control signal generation. There are two classes of instructions. One that are executed completely by the SPCB and one that are passed along to the

SPALU for partial or complete execution. The SP instructions are described herein below in detail.

When an instruction needs to be passed to the SPALU, the SPCB decodes the instruction and supplies the SPALU instruction code on the SPCBInst bus and asserts the SPALUGo signal.

When an instruction can be completely executed by the SPCB, the SPCB generates the appropriate control signals to the SP Program Counter, SP Address Generators and the SP Muxes in order to implement the specific move or jump instruction.

SPID Word Definition

The SPID word is a 40 bit word and is partitioned into various fields by the SPCB depending on the instruction code. The most significant 7 bits are always the SPCBInst Instruction word. The remaining 33 bits carry a different meaning based on the SPCBInst word. In some implementations, the width of the SPID may be reduced by 12 if there is no need to a move immediate instruction for 32 bit data.

SPCBData Word Definition

The SPCBData word (which is the remaining bits in the SPID word after we take out the SPID Instruction field) is partitioned into various fields depending on the accompanying SPCBInst word.

For example: The Jump, Call, Wait, WaitRJ instructions are followed by a Condition Code and a Jump Address. The Move Immediate instruction is followed by the constant value. The load Address Generator instructions are followed by the address to be loaded.

Implementation Information

The SPCB primarily takes the SPID word and brakes it up into various fields. Upon decoding the instruction field, it generates a combination of control signals from its 24 bit decode PAL. These control signals select the various muxes hat facilitate data movement and generate strobe signals that load values in various registers. New control signals can be added by widening the decode field and rearranging the PAL. The

SPCBInst is the only field that feeds into the PAL. The remaining fields of the SPID work pass through the SPCB and are directed to the other sub-modules of the State Processor.

State Processor Program Counter - SPPC

The Program Counter generates the address to the State Processor Instruction Database. It contained an Instruction Pointer (SPIP) which generates the SPID address. The instruction pointer can be incremented or loaded from a Jump Vector Multiplexer 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 SPALU.

After a Flow Signature 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.

In order to facilitate JUMP immediate instructions, the Program Counter takes an input field from the SPCB with the jump vector and loads the instruction pointer with the Jump Vector. Also, the SPALU can supply a jump vector via the SPALUData bus which in turn is loaded into the instruction pointer.

The State Processor supports "Call and Return Instructions" therefore the Program Counter block contains a two level stack. A two bit stack pointer points to the top of the stack that the Instruction Pointer is pushed to or popped from.

The SP Program Counter block contains:

The Instruction Pointer, The Flag Register (containing several bits used for conditional branching) and a Jump Vector MUX. It also contains a two level stack and a stack pointer.

The SPPC is N bits wide. This allows addressing of 2^n words in the SPID. N is defined in the AnalyzerConstants.v file by the AN_SPID_AWIDTH variable.

In addition, the Flag register holds a word supplied via the UFKB.

Implementation Information

The State Processor Instruction Pointer (SPIP) is an n bit up counter with reset, load, increment and add capability. It is clocked with the rising edge of MCLK and its output supplies the address pointing to the SPID.

Upon Reset, the SPIP is loaded with the Reset Vector.

When Instructions are executed, the SPIP is incremented at the rising edge of MCLK.

When Jump or Wait instructions are executed, the SPIP is loaded with a Jump Vector from the Jump Vector Mux.

When WaitJR (Jump Relative) instructions are executed, the relative address is added to the SPIP.

When Wait instructions are executed, the SPIP is halted until the condition code is met.

State Processor ALU - SPALU

The State Processor ALU contains all the Arithmetic, Logical and String Compare functions necessary to implement the State Processor instructions. The main blocks of the SP ALU 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.

Implementation Information

This block is implemented to be able to operate on multiple works and generates Increment and Decrement signals to the SPFK Address Generator in order to obtain new data to process.

State Processor Flow Key Address Generator - SPFKAG

The Flow Key Address Generator generates the address to where the State Processor is accessing in the Unified Flow Key Buffer.

The main blocks of the SPFKAG are:

a) The Flow Key Address Pointer Register

b) The ROM decode that generates addresses (implemented but not used for stage one)

To further illustrate the operation, consider the following example. If the UFKB contains 360 bytes organized in 64 bit words, the memory would be 45 locations of 8 bytes each (45X8=360). The address pointer needs to generate 45 addresses only. The width of the address generator would be 2^n=64 or n=6. Since we may only be interested in certain starting points in the memory field, we may only need to access say 8 or 16 locations directly and then reach the other locations by incrementing OR DECREMENTING the Address Pointer. The ROM would hold the values of these directly addressable fields. This way we save a few bits. The State Processor will be able to load the full address into the address pointer register.

The Flow Key (*i.e.*, flow signature) Pointer can perform both direct and indirect addressing. Indirect addressing is used to offset into a protocol's header. (Stage2)

Implementation Information

The SPFKAG can be loaded, incremented and decremented by the SPCB.

It can be incremented and decremented by the SPALU.

State Processor Flow Entry Address Generator - SPFEAG

The Flow Entry Address Generator provides the address where the State Processor is accessing the Flow Entry in the Cache. If a flow entry exists, the upper address bits come from the hash used to lookup the bucket in the Flow database. The middle bits come from the bucket entry found. The lower bits come from the offset the State Processor is using.

The main blocks of the SPFKAG are:

a) The Flow Key Pointer Register

b) The ROM decode that generates addresses

Implementation Information

The SPFEAG can be loaded, incremented and decremented by the SPCB.

It can be incremented and decremented by the SPALU.

State Processor UFKB Data Mux - SPMUXUFKB

The State Processor UFKB Data Mux - SPMUXUFKB selects the data source destined to the UFKB.

Implementation Information

The SP MUX UFKB multiplexes one of three sources of data into the UFKB. The three sources are: The ALU Output data bus, the lower Cache output data bus and the 32 bit SPCB Data. The select signal is a 2 bit signal.

State Processor Cache Data Mux - SPMUXCA

The State Processor Cache Data Mux - SPMUXCA – selects the data source destined to the Cache.

Implementation Information

The SP MuxCA multiplexes one of four sources of data into the Cache. The four sources are: The ALU Output data bus, the lower 32 bits of the UFKB data bus, the upper 32 bits of the UFKB data bus and the 32 bit SPCB Data. The select signal is a 2 bit signal. In order to allow for 16bit moves, the SPMUXCA incorporates two 16bit muxes that supply information to the lower and upper 16bits of the Cache.

The State Processor ALU Data Mux A - SPMUXA

Th State Processor ALU Data Mux A - SPMUXA – selects the data source destined to the UFKB.

Implementation Information

The SP ALU Mux A multiplexes one of three sources of 32 bit data into the A side of the ALU. The three sources are: The Cache data bus, the lower 32 bits of the UFKB data bus and the upper 32 bits of the UFKB data bus. The select signal is a 2 bit signal.

State Processor ALU Data Mux B - SPMUXB

The State Processor ALU Data Mux B – SPMUXB –selects the data source destined to the B side of the SP ALU.

Implementation Information

The SP ALU Mux B multiplexes one of two sources of 32 bit data into the B side of the ALU. The two sources are: The Cache data bus, and the SPCBData word. The select signal is a 1 bit signal.

State Processor Instruction Definitions

The following sections describe the instructions available in the State Processor. It should be noted that typically, no assembler is provided for the State Processor. This is because the engineer typically need not write code for this processor. The Compiler writes the code and loads it into the State Processor Instruction Database from the protocols defined in the Protocol List (PDL files).

State Processor Instruction Definition	
Instruction	**Description**
STAGE1 Instructions (a simpler implementation)	
In_Noop	No Operation
In_Wait	Wait for a condition to occur, jump absolute based on the condition
In_Call	Call a subroutine
In_Return	Return from a subroutine
In_WaitJR	Wait for a condition to occur, jump relative based on the condition
In_Jump	Jump to an immediate jump vector based on a condition

In_Move	Move Data from Location X, to Location Y
In_Load_FKAG	Load the FK Address Generator
In_Inc_FKAG	Increment the KF Address Generator
In_Dec_FKAG	Decrement the KF Address Generator
In_Load_FEAG	Load the FK Address Generator
In_Inc_FEAG	Increment the KF Address Generator
In_Dec_FEAG	Decrement the KF Address Generator
In_Set_SPDone	Set the SP Done Bit
STAGE1 ALU Instructions	
In_INC	Increment the value in the A Register
In_DEC	Decrement the value in the A Register
In_ADD	ADD Register A + Register B
In_SUB	Subtract Register A - Register B
In_AND	Bitwise OR Register A, Register B
In_OR	Bitwise OR Register A, Register B
In_XOR	Bitwise XOR Register A , Register B
In_COM	Bitwise Complement Register A
In_Simple_Compare	Compare Reg A, with Reg B. Returns a SPALU_MATCH if equal
STAGE2 ALU Instructions (more complex implementation)	
In_Compare	See if the string at a fixed location matches one in a reference string array
In_Compare_Continue	
In_Find	Find a string (or a set of strings) in a range
In_FindContinue	
In_AD2B	Convert an ASCII Decimal character to Binary
In_AD2BContinue	Convert an ASCII Decimal character to Binary
In_AH2B	Convert an ASCII Hex character to Binary

In_AH2BContinue	Convert an ASCII Hex character to Binary

The instructions are now described in more detail.

Noop

This instruction is the No Operation Instruction. No control signals are generated nor any of the condition code flags are tested.

Jump

This instruction causes the Instruction Pointer to be loaded with the address in the JumpAddress field of the State Processor Instruction Database word. This instruction is always conditional. Whether the branch is taken or not depends on the ConditionCode field in the instruction and the state of the Flag Register. If the Condition is not met, the Instruction Pointer is incremented.

Wait

This instruction causes the Instruction Pointer to be halted (loaded with the same value as before) until the condition or event that we are waiting for occurs. When the event occurs, the Instruction pointer is loaded with the address provided by the source causing the event. This instruction is always conditional.

In order to avoid being stuck at this instruction forever, one of the conditions can be a timeout which can preload the Instruction pointer with the Reset Vector.

Call

This instruction causes the Instruction Pointer to be loaded with the address in the JumpAddress field of the State Processor Instruction Database. At the same time the current address in the Instruction Pointer is pushed onto the 2 level stack.
This instruction may be made conditional Whether the call is taken (made) or not depends on the ConditionCode field in the instruction and the state of the flag register.

Return

This instruction causes the Instruction Pointer to be loaded with the address at the top of the stack. This instruction is always unconditional.

In_Set_SPDone

Set the SP Done Bit

Move

The move instruction in made up of a set of specific move instructions that deal with moving different size words from a source to a destination. These set of Move instructions have been developed to ensure the word sizes always match. There are 32 bit and 16 bit Move instructions

The Move instruction moves data from:

Immediate Data	to SP ALU B Register
Immediate Data	to Cache
Immediate Data	to UFKB
SP ALU Output	to UFKB
SP ALU Output	to Cache
Cache	to UFKB
Cache	to SP ALU A Register
Cache	to SP ALU B Register
UFKB	to Cache
UFKB	to SP ALU A Register

The execution of a MOVE instruction entails:

- The generation of the addresses to the sources and destinations (in the case of Flow Signature and Cache)
- The selection of the appropriate destination MUX.
- The generation of the appropriate Load or Write signal to the destination register or memory.

To continue the description of the instructions:

Address Generator Control Instruction

Flags Affected: UFKB_Nend – Address Generator End Count

In_Load_FKAG Load the FK Address Generator

In_Inc_FKAG Increment the KF Address Generator

In_Dec_FKAG Decrement the KF Address Generator

In_Load_FEAG Load the FK Address Generator

In_Inc_FEAG Increment the KF Address Generator

In_Dec_FEAG Decrement the KF Address Generator

In_Set_SPDone Set the SP Done Bit

STAGE 1 ALU Instructions (those in a simpler implementation)

Flags Affected: SPALU_Carry, SPALUMatch

In_INC Increment the value in the A Register. The Flags Affected: SPALU_Carry

In_DEC Decrement the value in the A Register
Flags Affected: SPALU_Carry

In_ADD ADD Register A + Register B
Flags Affected: SPALU_Carry

In_SUB Subtract Register A - Register B
Flags Affected: SPALU_Carry

In_AND Bitwise OR Register A, Register BFlags Affected:
SPALU_Carry=0

In_OR Bitwise OR Register A, Register B
Flags Affected: SPALU_Carry=0

In_XOR Bitwise XOR Register A , Register B
Flags Affected: SPALU_Carry=0

In_COM Bitwise Complement Register A
Flags Affected: SPALU_Carry=0

In_Simple_Compare

This instruction compares the contents of RegA with the contents of RegB and returns a MATCH if equal The instruction format is as follows:

STAGE 2 ALU Instructions (those in a more complex implementation)

The following is a list of the Stage2 ALU Instructions along with the instruction formats and related information.

In_Compare

This instruction provides information to the ALU-Search Engine to perform a compare operation and return a MATCH along with the matched string information. A Compare operation compares a WORD whose first character is located at a known location in the UFKB, and a known Reference String in the Reference String memory. Prior to executing this instruction, the SPUFKB address generator is loaded with the address pointing to the Target Character. Since the UFKB has multiple words in one location, an additional offset is provided which points to the exact location of the Target Character within a UFKB word. A location in the ALU Reference Memory will hold a list of reference characters to compare.

In_Compare_Continue

In_Find

This instruction provides information to the ALU-Search Engine to perform a Find operation and return a MATCH along with the matched string information and the location at which the target string was found.

The instruction format is as follows:

In_Find [Reference String Array Address], [UFKB Byte Offset], [Range]

Instruction Word Definition	
Bit	**Description**
In_Find	OpCode

N (size of Abus)	Reference String Array Address in the ALU Reference Memory. At this location, there is an array of one to four reference strings to be found. A Reference String Data Structure of the array is defined in the Reference Memory Data Structure section below. (Default N = 16)
Offset (2:0)	UFKB Byte Offset This is the offset address pointing to a byte in the selected UFKB word. The offset is used to determine which byte within the selected UFKB word is the first byte location to start the find operation. If the UFKB is 64 bits (8 bytes) this field would be 3 bits wide and point to the first target byte to start the find operation.
Range (7:0)	The Range, in number of byte, in the UFKB area to be searched. This means the number of bytes to search. If a full MATCH does not result after comparing this range, the find operation is concluded.

Reference String Memory Data Structure for FIND Operations

Bit Field	**Description**
# of Strings **(8 bits)**	**# of Strings in Array indicates the total number of strings in this array. Valid numbers are 0,1,2,3 for 1,2,3 or 4 strings.** **8 bits are allocated for future expansion and to simplify the implementation.**
Size of 1st String **(4 bits)**	This parameter indicates the size of the 1st string in bytes. The value placed here is N-1. Valid numbers are 0-F for a string as small as 1 character and as large as 0xF characters.
Size of 2nd String **(4 bits)**	This parameter indicates the size of the 2nd string in bytes. The value placed here is N-1. Valid numbers are 0-F for a string as small as 1 character and as large as 0xF characters.
Size of 3rd String **(4 bits)**	This parameter indicates the size of the 3rd string in bytes. The value placed here in N-1. Valid numbers are 0-F for a string as small as 1 character and as large as 0xF characters.
Size of 4th String **(4 bits)**	This parameter indicates the size of the 4th string in bytes. The value placed here in N-1. Valid numbers are 0-F for a string as small as 1 character and as large as 0xF characters.
String1	1 to 16 characters of string1.
String2	1 to 16 characters of string2.

String3	1 to 16 characters of string3.
String4	1 to 16 characters of string4.
Vector **(16 bits)**	This is a 16 bit vector returned to the Program Counter to point to an area in the SPID that processes the result of the FIND.

Reference String Memory Data Structure for FIND Operations (Diagram)

Input

The Reference String Array Address in the SP ALU Reference Memory. This is always a WORD location.

There can be one or more (up to four) reference strings.

The offset is used to determine the first byte location in the UFKB memory for starting the find operation.

The range specifies the field of search. I.e. how many bytes of the Flow Key (*i.e.*, flow signature) Buffer should be searched. This range is inclusive.

Output

When the search is complete, the Search Done bit is set.

The MATCH bit is set or reset based on the result of the search.

The ALU_DATA bus will hold the following information:

Jump_Vector[15:0] – this is a vector stored in the Reference String Array.

String Code[1:0] – this is the STRING CODE for the string that was found. (i.e. 0,1,2,3)

The location at which the string was found in the Flow Key Buffer is maintained. This is a combination of the UFKB word address + the byte location of the first character of the target found string.

The search is done if:

a) the first occurrence of any of the reference strings is found OR

b) there is no MATCH in the entire search range.

Consider the following example. Assume we wish to FIND a reference string in the payload area of the UFKB and search starting at byte location 5 of the payload and stop searching at byte location 100. Assume the reference string is located at location 0050h. The instruction format for this example would be as follows:

In_Load_FKAG, payload address

In_Find, 0050h, 5, 60h

The range would be 100 – 5 + 1 = 96 = 60h

Example 2

If we wish to search locations 12h to location 2Ah in the UFKB, the following instructions will be issues:

In_Load_FKAG 02H

In_Find [Reference String Address],2,19h

2Ah – 12h +1 = 19h

In_Find_Continue

This instruction follows a FIND instruction and tells the ALU-Search Engine to perform a Find operation starting from the location where the last string was found and return a MATCH along with the matched string information and the location at which the target string was found. The purpose for this instruction is to facilitate searching for a new reference string starting from the location where a previous search ended. Therefore, an offset is not provided since the Search Engine will remember the location where it finished its previous search.

The instruction format is as follows:

In_Find_Continue [Reference String Array Address], [0], [Range]

Instruction Word Definition

Bit	Description
In_Find	Opcode
N (size of Abus)	Reference String Array Address in the ALU Reference Memory. At this location, there is an array of one to four reference strings to be found. A Reference String Data Structure of the array is defined in the Reference Memory Data Structure section below. (Default N = 16)
Offset (2:0)	UFKB Byte Offset Always Zero.
Range (7:0)	The Range, in number of byte, in the UFKB area to be searched. This means the number of bytes to search. If a full MATCH does not result after comparing this range, the find operation is concluded.

As an example, assume we wish to FIND a string (String A) in the payload area of the UFKB and search starting at byte location 5 of the payload and stop searching at byte location 100. Assume the reference string (String A) is located at location 0050h. After finding the first reference string, assume we wish to continue searching for a new string (String B) in the following 30h bytes. Assume String B is located at location 0080h.

The instruction format for this example would be as follows:

In_Load_FKAG, payload address

In_Find, 0050h, 5, 60h

. . .

. . .

In_Find_Continue, 0080h, 5, 30h

The range would be 100 – 5 + 1 = 96 = 60h

ASCII Decimal to Binary

This instruction passes the location of an ASCII code string representing a decimal value. The result is the binary equivalent value.

ASCII Hex to Binary

This instruction passes the location of an ASCII code string representing a hex value. The result is the binary equivalent value.

Search Engine -- Architectural Overview

Search Engine in the preferred embodiment executes the IN_FIND and IN_FIND_CONTINUE instructions issued to the State Process ALU. The FIND Instructions searches an area of the UFKB and looks for up to four possible reference strings in the target (UFKB) area. The reference strings are stored in the ALU Reference String Memory.

The Search Engine continuously monitors the SPMuxBOut bus and SPALUGo signal to detect the In_Find and In_Find_Continue instructions. The In-Find instruction is a fresh search instruction (as explained elsewhere hereinabove in the State Processor description) whereas the In_Find_Continue is the continuation search instruction which continues a new search from the last UFKB location of the previously executed In_Find instruction. On the falling (or rising in other implmentaions) edge of SPALUGo control signal, the search engine checks SPMuxBOut bus's [31:25] bits to determine if the current command is In_Find or In_Find_Continue. The search engine assumes that the SP_Data_UFKB is setup to receive data, in word size, from UFKB through SPMUXA (see Architecture Block Diagram of the Search Engine SE_TOP in Fig. 21). Similarly, port SP_Data_RMB is setup to receive the reference string from the appropriate address of Reference String Memory.

As shown in Fig. 22, the Search Engine interface with the following blocks:

- ALU String Reference Memory – Where the reference strings are stored
- SPAUL Data Mux A – Through which the Target data is supplied (64 bits at

a time)

- SPALU Data Mux B – Through which the instruction Code is supplied
- Flow Key Address Generator – Used to increment and decrement the UFKB address
- State Processor Program Counter – Where the results are reported.

Search Engine Internal Block Diagram

Fig. 23 shows a block diagram of internal structure of the Search Engine.

Search Engine Sub Module Descriptions

The Instruction Decode Block – SE_INST

The Instruction Decode Block – SE_INST – is the Instruction Decode block which decodes the instruction code for In_Find and In_Find_Continue and starts the Search Engine upon the activation of the SPAUL_GO signal.

The Search Engine Reference Load Block – SE_LOAD

The Search Engine Reference Load – SE_LOAD – module is responsible for "priming" the reference string registers once an In_Find or In_FindContinue instruction is issued. It takes a Reference String Array from the Reference String memory and interprets it and load the reference string registers with the information.

State Machine for the Reference Memory Data Structure

The SE_LOAD is implemented as a state machine consisting of three states, the reset, idle, and the proc state. The module remains in the idle state between the reset and the issuing of the first In_Find instruction, and then between the completion of the In_Find or In_Find_Continue instruction and the next In_Find or In_Find_Continue instruction issue. When the desired instruction is issued, this module is placed in the proc state upon the assertion of the SPALUGo signal. In the proc state it first loads the first word from the starting location of the reference memory buffer (RMB), the starting location is assumed to be set up at the proper location prior to issuing of the instruction. The first word contains the number of strings to be searched, the size of each string, and the first or subsequent strings as shown in the reference memory format diagram below. Once the number of strings and the size of the strings are loaded, the loading process continues loading all of the strings. During the loading of the strings, the LOAD_KEY_DONE is negated. When the last word of the last reference string is being loaded, the LOAD_KEY signal is pulsed once indicating to the search_engine_module to start searching from the next clock cycle. The LOAD_KEY_DONE signal is asserted during the next clock cycle and the jump vector is loaded at the same time from String Reference Memory.

The Search Engine Increment Control Block – SE_INCR_CONTROL

The Search Engine Increment / Control module is responsible for incrementing the Flow Key Address Generator in order to supply new words from the UFKB to the Search Engine block. It monitors the found signals out of the Single Search Engine modules and reports results. IT also is responsible for calculating true ending address and determines the last byte to be checked in the last word based on the Range provided in the In_Find instruction. The true ending address is provided to the SE_4SEARCH module, which subsequently provides the same, to all of the four underneath search engines. This module also provides several signals, SPALU_Done, SPALU_Inc_FKAG, SPALU_Dec,FKAG, SPALU_Match, and SPALU_Data, to the rest of the system. The assertion of SPALU_Done signal indicates the search is completed. If the SPALU_Match signal is asserted at the same time then it is a successful search. The successful search also results in the SPALU_Data bus carrying the jump vector along with the search engine number which found the reference string. The longest time for the SPALU_Done to be asserted from the time the instruction is issued is N+11 clock cycles (N= number of words to be searched in the UFKB memory) +(11 clocks for pre-loading and pointer adjustment in case of successful search). In case of failed search, the UFKB address pointer will be pointing two words beyond the range. Note that it is necessary for the micro-sequencer to decrement the UFKB address pointer by two before another In_Find instruction can be issued on the same buffer. In case of a successful search, the address pointer does point to the proper word as it is adjusted before SPALU_Match is asserted. This module has three states; the reset, the idle, and the proc. Transition from the idle state to the proc state occurs when the go_ahead signal is issued. During the transition, the module checks if the instruction is In_Find or In_Find_Continue and accordingly computes the new true ending address. In case of In_Find instruction, it uses byte offset (SPMuxBOut[10:8]) and the range to compute the new ending address. In case of In_Find_Continue instruction, it uses the previous successful searches ending byte offset (foundx_byte[2:0]) and the range to compute the new ending address. The module maintains a counter internally to determine when the search is exhausted.

The 4 Search Module Block – SE_4SEARCH

The 4 Search Module Block (SE_4SEARCH module) is a wrapper that combines 4 Single Search modules in one. In the future, if more than 4 reference strings need to be compared simultaneously, this module can be easily extended. The block diagram is shown in Fig. 24.

The Single Search Module Block – SE_SSEARCH

Each of the Search Engine Single Search (SE_SSEARCH) modules performs a single reference string search. Using multiple copies of this module multiple distinct reference strings can be searched in a common source buffer. The module consists of a comparator matrix and a state machine. The matrix is capable of comparing a target string of three eight-byte words (loaded in three successive cycles, one word at a time) with a reference string up to 16 bytes long. Each of the reference string bytes is appended with a check bit, which indicates whether to check this byte, or not. If the check bit is set then the corresponding byte checking is disabled. As 64-bit words (8 bytes) are loaded into three registers in a pipelined fashion, the comparison takes place two clock cycles after they are fetched. Hence, the source (UFKB) address pointer needs to be adjusted if the search is successful. If the search is successful, the match_int signal becomes active and the position of the first byte of the reference string is placed out on the position[2:0] bus. The state machine performs several tasks every clock cycle. It consists of three states; the reset, the idle, and the process state. While in the idle state, the state machine waits for the go pulse from the SE_LOAD module. Once arrived, it switches to the process state. During the first clock cycle in the process state, if a match occurs then the position is checked against the byte offset. If the byte offset is greater then the position, then it is ignored, i.e. found is not asserted. Similarly, if it is the last word to be checked, then the end offset byte is checked with the position and the found is ignored if the position is greater then last byte to be checked in the range. Otherwise, the found signal is asserted when the match is found by the matrix module and the position is latched and forwarded to the higher level's SE_INCR_CONTROL module.

Single Search Engine Control Block – SE_CONTROL

The Search Engine Matrix Block – SE_MATRIX Single Search Engine Control Block is the state machine for the single search module.

Search Engine Matrix Block – SE_MATRIX

The Search Engine Matrix Block – SE_MATRIX is the core comparator matrix of the Search Engine Module. It consists of a reference axis and a target axis. The reference axis holds the Reference String. The target axis holds three words coming from the UFKB. When "searching starts", the matrix will resolve (or find) a reference string, up to 16 bytes long, anywhere in the target word axis. If a target string happens to cross a word boundary, the matrix will automatically find the word.

The State Processor Instructions – Discussion

In most common processing systems, the set of instructions implemented are general purpose in nature. All processing systems have a typical set of instructions related to the analysis and manipulation of the Instruction and Program Counters. These instructions include Jump, Call and Return. In addition, these same processing systems contain the appropriate instructions to analyze and manipulate registers and memory locations. These instructions include Increment, Decrement and Move, Compare and Logical manipulation.

The state processor of the preferred embodiment also includes such a basic set of standard instructions. All of the instructions and operations described above are found in the core set of instructions for our system.

However, the preferred embodiment state processor has some very specific functions that are required in order to evaluate the content of and data within packets on networks. There are four specific functions performed by the preferred embodiment state processor to meet these objectives. Two of these are specialized conversion instructions designed to interpret and transpose text elements in a specific for into a mathematical and numerical format. These instructions are AH2B (ASCII Hexadecimal to Binary) and AD2D (ASCII Decimal to Binary). These instructions are single cycle in nature. These instructions are novel and included to provide for the time sensitive nature of the functions performed by the preferred embodiment state processor.

In order to have the system make speed and meet the objective for classification, there are several special functions provided in the inventive State Processor. These functions primarily deal with seeking, locating, analyzing and evaluating sequences of strings. These strings can be either formatted or unformatted.

The primary high level instructions are the FIND and IN_FIND_CONTINUE sub-systems. These high level systems are broken down into 4 specific microcode functions. They include SE_LOAD, SE_INST, SE_INCR_CONTROL, and SE_4SEARCH.

These functions and the total system have been designed to make the State Processor capable of simultaneous searching of payload content from a packet send into the system. This enables the system to scale and meet any network speed requirements. These functions are very specialized and novel, as is their implementation and application..

The basic microcode for the instructions is implemented the following operational codes for the system Compiler. The simple instructions are Find and Find-Continue. Using both of these instructions all required string and pattern searches can be performed.

A simple example of these functions can be found in the review of the steps the state processor must go through in order to determine the application level of an HTTP stream.

A simple example of these functions can be found in the review of the steps the state processor must go through in order to determine the application level of an HTTP stream.

Once the state processor has gone through the first several packet exchanges, a flow signature, key and payload will enter the UFKB for processing by the state processor. The instruction pointer in the Cache for the flow record will point to the entry that contains the following set of CPL instructions in a binary form. Note that "--" indicates what follows is a comment. See the PDL reference Guide hereinbelow for details on syntax.

```
-- MSG Pending 1
0x8002  -- StateReferenceCode (look for URLs and User-Agents)
0x03      -- StateObjectCount (3)
0xEE00    -- StateObject (StringSearch - "LF-CR-LF") Early out LF-CR-LR
0x00       -- StateObjectOperand (Case = sensitive)
0x0384     -- StateObjectOperand (Weighting = 900 of 1000 packets)
0x00       -- StateObjectOperand (Offset = 0)
0x00       -- StateObjectOperand (LF Offset Flag = 0)
0xFF       -- StateObjectOperand (Range = 255)
0x00       -- StateObjectOperand (LF Range Flag = 0)
0x8003     -- StateObjectOperand (state = Server Reply)
0x01       -- StateObjectOperand (process next state in NEXT packet)
0xEE01    -- StateObject (StringSearch - "LF-LF") Early out LF-LF
0x00       -- StateObjectOperand (Case = sensitive)
0x0384     -- StateObjectOperand (Weighting = 900 of 1000 packets)
0x00       -- StateObjectOperand (Offset = 0)
0x00       -- StateObjectOperand (LF Offset Flag = 0)
0xFF       -- StateObjectOperand (Range = 255)
0x00       -- StateObjectOperand (LF Range Flag = 0)
0x8003     -- StateObjectOperand (state = Server Reply)
0x01       -- StateObjectOperand (process next state in NEXT packet)
0xE004    -- StateObject (StringSearch = "PCN-The Poin")
0x00       -- StateObjectOperand (Case = sensitive)
0x01F4     -- StateObjectOperand (Weighting = 500 of 1000 packets)
0x04       -- StateObjectOperand (Offset = 4)
0x01       -- StateObjectOperand (LF Offset Flag = 1)
0xFF       -- StateObjectOperand (Range = 255)
0x00       -- StateObjectOperand (LF Range Flag = 0)
0x61       -- StateObjectOperand (child = PointCast)
0x01       -- StateObjectOperand (process next state in NEXT packet)
0x8003    -- StateChildOrNextState (go here when this state complete)
0x001F    -- StateChildOrNextStatePercent (how often does this occur)
```

In order to decrease the amount of storage and utilize the cache in an effective manner, the string and byte fragments are associated with a specific hash location. In this example, the first two entries are specified by the 0xEE00 and 0xEE01 locations. These string elements are used throughout the state processor string analysis functions. They are actually incorporated into the logic of the system; they are only in the CPL for reference by the optimization system in the compiler. These strings are used to locate an early exit from a deep and complex string search.

In most HTTP messages, a specific end of line termination sequence can terminate the string search. In this example, we are using standard UNIX and DOS end of line terminators as early search termination strings. These are used in the current example for completeness.

Once a Flow Record enters the "MSG Pend 1" state, the next packet will cause the state processor to perform a string search for a group of substrings. In our simple example, the set of strings is reduced to the early termination sequences and the string "PCN-The Poin".

Notice that these operations have several options to modify the features and search types performed by the state processor search engine. Since the search system will be reviewing several strings in the same pass, each string must be weighted. The weighting is used to determine which content is loaded into the search engine comparison memory systems. The heavy weighted strings will be loaded and review first.

Also notice that each search string has offsets, end of line offsets search ranges, end of line search ranges and case sensitivity. All of these parameters are used to assist in proper loading of the search engine memories and proper order execution of the searches.

Last, notice that each search strings contains the next state to enter on a match and a selection of searching on the current or next packet in the flow. This is used to perform multiple states on the same packet in a flow or move to the next packet in the flow.

Upon entering the "MSG Pend 1" state, the search state will be loaded into the program counter of the state processor. Once this occurs, the state processor will fetch the required memory locations and begin the setup of each subsystem in the state engine. The search engine has a large number of selection muxes in order to enable several simultaneous loads of the primed memories for the search engine. In our example, the memories will be loaded with the "PCN-The Poin" binary values.

At this point, the system will begin comparing 64 bit elements, stepping through the memory by loading packet elements. This case of a packet that contains the string, the initial 64-bit pattern will create a match event. Once that event has occurred, the next 64-bit element from the "PCN-The-Poin" search string will be loaded into the search memory. The search engine will continue by review the next set of 64 bit elements within the payload of the packet memory. This process will continue until we have a match. In our case the packet will match the search string.

Once the match has occurred, the search engine has completed it's task and indicates to the state processor that the match has occurred. This match will cause the state processor to load the next state into the flow record via the cache system. Since the state is to occur on the next packet, the state processor marks the record in the cache and the entry in the UFKB as complete. The state processor finishes and moves to the next set of work without changing the actual flow signature or record pattern for a deeper application.

```
-- MSG Pend 2
0x8003  -- StateReferenceCode (look for Content-Type and Servers)
0x04     -- StateObjectCount (4)
0xEE00   -- StateObject (StringSearch - “LF-CR-LF”) Early out LF-CR-LR
0x00      -- StateObjectOperand (Case = sensitive)
0x0032    -- StateObjectOperand (Weighting = 50 of 1000 packets)
0x00      -- StateObjectOperand (Offset = 0)
0x00      -- StateObjectOperand (LF Offset Flag = 0)
0x00      -- StateObjectOperand (Range = 0)
0x00      -- StateObjectOperand (LF Range Flag = 0)
0x8003    -- StateObjectOperand (state = Server Reply)
0x01      -- StateObjectOperand (process next state in NEXT packet)
0xEE01   -- StateObject (StringSearch - “LF-LF”) Early out LF-LF
0x00      -- StateObjectOperand (Case = sensitive)
0x0032    -- StateObjectOperand (Weighting = 50 of 1000 packets)
0x00      -- StateObjectOperand (Offset = 0)
0x00      -- StateObjectOperand (LF Offset Flag = 0)
0x00      -- StateObjectOperand (Range = 0)
0x00      -- StateObjectOperand (LF Range Flag = 0)
0x8003    -- StateObjectOperand (state = Server Reply)
0x01      -- StateObjectOperand (process next state in NEXT packet)
0xE007   -- StateObject (StringSearch = “Content-Type:”)
0x00      -- StateObjectOperand (Case = sensitive)
0x0320    -- StateObjectOperand (Weighting = 800 of 1000 packets)
0x00      -- StateObjectOperand (Offset = 0)
0x00      -- StateObjectOperand (LF Offset Flag = 0)
0xFF      -- StateObjectOperand (Range = 255)
0x00      -- StateObjectOperand (LF Range Flag = 0)
0x63      -- StateObjectOperand (child = MIME)
0x00      -- StateObjectOperand (process next state in THIS packet)
0xE004   -- StateObject (StringSearch = “PCN-The Poin”)
0x00      -- StateObjectOperand (Case = sensitive)
0x01F4    -- StateObjectOperand (Weighting = 500 of 1000 packets)
0x04      -- StateObjectOperand (Offset = 4)
0x01      -- StateObjectOperand (LF Offset Flag = 1)
0xFF      -- StateObjectOperand (Range = 255)
0x00      -- StateObjectOperand (LF Range Flag = 0)
0x61      -- StateObjectOperand (child = PointCast)
0x01      -- StateObjectOperand (process next state in NEXT packet)
0x8000   -- StateChildOrNextState (where to go when this state done)
0x001F   -- StateChildOrNextStatePercent (how often does this occur)
```

The next state that occurs will be initiated by the state processor on the next packet that has a flow signature created in the UFKB. This packet will be an exchange from the server to the client. The state processor will utilize the string search engine to, again, review the content of the packet payload for key text. Once this has been completed, the final state is set in the flow record for this UFKB signature. This sets the classification for the application of the flow to a value related to "PointCast". Now the Flow is classified and no further classification is required. The state step is updated in the flow record found in the cache. All state processing for this flow is complete related to application classification.

The Cache memory

The cache memory is connected to keep a set of most-likely-to-be-accessed flow entries in the flow-entry database. The cache memory contains a fully associative, true least-recently-used cache memory. Full associatively is achieved through the use of a content addressable memory (CAM). The need for a fully associative cache arises from the fact that the hash used to generate the initial lookup into the flow-entry database spreads the flow entries pseudo-randomly throughout the memory. Each hash data value corresponds to a bucket containing N flow entries.

The cache memory can service two read transfers at once. If there are more than two read requests active at one time the cache memory services them in order. The content-addressable memory contains a hash data value associated with the corresponding bucket in the cache memory. When there is a cache hit, the content-addressable memory produces the most significant bits of the address in cache memory where the bucket is stored. The cache then accesses the cache memory at the address indicated after concatenating the lower address bits provided by the requesting module. The cache remembers that the requesting module had a cache hit and the memory location returned. Such allows a cache lookup for a requesting module to occur only once per request. When the requesting module requires a different bucket, it drops, then again raises its request and another content-addressable memory cycle is initiated. A least-recently-used (LRU) algorithm requires the content-addressable memory to also be a stack. When there is a cache hit the content-addressable memory location that produced the hit is put on the top of the stack. The other locations above the hit location are shifted down to fill in the gap. If there is a miss, the bottom location is read to determine the address in the cache memory to put in the new bucket. All locations shift down. The new hash data value and cache memory address are put at the top of the stack.

Cache System - Detailed Description

Typical prior-art cache systems are used to support expediting memory accesses to and from microprocessor systems. Because microprocessors mainly access memory in a sequential mode, a typical prior-art cache engine for these systems uses a very simple association for blocks of memory that are currently stored in the cache and their current state. This limited association enables such a prior-art cache system to aid the microprocessor in both sequential and limit random access memory requests.

While a normal microprocessor system needs to have a cache assist in mainly sequential memory accesses, the preferred hardware embodiment of the present invention has very special memory access properties. These differing requirements mainly are caused by, 1) the need to access memory by a specific hash for addressing bins and buckets and, 2) due to the high random access on a large pool of off-chip memory structures tat are used for the flow database 324.

In one aspect, the invention uses the premise that the network data itself will create the best signature and hash key in order to locate the proper flow record for managing the state and updating the associated statistics. This enhances the overall system performance. It also created an opportunity for including a novel method for rapidly accessing and managing the memory system.

The first major feature of the cache system of the preferred hardware embodiment of monitor 300 is a full association between the cached item and the random memory storage location. This type of cache, known as a fully associative cache, is novel.

This fully associative property of the inventive Cache system is achieved preferably by implementing CAMs (content addressable memories) as the core of the Cache memory addressing subsystem. This provides a good matches the nature of the information we are looking up from the Memory System. The CAM contains the hash value associated with the corresponding bucket in the cache memory. When there is a cache hit, the CAM produces the most significant bits of the address in cache memory where the bucket is stored. The CAMs are used to quickly access elements from the

Cache memory via the Hash values used to manage memory lookup and access. In addition, the hash value in the CAM and the related address create and association between the bucket and the actual memory location in the off-chip memory subsystem. In other words, the computed Hash can then be directly evaluated to see if the record is currently in the cache memory or not. If the cache does contain the value, then the memory is reported as valid with in the same cycle. This is accomplished in the preferred embodiment system by the use of fully associative cache systems that contain specialized CAM elements to pinpoint the exact direct memory address in a randomly accessed memory system.

The architecture of this Cache also enables simultaneous read by individual systems in overlapping cycles. The Cache can service two read transfers at one time. If there are more than two read requests active at one time the Cache services them in a priority order related to the timeliness requirements of the other engines in the Analyzer system. This is key to the architecture of the Analyze and creates the required environment for the system to make the speed required.

The least recently used (LRU) algorithm means that the CAM can advantageously also be a stack. When there is a cache hit the CAM location that produced the hit is put on the top of the stack. The other locations above the hit location are shifted down to fill in the gap. If there is a miss, the bottom location is read to determine the address in the cache memory to put the new bucket. All the locations shifted down as normally. Finally the new hash value and cache memory address are put at the top of the stack. CAM is being used to shift the 'most recently accessed' to the top. When an entry is in the cache, the CAM enables the system to automatically keep the most recent randomly access information. If this system were to be implemented with standard memory cells the LRU system would not be able to maintain the associations and meet speed using normal addressing methods.

The Pattern Parse and Extraction Database Format

A compressed 3-D representation is used to store the pattern parse and extraction database 308 used by the parser and the identifying information extractor.

The three dimensions of the data structure are:

1. Type identifier [0:M–1]. This is the identifier that identifies a type of protocol at a particular level. For example, 0 indicates an Ethernet frame. 64 indicates IP, 16 indicates a that the Ethernet packet is an IEEE type Ethernet packet, etc. M may be a large number, depending on how may protocols the packet parser can handle, and M may grow over time as more protocols are able to be recognized by the system.

2. Size [1:64]. The size of the field of interest within the packet, and

3. Location [1:512]. This is the offset location within the frame, expressed as a number of octets (bytes).

At each location, when data is present, the data in the form of a length, or a value, and when a value, also included are a list children (as type IDs) to search next, for each of the IDs in the list, a list of values that need to be compared to determine which child or children are to be searched, and the extraction operations to perform to build the identifying signature. Note that the size of this matrix is M by 64 by 512, which large since M may extend up to 10,000. Also, at most dimensions, there are no entries. In other words, this is a sparse matrix.

Virtual base layer is the entry point for the parser. There can be multiple entry points. For every packet that is acquired into the system, there is a header provided by the packet acquisition device that is supplying the packets into the parser, for example, a network interface card for an Ethernet LAN. The packet acquisition device would receive the packet from the network, and a mechanism in the acquisition device would know the type of network, *e.g.,* an Ethernet, and would place a header indicating this type of packet. This header is used to determine the virtual base layer entry point into the parser. Thus, the parser in addition to the packet knows the type of packet at base layer.

The zero node of the 3D structure has all the children. The parser will start at the virtual base, which may have one or more children. In the example script in virtual.pdl included herein, there is only one child, 01, indicating the Ethernet.

Initially, the search starts at the child of the virtual base, as obtained in the header supplied by the acquisition device, which in this case is ID value 01, as parsed out of the header. ID value 01 is Ethernet.

We now search through the 3D structure. The parser looks for the first entry that has a child in the location specified. The hardware supports 4 lengths searched at once in parallel.

In our case, suppose we find something at 1, 2, 12

This states that the ID value 01 (which means virtual base, which for this case, is Ethernet version 2, the only virtual child in virtual.pdl, one of the PDL files included herein). This ID value needs to have a child (the type field) examined at offset 12 which has length 2 bytes (octets).

The 2-byte "type" field is operated on by first checking to see if it is a length. It is a length if its value is less than or equal to $05DC_{16}$. This test is particular to the Ethernet packet format because there are older types (V 2) and newer types (IEEE) of Ethernet formats that differ. The system via the PDL files specifies two children – the Ethertypes, and the LLC-check. The LLC check is macro that operates to set the type length check function for this node, and to fill in a value of $05DC_{16}$ in the child of the node. While this capability is only used for Ethernet type packets, in the future other packets may end up being modified, and so this capability in the form of a macro in the PDL files enables such future packets to still be decoded. If it is a length, then we know that this is an IEEE type Ethernet frame, else, if the LENGTH operation fails, we look at the 2 byte field code, and it will be one of the codes shown in 1712 in Fig 17. For example, if the type is 0800 (Hex), then the protocol is IP. If the code is 0BAD (Hex) the protocol is VIP (VINES). To follow the example, suppose the code at 2,12 is 0800, indicating IP.

Note that when the parser operates on the data structure, the search proceeds is groups of four lengths, since the hardware presently searches up to four lengths simultaneously. So starting at

(1, 1, 1)	(1, 1, 2)	...
(1, 2, 1)	(1, 2, 2)	
(1, 3, 1)	(1, 3, 2)	
(1, 4, 1)	(1, 4, 2)	

The parser eventually gets a match of either a length operation (in the form of a maximum length) or a value. A match means the ID part of the of the matrix is populated. At (0, 2, 12) where, in the example, the match is of a value 0800 (Hex) indicating IP. The new ID (first dimension) for IP is 64. Note, the possible children are put in at compile time into the data structure. For each node, at compile time, the following information is included in the 3-D location in the 3-D data structure stored in pattern structures and extraction operations database:

a) a list children (as type IDs) to search next. For example, for an Ethernet type 2, the children are Ethertype (IP, IPX, *etc,* as shown in 1712 of Fig. 17). These children are compiled into the type codes. The code for IP is 64, that for IPX is 83, *etc.*

b) for each of the IDs in the list, a list of values that need to be compared. For example, $64{:}0800_{16}$ in the list indicates that the value to look for is 0800_{16} for the child to be type ID 64 (which is the IP protocol). $83{:}8137_{16}$ in the list indicates that the value to look for is 8137_{16} for the child to be type ID 83 (which is the IPX protocol), *etc.*

c) the extraction operations to perform to build the identifying signature for the flow. The format used is (offset, length, flow_signature_value_identifier), the flow_signature_value_identifier indicating where the extracted entry goes into in the signature, including what operations (AND, ORs, *etc.*) may need to be carried out. For example, if it is a hash key component, then operations need to be carried out to evaluate the hash key component. For example, for a type 2 Ethernet packet, the 2-byte type (1706 in Fig 17), a 1-byte hash (1708 in Fig. 17) of the type, the offset (1710 in Fig. 17) in the packet for the next level are used to form the signature, and the values for these in defining the extraction operations.

So at each stage of a search, the parser examines the packet and the 3-D structure to see if there's match. If not, the size is incremented (to maximum of 4) and then the offset is incremented. Note that in the preferred embodiment, the hardware parser is able to examine all four lengths simultaneously.

To continue with the Ethernet type-2 example, once the parser matches one of the possible children for the type, and in the example, the type is IP with a code 64, then the parser continues the search for the next level. The ID is 64, the length is unknown, and offset of known to be equal or larger than 14 bytes (12 offset for type, plus 2, the length of type), so the search of the 3-D structure commences as

(64, 1, 14)
(64, 2, 14)

and then there is a match (meaning a populated node).

Alternatively, suppose at (0, 2, 12) had a length 1211_{10}. Then this indicates this is an IEEE type Ethernet frame, which stores its type elsewhere. We now try for a new ID (that of an IEEE type Ethernet frame, type 16) and continue the search, which in this case starts at offset 14. so the search of the 3-D structure continues as

(16, 1, 14)
(16, 2, 14)

and then there is a match at (16, 2, 14) of 0800, which indicates the IP protocol at the next level, which is type 64, and the search continues, starting at (64, 1, 16).

Compression.

As noted above, the 3-D data structure is very large, and sparsely populated. For example, if 32 bytes are stored at each location, then the length is M by 64 by 512 by 32 bytes, which is M megabytes. If M = 10000, then this is about 10 gigabytes. A compressed form of storing the data structure thus is required.

One compression scheme that may be used is a modification of multi-dimensional run length encoding. An alternate is functionally equivalent: rather than have one overall 3-D table of nodes, store many smaller tables. The second scheme is used in the preferred embodiment.

The process of compression is now described. The compression is carried out by the optimizer component of the compiler. The building of the uncompressed table is first described.

The compiler first builds a table of all the links between protocols. Links consist of the connection between parent and child protocols. Each protocol can have zero or more children. If a protocol has children, a link is created that consists of the parent protocol, the child protocol, the child recognition pattern and the child recognition pattern size. The compiler first extracts child recognition patterns that are greater than two bytes long. Since there are a few of these they are handled separately. Next sub links are created for each link that has a child recognition pattern size of two. All the links are then formed into tables of 256 entries. The first step in the optimization is checking all the tables against all the other tables to find out which tables can share a table. This process creates the "folds". When a child recognition pattern is checked against a table there is always been expected fold. If the fold matches the information in the table, it is used to decide what to do next. If the fold does not match, we are finished.

The next step in the optimization is to find a minimum size for each table. The tables are then rearranged so that they fit in the minimum possible address space. At each step in the process there's no break between the parent and child protocols. This means that we can update the final tables with the information required for the slicer.

The pattern recognition engines database consists of a series of tables. Each table entry contains a node code. This node code can have four values. The first is a terminal node. A terminal node when found tells the pattern recognition engine that a protocol has been recognized. The second type of node is an intermediate node. An intermediate node means that a protocol has been partially recognized. The third type of node is a terminus node. A terminus node is used for a recognized protocol that has no children. Finally there is the null node. A null node is inserted in the table at each unused entry. That is, the "null" type node is used as an 'invalid flag' at leach 3-D location which tells us whether or the particular location (in 3D) has content, that is, a valid child recognition pattern (*i.e.*, an ID code).

Other fields in the table entry are a next table pointer, a next table length, the protocol and the fold. If the entry is a terminal or terminus node to protocol is used to index into another table. This table contains the information necessary for further processing. It contains the header length, offset, slicer command, and flags.

The slicer (also called extractor) instruction database consists of instruction, source address, destination address, and length. The slicer receives a command from the pattern recognition engine. This command is used as offset into the slicer instruction database. The instruction or Op code tells the slicer what to extract from the incoming packet and where to put it in the flow signature. Writing into certain fields of the flow signature automatically generates a hash. The instruction can also tell the slicer out to determine the connection status of certain protocols.

When a packet arrives at the parser, the virtual base has been prepended. The virtual base entry tells the packet recognition engine where to find the child recognition pattern. The pattern recognition engine then extracts child recognition patterns from packet and uses it as an address into the virtual base table. If the entry looked up by this method matches the fold value in the virtual base entry the lookup is deemed valid. The node code is then examined. If it is an intermediate node and next table field is used as the most significant bits of the address. The new fold is also extracted from the entry. The pattern recognition engine then extracts the next byte from the packet and uses it as least significant bits of the address. There is actually a little more to it then that because the size of the tables can vary. Tables can be from 2 to 256 entries in powers of two. If table is 256 bytes byte from the packet is unmodified. The table is 128 bytes the most significant bit of byte from packet is ignored. This process continues until the entire set of structures has been converted.

The system reduces the number of null nodes by first finding tables that can be shared. Tables that can be shared have no addresses in common. For example, if table 1 as entries up to address 16 and table 2 has no entries below 16 they can share a table. The fold value is used to distinguish between two types of entries. When a lookup is performed using that table, the parent protocols fold value is compared to the entries. If they match the entry is valid that parent protocol. If they do not match the entry is invalid. The second way reduce the number of null nodes is by sizing the tables. If a table has no more than 16 entries the table sizes four bits. Sixteen of these tables can be condensed into a single 256-entry table. Depending on the number of protocols with children and their child recognition patterns this method can reduce the number of entries by up to 80 percent.

The pattern recognition database is split into two parts. One part contains a single entry for each protocol. The entry consists of the slicer (extractor) command for that protocol, if there are children the first table to perform the lookup in, the size of that table, the expected fold value, the header length, child recognition pattern offset, and flags. To optimize the size of the memory areas for the data structures, the compiler sizes these fields based on the number of protocols, the number of tables, the number of folds, the maximum header length and the maximum offset. The second part of the pattern recognition database contains the tables (compressed 3-D structures) as described above.

Traffic Classification Capabilities

The invention allows for a very rich set of protocol classification and sub-classification in the process of analyzing and interpreting network traffic. In the preferred embodiment, this is accomplished by combining the maintenance of state information with a robust ability to interpret network data streams.

Without the ability to maintain state, an increasingly large amount of network traffic will be mis-classified, partially classified, or not classified at all with prior art traffic analysis and interpretation technologies. Pattern matching parser techniques used in many such technologies provide little help here given the growing complexity of today's network traffic.

One method of classification is parsing each datagram followed by interpreting assigned (or otherwise well-known) port/socket numbers to particular applications. Misclassification would then be common because of the as ephemeral nature of such ports/sockets. This has become especially noteworthy with the increasing proliferation of Web Browsers and the use of WinSock (Microsoft, Redmond, Washington). For example, BackWeb push-technology and Streamworks or VDOLive multimedia clients can use UDP ports that are either assigned to or used as defacto standards by other network applications such as Citrix, H.323 Gatekeeper, RealAudio, *etc.*

When the scope of interpretation is limited to a single packet, partial classification is a common limitation. For example, one could see TCP Port #1527 referenced in a network packet and know that is was an Oracle TNS Packet. Without having interpreted the initial Oracle TNS protocol exchange spanning multiple packets, one could not have known that it was indeed PeopleSoft running over SQL*Net running over Oracle TNS.

Another example is of partial classification is simple “IP Fragmentation”. Decoding the first fragment of an IP Datagram could easily determine that it further contained NFS over SunRPC over UDP. However, since subsequent fragments do not contain the UDP or SunRPC headers, they cannot be sub-classified for these protocols without having retained state and decoding information from the original (or first) fragment.

The inability to classify is becoming increasingly common as Network Applications use dynamic mechanisms to allocate and assign resources to various applications. There are a number of ways this can happen.

- In many cases, connections are established on a “truly” well-known port/socket of a server. The exchange on this connection serves to negotiate services requested/available and the address/port at which those services can be accessed. A second connection on the allocated/assigned address and port (almost always ephemeral) carries the bulk or volume of the data in the overall Network Session. Without the ability to interpret and analyze “data” in such allocation/assignment protocols connections, the volume traffic on the secondary connections cannot be distinguished from any other “un-interpretable” traffic. Microsoft’s Endpoint-Mapper, SunRPC’s Portmapper, and Oracle TNS are examples of such protocols.
- In other cases, available services and their locations (addresses and ports/sockets) are periodically announced. Without having interpreted and remembered the content of such announcements, traffic to/from them cannot be classified. Novell SAP and Apple’s Name Binding Protocol (NBP) are examples of such announcement-based approaches.

The art of traffic classification becomes further complicated when a multitude of the underlying challenges described above occurs for the same Network Data events. For example, NFS version 1 is transferring one of its typical 32-Kbyte blocks of data in a single IP Datagram and is hence fragmenting it (partial classification scenario). This transfer is occurring on an “ephemeral” UDP port of the server that was allocated via an initial exchange with the SunRPC Portmapper protocol (no classification scenario). Or, even worse, the “ephemeral” UDP port on the server turns out to be the same as one of the defacto standard UDP ports that "RealAudio" uses (mis-classification scenario).

Embodiments of the present invention surmount these challenges to provide accurate and thorough network traffic classification. There are many traffic in-progress traffic classification capabilities supported by aspects of the invention. The preferred embodiment of the invention also may be extended to support further sub-classifications.

Particular Protocols and Features supported

Each of the following protocols are supported. A set of PDL files may be built for any of these protocols. After compiling (and optimization), the including of the resulting databases is equivalent to having a separate "sub-engine" in the Parser/Extractor and in the Analyzer for the particular application/protocol, since when the databases compiled/optimized by using set of such PDL files for the particular application/protocol, when acting with the engine, are equivalent to a sub-engine being present.

IP/IPIP/IPIP4 Fragmentation

Fragmentation considerations address the area of partial classification. The first fragment of an IP Datagram can be decoded to determine further information on the nature of the underlying traffic contained within the packet. However, since the remaining fragments of the overall IP Datagram do not contain Transport, Session, and Application layer headers, they cannot be classified for these protocol layers without having retained state and decoding information from the original (or first) fragment. Internet fragmentation capabilities in the preferred implementation of the invention address these traffic considerations.

The analyzer component includes support for state maintenance and sub-classification retention for network packet fragments associated with the following protocols:

IP - Internet Protocol Version 4 datagram fragments

IPIP - IPIP datagram fragments Tunneled over IP

IPIP4 - IPIP4 datagram fragments Tunneled over IP

Key capabilities for these protocols include:

1. tracking fragments for their corresponding protocols;
2. passing on 1st fragments through normal decoding and state-based decoding;
3. retaining complete 1st fragment sub-classification information for datagrams which are not further classified as state based (e.g. NFS Version 2 over UDP on well-known port 2049) and applying this information to all subsequent fragments components;
4. retaining flow references for 1st fragment sub-classifications that further classify as state-based (e.g. Oracle TNS over TCP on a redirected, ephemeral port) and updating such flows for all subsequent fragment components; and
5. supporting concurrent fragmentation of data across multiple layers of Tunneling (e.g. IPIP4 fragments contained in IP fragments).

Sub-classifications: Note that these "sub-engines" don't really "classify" or "sub-classify" underlying protocols contained in fragments beyond that normally done by the standard IP Version 4 decoding of the "protocol type". They do however retain "sub-classification" information or flow references.

Support for IP Version 6 is easily added.

Microsoft Endpoint-Mapper

The Microsoft Endpoint-Mapper **actually** supports the Endpoint-Mapper protocol defined by the "*Distributed Computing Environment (DCE) 1.1 – Remote Procedure Call*" specification. The key node point in the protocol directory for this protocol, and related applications determined by its mappings, is "*endpoint-mapper*".

With "*endpoint-mapper*", connections are initially established on a well-known service port. The DCE-RPC Endpoint Mapper protocol is used on this connection to identify the target application requested by the client and set-up a second connection on an ephemeral port where the bulk of exchange and data transfer will occur with the target application.

Key capabilities for Microsoft Endpoint-Mapper include:

1. tracking connections to and exchange within the well-known Endpoint-Mapper.
2. distinguishing such "mapping" traffic from traffic on application connections subsequently "mapped".
3. detecting assignments of server application access assignments to various hosts and/or ports and creating sub-classifications for these access points.
4. classifying traffic seen on these access points:
   a) by the appropriate application under "*endpoint-mapper*", if the server application identifier in the mapping exchange is a **known** sub-application; or
   b) Minimally as "*endpoint-mapper*", if the server application is unknown.
5. allowing **known** sub-applications to be specified with respect to flow reporting with two levels of identification
   a) Level 1 – Endpoint Mapped "Application Group"
   b) Level 2 – Sub-application within the Application Group
6. supporting the "connection-oriented" mode of Endpoint-Mapper operations.

Sub-classifications: Sub-classifications under "endpoint-mapper" include the following in both the "*tcp*" and "*udp*" protocol subtrees:

endpoint-mapper	*→ dcerpc-mapper*		*(DCE RPC – Endpoint Mapping)*
	→ ms-exchange	*→ directory*	*(MS-Exchange Directory)*
		→ information-store	*(MS-Exhange Information Store)*
		→ mta	*(MS-Exchange MS-Mail MTA)*

New sub-classifications are easily added as new entries in the DCE RPC Sub-Engine's "Sub-Protocol Info" table, if the Universally Unique IDs (UUIDs) of the corresponding applications are known.

Certainly there are more applications other than MS-EXCHANGE using DCE-

RPC (also known as MS-RPC or Microsoft RPC since Microsoft adopted this RPC standard as opposed to SunRPC). As more notable applications are identified along with their assigned UUIDs, they may easily be added to the implementation, as would be clear to those in the art.

Support for the "connection-less" mode of Endpoint Mapper operation could also be implemented.

SunRPC PortMapper

The SunRPC PortMapper protocol is defined by the "RPC: Remote Procedure Call Specification Version 2 (RFC 1831)" standard. The key node point in the protocol directory for this protocol, and related applications determined by its mappings, is "*sunrpc*".

With SunRPC PortMapper, exchanges are initially performed on a well-known service port identify the target application requested by the client and set-up a subsequent ephemeral port (for use by either a connection or datagram service) where the bulk of exchange and data transfer will occur with the target application.

Key capabilities of this sub-engine include:

1. tracking exchanges with the well-known SunRPC PortMapper;
2. distinguishing such "mapping" traffic from traffic on application connections subsequently "mapped";
3. detecting assignments of server application access assignments to various hosts and/or ports and creating sub-classifications for these access points;
4. classifying traffic seen on these access points:
    a) by the appropriate application under "*sunrpc*", if the server application identifier in the mapping exchange is a **known** sub-application; or
    b) minimally as "*sunrpc*", if the server application is unknown;
5. allowing known sub-applications to be specified with respect to flow

reporting with a single levels of identification

a) Level 1 – Portmapped "Application".

Sub-classifications: Sub-classifications under "sunrpc" include the following in both the "tcp" and "udp" protocol subtrees:

sunrpc	→ *portmapper*	*(SunRPC – Port Mapping)*
	→ *rstat*	*(remote statistics)*
	→ *nfs*	*(network file service)*
	→ *ypserv*	*(yellow pages – server)*
	→ *ypbind*	*(yellow pages – bindings)*
	→ *ypupdated*	*(yellow pages – update daemon)*
	→ *ypxferd*	*(yellow pages – transfer daemon)*
	→ *mount*	*(remote file system mount)*
	→ *3270-mapper*	*(3270 terminal session mapper)*
	→ *rje-mapper*	*(remote job entry session mapper)*
	→ *nis*	*(next generation yellow pages)*
	→ *pcnfsd*	*(pcNFS daemon)*

New sub-classifications are easily added as new entries in the SunRPC Sub-Engine's "Sub-Protocol Info" table, if the SunRPC Program Number of the corresponding applications are known.

Other applications also use SunRPC, and as more such applications are identified along with their assigned SunRPC Program Numbers, they may easily be added to the implementation.

Enhancement of the SunRPC Sub-Engine to additionally support SET, UNSET, DUMP, and/or CALLIT SunRPC PortMapper primitives could be added to the implementation.

Oracle 6/7 Transparent Network Substrate (TNS)

The Transparent Network Substrate (TNS) protocol is defined by Oracle Corporation and is used as the underlying networks access framework for its Oracle Version 6 and Oracle Version 7 database product offerings. The key node points in the protocol directory for this protocol and applications determined by its mappings are "*oracl-tns*", "*oracl-tns2*", "*oracl-tns-srv*". These three node points reflect the three different "well-known" ports that serve to support initial access to Oracle TNS on Oracle Database servers. The first is a defacto, Oracle standard use. The next two access points

(TCP ports) are assigned to Oracle by IANA.

Oracle client applications initially connect to the database on a well-known, Oracle TNS service port. On this connection, they identify themselves by client host, user, and application. The Oracle Server may choose to accept the database session on this connection or "redirect" it to another ephemeral port. When redirected, a second connection to the ephemeral port will be established and where the subsequent bulk of exchange and data transfer with the database server will occur.

Key capabilities of for this application include:

1. tracking connections to and exchanges in well-known Oracle TNS port traffic;
2. learning the client application attempting to access the Oracle Database (e.g. PeopleSoft, Oracle Forms, etc.) to further classify traffic on the well-known Oracle TNS connections;
3. detecting "redirections" of connections to various hosts and/or ports and creating sub-classifications for these access points. Such "redirections" inherit the sub-classifications of the initial connections to the well-known Oracle TNS service;
4. classifying traffic to these access points is seen or when TNS sessions are "accepted" on the well-known TNS service port:
    a) by the appropriate client application under "*oracle-tns*" (or "*oracl-tns2*" or "*oracl-tns-srv*), if the client application identifier is a **known** sub-application; or
    b) minimally as "*oracle-tns*" (or "*oracl-tns2*" or "*oracl-tns-srv*), if the server application is unknown.
5. allowing known sub-applications to be specified with respect to flow reporting with two levels of identification
    a) Level 1 – Oracle client's "Application Group"
    b) Level 2 – Sub-application within the Application Group

Sub-classifications: Sub-classifications under "oracle-tns" include the following in the "tcp" subtree. Note that the same sub-classification occurs under the "oracl-tns2" and "oracl-tns-srv" nodes as well.

oracle-tns	→ *ms-odbc*	*(Microsoft ODBC)*
	→ *ms-ole*	*(Microsoft OLE)*
	→ *oracle-sqlplus*	*(Oracle SQLPlus)*
	→ *oracle-forms*	*(Oracle FORMS)*
	→ *peoplesoft*	*(PeopleSoft)*

New sub-classifications are easily added as new entries in the Oracle TNS Sub-Engine's "Sub-Protocol Info" table, if the Program Names (or names of the client programs' executables) of the corresponding client applications are known.

Further sub-classification of "PeopleSoft" may also be easily added, which would include breaking "*peoplesoft*" down into component applications.

There similarly are other native, client applications using Oracle TNS, and any such applications may easily be added by identifying such applications along with their assigned Program/Executable Names. For example, "*SAP R/3*" and "*Baan*", may be added.

The Oracle TNS sub-engine may be extended by building upon the application sub-classification capabilities presently supported. This will allow the "sub-engine" to further delve into the SQL*Net content to determine the actual client applications riding atop 4GL tools (such as Oracle FORMs) and access APIs (such as MS ODBC, and MS OLE).

H.323 Videoconferencing

H.323 is an umbrella standard, published by the International Telecommunication Union (ITU, formerly CCITT), for videoconferencing. H.323 entails one of the most complicated traffic classification challenges of today's networking protocols. This arises from its inherent multi-tier connection/data-stream architecture.

The key node points in the protocol directory for this protocol, and related applications determined by its mappings, are "*h323-host-call*" and "*h323-host-control*" for videoconference negotiation/set-up and "*rtp*" and "*rtcp*" for videoconference payload data transfer.

In H.323, connections are initially established on a well-known service port. The Q.931 protocol is used on this "H.323 Call Setup" connection to set-up a second connection on an ephemeral port. The second "H.323 Call Control" connection uses the H.245 protocol to negotiate audio and video capabilities (codecs) as well as to further set-up RTP/RTCP audio and video data streams over ephemeral UDP ports.

Key capabilities for this service include:

1. tracking connections to and exchanges on well-known H.323-host-call port (Q.931 protocol) traffic;
2. detecting assignments of H.245 access points to various hosts and/or ports and creating H.245 sub-classifications for these access points;
3. tracking connections to and exchanges with such assigned H.245 access points;
4. detecting the assignment of RTP/RTCP audio and video, UDP datastreams access points as well as the audio and video "codecs" negotiated for use on them and creating RTP/RTCP sub-classifications for these access points;
5. classifying traffic seen on these RTP/RTCP access points:
    a) by the appropriate "codec" under "*rtp*", if the negotiated codec is a **known** audio/video stream type; or
    b) minimally as "*rtp*", if the negotiated codec is unknown
6. allowing known sub-applications (audio/video datastreams) to be specified with respect to flow reporting with three levels of identification
    a) Level 1 – Datastream Class (e.g. audio, video, other…)
    b) Level 2 – Datastream Type within the Datastream Class
    c) Level 3 – Datastream Sub-Type within the Datastream Type

7. supporting the Q.931 "normal mode" of operation for "H.323 Call Setup connections".

Sub-classifications: "H.323 Call Setup" sub-classifications under "h323-host-call" include the following in the "tcp" subtree.

h323-host-call	*→ q931*	*(H.323 Call Setup)*
	→ q931-fast-start	*(H.323 Combined Setup and Control)*

"H.323 Call Control" sub-classifications under "*h323-host-control*" include the following in the "*tcp*" subtree.

h323-host-control	→ h245	(H.323 Call Control)

Audio and video datastream sub-classifications under "*rtp*" and "*rtcp*" include the following in the "*udp*" subtree:

rtcp	→			*(Audio/Video Stream Control sub-channel)*
rtp[	*→ audio*	*→ G.711*		*(Audio Transfer sub-channel)*
		→ G.722		
		→ G.728		
		→ G.729		
		→ MPEG1-audio		
		→ G.723		
		→ GSM		
	→ video	*→ H.261*	*→ QCIF*	*(Video Transfer sub-channel)*
			→ CIF	
		→ H.263	*→ SQCIF*	
			→ QCIF	
			→ CIF	
			→ 4CIF	
			→ 16CIF	
		→ MRV		

Standards for the audio stream sub-classifications indicated above are:

G.711 - 64 Kbps, 8K samples/sec, 8-bit companded PCM (A-law or µ -law), high quality, low complexity. Required for H.320 and H.323.

G.722 - ADPCM audio encode/decode (64 kbit/s, 7 kHz) .

G.723 - Speech coder at 6.3 and 5.3 Kbps data rate. Medium complexity. Required for H.324; Optional for H.323.

G.728 - 16 Kbps, LD-CELP, high quality speech coder, very high complexity. Optional for H.320 and H.323.

G.729 - 8Kbps, LD-CELP, high quality speech coder, medium complexity. G.DSVD is an interoperable subset.

GSM - Group Special Mobile -- European telephony standard, not ITU. Used by ProShare Video Conferencing software versions 1.0-1.8. 13Kbps, medium quality for voice only, low complexity.

Standards for the video stream sub-classifications indicated above are:

H.261 - Supports 352x288 (CIF or FCIF) and 176x144 (QCIF). DCT-based algorithm tuned for 2B to 6B ISDN communication. Required for H.320, H.323, and H.324.

H.263 - Much-improved derivative of H.261, tuned for POTS data rates. Mostly aimed at QCIF and Sub-QCIF (128x96 -- SQCIF). Optional for H.323 and H.324, although industry is focusing on it for POTS. Being added as an option to H.261.

MRV - Intel Indeo® video compression technology tuned for ISDN and LAN data rates.

Extensibility: New sub-classifications are easily added as new entries in the H.323 Sub-Engine's "Sub-Protocol Info" table, if the Audio/Video Capability Identifiers of the corresponding audio/video datastream are known.

There are still more audio/video datastream formats that can easily be included.

There is a mode of H.323 operation defined called "Q.931 Fast Start". In this mode, "H.323 Call Control" operations (normally performed under their own H.245 connection) are piggybacked over Q.931 in the "H.323 Call Setup" connection. The use of this mode of operation has historically been rare and infrequent in contemporary videoconferencing products. The H.323 sub-engine can easily be enhanced to support this mode of operation.

HTTP

The HTTP Protocol is the basis of common, present-day Web Browsers and has become a fundamental transport mechanism for many Internet applications. HTTP operates over TCP connections. Traditional/typical use of HTTP involves the establishment/tear-down of an individual HTTP connection for each element of exchange in a given user session activity (e.g. a web page will involve many TCP connections to effect the transfer of the various components of the activity). The key node points in the protocol directory for HTTP "*www-http*" and "*alternate-http*".

There are two ways to distinguish the nature of the higher-level, application information involved in on an HTTP connection:

- analyzing the HTTP content type; and
- interpreting of various fields in the HTTP command and responses

Key capabilities for this protocol include:

1. tracking connections to and exchanges in well-known HTTP Port traffic;
2. learning the nature of the application data being transferred or accessed to further classify traffic on such well-known HTTP connections;
3. learning the nature of the application by virtue of analyzing selected HTTP fields;
4. allowing known sub-applications to be specified with respect to flow reporting with two levels of identification:
    a) Level 1 – HTTP sub-application group (e.g. database, application, video, *etc.*)
    b) Level 2 – sub-application within the sub-application group
5. classifying HTTP traffic:
    a) by the appropriate sub-application within the sub-application group, if the sub-application identifier is **known**; or
    b) minimally by the sub-application group, if the negotiated sub-

application identifier is unknown.

Sub-classifications : Sub-classifications under "www-http" include the following in the "tcp" subtree. Note that the same sub-classification occurs under the "alternate-http" node as well.

www-http	*→ database*	*→ sybase-web-sql*		*(Sybase web.sql)*
		→ sybase-tunneled-tds		*(Sybase jConnect)*
		→ jdbc	*→ odbc-bridge*	*(JDBC-ODBC Bridge)*
			→ ibm-db2	*(IBM DB2 JDBC)*
			→ gupta-jdbc	*(Gupta SQLBase JDBC)*
			→ sybase-jdbc	*(Sybase jConnect)*
	→ application	*→ pointcast*		*(PointCast Network)*
		→ backweb		*(BackWeb)*
		→ datawindow		*(Sybase PowerBuilder)*
		→ edi-content		*(EDI)*
		→ edi-x12		*(EDI)*
		→ edifact		*(EDI)*
		→ excel		*(Microsoft Excel)*
		→ macbinhex40		*(Macintosh BINHEX)*
		→ mp3		*(MPEG-3 Audio)*
		→ mspowerpoint		*(Microsoft Powerpoint)*
		→ msword		*(Microsoft Word)*
		→ news-message-id		*(USENET News – rfc1036)*
		→ news-transmission		*(USENET News – rfc1036)*
		→ octet-stream		*(raw data, Java Applets)*
		→ oda		*(Office Document Architecture)*
		→ pdf		*(Adobe Acrobat)*
		→ postscript		*(Postscript)*
		→ powerbuilder		*(Sybase PowerBuilder)*
		→ quattro-pro		*(Lotus Quattro-Pro)*
		→ rtf		*(Rich Text Format)*
		→ sgml		*(SGML – rfc1874)*
		→ vnd-framemaker		*(Adobe FrameMaker)*
		→ vnd-lotus-1-2-3		*(Lotus 1-2-3)*
		→ vnd-lotus-approach		*(Lotus Approach)*
		→ vnd-lotus-freelance		*(Lotus Freelance Graphics)*
		→ vnd-lotus-organizer		*(Lotus Organizer)*
		→ vnd-lotus-wordpro		*(Lotus Word Pro)*
		→ vnd-mif		*(Adobe FrameMaker MIF-Format)*
		→ vnd-ms-excel		*(Microsoft Excel)*
		→ vnd-ms-powerpoint		*(Microsoft PowerPoint)*
		→ vnd-ms-project		*(Microsoft Project)*
		→ vnd-ms-word		*(Microsoft Word)*

	→ *vnd-powerbuilder*	*(Sybase PowerBuilder)*
	→ *vnd-rn-realplyer*	*(RealAudio)*
	→ *vnd-visio*	*(VISIO Graphics)*
	→ *wordperfect*	*(Corel WordPerfect)*
	→ *x-bcpio*	*(Old Unix CPIO Archive)*
	→ *x-compress*	*(Compressed Data*
	→ *x-cpio*	*(Posix-compliant CPIO Archive)*
	→ *x-csh*	*('C' Shell Program)*
	→ *x-director*	*(MacroMedia Shockwave)*
	→ *x-dvi*	*(TeX DVI Document)*
	→ *x-gtar*	*(GNU Tape Archive)*
	→ *x-gzip*	*(GNU Zip Compressed Data)*
	→ *x-javascrip*	*(Java Scripts)*
	→ *x-latex*	*(LaTeX Document)*
	→ *x-lotus-notes*	*(Lotus Notes)*
	→ *x-macbinary*	*(Macintosh Binary)*
	→ *x-mif*	*(Adobe FrameMaker MIF-Format)*
	→ *x-pncmd*	*(RealAudio)*
	→ *x-pn-realaudio*	*(RealAudio)*
	→ *x-powerpoint*	*(Microsoft Powerpoint)*
	→ *x-sh*	*(Bourne Shell Program)*
	→ *x-stuffit*	*(Macintosh StuffIt)*
	→ *x-tar*	*(Unix Tape Archive)*
	→ *x-tex*	*(TeX Document)*
	→ *x-troff*	*(TROFF Document)*
	→ *x-ustar*	*(Posix-compliant Tape Archive)*
	→ *x-zip-compressed*	*(ZIP Compressed Data)*
	→ *xpp5*	*(Microsoft Powerpoint)*
	→ *zip-archive*	*(ZIP Compressed Archive)*
	→ *x-netcdf*	*(Unidata netCDF)*
→ *audio*	→ *basic*	*(ULAW Audio Data)*
	→ *midi*	*(MIDI Audio Data)*
	→ *mpeg*	*(MPEG-2 Audio Data)*
	→ *vnd-rn-realaudio*	*(RealAudio)*
	→ *wav*	*(WAV Format Audio)*
	→ *x-aiff*	*(Apple AIFF Format Audio)*
	→ *x-midi*	*(MIDI Audio Data)*
	→ *x-mpeg*	*(MPEG-2 Audio Data)*
	→ *x-mpgurl*	*(MPEG Audio Data)*
	→ *x-pn-realaudio*	*(RealAudio)*
	→ *x-wav*	*(WAV Format Audio)*
→ *image*	→ *cgm*	*(Computer Graphics Metafile)*
	→ *g3fax*	*(Group 3 FAX)*
	→ *gif*	*(GIF Format Graphic)*
	→ *ief*	*(Image Exchange Format)*
	→ *jpeg*	*(JPEG Format Graphic)*
	→ *pict*	*(PICT Format Graphic)*

	→ png	*(Portable Network Graphics)*
	→ tiff	*(Apple TIFF Format Graphic)*
	→ vnd-rn-realflash	*(RealAudio)*
	→ vnd-rn-realpix	*(RealAudio)*
	→ x-bitmap	*(X Bitmap)*
	→ x-pixmap	*(X Pixmap)*
	→ x-quicktime	*(Apple QuickTime)*
	→ x-windowdump	*(X-Windows Dump Image)*
	→ x-xbm	*(X Bitmap)*
→ text	*→ enriched*	*(Enriched Text – rfc1896)*
	→ html	*(HTML – rfc1866)*
	→ plain	*(Plain Text)*
	→ richtext	*(RichText Format)*
	→ sgml	*(SGML – rfc1874)*
	→ tab-separated-value	*(Text with Tab Separations)*
	→ vnd-rn-text	*(RealAudio)*
	→ css	*(Cascading Style Sheet)*
→ video	*→ avi*	*(AVI Video)*
	→ mpeg	*(MPEG Video)*
	→ msvideo	*(Microsoft Media Video)*
	→ ms-video	*(Microsoft Media Video)*
	→ quicktime	*(Apple QuickTime)*
	→ vnd-rn-realvideo	*(RealAudio)*
	→ vnd-vivo	*(Vivo Acrtive Streaming Video)*
	→ x-ls-asf	*(Microsoft Media Video)*
	→ x-ls-asx	*(Microsoft Media Video)*
	→ x-mpeg	*(MPEG-Video)*
	→ x-ms-asf	*(Microsoft Media Video)*
	→ x-ms-asx	*(Microsoft Media Video)*
	→ x-msvideo	*(Microsoft Media Video)*
	→ x-sgi-movie	*(SGI MoviePlayer)*
→ x-world	*→ x-vrml*	*(VRML)*

New sub-classifications may be added to the HTTP capabilities. The following should be noted when doing so:

1. HTTP is a "text" based protocol

2. To support "minimum" execution overhead, when searching the HTTP Sub-Engine's "Sub-Protocol Info" database, a rather robust set of sequentially indexed, look-aside tables are employed.

    (a) The challenge here is to take a string from an HTTP packet (e.g.

Content Type) and match it with any one of approximately 110+ well known (as is the case with Content Type)

(b) And to do so within an embedded environment that is trying to keep up with the network packet rate at line speed.

(c) The supported search mechanism can identify a single match candidate sub-string by looking at typically no more than 3 to 5 characters of the sub-string from the HTTP packet.

3. Adding a sub-classification to the HTTP “Sub-Protocol Info” Database is simply a matter of adding a new entry if the “Content Type” or “JDBC URL Component” is known.

4. Updating and/or extending the “look-aside” tables requires extreme caution and accuracy.

Note that in these days, new “Content Types” are springing up almost every week. One feature of the invention is that as new applications are identified along with their designated Content Types, they may easily be added to the implementation.

WebNFS from Sun Microsystems, Inc., tunnels NFS file access over HTTP and is a good choice for inclusion into this sub-engine.

There are many other JDBC packages from various database manufactures and technology suppliers that are integrated with WWW. Oracle’s being the most noted at this time. As more are identified along with their designated JDBC URL Selectors, they may easily be added to the implementation.

BackWeb

BackWeb (BackWeb Technologies, Inc.) is a news/broadcast application. It may be configured to operate in either of 2 modes:

- HTTP only (see Section 3.6 above)
- UDP for access to BackWeb Servers & HTTP to access to 3rd party channels (polite mode)

BackWeb operates over UDP in what it calls its "Polite Client" mode. In this mode, BackWeb has an unusual mechanism of exchange that makes traffic in one direction very easy to see (well-known), but difficult to classify in the other direction.

The BackWeb sub-engine has been implemented specifically for BackWeb's UDP (Polite Mode) access protocol. The key node points in the protocol directory for BackWeb is "*backweb*".

Key capabilities for this protocol include:

1. tracking exchanges with BackWeb Servers in well-known BackWeb Server port traffic;
2. remembering the access points of traffic from BackWeb Clients and creating sub-classifications for these access points; and
3. classifying traffic seen on these access points:
   a) as "*backweb*"

Real-Time Streaming Protocol (RTSP)

The "Real-Time Streaming Protocol" is defined in RFC 2326. Like HTTP it is a "text" based protocol. Unlike HTTP, its principle purpose is to enable the controlled, on-demand delivery of real-time data, such as audio and video. The key node points in the protocol directory for RTSP will be "*rtsp*".

In function it acts similar to H.323's "Call Setup" and "Call Control" services, however, in a single connection on a well-known port. Ultimately, it serves to set up RTP/RTCP datastreams over UDP.

Key capabilities for this protocol include:

1. tracking exchanges with the well-known RTSP server;
2. detecting the assignment of RTP/RTCP audio and video, UDP datastreams access points as well as the audio and video "codecs" negotiated for use on them and creating RTP/RTCP sub-classifications for these access points;

3. classifying traffic seen on these RTP/RTCP access points:

   a) by the appropriate "codec" under "*rtp*", if the negotiated codec is a **known** audio/video stream type; or

   b) minimally as "*rtp*", if the negotiated codec is unknown.

4. allowing known sub-applications (audio/video datastreams) to be specified with respect to flow reporting with three levels of identification

   a) Level 1 – Datastream Class (e.g. audio, video, other…)

   b) Level 2 – Datastream Type within the Datastream Class

   c) Level 3 – Datastream Sub-Type within the Datastream Type

Sub-classifications: RTSP traffic is classified as "*rtsp*" in the "*tcp*" subtree. RTSP itself does not sub-classify any further.

New audio and video datastream sub-classifications under "*rtp*" include the following in the "*udp*" subtree.

rtp	→ *audio*	→ *1016* → *DVI4* → *L8* → *L16* → *LPC* → *MPA* → *VDVI* → *AIFF-C*	*(Audio Transfer sub-channel)*
	→ *video*	→ *CelB* → *JPEG* → *MPV* → *MP2T* → *nv*	*(Video Transfer sub-channel)*

Standards for the audio stream sub-classifications indicated above are:

1016 - frame based encoding using code-excited linear prediction (CELP) and is specified in Federal Standard FED-STD 1016

DVI4 - IMA ADPCM wave type, "IMA Recommended Practices for Enhancing

Digital Audio Compatibility in Multimedia Systems (version 3.0)"

L8 - L8 denotes linear audio data, using 8-bits of precision with an offset of 128, that is, the most negative signal is encoded as zero.

L16 - L16 denotes uncompressed audio data, using 16-bit signed representation with 65535 equally divided steps between minimum and maximum signal level, ranging from -32768 to 32767. The value is represented in two's complement notation and network byte order.

LPC - LPC designates an experimental linear predictive encoding contributed by Ron Frederick, Xerox PARC, which is based on an implementation written by Ron Zuckerman, Motorola, posted to the Usenet group comp.dsp on June 26, 1992.

MPA - MPA denotes MPEG-I or MPEG-II audio encapsulated as elementary streams. The encoding is defined in ISO standards ISO/IEC 11172-3 and 13818-3. The encapsulation is specified in work in progress.

VDVI - VDVI is a variable-rate version of DVI4, yielding speech bit rates of between 10 and 25 kb/s. It is specified for single-channel operation only.

AIFF-c - Apple Computer, "Audio interchange file format AIFF-C," Aug. 1991. (also ftp://ftp.sgi.com/sgi/aiff-c.9.26.91.ps.Z).

Standards for the **video** stream sub-classifications indicated above are:

CelB - The CELL-B encoding is a proprietary encoding proposed by Sun Microsystems. "RTP payload format of CellB video encoding," Work in Progress, Internet Engineering Task Force, Aug. 1995.

JPEG - The encoding is specified in ISO Standards 10918-1 and 10918-2.

MPV - Designates the use MPEG-I and MPEG-II video encoding elementary streams as specified in ISO Standards ISO/IEC 11172 and 13818-2, respectively.

MP2T - MP2T designates the use of MPEG-II transport streams, for either audio or video.

nv - The encoding is implemented in the program 'nv', version 4, developed at Xerox PARC

Extensibility New sub-classifications are easily added as new entries in the RTSP Sub-Engine's "Sub-Protocol Info" table, if the Payload Types of the corresponding audio/video stream are known.

Novell Service Advertising Protocol (SAP)

The Novell Service Advertising Protocol (SAP) is a protocol similar in nature to the "SUN RPC PortMapper" protocol. It is used to support the dynamic management and locating of "services" with regards to their locations (network addresses) and port assignments. The key node points in the protocol directory for Novell SAP is "*nov-sap*".

SAP uses a completely different protocol than the SUN RPC protocol PortMapper. Also, a fundamental difference from Sun RPC is that SAP periodically broadcasts services that are in its advertising database.

Key capabilities for this service include:

1. tracking SAP announcements periodically broadcast by Novell Netware servers;
2. distinguishing such "announcement" traffic from traffic on application connections subsequently "mapped";
3. detecting assignments of server application access assignments to various hosts and/or sockets and creating sub-classifications for these access points;
4. classifying traffic seen on these access points:
    a) By the appropriate application under "*nov-sap*", if the server application identifier in the announcement is a **known** sub-application.

    b) Minimally as "*nov-sap*", if the server application is unknown.
5. allowing known sub-applications to be specified with respect to flow reporting with two levels of identification:

a) Level 1 – SAP Mapped "Application Group"

b) Level 2 – Sub-application within the Application Group.

Sub-classifications under "*nov-sap*" will include the following in the "*ipx/nov-pep*" subtree.

nov-sap	→ *announce*	*(Novell SAP Announcements)*
	→ *ms-exchange*	*(Microsoft Exchange)*
	→ *sybase_sqlany*	*(Sybase SQL Anywhere)*
	→ *sybase_sqlenterprise*	*(Sybase SQL Enterprise)*
	→ *gupta-sqlbase*	*(Gupta SQLBase)*
	→ *ms-sna-server*	*(Microsoft SNA Server)*
	→ *ms-sql-server*	*(Microsoft SQL Server)*
	→ *citrix-app-server*	*(Citrix Application Server)*
	→ *citrix-app-server-nt*	*(Citrix Application Server for NT)*
	→ *hp-laserjet*	*(HP Laserjet Printer)*
	→ *advertising-print-svr*	*(Advertising Print Server)*
	→ *netware-sql-server*	*(Novell Netware SQL Server)*
	→ *remote-bridge*	*(Remote Bridge Router Service)*
	→ *bridge-server*	*(Bridge Server)*
	→ *print-queue*	*(Print Queue Server)*

New sub-classifications are easily added as new entries in the Novell SAP Sub-Engine's "Sub-Protocol Info" table, if the SAP IDs of the corresponding application are known.

MS-Media

MS-Media is a audio/video streaming, multimedia application (similar to RealAudio) from Microsoft. MS-Media may be configured to operate over UDP when transferring its payload. In this configuration, MS-Media has an unusual mechanism to allocate UDP resources for this purpose via an initial TCP connection.

The MS-Media sub-engine will be implemented specifically for MS-Media's access protocol.

Key capabilities for this service include:

1. tracking connections to and exchanges in well-known MS-Media port traffic;

2. detecting assignments of UDP access points to various hosts and/or ports and creating MS-Media sub-classifications for these access points; and

3. classifying traffic seen on these access points

    a) as “ms-media”.

Streamworks and VDOLive

Streamworks and VDOLive are multi-media, streaming applications, which transfer their payloads over UDP.

Like BackWeb, Streamworks and VDOLive employ unusual mechanisms of exchange that makes traffic one direction very easy to see (well-known), but difficult to classify in the other direction.

The BackWeb sub-engine may be expanded to further support Streamworks and VDOLive classification.

Re-using information from flows for maintaining statistics

It is advantageous to collect statistics rather than to count each and every packet. The process used in the embodiments of the invention to accumulate statistics enables specific metrics to be collected in real-time that otherwise would not be possible. Metrics related to bi-directional conversations must be maintained based on the entire flow for each exchange in the conversation. There are also several metrics that can not be acquired without a complete understanding of the state that the conversation is in when the metric is captured.

Most prior-art systems related to network traffic when the use statistics collect only end-point and end-of-session related statistics. Examples of commonly used metrics include packet counts, byte counts, session connection time, session timeouts, session and transport response times and others. All of these deal with events that can be directly related to an event in a single packet. These prior-art systems cannot collect some important performance metrics that are related to a set and sequence of packets in a network.

In another aspect of the invention, the monitor 300 provides the ability to collect metrics that are related to a sequence of events. A good example is relative jitter. Measuring the time from the end of one packet in one direction to another packet with the same signature in the same direction collects data that relates normal jitter. This type of jitter metric is good for measuring broad signal quality in a packet network. However, it is not specific to the payload or data item being transported in a cluster of packets. Using the state processing as described herein, monitor 300 can be programmed to collect the same jitter metric for a group of packets in a flow that are all related to a specific data payload. This allows the inventive system to provide metrics more focused on the type of quality related to a set of packets. This is much more useful when evaluating the performance of a system in a network than metrics related to single packets.

Specifically, the monitor system 300 can be programmed to maintain any type of metric at any point in a conversation. Also the system 300 can have the actual statistics programmed into the state at any point. This enables the monitor system to collect the standard metrics related to network usage and performance, as well as metrics related to specific states or sequences.

Some of the specific metrics that can be collected only with states are events related to a group of traffic in one direction, events related to the status of a communication sequence in one or both directions, events related to the exchange of packets for a specific application in a specific sequence. This is only a small sample of the metrics that requires an engine that can relate the state of a flow to a set of metrics.

In addition, because the monitor 300 provides greater visibility to the specific application in a conversation or flow, the monitor 300 can be programmed to collect metrics that may be specific to that type of application or service. In other word, if a flow is for an Oracle Database server, an embodiment of monitor 300 could collect the number of packets required to complete a transaction. Only with both state and application classification can this type of metric be derived from the network.

Because the monitor 300 can be programmed to collect a diverse set of metrics, the system can be used as a data source for metrics required in a number of environments. In particular, the metrics our system collects could be used to monitor and analyze the quality and performance of traffic flows related to a specific set of applications. Other implementation could include metrics related to bill and charge-back for specific traffic flow and events with the traffic flows. These are important for charging within a network system. Also, troubleshooting and capacity planning related directly to a focused application and service. The monitor system can be programmed to collect all of this type of metrics due to the ability to relate traffic to a specific point in time or point in a sequence of events.

Fig. 15 describes how the monitor system can be set up with a host processor. The host processor would do part of the analysis.

This following section describes how the monitor of the invention can be used to monitor the Quality of Service (QOS) by providing QOS Metrics.

Quality of Service Traffic Statistics (Metrics)

This next section defines the common structure that may be applied for the Quality of Service (QOS) Metrics according to one aspect of the invention. It also defines the original (or base) set of metrics that may be implemented in an embodiment of the invention to support QOS.

In summary, the QOS Metrics defined in this part of the description are broken into the following Metrics Groups:

Traffic Metrics

CSTraffic

SCTraffic

Jitter Metrics

CSJitter

SCJitter

Exchange Response Metrics

CSExchangeResponseTimeStartToStart

CSExchangeResponseTimeEndToStart

CSExchangeResponseTimeStartToEnd

SCExchangeResponseTimeStartToStart

SCExchangeResponseTimeEndToStart

SCExchangeResponseTimeStartToEnd

Transaction Response Metrics

CSTransactionResponseTimeStartToStart

CSApplicationResponseTimeEndToStart

CSApplicationResponseTimeStartToEnd

SCTransactionResponseTimeStartToStart

SCApplicationResponseTimeEndToStart

SCApplicationResponseTimeStartToEnd

Connection Metrics

ConnectionEstablishment

ConnectionGracefulTermination

ConnectionTimeoutTermination

Connection Sequence Metrics

CSConnectionRetransmissions

SCConnectionRetransmissions

CSConnectionOutOfOrders

SCConnectionOutOfOrders

Connection Window Metrics

CSConnectionWindow

SCConnectionWindow

CSConnectionFrozenWindows

SCConnectionFrozenWindows

CSConnectionClosedWindows

SCConnectionClosedWindows

QOS Metric Structure and Methods

Metrics Perspective

When dealing with time based metrics on application data packets ideally if all the timestamps and related data could be stored and forwarded for later analysis. However when faced with thousands of conversations per second on ever faster networks, storing all the data, even if compressed, would take too much processing, memory, and manager down load time to be practical.

In one aspect of the invention, statistical analysis may advantageously be applied to time based metrics for traffic analysis.

Network data is modeled as a population and not a sample. In collecting data fror processing, a population, *i.e.,* all the data, must be processed. Because of the nature of application protocols, just sampling some of the packets will not give good results. Missing just one critical packet, such as one the specified an additional port that data will be transmitted on, or what application will be run, can cause much valid data to be lost.

The time-based metrics, the statistical metrics process collects will come from examining the entire group of data, *i.e.,* the population. The population is finite. The statistical metrics process seeks only to provide information that will describe the actual data. Analysis of that data is preferably left to the management station that may run on a host (see Fig. 15).

The simplest form of representing a group of data is by frequency distributions in sub-ranges. Statistics provides inventive advantageous ways of analyzing this type of data. In the preferred embodiment, there are some rules in creating the sub-ranges. First the range needs to be known. Second a sub-range size needs to be determined. Fixed sub-range sizes are best, variable may be used if needed, however the statistics texts tend to only refer to operations of fixed size sub-ranges. This method of describing data is expensive for a statistical metrics process to implement. First the statistical metrics process is processing a great amount of data at a time, storing the data and determining the range, then the sub-ranges and then filling in the data after the fact takes too much storage and too much time. Fixing the range and sub-range sizes in the beginning can be problematical as the statistical metrics process may have to adjust the values for each of the applications it collects data on. That number can be in the thousands. Additional complexity arises in adding new protocols and even in describing the sub-ranges themselves to the management application.

In addition to frequency distribution, statistical analysis provides for measurements such a mean and standard deviation that can be obtained by summation functions on the individual data elements in a population. Also note that frequency distributions using sub-ranges, by their very nature, may introduce error that is not present by directly analysis via summation type formulas.

The metric provided by the statistical metrics process will provide data that can be used to calculate the most basic and useful statistical measurements. In the preferred embodiment, the statistical metrics process will not perform the calculations and provide the statistical measurement directly, while in other embodiments, direct measurement is provided. There are several reason why this is not preferred. First is that to find the final measurement can be expensive in terms of computation and representation. There are divisions and square roots and the measurements are expressed as floating point values. Second is that by providing the variables to the statistical functions, those variables are scaleable. It is possible to combine smaller intervals into larger ones.

An example is the arithmetic mean or average. This is the sum of the data divided by the number of data elements.

$$\overline{X} = \frac{\sum x}{N}$$

The metric provided by the statistical metrics process will provide 2 OIDs, the first the sum of the x, the second the number of elements N. The management station can perform the division to obtain the average. Given two samples, they can be combined by adding the sum of the x's and by adding the number of elements to get a combined sum and number of elements. The average formula then works just the same. Also the sum of the x and the number of element variables are used in calculating other statistical measurement values as well.

Metric Structure

The data structure elements of the metric have been chosen to maximize the amount of data available while minimizing the amount of memory needed to store the metric and minimizing the CPU processing requirement needed to generate the metric.

The metric data structure contains five unsigned integer datum.

- N count of the number of data points for the metric
- Σ X sum of all the data point values for the metric
- $\Sigma (X^2)$ sum of all the data point values squared for the metric

- X_{max} maximum data point value for the metric
- X_{min} minimum data point value for the metric
- Trend An enumerated type {increasing, flat, decreasing, unknown}

A performance metric is used to describe events over a time interval. The events, data points, can be processed immediately into the metric and do not have to be stored for later processing. For example to count the number of events in a time interval it is sufficient to increment a counter for each event, it is not necessary to cache all the events and then count them at the end of the interval. The metric is also designed to be easily scaleable in terms of combining adjacent intervals. For example if an statistical metrics process created a specific metric every 30 seconds and a user table interval was set to 60 seconds, the 60 second metric could be obtained by combining the two 30 second metrics. The following rules will be applied when combining adjacent metrics.

- N ΣN
- ΣX $\Sigma(\Sigma (X))$
- $\Sigma (X^2)$ $\Sigma(\Sigma (X^2))$
- X_{max} $MAX(X_{max})$
- X_{min} $MIN(X_{min})$
- Trend Implementation specific

The following approximates the CPU processing requirements needed to update a specific metric.

- 3 to 4 additions
- 1 multiplication
- 2 comparisons
- 3 to 6 assignments.

The metric structure gives a generic framework upon which the actual performance metrics will be defined. Each specific performance metric definition must address the specific significance, if any, given to each of the metric datum. While a specific metric definition should try to conform to the generic framework, it is ok for a metric datum to not be used, and to have no meaning, for a specific metric. In such cases the datum will default to a 0 value, or unknown in the case of the trend variable.

Trend is unique in that it is an enumerated type rather than a directly updated integer value. The reason for this is that the recommended method of generating this information is to subtract the first value of the interval from the last value of the interval. The number calculated has little value other than examining its sign to determine a crude indication of trend. It cannot be interpreted as a slope of a line fitted to the data points.

Metric Analysis

The actual meaning of a specific metric structure is determined by the definition of the specific metric. The following is a discussion of the operations and observations that can be performed on a generic metric. This means that the following may or may not apply and/or have meaning when applied to any specific metric.

The following observations and analysis techniques are not all inclusive. Rather these are the ones we have come up with at the time of writing this document.

- Number.

$$N$$

- Frequency.

$$\frac{N}{TimeInterval}$$

The time interval is the time interval specified in the control table. It is not a metric datum, but it is associated with the metric.

- Maximum

$$X_{\max}$$

- Minimum

$$X_{\min}$$

- Range

$$R = X_{\max} - X_{\min}$$

- Arithmetic Mean

$$\overline{X} = \frac{\sum X}{N}$$

- Root Mean Square

$$RMS = \sqrt{\frac{\sum\left(X^2\right)}{N}}$$

- Variance

$$\sigma^2 = \frac{\sum\left(X - \overline{X}\right)^2}{N} = \frac{\left(\sum X^2\right) - 2\overline{X}\left(\sum X\right) + N\left(\overline{X}^2\right)}{N}$$

- Standard Deviation

$$\sigma = \sqrt{\frac{\sum\left(\left(X - \overline{X}\right)^2\right)}{N}} = \sqrt{\frac{\left(\sum\left(X^2\right)\right) - 2\overline{X}\left(\sum X\right) + N\left(\overline{X}^2\right)}{N}}$$

- Trend

  There are two types of trending information. The trend between polled intervals and the trend within an interval. Trending between polled intervals is a management application function. Typically the management station would trend on the average of the reported interval. The trend within an interval is presented as an enumerated type and can easily be generated by subtracting the first value in the interval from the last and assigning trend based on the sign value.

Alternate Embodiments

One or more of the following different data elements may be included in various implementation of the metric. The following is what was considered but did not make it into the metric.

- Sum of the deltas. The trend enumeration can be based on this easy calculation. It didn't make it because it could be negative, which would have meant another mib variable to specify sign information. And the number is an ambiguous measure of slope as seen by comparing the following two series of values. The sum of the delta in both cases is 6-2 = 4.

  - Series A: 2, 6, 10, 6, 6, 6, 6, 6, 6, 6
  - Series B: 2, 2, 2, 2, 6, 10, 10, 10, 10, 6

- Sum of the absolute values of the delta values. This would provide a measurement of the overall movement within an interval. A value for the average change could be calculated. This measurement gives no indication of trend or grouping of data within the interval.

- Sum of positive delta values and sum of the negative delta values. These may not give much more useful information than the sum of the deltas and require 2 data elements to represent. Expanding each of these with an associated count and maximum would give nice information, but at a total of 6 data elements for this data alone. It is potentially expensive in terms of memory.

- The statistical measurement of skew can be obtained by adding $\Sigma(X^3)$ to the existing metric. This requires an additional multiply, and additional mib variable, and possibly overflow problems if X is sufficiently large.

- The statistical measurement of kurtosis can be obtained by adding $\Sigma(X^3)$ and $\Sigma(X^4)$ to the existing metric. This would require two additional multiplies, 2 additional mib variables, and an even larger chance of overflow is X is sufficiently large. And in this case large is really not so large.

- Data to calculate a slope of a least-squares line through the data would have taken 3 additional data elements, and two multiplies. Also in order to be scaleable to a control table interval would have required the sum of squaring of a potentially large time values causing overflow within the metric data element.

Various metrics are now described

Traffic Metrics

CSTraffic

Definition

This metric contains information about the volume of traffic measured for a given application and either a specific Client-Server Pair or a specific Server and all of its clients.

This information duplicates, somewhat, that which may be found in the standard, RMON II, AL/NL Matrix Tables. It has been included here for convenience to applications and the associated benefit of improved performance by avoiding the need to access different functional RMON areas when performing QOS Analysis.

Metric Specification

Metric	Applicability	Units	Description
N	Applicable	Packets	Count of the # of Packets from the Client(s) to the Server
Σ	Applicable	Octets	Sum total of the # of Octets in these packets from the Client(s) to the Server.
Maximum	Not Applicable		
Minimum	Not Applicable		

SCTraffic

Definition

This metric contains information about the volume of traffic measured for a given application and either a specific Client-Server Pair or a specific Server and all of its clients.

This information duplicates, somewhat, that which may be found in the standard, RMON II, AL/NL Matrix Tables. It has been included here for convenience to applications and the associated benefit of improved performance by avoiding the need to access different functional RMON areas when performing QOS Analysis.

Jitter Metrics

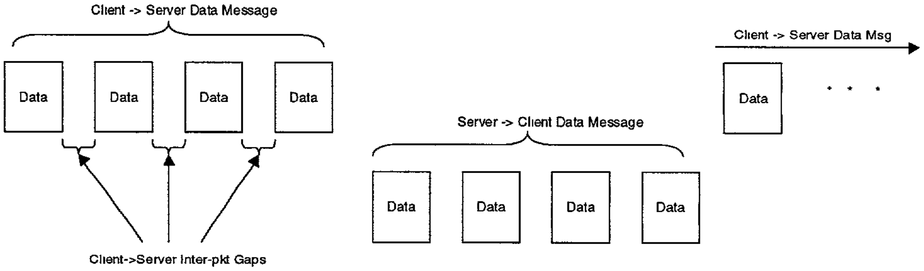
CSJitter

Definition

This metric contains information about the Jitter (e.g. Inter-packet Gap) measured for data packets for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, *CSJitter* measures the Jitter for Data Messages from the Client to the Server.

A Data Message starts with the 1st Transport Protocol Data Packet/Unit (TPDU) from the Client to the Server and is demarcated (or terminated) by 1st subsequent Data Packet in the other direction. Client to Server Inter-packet Gaps are measured between Data packets within the Message. Note that ACKnowledgements are not considered within the measurement of this metric.

Also, there is no consideration in the measurement for retransmissions or out-of-order data packets. The interval between the last packet in a Data Message from the Client to the Server and the 1st packet of the Next Message in the same direction is not interpreted as an Inter-Packet Gap.


Client -> Server Data Message
Data
Data
Data
Data
Client->Server Inter-pkt Gaps
Server -> Client Data Message
Data
Data
Data
Data
Client -> Server Data Msg
Data
. . .

Jitter Metrics

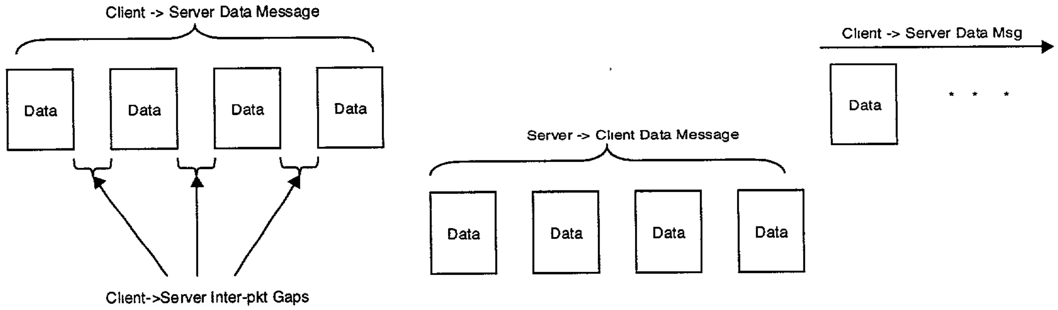
CSJitter

Definition

This metric contains information about the Jitter (e.g. Inter-packet Gap) measured for data packets for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, *CSJitter* measures the Jitter for Data Messages from the Client to the Server.

A Data Message starts with the 1st Transport Protocol Data Packet/Unit (TPDU) from the Client to the Server and is demarcated (or terminated) by 1st subsequent Data Packet in the other direction. Client to Server Inter-packet Gaps are measured between Data packets within the Message. Note that ACKnowledgements are not considered within the measurement of this metric.

Also, there is no consideration in the measurement for retransmissions or out-of-order data packets. The interval between the last packet in a Data Message from the Client to the Server and the 1st packet of the Next Message in the same direction is not interpreted as an Inter-Packet Gap.


Client -> Server Data Message
Data
Data
Data
Data
Client->Server Inter-pkt Gaps
Server -> Client Data Message
Data
Data
Data
Data
Client -> Server Data Msg
Data
. . .

Jitter Metrics

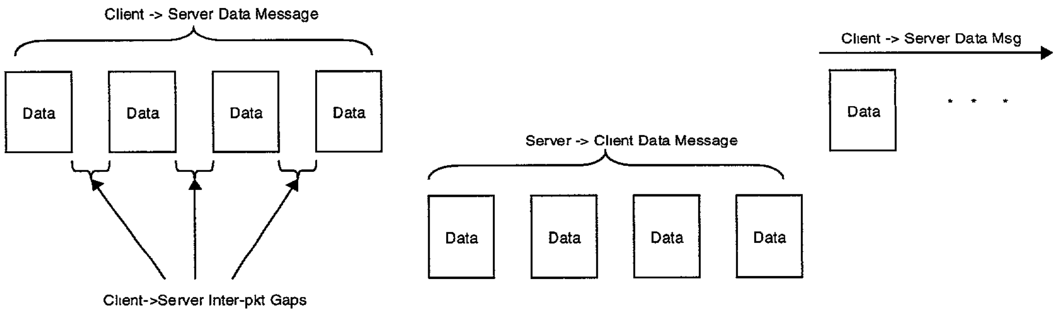
CSJitter

Definition

This metric contains information about the Jitter (e.g. Inter-packet Gap) measured for data packets for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, *CSJitter* measures the Jitter for Data Messages from the Client to the Server.

A Data Message starts with the 1st Transport Protocol Data Packet/Unit (TPDU) from the Client to the Server and is demarcated (or terminated) by 1st subsequent Data Packet in the other direction. Client to Server Inter-packet Gaps are measured between Data packets within the Message. Note that ACKnowledgements are not considered within the measurement of this metric.

Also, there is no consideration in the measurement for retransmissions or out-of-order data packets. The interval between the last packet in a Data Message from the Client to the Server and the 1st packet of the Next Message in the same direction is not interpreted as an Inter-Packet Gap.


Client -> Server Data Message
Data
Data
Data
Data
Client -> Server Data Msg
Data
. . .
Server -> Client Data Message
Data
Data
Data
Data
Client->Server Inter-pkt Gaps

Metric Specification

Metric	Applicability	Units	Description
N	Applicable	Inter-Packet Gaps	Count of the # of Inter-Packet Gaps measured for Data from the Client(s) to the Server
Σ	Applicable	uSeconds	Sum total of the Delta Times in these Inter-Packet Gaps
Maximum	Applicable	uSeconds	The maximum Delta Time of Inter-Packet Gaps measured
Minimum	Applicable	uSeconds	The minimum Delta Time of Inter-Packet Gaps measured.

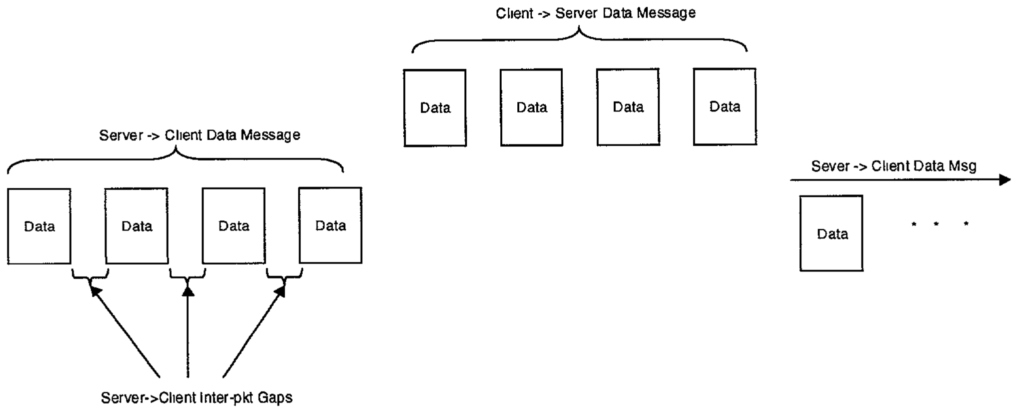
SCJitter

Definition

This metric contains information about the Jitter (e.g. Inter-packet Gap) measured for data packets for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, *SCJitter* measures the Jitter for Data Messages from the Client to the Server.

A Data Message starts with the 1st Transport Protocol Data Packet/Unit (TPDU) from the Server to the Client and is demarcated (or terminated) by 1st subsequent Data Packet in the other direction. Server to Client Inter-packet Gaps are measured between Data packets within the Message. Note that ACKnowledgements are not considered within the measurement of this metric.

Also, there is no consideration in the measurement for retransmissions or out-of-order data packets. The interval between the last packet in a Data Message from the


Client -> Server Data Message
Data
Data
Data
Data
Server -> Client Data Message
Data
Data
Data
Data
Sever -> Client Data Msg
Data
. . .
Server->Client Inter-pkt Gaps

Server to the Client and the 1st packet of the Next Message in the same direction is not interpreted as an Inter-Packet Gap.

Metric Specification

Metric	**Applicability**	**Units**	**Description**
N	Applicable	Inter-Packet Gaps	Count of the # of Inter-Packet Gaps measured for Data from the Server to the Client(s).
Σ	Applicable	uSeconds	Sum total of the Delta Times in these Inter-Packet Gaps.
Maximum	Applicable	uSeconds	The maximum Delta Time of Inter-Packet Gaps measured
Minimum	Applicable	uSeconds	The minimum Delta Time of Inter-Packet Gaps measured.

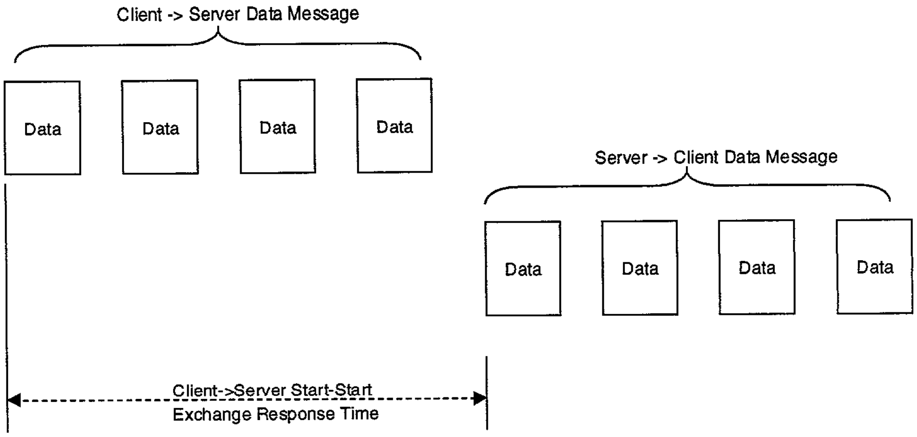
Exchange Response Metrics

CSExchangeResponseTimeStartToStart

Definition

This metric contains information about the Transport-level response time measured for data packets for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, *CSExchangeResponseTimeStartToStart* measures the response time between **start** of Data Messages from the Client to the Server and the **start** of their subsequent response Data Messages from the Server to the Client.

A Client->Server Data Message starts with the 1st Transport Protocol Data Packet/Unit (TPDU) from the Client to the Server and is demarcated (or terminated) by 1st subsequent Data Packet in the other direction. The total time between the start of the Client->Server Data Message and the start of the Server->Client Data Message is measured with this metric. Note that ACKnowledgements are not considered within the measurement of this metric.


Client -> Server Data Message
Data
Data
Data
Data
Server -> Client Data Message
Data
Data
Data
Data
Client->Server Start-Start
Exchange Response Time

Also, there is no consideration in the measurement for retransmissions or out-of-order data packets.

Metric Specification

Metric	Applicability	Units	Description
N	Applicable	Client-> Server Messages	Count of the # Client->Server Messages measured for Data Exchanges from the Client(s) to the Server
Σ	Applicable	uSeconds	Sum total of the Start-to-Start Delta Times in these Exchange Response Times
Maximum	Applicable	uSeconds	The maximum Start-to-Start Delta Time of these Exchange Response Times
Minimum	Applicable	uSeconds	The minimum Start-to-Start Delta Time of these Exchange Response Times

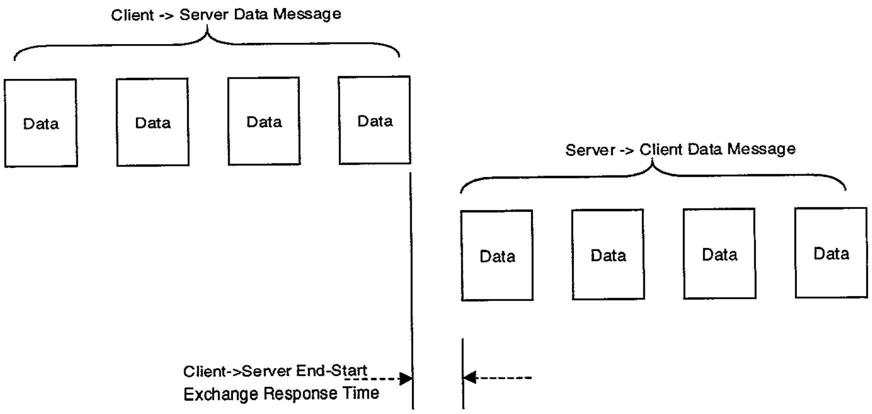
CSExchangeResponseTimeEndToStart

Definition

This metric contains information about the Transport-level response time measured for data packets for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, *CSExchangeResponseTimeEndToStart* measures the response time between **end** of Data Messages from the Client to the Server and the **start** of their subsequent response Data Messages from the Server to the Client.

A Client->Server Data Message starts with the 1st Transport Protocol Data Packet/Unit (TPDU) from the Client to the Server and is demarcated (or terminated) by

1st subsequent Data Packet in the other direction. The total time between the end of the Client->Server Data Message and the start of the Server->Client Data Message is measured with this metric. Note that ACKnowledgements are not considered within the measurement of this metric.


Client -> Server Data Message
Data
Data
Data
Data
Server -> Client Data Message
Data
Data
Data
Data
Client->Server End-Start Exchange Response Time

Also, there is no consideration in the measurement for retransmissions or out-of-order data packets.

Metric Specification

Metric	**Applicability**	**Units**	**Description**
N	Applicable	Client-> Server Messages	Count of the <u># Client->Server Messages</u> measured for Data Exchanges from the Client(s) to the Server
Σ	Applicable	uSeconds	Sum total of the <u>End-to-Start Delta Times</u> in these Exchange Response Times
Maximum	Applicable	uSeconds	The maximum <u>End-to-Start Delta Time</u> of these Exchange Response Times
Minimum	Applicable	uSeconds	The minimum <u>End-to-Start Delta Time</u> of these Exchange Response Times

CSExchangeResponseTimeStartToEnd

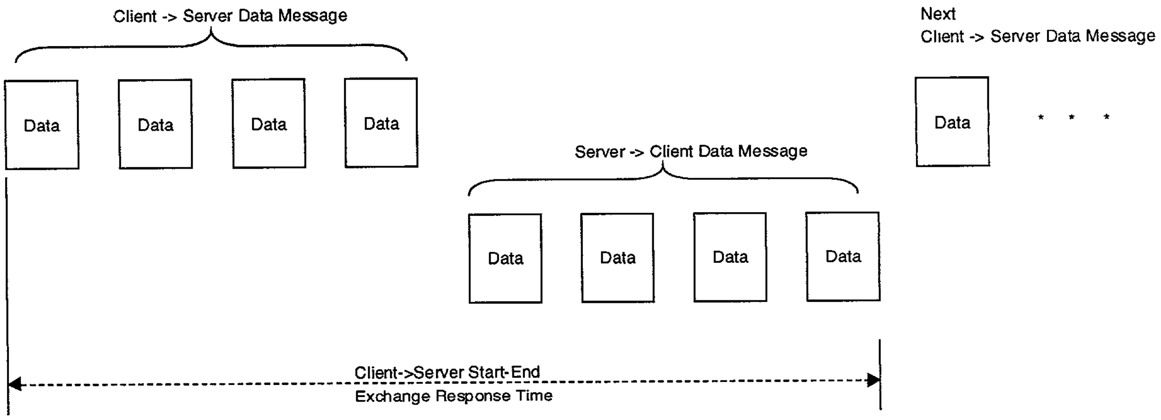
Definition

This metric contains information about the Transport-level response time measured for data packets for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, *CSExchangeResponseTimeEndToStart* measures the response time between **Start** of

Data Messages from the Client to the Server and the **End** of their subsequent response Data Messages from the Server to the Client.

A Client->Server Data Message starts with the 1st Transport Protocol Data Packet/Unit (TPDU) from the Client to the Server and is demarcated (or terminated) by 1st subsequent Data Packet in the other direction. The end of the Response Message in the other direction (e.g. from the Server to the Client) is demarcated by the last data of the Message prior to the 1st data packet of the **next** Client to Server Message. The total time between the start of the Client->Server Data Message and the end of the Server->Client Data Message is measured with this metric. Note that ACKnowledgements are not considered within the measurement of this metric.

Also, there is no consideration in the measurement for retransmissions or out-of-order data packets.



Metric Specification

Metric	Applicability	Units	Description
N	Applicable	Client-> Server Message Exchanges	Count of the # Client->Server and Server-> Client Exchange message pairs measured for Data Exchanges from the Client(s) to the Server
Σ	Applicable	uSeconds	Sum total of the Start-to-End Delta Times in these Exchange Response Times
Maximum	Applicable	uSeconds	The maximum Start-to-End Delta Time of these Exchange Response Times
Minimum	Applicable	uSeconds	The minimum Start-to-End Delta Time of these Exchange Response Times

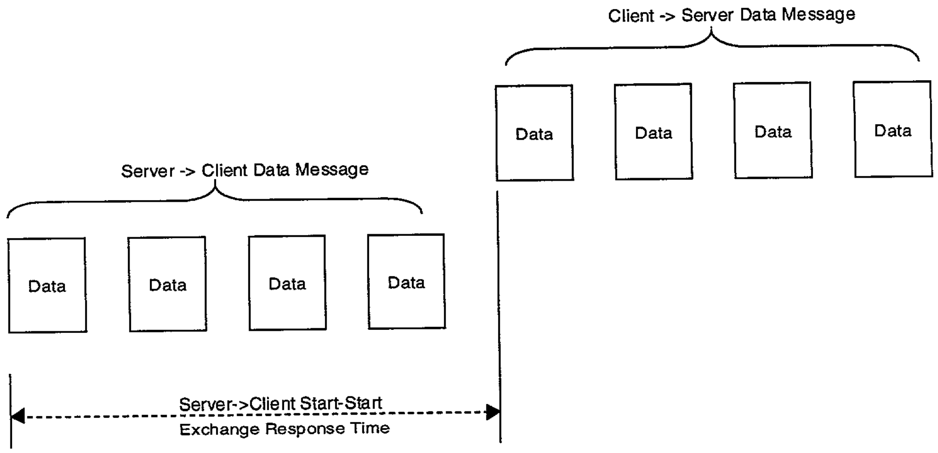
SCExchangeResponseTimeStartToStart

Definition

This metric contains information about the Transport-level response time measured for data packets for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, *SCExchangeResponseTimeStartToStart* measures the response time between **start** of Data Messages from the Server to the Client and the **start** of their subsequent response Data Messages from the Client to the Server.

A Server->Client Data Message starts with the 1st Transport Protocol Data Packet/Unit (TPDU) from the Server to the Client and is demarcated (or terminated) by 1st subsequent Data Packet in the other direction. The total time between the start of the Server->Client Data Message and the start of the Client->Sever Data Message is measured with this metric. Note that ACKnowledgements are not considered within the measurement of this metric.

Also, there is no consideration in the measurement for retransmissions or out-of-order data packets.


Client -> Server Data Message
Data
Data
Data
Data
Server -> Client Data Message
Data
Data
Data
Data
Server->Client Start-Start
Exchange Response Time

Metric Specification

Metric	Applicability	Units	Description
N	Applicable	Server-> Client Messages	Count of the # Server->Client Messages measured for Data Exchanges from the Client(s) to the Server
Σ	Applicable	uSeconds	Sum total of the Start-to-Start Delta Times in these Exchange Response Times
Maximum	Applicable	uSeconds	The maximum Start-to-Start Delta Time of these Exchange Response Times
Minimum	Applicable	uSeconds	The minimum Start-to-Start Delta Time of these Exchange Response Times

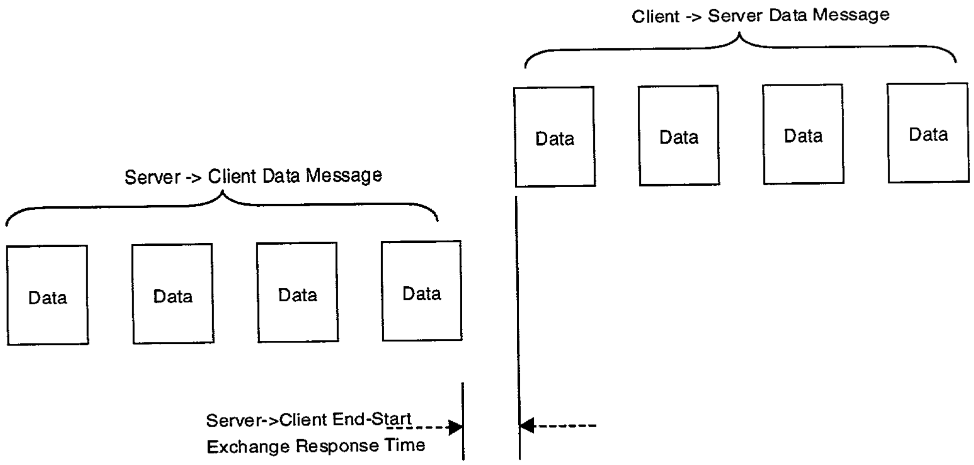
SCExchangeResponseTimeEndToStart

Definition

This metric contains information about the Transport-level response time measured for data packets for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, *SCExchangeResponseTimeEndToStart* measures the response time between **end** of Data Messages from the Server to the Client and the **start** of their subsequent response Data Messages from the Client to the Server.

A Server->Client Data Message starts with the 1st Transport Protocol Data Packet/Unit (TPDU) from the Server to the Client and is demarcated (or terminated) by

1st subsequent Data Packet in the other direction. The total time between the end of the Server->Client Data Message and the start of the Client->Server Data Message is measured with this metric. Note that ACKnowledgements are not considered within the measurement of this metric.



Also, there is no consideration in the measurement for retransmissions or out-of-order data packets.

Metric Specification

Metric	**Applicability**	**Units**	**Description**
N	Applicable	Server-> Client Messages	Count of the # Server->Client Messages measured for Data Exchanges from the Client(s) to the Server
Σ	Applicable	uSeconds	Sum total of the End-to-Start Delta Times in these Exchange Response Times
Maximum	Applicable	uSeconds	The maximum End-to-Start Delta Time of these Exchange Response Times
Minimum	Applicable	uSeconds	The minimum End-to-Start Delta Time of these Exchange Response Times

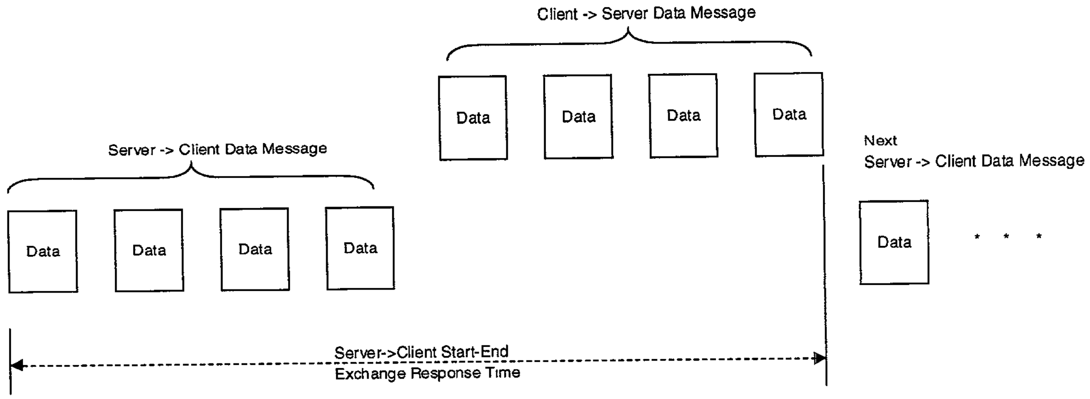
SCExchangeResponseTimeStartToEnd

Definition

This metric contains information about the Transport-level response time measured for data packets for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, *SCExchangeResponseTimeEndToStart* measures the response time between **Start** of Data Messages from the Server to the Client and the **End** of their subsequent response Data Messages from the Client to the Server.

A Server->Client Data Message starts with the 1st Transport Protocol Data Packet/Unit (TPDU) from the Server to the Client and is demarcated (or terminated) by 1st subsequent Data Packet in the other direction. The end of the Response Message in the other direction (e.g. from the Server to the Client) is demarcated by the last data of the Message prior to the 1st data packet of the **next** Server to Client Message. The total time between the start of the Server->Client Data Message and the end of the Client->Server Data Message is measured with this metric. Note that ACKnowledgements are not considered within the measurement of this metric.

Also, there is no consideration in the measurement for retransmissions or out-of-order data packets.



Metric Specification

Metric	Applicability	Units	Description
N	Applicable	Client->Svr Transaction Requests	Count of the <u># Client->Server Transaction Requests</u> measured for Application requests from the Client(s) to the Server
Σ	Applicable	uSeconds	Sum total of the <u>End-to-Start Delta Times</u> in these Application Response Times
Maximum	Applicable	uSeconds	The maximum <u>End-to-Start Delta Time</u> of these Application Response Times
Minimum	Applicable	uSeconds	The minimum <u>End-to-Start Delta Time</u> of these Application Response Times

CSApplicationResponseTimeStartToEnd

Definition

This metric contains information about the **Application-level response time** measured for application transactions for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, *CSTransactionResponseTimeStartToEnd* measures the response time between **Start** of an application transaction from the Client to the Server and the **End** of their subsequent transaction response from the Server to the Client.

A Client->Server transaction starts with the 1st Transport Protocol Data Packet/Unit (TPDU) a transaction request from the <u>Client to the Server</u> and is demarcated (or terminated) by 1st subsequent data packet of the response to the transaction request. The end of the Transaction Response in the other direction (e.g. from the Server to the Client) is demarcated by the last data of the transaction response <u>prior to the 1st data of the **next** Client to Server Transaction Request</u>. The total time between the start of the Client->Server transaction request and the end of the Server->Client transaction response is measured with this metric.

Metric Specification

Metric	Applicability	Units	Description
N	Applicable	Client-Server Message Exchanges	Count of the # Server->Client and Client-> Server Exchange message pairs measured for Data Exchanges from the Server to the Client(s)
Σ	Applicable	uSeconds	Sum total of the Start-to-End Delta Times in these Exchange Response Times
Maximum	Applicable	uSeconds	The maximum Start-to-End Delta Time of these Exchange Response Times
Minimum	Applicable	uSeconds	The minimum Start-to-End Delta Time of these Exchange Response Times

Transaction Response Metrics

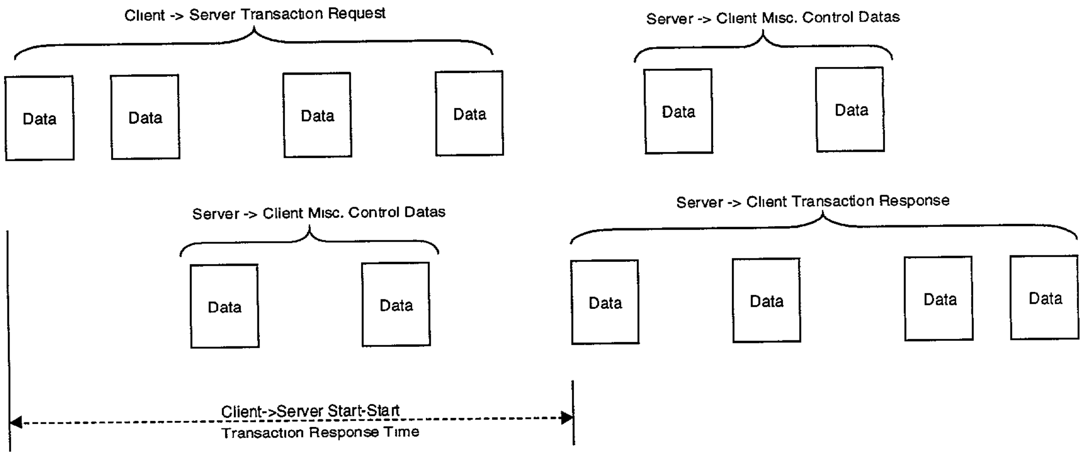
CSTransactionResponseTimeStartToStart

Definition

This metric contains information about the **Application-level response time** measured for application transactions for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, *CSTransactionResponseTimeStartToStart* measures the response time between **start** of an application transaction from the Client to the Server and the **start** of their subsequent transaction response from the Server to the Client.

A Client->Server transaction starts with the 1st Transport Protocol Data Packet/Unit (TPDU) of a transaction request from the Client to the Server and is demarcated (or terminated) by 1st subsequent data packet of the response to the transaction request. The total time between the start of the Client->Server transaction request and the start of the actual transaction response from the Server->Client is measured with this metric.

This metric is considered a “best-effort” measurement. Systems implementing this metric should make a “best-effort” to demarcate the start and end of requests and responses with the specific application’s definition of a logical transaction. The lowest level of support for this metric would make this metric the equivalent of *CSExchangeResponseTimeStartToStart.*


Client -> Server Transaction Request
Data
Data
Data
Data
Server -> Client Misc. Control Datas
Data
Data
Server -> Client Misc. Control Datas
Data
Data
Server -> Client Transaction Response
Data
Data
Data
Data
Client->Server Start-Start
Transaction Response Time

Metric Specification

Metric	Applicability	Units	Description
N	Applicable	Client->Svr Transaction Requests	Count of the # Client->Server Transaction Requests measured for Application requests from the Client(s) to the Server
Σ	Applicable	uSeconds	Sum total of the Start-to-Start Delta Times in these Application Response Times
Maximum	Applicable	uSeconds	The maximum Start-to-Start Delta Time of these Application Response Times
Minimum	Applicable	uSeconds	The minimum Start-to-Start Delta Time of these Application Response Times

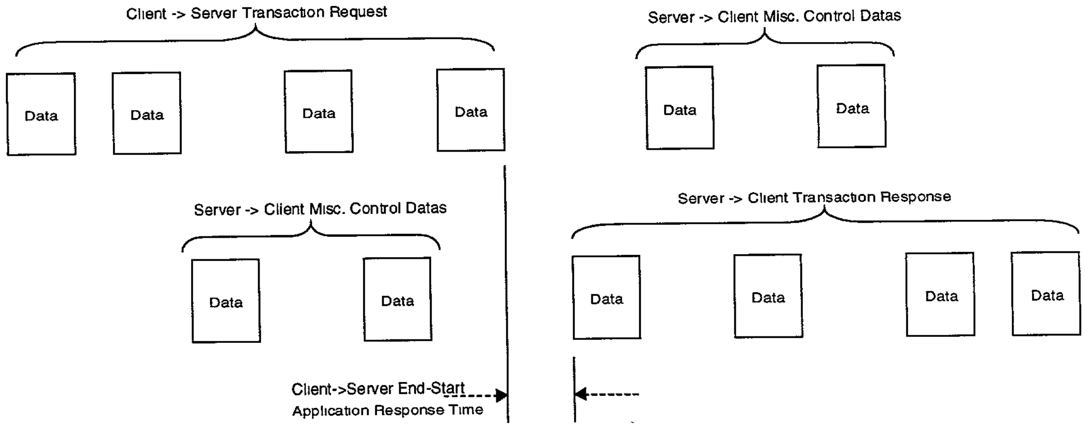
CSApplicationResponseTimeEndToStart

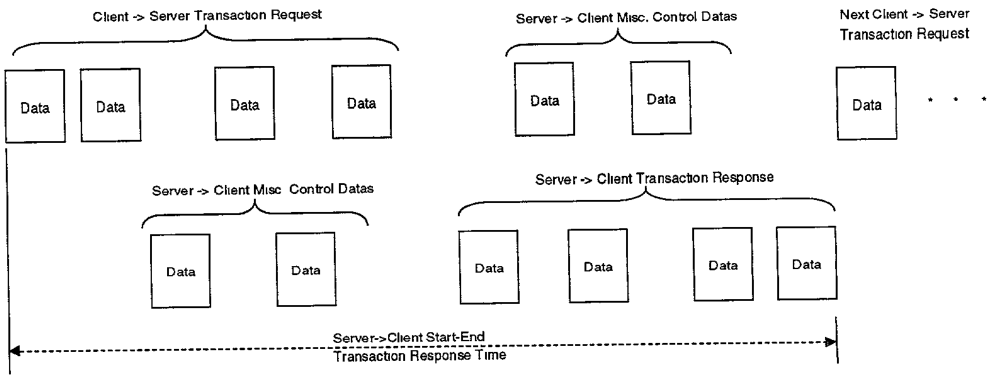
Definition

This metric contains information about the **Application-level response time** measured for application transactions for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, *CSApplicationResponseTimeEndToStart* measures the response time between **end** of an application transaction from the Client to the Server and the **start** of their subsequent transaction response from the Server to the Client.

A Client->Server transaction starts with the 1st Transport Protocol Data Packet/Unit (TPDU) of a transaction request from the Client to the Server and is demarcated (or terminated) by 1st subsequent data packet of the response to the transaction request The total time between the end of the Client->Server transaction request and the start of the actual transaction response from the Server->Client is measured with this metric

This metric is considered a "best-effort" measurement. Systems implementing this metric should make a "best-effort" to demarcate the start and end of requests and responses with the specific application's definition of a logical transaction. The lowest level of support for this metric would make this metric the equivalent of *CSExchangeResponseTimeEndToStart.*


Client -> Server Transaction Request
Data
Data
Data
Data
Server -> Client Misc. Control Datas
Data
Data
Server -> Client Misc. Control Datas
Data
Data
Server -> Client Transaction Response
Data
Data
Data
Data
Client->Server End-Start
Application Response Time


Client -> Server Transaction Request
Server -> Client Misc. Control Datas
Next Client -> Server Transaction Request
Data
Data
Data
Data
Data
Data
Data
. . .
Server -> Client Misc Control Datas
Server -> Client Transaction Response
Data
Data
Data
Data
Data
Data
Server->Client Start-End Transaction Response Time

This metric is considered a "best-effort" measurement. Systems implementing this metric should make a "best-effort" to demarcate the start and end of requests and responses with the specific application's definition of a logical transaction. The lowest level of support for this metric would make this metric the equivalent of *CSExchangeResponseTimeStartToEnd.*

Metric Specification

Metric	Applicability	Units	Description
N	Applicable	Client-> Server Transactions	Count of the # Client<->Server request/response pairs measured for transactions from the Client(s) to the Server
Σ	Applicable	uSeconds	Sum total of the Start-to-End Delta Times in these Application Response Times
Maximum	Applicable	uSeconds	The maximum Start-to-End Delta Time of these Application Response Times
Minimum	Applicable	uSeconds	The minimum Start-to-End Delta Time of these Application Response Times

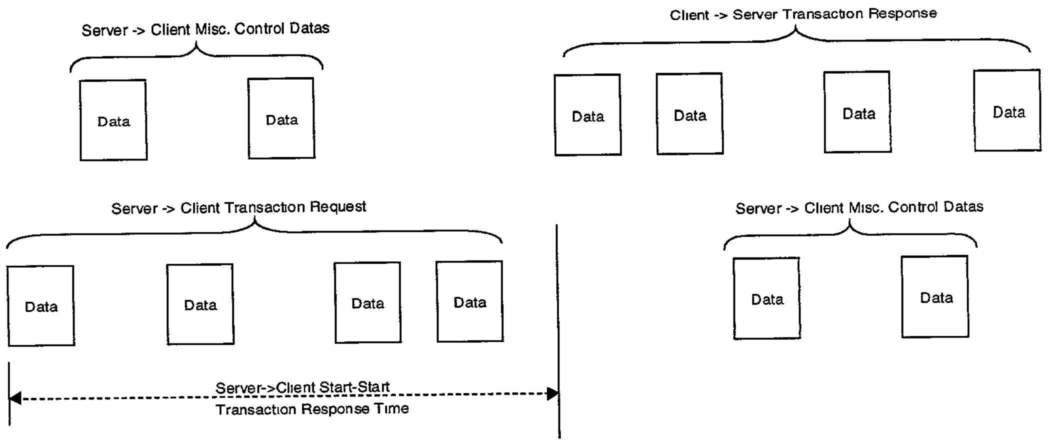
SCTransactionResponseTimeStartToStart

Definition

This metric contains information about the **Application-level response time** measured for application transactions for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, *SCTransactionResponseTimeStartToStart* measures the response time between **start** of an application transaction from the Server to the Client and the **start** of their subsequent transaction response from the Client to the Server.

A Server->Client transaction starts with the 1st Transport Protocol Data Packet/Unit (TPDU) of a transaction request from the Server to the Client and is demarcated (or terminated) by 1st subsequent data packet of the response to the transaction request. The total time between the start of the Server->Client transaction request and the start of the actual transaction response from the Client->Server is measured with this metric.

This metric is considered a "best-effort" measurement. Systems implementing this metric should make a "best-effort" to demarcate the start and end of requests and responses with the specific application's definition of a logical transaction. The lowest level of support for this metric would make this metric the equivalent of *SCExchangeResponseTimeStartToStart*.


Server -> Client Misc. Control Datas
Data
Data
Client -> Server Transaction Response
Data
Data
Data
Data
Server -> Client Transaction Request
Data
Data
Data
Data
Server -> Client Misc. Control Datas
Data
Data
Server->Client Start-Start
Transaction Response Time

Metric Specification

Metric	Applicability	Units	Description
N	Applicable	Svr->Client Transaction Requests	Count of the # Server->Client Transaction Requests measured for Application requests from the Server to the Client(s)
Σ	Applicable	uSeconds	Sum total of the Start-to-Start Delta Times in these Application Response Times
Maximum	Applicable	uSeconds	The maximum Start-to-Start Delta Time of these Application Response Times
Minimum	Applicable	uSeconds	The minimum Start-to-Start Delta Time of these Application Response Times

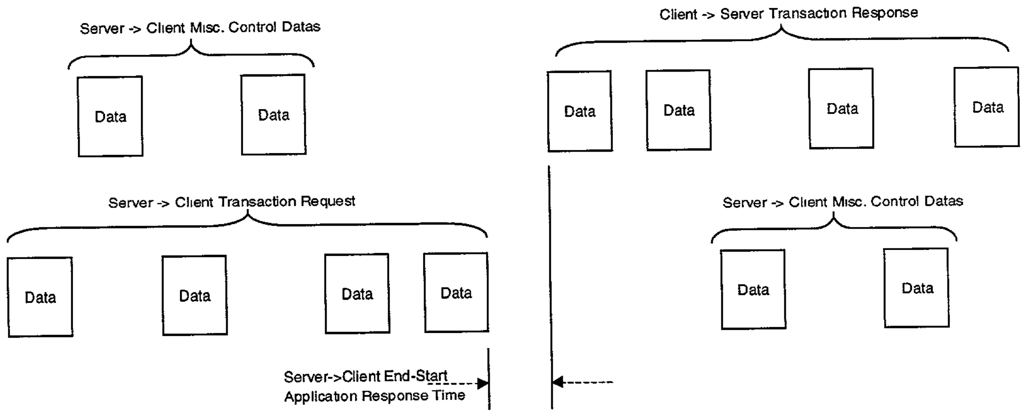
SCApplicationResponseTimeEndToStart

Definition

This metric contains information about the **Application-level response time** measured for application transactions for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, *SCApplicationResponseTimeEndToStart* measures the response time between **end** of an application transaction from the Server to the Client and the **start** of their subsequent transaction response from the Client to the Server.

A Server->Client transaction starts with the 1st Transport Protocol Data Packet/Unit (TPDU) of a transaction request from the Server to the Client and is demarcated (or terminated) by 1st subsequent data packet of the response to the transaction request The total time between the end of the Server->Client transaction request and the start of the actual transaction response from the Client->Server is measured with this metric

This metric is considered a "best-effort" measurement. Systems implementing this metric should make a "best-effort" to demarcate the start and end of requests and responses with the specific application's definition of a logical transaction. The lowest level of support for this metric would make this metric the equivalent of *SCExchangeResponseTimeEndToStart*.


Server -> Client Misc. Control Datas
Data
Data
Client -> Server Transaction Response
Data
Data
Data
Data
Server -> Client Transaction Request
Data
Data
Data
Data
Server -> Client Misc. Control Datas
Data
Data
Server->Client End-Start
Application Response Time

Metric Specification

Metric	Applicability	Units	Description
N	Applicable	Svr->Client Transaction Requests	Count of the # Server->Client Transaction Requests measured for Application requests from the Server to the Client(s)
Σ	Applicable	uSeconds	Sum total of the End-to-Start Delta Times in these Application Response Times
Maximum	Applicable	uSeconds	The maximum End-to-Start Delta Time of these Application Response Times
Minimum	Applicable	uSeconds	The minimum End-to-Start Delta Time of these Application Response Times

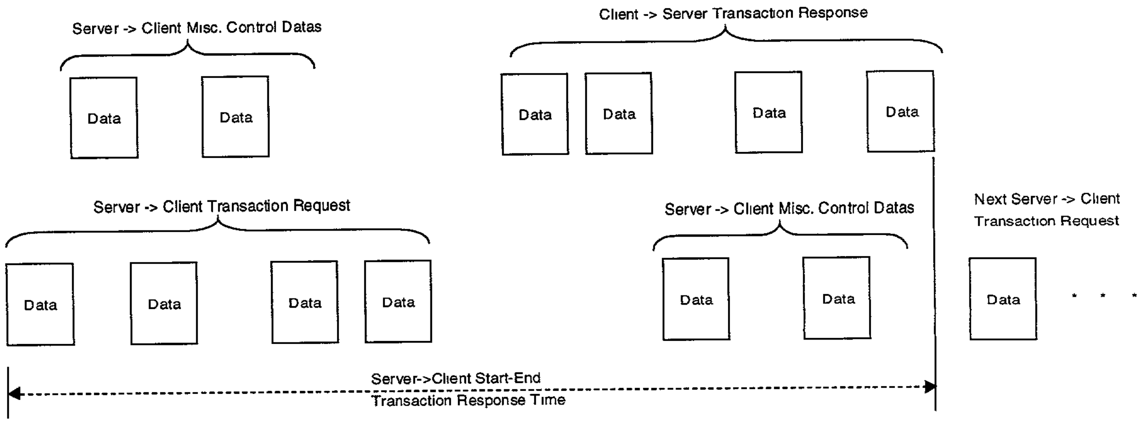
SCApplicationResponseTimeStartToEnd

Definition

This metric contains information about the **Application-level response time** measured for application transactions for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, *SCTransactionResponseTimeStartToEnd* measures the response time between **Start** of an application transaction from the Server to the Client and the **End** of their subsequent transaction response from the Client to the Server.

A Server->Client transaction starts with the 1st Transport Protocol Data Packet/Unit (TPDU) a transaction request from the Server to the Client and is demarcated (or terminated) by 1st subsequent data packet of the response to the transaction request. The end of the Transaction Response in the other direction (e.g. from the Client to the Server) is demarcated by the last data of the transaction response prior to the 1st data of the **next** Server to Client Transaction Request. The total time between the start of the Server->Client transaction request and the end of the Client->Server transaction response is measured with this metric.

This metric is considered a "best-effort" measurement. Systems implementing this metric should make a "best-effort" to demarcate the start and end of requests and responses with the specific application's definition of a logical transaction. The lowest level of support for this metric would make this metric the equivalent of *SCExchangeResponseTimeStartToEnd.*


Server -> Client Misc. Control Datas
Data
Data
Client -> Server Transaction Response
Data
Data
Data
Data
Server -> Client Transaction Request
Data
Data
Data
Data
Server -> Client Misc. Control Datas
Data
Data
Next Server -> Client Transaction Request
Data
. . .
Server->Client Start-End Transaction Response Time

Metric Specification

Metric	Applicability	Units	Description
N	Applicable	Server-> Client Transactions	Count of the # Server<->Client request/response pairs measured for transactions from the Server to the Client(s)
Σ	Applicable	uSeconds	Sum total of the Start-to-End Delta Times in these Application Response Times
Maximum	Applicable	uSeconds	The maximum Start-to-End Delta Time of these Application Response Times
Minimum	Applicable	uSeconds	The minimum Start-to-End Delta Time of these Application Response Times

Connection Metrics

ConnectionEstablishment

Definition

This metric contains information about the transport-level connection establishment for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, *ConnectionsEstablishment* measures number of connections established the Client(s) to the Server. The information contain, in essence, includes:

- # Transport Connections Successfully established
- Set-up Times of the established connections
- Max. # of Simultaneous established connections.
- # Failed Connection establishment attempts (due to either timeout or rejection)

Note that the "# of CURRENT Established Transport Connections" may be derived from this metric along with the *ConnectionGracefulTermination* and *ConnectionTimeoutTermination* metrics, as follows:

# current connections :== "# successfully established"
- "# terminated gracefully"
- "# terminated by time-out"

The set-up time of a connection is defined to be the delta time between the first transport-level, Connection Establishment Request (*i.e.*, SYN, CR-TPDU, etc.) and the first Data Packet exchanged on the connection.

Metric Specification

Metric	**Applicability**	**Units**	**Description**
N	Applicable	Connections	Count of the # Connections Established from the Client(s) to the Server
Σ	Applicable	uSeconds	Sum total of the Connection Set-up Times in these Established connections
Maximum	Applicable	Connections	Count of the MAXIMUM simultaneous # Connections Established from the Client(s) to the Server
Minimum	Not Applicable	Connections	Count of the Failed simultaneous # Connections Established from the Client(s) to the Server

ConnectionGracefulTermination

Definition

This metric contains information about the transport-level connections terminated gracefully for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, *ConnectionsGracefulTermination* measures gracefully terminated connections both in volume and summary connection duration. The information contain, in essence, includes:

- # Gracefully terminated Transport Connections
- Durations (lifetimes) of gracefully terminated connections.

Metric Specification

Metric	Applicability	Units	Description
N	Applicable	Connections	Count of the # Connections Gracefully Terminated between Client(s) to the Server
Σ	Applicable	mSeconds	Sum total of the Connection Durations (Lifetimes) of these terminated connections
Maximum	Not Applicable		
Minimum	Not Applicable		

ConnectionTimeoutTermination

Definition

This metric contains information about the transport-level connections terminated non-gracefully (e.g. Timed-Out) for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, *ConnectionsTimeoutTermination* measures previously established and timed-out connections both in volume and summary connection duration. The information contain, in essence, includes:

- # Timed-out Transport Connections
- Durations (lifetimes) of timed-out terminated connections.

The duration factor of this metric is considered a "best-effort" measurement. Independent network monitoring devices cannot really know when network entities actually detect connection timeout conditions and hence may need to extrapolate or estimate when connection timeouts actually occur.

Metric Specification

Metric	Applicability	Units	Description
N	Applicable	Connections	Count of the # Connections Timed-out between Client(s) to the Server
Σ	Applicable	mSeconds	Sum total of the Connection Durations (Lifetimes) of these terminated connections
Maximum	Not Applicable		
Minimum	Not Applicable		

Connection Sequence Metrics

CSConnectionRetransmissions

Definition

This metric contains information about the transport-level connection health for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, CS*ConnectionRetransmissions* measures number of actual events within established connection lifetimes in which Transport, data-bearing PDUs (packets) from the Client->Server were retransmitted.

Note that retransmission events as seen by the Network Monitoring device indicate the "duplicate" presence of a TPDU as observed on the network.

Metric Specification

Metric	Applicability	Units	Description
N	Applicable	Events	Count of the # Data TPDU retransmissions from the Client(s) to the Server
Σ	Not Applicable		
Maximum	Not Applicable		
Minimum	Not Applicable		

SCConnectionRetransmissions

Definition

This metric contains information about the transport-level connection health for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, SC*ConnectionRetransmissions* measures number of actual events within established connection lifetimes in which Transport, data-bearing PDUs (packets) from the Server->Client were retransmitted.

Note that retransmission events as seen by the Network Monitoring device indicate the "duplicate" presence of a TPDU as observed on the network.

Metric Specification

Metric	**Applicability**	**Units**	**Description**
N	Applicable	Events	Count of the # Data TPDU retransmissions from the Server to the Client(s)
Σ	Not Applicable		
Maximum	Not Applicable		
Minimum	Not Applicable		

CSConnectionOutOfOrders

Definition

This metric contains information about the transport-level connection health for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, CS*ConnectionOutOfOrders* measures number of actual events within established connection lifetimes in which Transport, data-bearing PDUs (packets) from the Client->Server were detected as being out of sequential order.

Note that retransmissions (or duplicates) are considered to be different than out-of-order events and are tracked separately in the CS*ConnectionRetransmissions* metric.

Metric Specification

Metric	Applicability	Units	Description
N	Applicable	Events	Count of the # Out-of-Order TPDU events from the Client(s) to the Server
Σ	Not Applicable		
Maximum	Not Applicable		
Minimum	Not Applicable		

SCConnectionOutOfOrders

Definition

This metric contains information about the transport-level connection health for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, SC*ConnectionOutOfOrders* measures number of actual events within established connection lifetimes in which Transport, data-bearing PDUs (packets) from the Server->Client were detected as being out of sequential order.

Note that retransmissions (or duplicates) are considered to be different than out-of-order events and are tracked separately in the SC*ConnectionRetransmissions* metric.

Metric Specification

Metric	Applicability	Units	Description
N	Applicable	Events	Count of the # Out-of-Order TPDU events from the Server to the Client(s)
Σ	Not Applicable		
Maximum	Not Applicable		
Minimum	Not Applicable		

Connection Window Metrics

CSConnectionWindow

Definition

This metric contains information about the transport-level connection windows for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, CS*ConnectionWindow* measures number of Transport-level Acknowledges within established connection lifetimes and their relative sizes from the Client->Server.

Note that the number of DATA TPDUs (packets) may be estimated by differencing the Acknowledge count of this metric and the overall traffic from the Client to the Server (see *CSTraffic* above). A slight error in this calculation may occur due to Connection Establishment and Termination TPDUS, but it should not be significant.

Metric Specification

Metric	Applicability	Units	Description
N	Applicable	Events	Count of the # ACK TPDU retransmissions from the Client(s) to the Server
Σ	Not Applicable	Increments	Sum total of the Window Sizes of the Acknowledges
Maximum	Not Applicable	Increments	The maximum Window Size of these Acknowledges
Minimum	Not Applicable	Increments	The minimum Window Size of these Acknowledges

SCConnectionWindow

Definition

This metric contains information about the transport-level connection windows for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, SS*ConnectionWindow* measures number of Transport-level Acknowledges within established connection lifetimes and their relative sizes from the Server->Client.

Note that the number of DATA TPDUs (packets) may be estimated by differencing the Acknowledge count of this metric and the overall traffic from the Client to the Server (see *SCTraffic* above).. A slight error in this calculation may occur due to Connection Establishment and Termination TPDUS, but it should not be significant.

Metric Specification

Metric	**Applicability**	**Units**	**Description**
N	Applicable	Events	Count of the # ACK TPDU retransmissions from the Server to the Client(s)
Σ	Applicable	Increments	Sum total of the Window Sizes of the Acknowledges
Maximum	Applicable	Increments	The maximum Window Size of these Acknowledges
Minimum	Applicable	Increments	The minimum Window Size of these Acknowledges

CSConnectionFrozenWindows

Definition

This metric contains information about the transport-level connection windows for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, CS*ConnectionWindow* measures number of Transport-level Acknowledges from Client->Server within established connection lifetimes which validly acknowledge data, but either

- failed to increase the upper window edge,
- reduced the upper window edge

Metric Specification

Metric	Applicability	Units	Description
N	Applicable	Events	Count of the # ACK TPDU with frozen/reduced windows from the Client(s) to the Server
Σ	Not Applicable		
Maximum	Not Applicable		
Minimum	Not Applicable		

SCConnectionFrozenWindows

Definition

This metric contains information about the transport-level connection windows for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, SC*ConnectionWindow* measures number of Transport-level Acknowledges from Server->Client within established connection lifetimes which validly acknowledge data, but either

- failed to increase the upper window edge,
- reduced the upper window edge

Metric Specification

Metric	Applicability	Units	Description
N	Applicable	Events	Count of the # ACK TPDU with frozen/reduced windows from the Client(s) to the Server
Σ	Not Applicable		
Maximum	Not Applicable		
Minimum	Not Applicable		

CSConnectionClosedWindows

Definition

This metric contains information about the transport-level connection windows for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, CS*ConnectionWindow* measures number of Transport-level Acknowledges from Client->Server within established connection lifetimes which fully closed the acknowledge/sequence window.

Metric Specification

Metric	**Applicability**	**Units**	**Description**
N	Applicable	Events	Count of the # ACK TPDU with Closed windows from the Client(s) to the Server
Σ	Not Applicable		
Maximum	Not Applicable		
Minimum	Not Applicable		

SCConnectionClosedWindows

Definition

This metric contains information about the transport-level connection windows for a given application and either a specific Client-Server Pair or a specific Server and all of its clients. Specifically, SC*ConnectionWindow* measures number of Transport-level Acknowledges from Server->Client within established connection lifetimes which fully closed the acknowledge/sequence window.

Metric Specification

Metric	**Applicability**	**Units**	**Description**
N	Applicable	Events	Count of the # ACK TPDU with Closed windows from the Client(s) to the Server
Σ	Not Applicable		
Maximum	Not Applicable		
Minimum	Not Applicable		

Some common definitions

The definitions below are of terms that would be well-known to those of ordinary skill in the art, and are only presented here for completeness so that people less acquainted with the art also may be able to understand the description.

The term **RMON** derives from a standard that was first developed in 1992 by the Internet Engineering Task Force (IETF) as an extension to the Simple Network Management Protocol (SNMP) Management Information Base (MIB). These MIB extensions are referred to as the **R**emote **MON**itoring MIB; which is commonly abbreviated to **RMON**. The IETF defines 10 RMON Groups for the gathering of information on Ethernet and Token Ring networks, and in 1997 a second RFC was adopted that allowed the gathering of information at all 7 layers. There is no IETF definition for RMON on FDDI networks or Wide Area Networks, such as Frame Relay, but the probes follow the same structures and conventions as the original RMON definitions, providing this capability over many network types.

Tunneling is understood to mean transmitting data structured in one protocol format within the format of another protocol. Tunneling allows other types of transmission streams to be carried within the prevailing protocol. For example, IP tunneling is carrying a foreign protocol within a TCP/IP packet. For example, IPX can be encapsulated and transmitted via TCP/IP. 2TP (Layer 2 Tunneling Protocol) A protocol from the Internet Engineering Task Force (IETF) for creating virtual private networks (VPNs) over the Internet. It supports non-IP protocols such as AppleTalk and IPX as well as the IPSec security protocol. It is a combination of the Point-to-Point Tunneling Protocol (Microsoft Corporation, Redmond, Washington) and Layer 2 Forwarding (L2F) technology (Cisco Systems, San Jose, California).

DCOM (Distributed Component Object Model), formerly called Network OLE (Microsoft Corporation, Redmond, Washington), is Microsoft's technology for distributed objects. DCOM is based on COM, Microsoft's component software architecture, which defines the object interfaces. DCOM defines the remote procedure call which allows those objects to be run remotely over the network. DCOM began shipping with Windows NT 4.0 and is Microsoft's counterpart to **CORBA** (Common Object Request Broker Architecture), a standard from the Object Management Group (OMG) for communicating between distributed objects (objects are self-contained software modules). CORBA provides a way to execute programs (objects) written in different programming languages running on different platforms no matter where they reside in the network.

Sun-RPC (Sun's Remote Procedure Call) 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. The calling programming sends a message and data to the remote program, which is executed, and results are passed back to the calling program. This type of interface is designed to allow programs to communicate with each another while freeing the programmer from the networking details. Microsoft's DCOM was modeled after the RPC in DCE. CORBA also provides this capability.

CAM is the same as associative storage. associative storage This is storage that is accessed by comparing the content of the data stored in it rather than by addressing predetermined locations.

UDP (User Datagram Protocol) A protocol within the TCP/IP protocol suite that is used in place of TCP when a reliable delivery is not required. For example, UDP is used for realtime audio and video traffic where lost packets are simply ignored, because there is no time to retransmit. If UDP is used and a reliable delivery is required, packet sequence checking and error notification must be written into the applications.

RTP (Realtime Transport Protocol) An IP protocol that supports realtime transmission of voice and video. An RTP packet rides on top of UDP and includes timestamping and synchronization information in its header for proper reassembly at the receiving end. Realtime Control Protocol (RTCP) is a companion protocol that is used to maintain QoS. RTP nodes analyzes network conditions and periodically send each other RTCP packets that report on network congestion.

RTP Packet. In a UDP/IP stack, the RTP header is created first and then the packet is moved down the stack to UDP and IP. This shows the RTP packet within an Ethernet frame ready for transmission over the network.

Port number. In a TCP/IP-based network such as the Internet, it is a number assigned to an application program running in the computer. The number is used to link the incoming data to the correct service. Well-known ports are standard port numbers used by everyone; for example, port 80 is used for HTTP traffic (Web traffic).

Binding. In a communications network, to establish a software connection between one protocol and another. Data flows from the application to the transport protocol to the network protocol to the data link protocol and then onto the network. Binding the protocols creates the internal pathway.

Frame relay. A high-speed packet switching protocol used in wide area networks (WANs). It has become popular for LAN to LAN connections across remote distances, and services are provided by all the major carriers. Frame relay is faster than traditional X.25 networks, because it was designed for today's reliable circuits and performs less rigorous error detection. Frame relay provides for a granular service up to DS3 rates of 44.736 Mbps and is suited for data and image transfer. Because of its variable-length packet architecture, it is not the most efficient technology for realtime voice and video.

A **connection oriented** communications architecture is one that requires an establishment of the session between two nodes before transmission can begin. When the communications is completed, the session is ended (torn down). All circuit-switched networks are connection oriented because they require a dedicated channel for the duration of the session. In addition, packet-switched X.25, frame relay and ATM networks are also considered connection oriented, because they require receiving nodes to acknowledge their ability to support the transmission before data can be sent.

A **connectionless communications architecture**, on the other hand, is one that does not require the establishment of a session between two nodes before transmission can begin. The transmission of frames within a local area network (LAN), such as Ethernet, Token Ring and FDDI, is connectionless. The terms connection-oriented and connectionless oriented also apply to the different protocol levels. For example, common TCP/IP is composed of TCP (Transmission Control Protocol), a connection-oriented protocol that passes its data to the next lower layer, IP (Internet Protocol), a connectionless protocol. TCP sets up a connection at both ends and guarantees reliable delivery of the full message sent. TCP tests for errors and requests retransmission if necessary, because IP does not. UDP packets within a TCP/IP network are also connectionless.

Some PDL Files.

The following pages include some PDL files as examples. Included herein are the PDL contents of the following files. A reference to PDL is also included herein. Note that any contents on any line following two hyphen (--) are ignored by the compiler. That is, they are comments.

common.pdl;

flows.pdl;

virtual.pdl;

ethernet.pdl;

IEEE8032.pdl and IEEE8033.pdl (ethertype files);

IP.pdl;

TCP.pdl and UDP.pdl;

RPC.pdl;

NFS.pdl; and

HTTP.pdl.

```
--------------------------------------------------------------------------
--
--  Common.pdl - Common protocol definitions
--
--  Description:
--     This file contains some field definitions for commonly used fields
--     in various network protocols.
--
--  Copyright:
--     Copyright (c) 1996-1999 Apptitude, Inc.
--       (formerly Technically Elite, Inc.)
--     All rights reserved.
--
--  RCS:
--     $Id: Common.pdl,v 1.7 1999/04/13 15:47:56 skip Exp $
--
--------------------------------------------------------------------------
Int4   FIELD
       SYNTAX INT(4)

Int8   FIELD
       SYNTAX INT(8)

Int16  FIELD
       SYNTAX INT(16)

Int24  FIELD
       SYNTAX INT(24)

Int32  FIELD
       SYNTAX INT(32)

Int64  FIELD
       SYNTAX INT(64)

UInt8  FIELD
       SYNTAX UNSIGNED INT(8)

UInt16 FIELD
       SYNTAX UNSIGNED INT(16)

UInt24 FIELD
       SYNTAX UNSIGNED INT(24)

UInt32 FIELD
       SYNTAX UNSIGNED INT(32)

UInt64 FIELD
       SYNTAX UNSIGNED INT(64)

SInt16 FIELD
       SYNTAX INT(16)
       FLAGS  SWAPPED

SUInt16       FIELD
       SYNTAX UNSIGNED INT(16)
       FLAGS  SWAPPED

SInt32 FIELD
       SYNTAX INT(32)
       FLAGS  SWAPPED

ByteStr1      FIELD
              SYNTAX BYTESTRING(1)

ByteStr2      FIELD
              SYNTAX BYTESTRING(2)

ByteStr4      FIELD
              SYNTAX BYTESTRING(4)
```

```
Pad1            FIELD
                SYNTAX BYTESTRING(1)
                FLAGS  NOSHOW

Pad2            FIELD
                SYNTAX BYTESTRING(2)
                FLAGS  NOSHOW

Pad3            FIELD
                SYNTAX BYTESTRING(3)
                FLAGS  NOSHOW

Pad4            FIELD
                SYNTAX BYTESTRING(4)
                FLAGS  NOSHOW

Pad5            FIELD
                SYNTAX BYTESTRING(5)
                FLAGS  NOSHOW

macAddress    FIELD
        SYNTAX          BYTESTRING(6)
        DISPLAY-HINT    "1x:"
        LOOKUP          MACADDRESS
        DESCRIPTION
             "MAC layer physical address"

ipAddress     FIELD
        SYNTAX          BYTESTRING(4)
        DISPLAY-HINT    "1d."
        LOOKUP          HOSTNAME
        DESCRIPTION
             "IP address"

ipv6Address   FIELD
        SYNTAX          BYTESTRING(16)
        DISPLAY-HINT    "1d."
        DESCRIPTION
             "IPV6 address"
```

```
-----------------------------------------------------------------------
--
--  Flows.pdl - General FLOW definitions
--
--  Description:
--     This file contains general flow definitions.
--
--  Copyright:
--     Copyright (c) 1998-1999 Apptitude, Inc.
--       (formerly Technically Elite, Inc.)
--     All rights reserved.
--
--  RCS:
--     $Id: Flows.pdl,v 1.12 1999/04/13 15:47:57 skip Exp $
--
-----------------------------------------------------------------------

chaosnet  FLOW

spanningTree FLOW

sna      FLOW

oracleTNS FLOW
         PAYLOAD { INCLUDE-HEADER, LENGTH=256 }

ciscoOUI  FLOW

---------------------------------------------------------
-- IP Protocols
---------------------------------------------------------

igmp     FLOW

GGP      FLOW

ST       FLOW

UCL      FLOW

egp      FLOW

igp      FLOW

BBN-RCC-MON   FLOW

NVP2    FLOW

PUP     FLOW

ARGUS   FLOW

EMCON   FLOW

XNET    FLOW

MUX     FLOW

DCN-MEAS FLOW

HMP     FLOW

PRM     FLOW

TRUNK1  FLOW

TRUNK2  FLOW

LEAF1   FLOW

LEAF2   FLOW
```

RDP	FLOW
IRTP	FLOW
ISO-TP4	FLOW
NETBLT	FLOW
MFE-NSP	FLOW
MERIT-INP	FLOW
SEP	FLOW
PC3	FLOW
IDPR	FLOW
XTP	FLOW
DDP	FLOW
IDPR-CMTP	FLOW
TPPlus	FLOW
IL	FLOW
SIP	FLOW
SDRP	FLOW
SIP-SR	FLOW
SIP-FRAG	FLOW
IDRP	FLOW
RSVP	FLOW
MHRP	FLOW
BNA	FLOW
SIPP-ESP	FLOW
SIPP-AH	FLOW
INLSP	FLOW
SWIPE	FLOW
NHRP	FLOW
CFTP	FLOW
SAT-EXPAK	FLOW
KRYPTOLAN	FLOW
RVD	FLOW
IPPC	FLOW
SAT-MON	FLOW
VISA	FLOW
IPCV	FLOW

```
CPNX    FLOW
CPHB    FLOW
WSN     FLOW
PVP     FLOW
BR-SAT-MON      FLOW
SUN-ND  FLOW
WB-MON  FLOW
WB-EXPAK FLOW
ISO-IP  FLOW
VMTP    FLOW
SECURE-VMTP     FLOW
TTP     FLOW
NSFNET-IGP      FLOW
DGP     FLOW
TCF     FLOW
IGRP    FLOW
OSPFIGP         FLOW
Sprite-RPC      FLOW
LARP    FLOW
MTP     FLOW
AX25    FLOW
IPIP    FLOW
MICP    FLOW
SCC-SP  FLOW
ETHERIP         FLOW
encap   FLOW
GMTP    FLOW

----------------------------------------------------------
-- UDP Protocols
----------------------------------------------------------
compressnet  FLOW
rje     FLOW
echo    FLOW
discard         FLOW
systat FLOW
daytime         FLOW
qotd    FLOW
```

```
msp     FLOW

chargen         FLOW

biff    FLOW

who     FLOW

syslog FLOW

loadav FLOW

notify FLOW

acmaint_dbd   FLOW

acmaint_transd        FLOW

puparp FLOW

applix FLOW

ock     FLOW


------------------------------------------------------------
-- TCP Protocols
------------------------------------------------------------
tcpmux FLOW

telnet FLOW
        CONNECTION { INHERITED }

privMail        FLOW

nsw-fe FLOW

msg-icp         FLOW

msg-auth        FLOW

dsp     FLOW

privPrint       FLOW

time    FLOW

rap     FLOW

rlp     FLOW

graphics        FLOW

nameserver      FLOW

nicname         FLOW

mpm-flags       FLOW

mpm     FLOW

mpm-snd         FLOW

ni-ftp FLOW

auditd FLOW

finger FLOW

re-mail-ck      FLOW
```

```
la-maint      FLOW
xns-time      FLOW
xns-ch FLOW
isi-gl FLOW
xns-auth      FLOW
privTerm      FLOW
xns-mail      FLOW
privFile      FLOW
ni-mail       FLOW
acas   FLOW
covia  FLOW
tacacs-ds     FLOW
sqlnet FLOW
gopher FLOW
netrjs-1      FLOW
netrjs-2      FLOW
netrjs-3      FLOW
netrjs-4      FLOW
privDial      FLOW
deos   FLOW
privRJE       FLOW
vettcp FLOW
hosts2-ns     FLOW
xfer   FLOW
ctf    FLOW
mit-ml-dev    FLOW
mfcobol       FLOW
kerberos      FLOW
su-mit-tg     FLOW
dnsix  FLOW
mit-dov       FLOW
npp    FLOW
dcp    FLOW
objcall       FLOW
supdup FLOW
```

```
dixie  FLOW
swift-rvf      FLOW
tacnews        FLOW
metagram       FLOW
newacct        FLOW
hostname       FLOW
iso-tsap       FLOW
gppitnp        FLOW
csnet-ns       FLOW
threeCom-tsmux        FLOW
rtelnet        FLOW
snagas FLOW
mcidas FLOW
auth   FLOW
audionews     FLOW
sftp   FLOW
ansanotify    FLOW
uucp-path     FLOW
sqlserv       FLOW
cfdptkt       FLOW
erpc   FLOW
smakynet      FLOW
ntp    FLOW
ansatrader    FLOW
locus-map     FLOW
unitary       FLOW
locus-con     FLOW
gss-xlicen    FLOW
pwdgen FLOW
cisco-fna     FLOW
cisco-tna     FLOW
cisco-sys     FLOW
statsrv       FLOW
ingres-net    FLOW
loc-srv       FLOW
profile       FLOW
```

```
emfis-data    FLOW
emfis-cntl    FLOW
bl-idm FLOW
imap2  FLOW
news   FLOW
uaac   FLOW
iso-tp0       FLOW
iso-ip FLOW
cronus FLOW
aed-512       FLOW
sql-net       FLOW
hems   FLOW
bftp   FLOW
sgmp   FLOW
netsc-prod    FLOW
netsc-dev     FLOW
sqlsrv FLOW
knet-cmp      FLOW
pcmail-srv    FLOW
nss-routing   FLOW
sgmp-traps    FLOW
cmip-man      FLOW
cmip-agent    FLOW
xns-courier   FLOW
s-net  FLOW
namp   FLOW
rsvd   FLOW
send   FLOW
print-srv     FLOW
multiplex     FLOW
cl-1   FLOW
xyplex-mux    FLOW
mailq  FLOW
vmnet  FLOW
genrad-mux    FLOW
```

```
xdmcp  FLOW
nextstep      FLOW
bgp    FLOW
ris    FLOW
unify  FLOW
audit  FLOW
ocbinder      FLOW
ocserver      FLOW
remote-kis    FLOW
kis    FLOW
aci    FLOW
mumps  FLOW
qft    FLOW
gacp   FLOW
prospero      FLOW
osu-nms       FLOW
srmp   FLOW
irc    FLOW
dn6-nlm-aud   FLOW
dn6-smm-red   FLOW
dls    FLOW
dls-mon       FLOW
smux   FLOW
src    FLOW
at-rtmp       FLOW
at-nbp FLOW
at-3   FLOW
at-echo       FLOW
at-5   FLOW
at-zis FLOW
at-7   FLOW
at-8   FLOW
tam    FLOW
z39-50 FLOW
anet   FLOW
vmpwscs       FLOW
```

```
softpc FLOW
atls   FLOW
dbase  FLOW
mpp    FLOW
uarps  FLOW
imap3  FLOW
fln-spx       FLOW
rsh-spx       FLOW
cdc    FLOW
sur-meas      FLOW
link   FLOW
dsp3270       FLOW
pdap   FLOW
pawserv       FLOW
zserv  FLOW
fatserv       FLOW
csi-sgwp      FLOW
clearcase     FLOW
ulistserv     FLOW
legent-1      FLOW
legent-2      FLOW
hassle FLOW
nip    FLOW
tnETOS FLOW
dsETOS FLOW
is99c  FLOW
is99s  FLOW
hp-collector  FLOW
hp-managed-node      FLOW
hp-alarm-mgr  FLOW
arns   FLOW
ibm-app       FLOW
asa    FLOW
aurp   FLOW
unidata-ldm   FLOW
```

```
ldap    FLOW
uis     FLOW
synotics-relay       FLOW
synotics-broker      FLOW
dis     FLOW
embl-ndt        FLOW
netcp   FLOW
netware-ip      FLOW
mptn    FLOW
kryptolan       FLOW
work-sol        FLOW
ups     FLOW
genie   FLOW
decap   FLOW
nced    FLOW
ncld    FLOW
imsp    FLOW
timbuktu        FLOW
prm-sm FLOW
prm-nm FLOW
decladebug      FLOW
rmt     FLOW
synoptics-trap       FLOW
smsp    FLOW
infoseek        FLOW
bnet    FLOW
silverplatter FLOW
onmux   FLOW
hyper-g         FLOW
ariel1 FLOW
smpte   FLOW
ariel2 FLOW
ariel3 FLOW
opc-job-start FLOW
opc-job-track FLOW
icad-el         FLOW
```

```
smartsdp        FLOW
svrloc FLOW
ocs_cmu         FLOW
ocs_amu         FLOW
utmpsd FLOW
utmpcd FLOW
iasd    FLOW
nnsp    FLOW
mobileip-agent          FLOW
mobilip-mn      FLOW
dna-cml         FLOW
comscm FLOW
dsfgw   FLOW
dasp    FLOW
sgcp    FLOW
decvms-sysmgt FLOW
cvc_hostd       FLOW
https   FLOW
        CONNECTION { INHERITED }
snpp    FLOW
microsoft-ds    FLOW
ddm-rdb         FLOW
ddm-dfm         FLOW
ddm-byte        FLOW
as-servermap    FLOW
tserver         FLOW
exec    FLOW
        CONNECTION { INHERITED }
login   FLOW
        CONNECTION { INHERITED }
cmd     FLOW
        CONNECTION { INHERITED }
printer         FLOW
        CONNECTION { INHERITED }
talk    FLOW
        CONNECTION { INHERITED }
ntalk   FLOW
        CONNECTION { INHERITED }
utime   FLOW
```

```
efs     FLOW
timed   FLOW
tempo   FLOW
courier         FLOW
conference      FLOW
netnews         FLOW
netwall         FLOW
apertus-ldp     FLOW
uucp    FLOW
uucp-rlogin     FLOW
klogin  FLOW
kshell  FLOW
new-rwho        FLOW
dsf     FLOW
remotefs        FLOW
rmonitor        FLOW
monitor         FLOW
chshell         FLOW
p9fs    FLOW
whoami  FLOW
meter   FLOW
ipcserver       FLOW
urm     FLOW
nqs     FLOW
sift-uft        FLOW
npmp-trap       FLOW
npmp-local      FLOW
npmp-gui        FLOW
ginad   FLOW
doom    FLOW
mdqs    FLOW
elcsd   FLOW
entrustmanager          FLOW
netviewdm1      FLOW
netviewdm2      FLOW
```

```
netviewdm3    FLOW
netgw  FLOW
netrcs FLOW
flexlm FLOW
fujitsu-dev   FLOW
ris-cm FLOW
kerberos-adm  FLOW
rfile  FLOW
pump   FLOW
qrh    FLOW
rrh    FLOW
tell   FLOW
nlogin FLOW
con    FLOW
ns     FLOW
rxe    FLOW
quotad FLOW
cycleserv     FLOW
omserv FLOW
webster       FLOW
phonebook     FLOW
vid    FLOW
cadlock       FLOW
rtip   FLOW
cycleserv2    FLOW
submit FLOW
rpasswd       FLOW
entomb FLOW
wpages FLOW
wpgs   FLOW
concert       FLOW
mdbs_daemon   FLOW
device FLOW
xtreelic      FLOW
maitrd FLOW
busboy FLOW
```

```
garcon FLOW
puprouter     FLOW
socks  FLOW
```

```
---------------------------------------------------------------------------
--
--  Virtual.pdl - Virtual Layer definition
--
--  Description:
--     This file contains the definition for the VirtualBase layer used
--     by the embodiment.
--
--  Copyright:
--     Copyright (c) 1998-1999 Apptitude, Inc.
--       (formerly Technically Elite, Inc.)
--     All rights reserved.
--
--  RCS:
--     $Id: Virtual.pdl,v 1.13 1999/04/13 15:48:03 skip Exp $
--
---------------------------------------------------------------------------
-- This includes two things: the flow signature (called FLOWKEY) that the
-- system that is going to use.
--
-- note that not all elements are in the HASH. Reason is that these non-HASHED
 -- elements may be varied without the HASH changing, whihc allows the system
 -- to look up multiple buckets with a single HASH. That is, the MeyMatchFlag,
 -- StateStatus Flag and MuliPacketID may be varied.
 --

FLOWKEY {
    KeyMatchFlags, -- to tell the system which of the in-HASH elements have to
 -- match for the this particular flow record.
                        -- Flows for which complete signatures may not yet have
                        -- been generated may then be stored in the system
--
StateStatusFlags,

    GroupId1             IN-HASH, -- user defined
    GroupId2             IN-HASH, -- user defined

    DLCProtocol          IN-HASH, , -- data link protocol - lowest level we
                                    -- evaluate. It is the type for the
-- Ethernet V 2
    NetworkProtocol      IN-HASH,   -- IP, etc.
    TunnelProtocol       IN-HASH,   -- IP over IPX, etc.
    TunnelTransport      IN-HASH,
    TransportProtocol    IN-HASH,
    ApplicationProtocol  IN-HASH,

    DLCAddresses(8)      IN-HASH, -- lowest level address
    NetworkAddresses(16) IN-HASH,
    TunnelAddresses(16)  IN-HASH,
    ConnectionIds        IN-HASH,

    MultiPacketId                 -- used for fragmentaion purposes
}
-- now define all of the children. In this example, only one virtual
-- child - Ethernet.
--
virtualChildren    FIELD
             SYNTAX INT(8) { ethernet(1) }

-- now define the base for the children. In this case, it is the same as
-- for the overall system. There may be multiples.
--

VirtualBase  PROTOCOL
::= { VirtualChildren=virtualChildren }

--
-- The following is the header that every packet has to have and
-- that is placed into the system by the packet acquisition system.
--
```

```
VirtualBase   FLOW
              HEADER { LENGTH=8 }
              CHILDREN { DESTINATION=VirtualChildren } -- this will be
-- Ethernet for this example.
--
-- the VirtualBAse will be 01 for these packets.
```

```
----------------------------------------------------------------------------
--
--  Ethernet.pdl - Ethernet frame definition
--
--  Description:
--     This file contains the definition for the Ethernet frame.  In this
--  PDL file, the decision on EtherType vs. IEEE is made.  If this is
--  EtherType, the selection is made from this file.  It would be possible
--  to move the EtherType selection to another file, if that would assist
--  in the modularity.
--
--  Copyright:
--     Copyright (c) 1994-1998 Apptitude, Inc.
--       (formerly Technically Elite, Inc.)
--     All rights reserved.
--
--  RCS:
--     $Id: Ethernet.pdl,v 1.13 1999/01/26 15:15:57 skip Exp $
--
----------------------------------------------------------------------------

--
-- Enumerated type of a 16 bit integer that contains all of the
-- possible values of interest in the etherType field of an
-- Ethernet V2 packet.
--
etherType      FIELD
               SYNTAX         INT(16) { xns(0x0600), ip(0x0800),
                              chaosnet(0x0804), arp(0x0806),
                              vines(0xbad),
                              vinesLoop(0x0bae), vinesLoop(0x80c4),
                              vinesEcho(0xbaf), vinesEcho(0x80c5),
                              netbios(0x3c00), netbios(0x3c01),
                              netbios(0x3c02), netbios(0x3c03),
                              netbios(0x3c04), netbios(0x3c05),
                              netbios(0x3c06), netbios(0x3c07),
                              netbios(0x3c08), netbios(0x3c09),
                              netbios(0x3c0a), netbios(0x3c0b),
                              netbios(0x3c0c), netbios(0x3c0d),
                              dec(0x6000), mop(0x6001), mop2(0x6002),
                              drp(0x6003), lat(0x6004), decDiag(0x6005),
                              lavc(0x6007), rarp(0x8035), appleTalk(0x809b),
                              sna(0x80d5), aarp(0x80f3), ipx(0x8137),
                              snmp(0x814c), ipv6(0x86dd), loopback(0x9000) }
               DISPLAY-HINT   “1x:”
               LOOKUP         FILE “EtherType.cf”
               DESCRIPTION
                   “Ethernet type field”

--
-- The unformatted data field in and Ethernet V2 type frame
--
etherData      FIELD
               SYNTAX         BYTESTRING(46..1500)
               ENCAP          etherType
               DISPLAY-HINT   “HexDump”
               DESCRIPTION
                   “Ethernet data”

--
-- The layout and structure of an Ethernet V2 type frame with
-- the address and protocol fields in the correct offset position
--
ethernet       PROTOCOL
               DESCRIPTION
                   “Protocol format for an Ethernet frame”
               REFERENCE      “RFC 894”
::= { MacDest=macAddress, MacSrc=macAddress, EtherType=etherType,
      Data=etherData }

--
```

```
-- The elements from this Ethernet frame used to build a flow key
-- to classify and track the traffic.  Notice that the total length
-- of the header for this type of packet is fixed and at 14 bytes or
-- octets in length.  The special field, LLC-CHECK, is specific to
-- Ethernet frames for the decoding of the base Ethernet type value.
-- If it is NOT LLC, the protocol field in the flow is set to the
-- EtherType value decoded from the packet.
--
ethernet        FLOW
                HEADER { LENGTH=14 }
                DLC-LAYER {
                    SOURCE=MacSrc,
                    DESTINATION=MacDest,
                    TUNNELING,
                    PROTOCOL
                }
                CHILDREN { DESTINATION=EtherType, LLC-CHECK=llc }
```

```
-----------------------------------------------------------------------
--
--  IEEE8022.pdl - IEEE 802.2 frame definitions
--
--  Description:
--     This file contains the definition for the IEEE 802.2 Link Layer
--     protocols including the SNAP (Sub-network Access Protocol).
--
--  Copyright:
--     Copyright (c) 1994-1998 Apptitude, Inc.
--       (formerly Technically Elite, Inc.)
--     All rights reserved.
--
--  RCS:
--     $Id: IEEE8022.pdl,v 1.18 1999/01/26 15:15:58 skip Exp $
--
--------------------------------------------------------------------------

--
-- IEEE 802.2 LLC
--
llcSap FIELD
       SYNTAX        INT(16) { ipx(0xFFFF), ipx(0xE0E0), isoNet(0xFEFE),
                     netbios(0xF0F0), vsnap(0xAAAA), ip(0x0606),
                     vines(0xBCBC), xns(0x8080), spanningTree(0x4242),
                     sna(0x0c0c), sna(0x0808), sna(0x0404) }
       DISPLAY-HINT  "1x:"
       DESCRIPTION
           "Service Access Point"

llcControl   FIELD
       -- This is a special field. When the decoder encounters this field, it
       -- invokes the hard-coded LLC decoder to decode the rest of the packet.
       -- This is necessary because LLC decoding requires the ability to
       -- handle forward references which the current PDL format does not
       -- support at this time.
       SYNTAX        UNSIGNED INT(8)
       DESCRIPTION
           "Control field"

llcPduType   FIELD
       SYNTAX BITSTRING(2) { llcInformation(0), llcSupervisory(1),
              llcInformation(2), llcUnnumbererd(3) }

llcData      FIELD
       SYNTAX        BYTESTRING(38..1492)
       ENCAP         llcPduType
       FLAGS         SAMELAYER
       DISPLAY-HINT  "HexDump"

llc    PROTOCOL
       SUMMARIZE
           "$llcPduType == llcUnnumbered" :
              "LLC ($SAP) $Modifier"
           "$llcPduType == llcSupervisory" :
              "LLC ($SAP) $Function N(R)=$NR"
           "$llcPduType == 0|2" :
              "LLC ($SAP) N(R)=$NR N(S)=$NS"
           "Default" :
              "LLC ($SAP) $llcPduType"
       DESCRIPTION
           "IEEE 802.2 LLC frame format"
::= { SAP=llcSap, Control=llcControl, Data=llcData }

llc     FLOW
       HEADER { LENGTH=3 }
       DLC-LAYER { PROTOCOL }
       CHILDREN { DESTINATION=SAP }

llcUnnumberedData FIELD
       SYNTAX        BYTESTRING(0..1500)
```

```
        ENCAP        llcSap
        DISPLAY-HINT "HexDump"

llcUnnumbered PROTOCOL
        SUMMARIZE
            "Default" :
               "LLC ($SAP) $Modifier"
::= { Data=llcUnnumberedData }

llcSupervisoryData  FIELD
        SYNTAX       BYTESTRING(0..1500)
        DISPLAY-HINT "HexDump"

llcSupervisory      PROTOCOL
        SUMMARIZE
            "Default" :
               "LLC ($SAP) $Function N(R)=$NR"
::= { Data=llcSupervisoryData }

llcInformationData  FIELD
        SYNTAX       BYTESTRING(0..1500)
        ENCAP        llcSap
        DISPLAY-HINT "HexDump"

llcInformation      PROTOCOL
        SUMMARIZE
            "Default" :
               "LLC ($SAP) N(R)=$NR N(S)=$NS"
::= { Data=llcInformationData }

--
-- SNAP
--
snapOrgCode   FIELD
        SYNTAX       BYTESTRING(3) { snap("00:00:00"), ciscoOUI("00:00:0C"),
                     appleOUI("08:00:07") }
        DESCRIPTION
            "Protocol ID or Organizational Code"

vsnapData    FIELD
        SYNTAX       BYTESTRING(46..1500)
        ENCAP        snapOrgCode
        FLAGS        SAMELAYER
        DISPLAY-HINT "HexDump"
        DESCRIPTION
            "SNAP LLC data"

vsnap  PROTOCOL
        DESCRIPTION
            "SNAP LLC Frame"
::= { OrgCode=snapOrgCode, Data=vsnapData }

vsnap        FLOW
             HEADER { LENGTH=3 }
             DLC-LAYER { PROTOCOL }
             CHILDREN { DESTINATION=OrgCode }

snapType      FIELD
             SYNTAX   INT(16) { xns(0x0600), ip(0x0800), arp(0x0806),
                      vines(0xbad),
                      mop(0x6001), mop2(0x6002), drp(0x6003),
                      lat(0x6004), decDiag(0x6005), lavc(0x6007),
                      rarp(0x8035), appleTalk(0x809B), sna(0x80d5),
                      aarp(0x80F3), ipx(0x8137), snmp(0x814c), ipv6(0x86dd) }
             DISPLAY-HINT    "1x:"
             LOOKUP          FILE "EtherType.cf"
             DESCRIPTION
               "SNAP type field"

snapData      FIELD
        SYNTAX       BYTESTRING(46..1500)
        ENCAP          snapType
        DISPLAY-HINT  "HexDump"
        DESCRIPTION
            "SNAP data"

snap    PROTOCOL
        SUMMARIZE
            "$OrgCode == 00:00:00" :
               "SNAP Type=$SnapType"
            "Default" :
               "VSNAP Org=$OrgCode Type=$SnapType"
        DESCRIPTION
            "SNAP Frame"
::= { SnapType=snapType, Data=snapData }

snap     FLOW
        HEADER { LENGTH=2 }
        DLC-LAYER { PROTOCOL }
        CHILDREN { DESTINATION=SnapType }
```

```
------------------------------------------------------------------------
--
--  IEEE8023.pdl - IEEE 802.3 frame definitions
--
--  Description:
--     This file contains the definition for the IEEE 802.3 (Ethernet)
--     protocols.
--
--  Copyright:
--     Copyright (c) 1994-1998 Apptitude, Inc.
--       (formerly Technically Elite, Inc.)
--     All rights reserved.
--
--  RCS:
--     $Id: IEEE8023.pdl,v 1.7 1999/01/26 15:15:58 skip Exp $
--
---------------------------------------------------------------------------

--
-- IEEE 802.3
--
ieee8023Length      FIELD
       SYNTAX UNSIGNED INT(16)

ieee8023Data  FIELD
       SYNTAX        BYTESTRING(38..1492)
       ENCAP         =llc
       LENGTH        "$ieee8023Length"
       DISPLAY-HINT  "HexDump"

ieee8023      PROTOCOL
       DESCRIPTION
           "IEEE 802.3 (Ethernet) frame"
       REFERENCE     "RFC 1042"
::= { MacDest=macAddress, MacSrc=macAddress, Length=ieee8023Length,
       Data=ieee8023Data }
```

```
--------------------------------------------------------------------------
--
--  IP.pdl - Internet Protocol (IP) definitions
--
--  Description:
--     This file contains the packet definitions for the Internet
--     Protocol.  These elements are all of the fields, templates and
--  processes required to recognize, decode and classify IP datagrams
--  found within packets.
--
--  Copyright:
--     Copyright (c) 1994-1998 Apptitude, Inc.
--       (formerly Technically Elite, Inc.)
--     All rights reserved.
--
--  RCS:
--     $Id: IP.pdl,v 1.14 1999/01/26 15:15:58 skip Exp $
--
--------------------------------------------------------------------------

--
-- The following are the fields that make up an IP datagram.
-- Some of these fields are used to recognize datagram elements, build
-- flow signatures and determine the next layer in the decode process.
--
ipVersion       FIELD
                SYNTAX INT(4)
                DEFAULT       "4"

ipHeaderLength       FIELD
                SYNTAX INT(4)

ipTypeOfService      FIELD
                SYNTAX BITSTRING(8) { minCost(1), maxReliability(2),
                       maxThruput(3), minDelay(4) }

ipLength        FIELD
                SYNTAX UNSIGNED INT(16)

--
-- This field will tell us if we need to do special processing to support
-- the payload of the datagram existing in multiple packets.
--
ipFlags              FIELD
                SYNTAX BITSTRING(3) { moreFrags(0), dontFrag(1) }

ipFragmentOffset FIELD
                SYNTAX INT(13)

--
-- This field is used to determine the children or next layer of the
-- datagram.
--
ipProtocol      FIELD
                SYNTAX INT(8)
                LOOKUP FILE "IpProtocol.cf"

ipData          FIELD
                SYNTAX       BYTESTRING(0..1500)
                ENCAP        ipProtocol
                DISPLAY-HINT "HexDump"

--
-- Detailed packet layout for the IP datagram.  This includes all fields
-- and format.  All offsets are relative to the beginning of the header.
--
ip     PROTOCOL
       SUMMARIZE
           "$FragmentOffset != 0":
               "IPFragment ID=$Identification Offset=$FragmentOffset"
           "Default" :
                 “IP Protocol=$Protocol”
        DESCRIPTION
             “Protocol format for the Internet Protocol”
        REFERENCE      “RFC 791”
::= { Version=ipVersion, HeaderLength=ipHeaderLength,
        TypeOfService=ipTypeOfService, Length=ipLength,
        Identification=UInt16, IpFlags=ipFlags,
        FragmentOffset=ipFragmentOffset, TimeToLive=Int8,
        Protocol=ipProtocol, Checksum=ByteStr2,
        IpSrc=ipAddress, IpDest=ipAddress, Options=ipOptions,
        Fragment=ipFragment, Data=ipData }

--
-- This is the description of the signature elements required to build a flow
-- that includes the IP network layer protocol.  Notice that the flow builds on
-- the lower layers.  Only the fields required to complete IP are included.
-- This flow requires the support of the fragmentation engine as well as the
-- potential of having a tunnel.  The child field is found from the IP
-- protocol field.
--
ip      FLOW
        HEADER { LENGTH=HeaderLength, IN-WORDS }
        NET-LAYER {
             SOURCE=IpSrc,
             DESTINATION=IpDest,
             FRAGMENTATION=IPV4,
             TUNNELING
        }
        CHILDREN { DESTINATION=Protocol }

ipFragData      FIELD
                SYNTAX          BYTESTRING(1..1500)
                LENGTH          “$ipLength - $ipHeaderLength * 4”
                DISPLAY-HINT    “HexDump”

ipFragment      GROUP
                OPTIONAL        “$FragmentOffset != 0”
::= { Data=ipFragData }

ipOptionCode    FIELD
                SYNTAX INT(8) { ipRR(0x07), ipTimestamp(0x44),
                        ipLSRR(0x83), ipSSRR(0x89) }
                DESCRIPTION
                     “IP option code”

ipOptionLength          FIELD
                SYNTAX UNSIGNED INT(8)
                DESCRIPTION
                     “Length of IP option”

ipOptionData    FIELD
                SYNTAX          BYTESTRING(0..1500)
                ENCAP           ipOptionCode
                DISPLAY-HINT    “HexDump”

ipOptions       GROUP
                LENGTH          “($ipHeaderLength * 4) - 20”
::= { Code=ipOptionCode, Length=ipOptionLength, Pointer=UInt8,
        Data=ipOptionData }
```

```
----------------------------------------------------------------------
--
--  TCP.pdl - Transmission Control Protocol (TCP) definitions
--
--  Description:
--     This file contains the packet definitions for the Transmission
--     Control Protocol.  This protocol is a transport service for
--  the IP protocol.  In addition to extracting the protocol information
--  the TCP protocol assists in the process of identification of connections
--  for the processing of states.
--
--  Copyright:
--     Copyright (c) 1994-1998 Apptitude, Inc.
--       (formerly Technically Elite, Inc.)
--     All rights reserved.
--
--  RCS:
--     $Id: TCP.pdl,v 1.9 1999/01/26 15:16:02 skip Exp $
--
----------------------------------------------------------------------
--
-- This is the 16 bit field where the child protocol is located for
-- the next layer beyond TCP.
--
tcpPort FIELD
       SYNTAX UNSIGNED INT(16)
       LOOKUP FILE "TcpPort.cf"

tcpHeaderLen FIELD
       SYNTAX INT(4)

tcpFlags FIELD
       SYNTAX BITSTRING(12) { fin(0), syn(1), rst(2), psh(3), ack(4), urg(5) }

tcpData FIELD
       SYNTAX        BYTESTRING(0..1564)
       LENGTH        "($ipLength - ($ipHeaderLength * 4)) - ($tcpHeaderLen * 4)"
       ENCAP         tcpPort
       DISPLAY-HINT  "HexDump"

--
-- The layout of the TCP datagram found in a packet.  Offset based on the
-- beginning of the header for TCP.
--
tcp PROTOCOL
       SUMMARIZE
           "Default" :
              "TCP ACK=$Ack WIN=$WindowSize"
       DESCRIPTION
           "Protocol format for the Transmission Control Protocol"
       REFERENCE     "RFC 793"
::= { SrcPort=tcpPort, DestPort=tcpPort, SequenceNum=UInt32,
       Ack=UInt32, HeaderLength=tcpHeaderLen, TcpFlags=tcpFlags,
       WindowSize=UInt16, Checksum=ByteStr2,
       UrgentPointer=UInt16, Options=tcpOptions, Data=tcpData }

--
-- The flow elements required to build a key for a TCP datagram.
-- Noticed that this FLOW description has a CONNECTION section. This is
-- used to describe what connection state is reached for each setting
-- of the TcpFlags field.
--
tcp      FLOW
        HEADER { LENGTH=HeaderLength, IN-WORDS }
        CONNECTION {
            IDENTIFIER=SequenceNum,
            CONNECT-START="TcpFlags:1",
            CONNECT-COMPLETE="TcpFlags:4",
            DISCONNECT-START="TcpFlags:0",
            DISCONNECT-COMPLETE="TcpFlags:4"
        }
        PAYLOAD { INCLUDE-HEADER }
        CHILDREN { DESTINATION=DestPort, SOURCE=SrcPort }

tcpOptionKind FIELD
              SYNTAX UNSIGNED INT(8) { tcpOptEnd(0), tcpNop(1), tcpMSS(2),
                            tcpWscale(3), tcpTimestamp(4) }
              DESCRIPTION
                  "Type of TCP option"

tcpOptionData FIELD
              SYNTAX        BYTESTRING(0..1500)
              ENCAP         tcpOptionKind
              FLAGS         SAMELAYER
              DISPLAY-HINT  "HexDump"

tcpOptions    GROUP
              LENGTH        "($tcpHeaderLen * 4) - 20"
--            SUMMARIZE
--                "Default" :
--                   "Option=$Option, Len=$OptionLength, $OptionData"
::= { Option=tcpOptionKind, OptionLength=UInt8, OptionData=tcpOptionData }

tcpMSS        PROTOCOL
::= { MaxSegmentSize=UInt16 }
```

```
-----------------------------------------------------------------------------
--
--  UDP.pdl - User Datagram Protocol (UDP) definitions
--
--  Description:
--     This file contains the packet definitions for the User Datagram
--     Protocol.
--
--  Copyright:
--     Copyright (c) 1994-1998 Apptitude, Inc.
--       (formerly Technically Elite, Inc.)
--     All rights reserved.
--
--  RCS:
--     $Id: UDP.pdl,v 1.9 1999/01/26 15:16:02 skip Exp $
--
-----------------------------------------------------------------------------
udpPort      FIELD
       SYNTAX UNSIGNED INT(16)
       LOOKUP FILE "UdpPort.cf"

udpLength FIELD
       SYNTAX        UNSIGNED INT(16)

udpData FIELD
       SYNTAX        BYTESTRING(0..1500)
       ENCAP         udpPort
       DISPLAY-HINT  "HexDump"

udp    PROTOCOL
       SUMMARIZE
           "Default" :
              "UDP Dest=$DestPort Src=$SrcPort"
       DESCRIPTION
           "Protocol format for the User Datagram Protocol."
       REFERENCE     "RFC 768"
 ::= { SrcPort=udpPort, DestPort=udpPort, Length=udpLength,
       Checksum=ByteStr2, Data=udpData }

udp    FLOW
       HEADER { LENGTH=8 }
       CHILDREN { DESTINATION=DestPort, SOURCE=SrcPort }
```

```
------------------------------------------------------------------------
--
-- RPC.pdl - Remote Procedure Calls (RPC) definitions
--
-- Description:
--    This file contains the packet definitions for Remote Procedure
--    Calls.
--
-- Copyright:
--    Copyright (c) 1994-1999 Apptitude, Inc.
--      (formerly Technically Elite, Inc.)
--    All rights reserved.
--
-- RCS:
--    $Id: RPC.pdl,v 1.7 1999/01/26 15:16:01 skip Exp $
--
------------------------------------------------------------------------
rpcType       FIELD
       SYNTAX UNSIGNED INT(32) { rpcCall(0), rpcReply(1) }

rpcData       FIELD
       SYNTAX        BYTESTRING(0..100)
       ENCAP         rpcType
       FLAGS         SAMELAYER
       DISPLAY-HINT  “HexDump”

rpc    PROTOCOL
       SUMMARIZE
           “$Type == rpcCall” :
             “RPC $Program”
           “$ReplyStatus == rpcAcceptedReply” :
             “RPC Reply Status=$Status”
           “$ReplyStatus == rpcDeniedReply” :
             “RPC Reply Status=$Status, AuthStatus=$AuthStatus”
           “Default” :
             “RPC $Program”
       DESCRIPTION
           “Protocol format for RPC”
       REFERENCE
           “RFC 1057”
::= { XID=UInt32, Type=rpcType, Data=rpcData }

rpc    FLOW
       HEADER { LENGTH=0 }
       PAYLOAD { DATA=XID, LENGTH=256 }

-------------
-- RPC Call
-------------
rpcProgram FIELD
       SYNTAX UNSIGNED INT(32) { portMapper(100000), nfs(100003),
              mount(100005), lockManager(100021), statusMonitor(100024) }

rpcProcedure GROUP
       SUMMARIZE
           “Default” :
             “Program=$Program, Version=$Version, Procedure=$Procedure”
::= { Program=rpcProgram, Version=UInt32, Procedure=UInt32 }

rpcAuthFlavor FIELD
       SYNTAX UNSIGNED INT(32) { null(0), unix(1), short(2) }

rpcMachine    FIELD
       SYNTAX LSTRING(4)

rpcGroup      GROUP
       LENGTH “$NumGroups * 4”
::= { Gid=Int32 }

rpcCredentials      GROUP
       LENGTH “$CredentialLength”
```

```
::= { Stamp=UInt32, Machine=rpcMachine, Uid=Int32, Gid=Int32,
      NumGroups=UInt32, Groups=rpcGroup }

rpcVerifierData     FIELD
      SYNTAX        BYTESTRING(0..400)
      LENGTH        "$VerifierLength"

rpcEncap      FIELD
      SYNTAX COMBO Program Procedure
      LOOKUP FILE "RPC.cf"

rpcCallData   FIELD
      SYNTAX        BYTESTRING(0..100)
      ENCAP         rpcEncap
      DISPLAY-HINT  "HexDump"

rpcCall       PROTOCOL
      DESCRIPTION
          "Protocol format for RPC call"
::= { RPCVersion=UInt32, Procedure=rpcProcedure,
      CredentialAuthFlavor=rpcAuthFlavor, CredentialLength=UInt32,
      Credentials=rpcCredentials,
      VerifierAuthFlavor=rpcAuthFlavor, VerifierLength=UInt32,
      Verifier=rpcVerifierData, Encap=rpcEncap, Data=rpcCallData }

-------------
-- RPC Reply
-------------
rpcReplyStatus      FIELD
      SYNTAX INT(32) { rpcAcceptedReply(0), rpcDeniedReply(1) }

rpcReplyData  FIELD
      SYNTAX        BYTESTRING(0..40000)
      ENCAP         rpcReplyStatus
      FLAGS         SAMELAYER
      DISPLAY-HINT  "HexDump"

rpcReply      PROTOCOL
      DESCRIPTION
          "Protocol format for RPC reply"
::= { ReplyStatus=rpcReplyStatus, Data=rpcReplyData }

rpcAcceptStatus     FIELD
      SYNTAX INT(32) { Success(0), ProgUnavail(1), ProgMismatch(2),
             ProcUnavail(3), GarbageArgs(4), SystemError(5) }

rpcAcceptEncap      FIELD
      SYNTAX BYTESTRING(0)
      FLAGS  NOSHOW

rpcAcceptData FIELD
      SYNTAX        BYTESTRING(0..40000)
      ENCAP         rpcAcceptEncap
      DISPLAY-HINT  "HexDump"

rpcAcceptedReply PROTOCOL
::= { VerifierAuthFlavor=rpcAuthFlavor, VerifierLength=UInt32,
      Verifier=rpcVerifierData, Status=rpcAcceptStatus,
      Encap=rpcAcceptEncap, Data=rpcAcceptData }

rpcDeniedStatus     FIELD
      SYNTAX INT(32) { rpcVersionMismatch(0), rpcAuthError(1) }

rpcAuthStatus FIELD
      SYNTAX INT(32) { Okay(0), BadCredential(1), RejectedCredential(2),
             BadVerifier(3), RejectedVerifier(4), TooWeak(5),
             InvalidResponse(6), Failed(7) }

rpcDeniedReply      PROTOCOL
::= { Status=rpcDeniedStatus, AuthStatus=rpcAuthStatus }
```

```
--------------------
-- RPC Transactions
--------------------
rpcBindLookup PROTOCOL
        SUMMARIZE
             “Default” :
                “RPC GetPort Prog=$Prog, Ver=$Ver, Proto=$Protocol”
::= { Prog=rpcProgram, Ver=UInt32, Protocol=UInt32 }

rpcBindLookupReply PROTOCOL
        SUMMARIZE
             “Default” :
                “RPC GetPortReply Port=$Port”
::= { Port=UInt32 }
```

```
--------------------------------------------------------------------------------
--
--  NFS.pdl - Network File System (NFS) definitions
--
--  Description:
--     This file contains the packet definitions for the Network File
--     System.
--
--  Copyright:
--     Copyright (c) 1994-1998 Apptitude, Inc.
--       (formerly Technically Elite, Inc.)
--     All rights reserved.
--
--  RCS:
--     $Id: NFS.pdl,v 1.3 1999/01/26 15:15:59 skip Exp $
--
--------------------------------------------------------------------------------
nfsString    FIELD
       SYNTAX LSTRING(4)

nfsHandle    FIELD
       SYNTAX        BYTESTRING(32)
       DISPLAY-HINT  “16x\n         “

nfsData              FIELD
       SYNTAX        BYTESTRING(0..100)
       DISPLAY-HINT  “HexDump”

nfsAccess    PROTOCOL
       SUMMARIZE
           “Default” :
              “NFS Access $Filename”
::= { Handle=nfsHandle, Filename=nfsString }

nfsStatus    FIELD
       SYNTAX INT(32) { OK(0), NoSuchFile(2) }

nfsAccessReply       PROTOCOL
       SUMMARIZE
           “Default” :
              “NFS AccessReply $Status”
::= { Status=nfsStatus }

nfsMode              FIELD
       SYNTAX UNSIGNED INT(32)
       DISPLAY-HINT  “4o”

nfsCreate    PROTOCOL
       SUMMARIZE
           “Default” :
              “NFS Create $Filename”
::= { Handle=nfsHandle, Filename=nfsString, Filler=Int8, Mode=nfsMode,
       Uid=Int32, Gid=Int32, Size=Int32, AccessTime=Int64, ModTime=Int64 }

nfsFileType  FIELD
       SYNTAX INT(32) { Regular(1), Directory(2) }

nfsCreateReply       PROTOCOL
       SUMMARIZE
           “Default” :
              “NFS CreateReply $Status”
::= { Status=nfsStatus, Handle=nfsHandle, FileType=nfsFileType,
       Mode=nfsMode, Links=UInt32, Uid=Int32, Gid=Int32, Size=Int32,
       BlockSize=Int32, NumBlocks=Int64, FileSysId=UInt32, FileId=UInt32,
       AccessTime=Int64, ModTime=Int64, InodeChangeTime=Int64 }

nfsRead      PROTOCOL
       SUMMARIZE
           “Default” :
              “NFS Read Offset=$Offset Length=$Length”
::= { Length=Int32, Handle=nfsHandle, Offset=UInt64, Count=Int32 }
```

```
nfsReadReply  PROTOCOL
        SUMMARIZE
             "Default" :
                "NFS ReadReply $Status"
::= { Status=nfsStatus, FileType=nfsFileType,
        Mode=nfsMode, Links=UInt32, Uid=Int32, Gid=Int32, Size=Int32,
        BlockSize=Int32, NumBlocks=Int64, FileSysId=UInt32, FileId=UInt32,
        AccessTime=Int64, ModTime=Int64, InodeChangeTime=Int64 }

nfsWrite PROTOCOL
        SUMMARIZE
             "Default" :
                "NFS Write Offset=$Offset"
::= { Handle=nfsHandle, Offset=Int32, Data=nfsData }

nfsWriteReply PROTOCOL
        SUMMARIZE
             "Default" :
                "NFS WriteReply $Status"
::= { Status=nfsStatus, FileType=nfsFileType,
        Mode=nfsMode, Links=UInt32, Uid=Int32, Gid=Int32, Size=Int32,
        BlockSize=Int32, NumBlocks=Int64, FileSysId=UInt32, FileId=UInt32,
        AccessTime=Int64, ModTime=Int64, InodeChangeTime=Int64 }

nfsReadDir    PROTOCOL
        SUMMARIZE
             "Default" :
                "NFS ReadDir"
::= { Handle=nfsHandle, Cookie=Int32, Count=Int32 }

nfsReadDirReply      PROTOCOL
        SUMMARIZE
             "Default" :
                "NFS ReadDirReply $Status"
::= { Status=nfsStatus, Data=nfsData }

nfsGetFileAttr       PROTOCOL
        SUMMARIZE
             "Default" :
                "NFS GetAttr"
::= { Handle=nfsHandle }

nfsGetFileAttrReply PROTOCOL
        SUMMARIZE
             "Default" :
                "NFS GetAttrReply $Status $FileType"
::= { Status=nfsStatus, FileType=nfsFileType,
        Mode=nfsMode, Links=UInt32, Uid=Int32, Gid=Int32, Size=Int32,
        BlockSize=Int32, NumBlocks=Int64, FileSysId=UInt32, FileId=UInt32,
        AccessTime=Int64, ModTime=Int64, InodeChangeTime=Int64 }

nfsReadLink   PROTOCOL
        SUMMARIZE
             "Default" :
                "NFS ReadLink"
::= { Handle=nfsHandle }

nfsReadLinkReply PROTOCOL
        SUMMARIZE
             "Default" :
                "NFS ReadLinkReply Path=$Path"
::= { Status=nfsStatus, Path=nfsString }

nfsMount      PROTOCOL
        SUMMARIZE
             "Default" :
                "NFS Mount $Path"
::= { Path=nfsString }

nfsMountReply PROTOCOL
```

```
        SUMMARIZE
            "Default" :
                "NFS MountReply $MountStatus"
::= { MountStatus=nfsStatus, Handle=nfsHandle }

nfsStatFs    PROTOCOL
        SUMMARIZE
            "Default" :
                "NFS StatFs"
::= { Handle=nfsHandle }

nfsStatFsReply      PROTOCOL
        SUMMARIZE
            "Default" :
                "NFS StatFsReply $Status"
::= { Status=nfsStatus, TransferSize=UInt32, BlockSize=UInt32,
        TotalBlocks=UInt32, FreeBlocks=UInt32, AvailBlocks=UInt32 }

nfsRemoveDir PROTOCOL
        SUMMARIZE
            "Default" :
                "NFS RmDir $Name"
::= { Handle=nfsHandle, Name=nfsString }

nfsRemoveDirReply PROTOCOL
        SUMMARIZE
            "Default" :
                "NFS RmDirReply $Status"
::= { Status=nfsStatus }

nfsMakeDir   PROTOCOL
        SUMMARIZE
            "Default" :
                "NFS MkDir $Name"
::= { Handle=nfsHandle, Name=nfsString }

nfsMakeDirReply PROTOCOL
        SUMMARIZE
            "Default" :
                "NFS MkDirReply $Status"
::= { Status=nfsStatus }

nfsRemove    PROTOCOL
        SUMMARIZE
            "Default" :
                "NFS Remove $Name"
::= { Handle=nfsHandle, Name=nfsString }

nfsRemoveReply PROTOCOL
        SUMMARIZE
            "Default" :
                "NFS RemoveReply $Status"
::= { Status=nfsStatus }
```

```
----------------------------------------------------------------------------
--
--  HTTP.pdl - Hypertext Transfer Protocol (HTTP) definitions
--
--  Description:
--      This file contains the packet definitions for the Hypertext Transfer
--      Protocol.
--
--  Copyright:
--      Copyright (c) 1994-1999 Apptitude, Inc.
--      (formerly Technically Elite, Inc.)
--      All rights reserved.
--
--  RCS:
--      $Id: HTTP.pdl,v 1.13 1999/04/13 15:47:57 skip Exp $
--
----------------------------------------------------------------------------
httpData FIELD
         SYNTAX         BYTESTRING(1..1500)
         LENGTH            "($ipLength - ($ipHeaderLength * 4)) - ($tcpHeaderLen * 4)"
         DISPLAY-HINT   "Text"
         FLAGS          NOLABEL

http    PROTOCOL
        SUMMARIZE
            "$httpData m/^GET|^HTTP|^HEAD|^POST/" :
                "HTTP $httpData"
            "$httpData m/^[Dd]ate|^[Ss]erver|^[Ll]ast-[Mm]odified/" :
                "HTTP $httpData"
            "$httpData m/^[Cc]ontent-/" :
                "HTTP $httpData"
            "$httpData m/^<HTML>/" :
                "HTTP [HTML document]"
            "$httpData m/^GIF/" :
                "HTTP [GIF image]"
            "Default" :
                "HTTP [Data]"
        DESCRIPTION
            "Protocol format for HTTP."
::= { Data=httpData }

http    FLOW
        CONNECTION { INHERITED }
        PAYLOAD { INCLUDE-HEADER, DATA=Data, LENGTH=256 }
        STATES
           "S0: CHECKCONNECT, GOTO S1
                DEFAULT NEXT S0

            S1: WAIT 2, GOTO S2, NEXT S1
                DEFAULT NEXT S0

            S2: MATCH
                  '\n\r\n'           900 0 0 255 0, NEXT S3
                  '\n\n'             900 0 0 255 0, NEXT S3
                  'POST /tds?'        50 0 0 127 1, CHILD sybaseWebsql
                  '.hts HTTP/1.0'     50 4 0 127 1, CHILD sybaseJdbc
                  'jdbc:sybase:Tds'   50 4 0 127 1, CHILD sybaseTds
                  'PCN-The Poin'     500 4 1 255 0, CHILD pointcast
                  't: BW-C-'         100 4 1 255 0, CHILD backweb
                  DEFAULT NEXT S3

            S3: MATCH
                  '\n\r\n'            50 0 0   0 0, NEXT S3
                  '\n\n'              50 0 0   0 0, NEXT S3
                  'Content-Type:'    800 0 0 255 0, CHILD mime
                  'PCN-The Poin'     500 4 1 255 0, CHILD pointcast
                  't: BW-C-'         100 4 1 255 0, CHILD backweb
                  DEFAULT NEXT S0"

sybaseWebsql   FLOW
               STATE-BASED
```

```
sybaseJdbc      FLOW
                STATE-BASED

sybaseTds       FLOW
                STATE-BASED

pointcast       FLOW
                STATE-BASED

backweb         FLOW
                STATE-BASED

mime            FLOW
                STATE-BASED
                STATES
                  "S0:   MATCH
                           'application' 900 0 0   1 0, CHILD mimeApplication
                           'audio'       900 0 0   1 0, CHILD mimeAudio
                           'image'        50 0 0   1 0, CHILD mimeImage
                           'text'         50 0 0   1 0, CHILD mimeText
                           'video'        50 0 0   1 0, CHILD mimeVideo
                           'x-world'     500 4 1 255 0, CHILD mimeXworld
                         DEFAULT GOTO S0"

mimeApplication FLOW
                  STATE-BASED

mimeAudio   FLOW
            STATE-BASED
            STATES
             "S0: MATCH
                   'basic'              100 0 0 1 0, CHILD pdBasicAudio
                   'midi'               100 0 0 1 0, CHILD pdMidi
                   'mpeg'               100 0 0 1 0, CHILD pdMpeg2Audio
                   'vnd.rn-realaudio'   100 0 0 1 0, CHILD pdRealAudio
                   'wav'                100 0 0 1 0, CHILD pdWav
                   'x-aiff'             100 0 0 1 0, CHILD pdAiff
                   'x-midi'             100 0 0 1 0, CHILD pdMidi
                   'x-mpeg'             100 0 0 1 0, CHILD pdMpeg2Audio
                   'x-mpgurl'           100 0 0 1 0, CHILD pdMpeg3Audio
                   'x-pn-realaudio'     100 0 0 1 0, CHILD pdRealAudio
                   'x-wav'              100 0 0 1 0, CHILD pdWav
                 DEFAULT GOTO S0"

mimeImage   FLOW
            STATE-BASED

mimeText    FLOW
            STATE-BASED

mimeVideo   FLOW
            STATE-BASED

mimeXworld FLOW
            STATE-BASED

pdBasicAudio FLOW
              STATE-BASED

pdMidi         FLOW
              STATE-BASED

pdMpeg2Audio FLOW
              STATE-BASED

pdMpeg3Audio FLOW
              STATE-BASED

pdRealAudio  FLOW
              STATE-BASED
```

```
pdWav          FLOW
              STATE-BASED

pdAiff         FLOW
              STATE-BASED
```

Traffic Classification System

Protocol Definition Language (PDL) Reference Guide

Version A0.02

VERSION A0.02

Included herein is this reference on the page description laguage (PDL) whihc, in one aspect of the invention, permits the automatic generation of the databases used by the parser and analyzer sub-systems, and also allows for including new and modified protocols and applications to the capabliity of the monitor.

COPYRIGHT NOTICE

1. INTRODUCTION

The inventive Protocol Definition Language (PDL) is a special purpose language used to describe network protocols and all the fields within the protocol headers.

Within this document, protocol descriptions (PDL files) are referred to as *PDL* or *rules* when there in no risk of confusion with other types of descriptions.

PDL uses both form and organization similar to the data structure definition part of the C programming language and the PERL scripting language. Since PDL was derived from a language used to decode network packet contact, the authors have mixed the language format with the requirements of packet decoding. This results in an expressive language that is very familiar and comfortable for describing packet content and the details required representing a flow.

1.1 Summary

The PDL is a non-procedural Forth Generation language (4GL). This means is describes *what* needs to be done without describing *how* to do it. The details of *how* are hidden in the compiler and the Compiled Protocol Layout (CPL) optimization utility.

In addition, it is used to describe network flows by defining which fields are the address fields, which are the protocol type fields, etc.

Once a PDL file is written, it is compiled using the Netscope compiler (**nsc**), which produces the *MeterFlow* database (MeterFlow.db) and the Netscope database (Netscope.db). The MeterFlow database contains the flow definitions and the Netscope database contains the protocol header definitions.

These databases are used by programs like: **mfkeys**, which produces flow keys (also called flow signatures); **mfcpl**, which produces flow definitions in CPL format; **mfpkts** which produces sample packets of all known protocols; and **netscope**, which decodes Sniffer™ and tcpdump files.

Due to its size, electronic media copies of the documentation are not provided but can be made available if necessary.

1.2 Document Conventions

The following conventions will be used throughout this document:

Small `courier` typeface indicates C code examples or function names. Functions are written with parentheses after them [**`function()`**], variables are written just as their names [**`variables`**], and structure names are written prefixed with "**`struct`**" [**`struct packet`**].

Italics indicate a filename (for instance, *mworks/base/h/base.h*). Filenames will usually be written relative to the root directory of the distribution.

Constants are expressed in decimal, unless written "`0x...`", the C language notation for hexadecimal numbers.

2. PROGRAM STRUCTURE

A *MeterFlow* PDL decodes and flow set is a non-empty sequence of statements.

There are four basic types of statements or definitions available in *MeterFlow* PDL:

FIELD,
GROUP,
PROTOCOL and
FLOW.

.

2.1 FIELD Definitions

The FIELD definition is used to define a specific string of bits or bytes in the packet. The FIELD definition has the following format:

```
  Name    FIELD
          SYNTAX Type [ { Enums } ]
          DISPLAY-HINT "FormatString"
          LENGTH "Expression"
          FLAGS FieldFlags
          ENCAP FieldName [ , FieldName2 ]
          LOOKUP LookupType [ Filename ]
          ENCODING EncodingType
          DEFAULT "value"
          DESCRIPTION "Description"
```

Where only the **FIELD** and **SYNTAX** lines are required. All the other lines are attribute lines, which define special characteristics about the **FIELD**. Attribute lines are optional and may appear in any order. Each of the attribute lines are described in detail below:

2.1.1 SYNTAX Type [{ Enums }]

This attribute defines the type and, if the type is an INT, BYTESTRING, BITSTRING, or SNMPSEQUENCE type, the enumerated values for the FIELD. The currently defined types are:

INT(*numBits*)	Integer that is *numBits* bits long.
UNSIGNED INT(*numBits*)	Unsigned integer that is *numBits* bits long.
BYTESTRING(*numBytes*)	String that is *numBytes* bytes long.
BYTESTRING(*R1..R2*)	String that ranges in size from *R1* to *R2* bytes.
BITSTRING(*numBits*)	String that is *numBits* bits long.

LSTRING(*lenBytes*)	String with *lenBytes* header.
NSTRING	Null terminated string.
DNSSTRING	DNS encoded string.
SNMPOID	SNMP Object Identifier.
SNMPSEQUENCE	SNMP Sequence.
SNMPTIMETICKS	SNMP TimeTicks.
COMBO *field1 field2*	Combination pseudo field.

2.1.2 DISPLAY-HINT "FormatString"

This attribute is for specifying how the value of the FIELD is displayed. The currently supported formats are:

Numx	Print as a num byte hexidecimal number.
Numd	Print as a num byte decimal number.
Numo	Print as a num byte octal number.
Numb	Print as a num byte binary number.
Numa	Print num bytes in ASCII format.
Text	Print as ASCII text.
HexDump	Print in hexdump format.

2.1.3 LENGTH "Expression"

This attribute defines an expression for determining the FIELD's length. Expressions are arithmetic and can refer to the value of other FIELD's in the packet by adding a $ to the referenced field's name. For example, "($tcpHeaderLen *4) – 20" is a valid expression if tcpHeaderLen is another field defined for the current packet.

2.1.4 FLAGS FieldFlags

The attribute defines some special flags for a FIELD. The currently supported FieldFlags are:

SAMELA YER	Display field on the same layer as the previous field.
NOLABEL	Don't display the field name with the value.

NOSHOW	Decode the field but don't display it.
SWAPPED	The integer value is swapped.

2.1.5 ENCAP FieldName [, FieldName2]

This attribute defines how one packet is encapsulated inside another. Which packet is determined by the value of the FieldName field. If no packet is found using FieldName then FieldName2 is tried.

2.1.6 LOOKUP LookupType [Filename]

This attribute defines how to lookup the name for a particular FIELD value. The currently supported LookupTypes are:

SERVICE	Use getservbyport().
HOSTNAME	Use gethostbyaddr().
MACADDRESS	Use $METERFLOW/conf/mac2ip.cf.
FILE *file*	Use *file* to lookup value.

2.1.7 ENCODING EncodingType

This attribute defines how a FIELD is encoded. Currently, the only supported EncodingType is BER (for Basic Encoding Rules defined by ASN.1).

2.1.8 DEFAULT “value”

This attribute defines the default value to be used for this field when generating sample packets of this protocol.

2.1.9 DESCRIPTION "Description"

This attribute defines the description of the FIELD. It is used for informational purposes only.

2.2 GROUP Definitions

The GROUP definition is used to tie several related FIELDs together. The GROUP definition has the following format:

```
   Name     GROUP
            LENGTH "Expression"
            OPTIONAL "Condition"
            SUMMARIZE "Condition" : "FormatString" [
            "Condition" : "FormatString"... ]
```

```
DESCRIPTION "Description"
::= { Name=FieldOrGroup [ ,
Name=FieldOrGroup... ] }
```

Where only the GROUP and ::= lines are required. All the other lines are attribute lines, which define special characteristics for the GROUP. Attribute lines are optional and may appear in any order. Each attribute line is described in detail below:

2.2.1 LENGTH "Expression"

This attribute defines an expression for determining the GROUP's length. Expressions are arithmetic and can refer to the value of other FIELD's in the packet by adding a $ to the referenced field's name. For example, "($tcpHeaderLen *4) – 20" is a valid expression if tcpHeaderLen is another field defined for the current packet.

2.2.2 OPTIONAL "Condition"

This attribute defines a condition for determining whether a GROUP is present or not. Valid conditions are defined in the Conditions section below.

2.2.3 SUMMARIZE "Condition" : "FormatString" ["Condition" : "FormatString"...]

This attribute defines how a GROUP will be displayed in Detail mode. A different format (FormatString) can be specified for each condition (Condition). Valid conditions are defined in the Conditions section below. Any FIELD's value can be referenced within the FormatString by proceeding the FIELD's name with a $. In addition to FIELD names there are several other special $ keywords:

$LAYER	Displays the current protocol layer.
$GROUP	Displays the entire GROUP as a table.
$LABEL	Displays the GROUP label.
$*field*	Displays the *field* value (use enumerated name if available).
$:*field*	Displays the *field* value (in raw format).

2.2.4 DESCRIPTION "Description"

This attribute defines the description of the GROUP. It is used for informational purposes only.

2.2.5 ::= { Name=FieldOrGroup [, Name=FieldOrGroup...] }

This defines the order of the fields and subgroups within the GROUP.

2.3 PROTOCOL Definitions

The PROTOCOL definition is used to define the order of the FIELDs and GROUPs within the protocol header. The PROTOCOL definition has the following format:

```
Name    PROTOCOL
        SUMMARIZE "Condition" : "FormatString" [
        "Condition" : "FormatString"... ]
        DESCRIPTION "Description"
        REFERENCE "Reference"
        ::= { Name=FieldOrGroup [ ,
        Name=FieldOrGroup... ] }
```

Where only the PROTOCOL and ::= lines are required. All the other lines are attribute lines, which define special characteristics for the PROTOCOL. Attribute lines are optional and may appear in any order. Each attribute line is described in detail below:

2.3.1 SUMMARIZE "Condition" : "FormatString" ["Condition" : "FormatString"...]

This attribute defines how a PROTOCOL will be displayed in Summary mode. A different format (FormatString) can be specified for each condition (Condition). Valid conditions are defined in the Conditions section below. Any FIELD's value can be referenced within the FormatString by proceeding the FIELD's name with a $. In addition to FIELD names there are several other special $ keywords:

$LAYER	Displays the current protocol layer.
$VARBIND	Displays the entire SNMP VarBind list.
$*field*	Displays the *field* value (use enumerated name if available).
$:*field*	Displays the *field* value (in raw format).
$#*field*	Counts all occurrences of *field.*
$**field*	Lists all occurrences of *field.*

2.3.2 DESCRIPTION "Description"

This attribute defines the description of the PROTOCOL. It is used for informational purposes only.

2.3.3 REFERENCE "Reference"

This attribute defines the reference material used to determine the protocol format. It is used for informational purposes only.

2.3.4 ::= { Name=FieldOrGroup [, Name=FieldOrGroup...] }

This defines the order of the FIELDs and GROUPs within the PROTOCOL.

2.4 FLOW Definitions

The FLOW definition is used to define a network flow by describing where the address, protocol type, and port numbers are in a packet. The FLOW definition has the following format:

```
Name     FLOW
         HEADER { Option [, Option...] }
         DLC-LAYER { Option [, Option...] }
         NET-LAYER { Option [, Option...] }
         CONNECTION { Option [, Option...] }
         PAYLOAD { Option [, Option...] }
         CHILDREN { Option [, Option...] }
         STATE-BASED
         STATES "Definitions"
```

Where only the FLOW line is required. All the other lines are attribute lines, which define special characteristics for the FLOW. Attribute lines are optional and may appear in any order. However, at least one attribute line must be present. Each attribute line is described in detail below:

2.4.1 HEADER { Option [, Option...] }

This attribute is used to describe the length of the protocol header. The currently supported Options are:

LENGTH=*number*	Header is a fixed length of size *number.*
LENGTH=*field*	Header is variable length determined by value of *field.*
IN-WORDS	The units of the header length are in 32-bit words rather than bytes.

2.4.2 DLC-LAYER { Option [, Option...] }

If the protocol is a data link layer protocol, this attribute describes it. The currently supported Options are:

DESTINATION=*field*	Indicates which *field* is the DLC destination address.
SOURCE=*field*	Indicates which *field* is the DLC source address.
PROTOCOL	Indicates this is a data link layer protocol.

TUNNELING	Indicates this is a tunneling protocol.

2.4.3 NET-LAYER { Option [, Option…] }

If the protocol is a network layer protocol, then this attribute describes it. The currently supported Options are:

DESTINATION=*field*	Indicates which *field* is the network destination address.
SOURCE=*field*	Indicates which *field* is the network source address.
TUNNELING	Indicates this is a tunneling protocol.
FRAGMENTATION=*type*	Indicates this protocol supports fragmentation. There are currently two fragmentation types: IPV4 and IPV6.

2.4.4 CONNECTION { Option [, Option…] }

If the protocol is a connection-oriented protocol, then this attribute describes how connections are established and torn down. The currently supported Options are:

IDENTIFIER=*field*	Indicates the connection identifier *field*.
CONNECT-START="*flag*"	Indicates when a connection is being initiated.
CONNECT-COMPLETE="*flag*"	Indicates when a connection has been established.
DISCONNECT-START="*flag*"	Indicates when a connection is being torn down.
DISCONNECT-COMPLETE="*flag*"	Indicates when a connection has been torn down.
INHERITED	Indicates this is a connection-oriented protocol but the parent protocol is where the connection is established.

2.4.5 PAYLOAD { Option [, Option…] }

This attribute describes how much of the payload from a packet of this type should be stored for later use during analysis. The currently supported Options are:

INCLUDE-	Indicates that the protocol header should be included.

HEADER	
LENGTH=*number*	Indicates how many bytes of the payload should be stored.
DATA=*field*	Indicates which *field* contains the payload.

2.4.6 CHILDREN { Option [, Option…] }

This attribute describes how children protocols are determined. The currently supported Options are:

DESTINATION=*field*	Indicates which *field* is the destination port.
SOURCE=*field*	Indicates which *field* is the source port.
LLCCHECK=*flow*	Indicates that if the DESTINATION field is less than 0x05DC then use *flow* instead of the current flow definition.

2.4.7 STATE-BASED

This attribute indicates that the flow is a state-based flow.

2.4.8 STATES "Definitions"

This attribute describes how children flows of this protocol are determined using states. See the State Definitions section below for how these states are defined.

2.5 CONDITIONS

Conditions are used with the OPTIONAL and SUMMARIZE attributes and may consist of the following:

Value1 == Value2	Value1 equals Value2. Works with string values.
Value1 != Value2	Value1 does not equal Value2. Works with string values.
Value1 <= Value2	Value1 is less than or equal to Value2.
Value1 >= Value2	Value1 is greater than or equal to Value2.
Value1 < Value2	Value1 is less than Value2.

Value1 > Value2	Value1 is greater than Value2.
Field m/regex/	Field matches the regular expression regex.

Where *Value1* and *Value2* can be either FIELD references (field names preceded by a $) or constant values. Note that compound conditional statements (using AND and OR) are not currently supported.

2.6 STATE DEFINITIONS

Many applications running over data networks utilize complex methods of classifying traffic through the use of multiple states. State definitions are used for managing and maintaining learned states from traffic derived from the network.

The basic format of a state definition is:

```
StateName:  Operand Parameters [ Operand Parameters…]
```

The various states of a particular flow are described using the following operands:

2.6.1 CHECKCONNECT, *operand*

Checks for connection. Once connected executes *operand.*

2.6.2 GOTO *state*

Goes to *state*, using the current packet.

2.6.3 NEXT *state*

Goes to *state*, using the next packet.

2.6.4 DEFAULT *operand*

Executes *operand* when all other operands fail.

2.6.5 CHILD *protocol*

Jump to child *protocol* and perform state-based processing (if any) in the child.

2.6.6 WAIT *numPackets, operand1, operand2*

Waits the specified number of packets. Executes *operand1* when the specified number of packets have been received. Executes *operand2* when a packet is received but it is less than the number of specified packets.

2.6.7 MATCH *'string' weight offset LF-offset range LF-range, operand*

Searches for a *string* in the packet, executes *operand* if found.

2.6.8 CONSTANT *number offset range, operand*

Checks for a constant in a packet, executes *operand* if found.

2.6.9 EXTRACTIP *offset destination, operand*

Extracts an IP address from the packet and then executes *operand.*

2.6.10 EXTRACTPORT *offset destination, operand*

Extracts a port number from the packet and then executes *operand.*

2.6.11 CREATEREDIRECTEDFLOW, *operand*

Creates a redirected flow and then executes *operand.*

3. EXAMPLE PDL RULES

The following section contains several examples of PDL Rule files.

3.1 Ethernet

The following is an example of the PDL for Ethernet:

```
MacAddress    FIELD
              SYNTAX       BYTESTRING(6)
              DISPLAY-HINT "1x:"
              LOOKUP       MACADDRESS
              DESCRIPTION
                     "MAC layer physical address"

etherType     FIELD
              SYNTAX       INT(16)
              DISPLAY-HINT "1x:"
              LOOKUP       FILE "EtherType.cf"
              DESCRIPTION
                     "Ethernet type field"

etherData     FIELD
              SYNTAX       BYTESTRING(46..1500)
              ENCAP               etherType
              DISPLAY-HINT "HexDump"
              DESCRIPTION
                  "Ethernet data"

ethernet      PROTOCOL
              DESCRIPTION
                  "Protocol format for an Ethernet frame"
              REFERENCE    "RFC 894"
::= { MacDest=macAddress, MacSrc=macAddress, EtherType=etherType,
       Data=etherData }

ethernet      FLOW
              HEADER { LENGTH=14 }
              DLC-LAYER {
                  SOURCE=MacSrc,
                  DESTINATION=MacDest,
                  TUNNELING,
                  PROTOCOL
              }
              CHILDREN { DESTINATION=EtherType, LLC-CHECK=llc }
```

3.2 IP Version 4

Here is an example of the PDL for the IP protocol:

```
ipAddress     FIELD
              SYNTAX       BYTESTRING(4)
              DISPLAY-HINT "1d."
              LOOKUP       HOSTNAME
              DESCRIPTION
                  "IP address"

ipVersion     FIELD
              SYNTAX INT(4)
              DEFAULT      "4"

ipHeaderLength      FIELD
                    SYNTAX INT(4)

ipTypeOfService     FIELD
                    SYNTAX       BITSTRING(8) { minCost(1),
                                 maxReliability(2), maxThruput(3), minDelay(4)
                                 }

ipLength            FIELD
                    SYNTAX UNSIGNED INT(16)

ipFlags             FIELD
                    SYNTAX BITSTRING(3) { moreFrags(0), dontFrag(1) }

IpFragmentOffset          FIELD
                          SYNTAX INT(13)

ipProtocol    FIELD
              SYNTAX INT(8)
              LOOKUP FILE "IpProtocol.cf"

ipData FIELD
              SYNTAX       BYTESTRING(0..1500)
              ENCAP               ipProtocol
              DISPLAY-HINT "HexDump"

ip      PROTOCOL
        SUMMARIZE
        "$FragmentOffset != 0":
              "IPFragment ID=$Identification Offset=$FragmentOffset"
        "Default" :
              "IP Protocol=$Protocol"
        DESCRIPTION
            "Protocol format for the Internet Protocol"
        REFERENCE    "RFC 791"
::= { Version=ipVersion, HeaderLength=ipHeaderLength,
        TypeOfService=ipTypeOfService, Length=ipLength,
        Identification=UInt16, IpFlags=ipFlags,
        FragmentOffset=ipFragmentOffset, TimeToLive=Int8,
        Protocol=ipProtocol, Checksum=ByteStr2,
        IpSrc=ipAddress, IpDest=ipAddress, Options=ipOptions,
        Fragment=ipFragment, Data=ipData }

ip      FLOW
        HEADER { LENGTH=HeaderLength, IN-WORDS }
        NET-LAYER {
            SOURCE=IpSrc,
            DESTINATION=IpDest,
            FRAGMENTATION=IPV4,
            TUNNELING
        }
        CHILDREN { DESTINATION=Protocol }

ipFragData    FIELD
              SYNTAX        BYTESTRING(1..1500)
              LENGTH        "ipLength - ipHeaderLength * 4"
              DISPLAY-HINT  "HexDump"

ipFragment    GROUP
              OPTIONAL      "$FragmentOffset != 0"
::= { Data=ipFragData }

ipOptionCode  FIELD
                     SYNTAX INT(8) { ipRR(0x07), ipTimestamp(0x44),
                                     ipLSRR(0x83), ipSSRR(0x89) }
                     DESCRIPTION
                         "IP option code"

ipOptionLength       FIELD
                     SYNTAX UNSIGNED INT(8)
                     DESCRIPTION
                         "Length of IP option"

ipOptionData  FIELD
                     SYNTAX        BYTESTRING(0..1500)
                     ENCAP                ipOptionCode
                     DISPLAY-HINT  "HexDump"

ipOptions     GROUP
              LENGTH        "(ipHeaderLength * 4) - 20"
::= { Code=ipOptionCode, Length=ipOptionLength, Pointer=UInt8,
        Data=ipOptionData }
```

3.3 TCP

Here is an example of the PDL for the TCP protocol:

```
tcpPort FIELD
        SYNTAX UNSIGNED INT(16)
        LOOKUP FILE "TcpPort.cf"

tcpHeaderLen FIELD
        SYNTAX INT(4)

tcpFlags FIELD
        SYNTAX BITSTRING(12) { fin(0), syn(1), rst(2), psh(3),
                                        ack(4), urg(5) }

tcpData FIELD
        SYNTAX BYTESTRING(0..1564)
        LENGTH "($ipLength-($ipHeaderLength*4))-($tcpHeaderLen*4)"
        ENCAP        tcpPort
        DISPLAY-HINT "HexDump"

tcp     PROTOCOL
        SUMMARIZE
             "Default" :
                "TCP ACK=$Ack WIN=$WindowSize"
        DESCRIPTION
             "Protocol format for the Transmission Control Protocol"
        REFERENCE     "RFC 793"
 ::= { SrcPort=tcpPort, DestPort=tcpPort, SequenceNum=UInt32,
        Ack=UInt32, HeaderLength=tcpHeaderLen, TcpFlags=tcpFlags,
        WindowSize=UInt16, Checksum=ByteStr2,
        UrgentPointer=UInt16, Options=tcpOptions, Data=tcpData }

tcp     FLOW
        HEADER { LENGTH=HeaderLength, IN-WORDS }
        CONNECTION {
              IDENTIFIER=SequenceNum,
              CONNECT-START="TcpFlags:1",
              CONNECT-COMPLETE="TcpFlags:4",
              DISCONNECT-START="TcpFlags:0",
              DISCONNECT-COMPLETE="TcpFlags:4"
         }
        PAYLOAD { INCLUDE-HEADER }
        CHILDREN { DESTINATION=DestPort, SOURCE=SrcPort }

tcpOptionKind FIELD
              SYNTAX UNSIGNED INT(8) { tcpOptEnd(0), tcpNop(1),
                                 tcpMSS(2), tcpWscale(3), tcpTimestamp(4) }
              DESCRIPTION
                  "Type of TCP option"

tcpOptionData FIELD
              SYNTAX BYTESTRING(0..1500)
              ENCAP        tcpOptionKind
              FLAGS        SAMELAYER
              DISPLAY-HINT "HexDump"

tcpOptions    GROUP
              LENGTH       "($tcpHeaderLen * 4) - 20"
::= { Option=tcpOptionKind, OptionLength=UInt8,
        OptionData=tcpOptionData }

tcpMSS PROTOCOL
::= { MaxSegmentSize=UInt16 }
```

3.4 HTTP (with State)

Here is an example of the PDL for the HTTP protocol:

```
httpData FIELD
   SYNTAX  BYTESTRING(1..1500)
   LENGTH    "($ipLength - ($ipHeaderLength * 4)) - ($tcpHeaderLen * 4)"
   DISPLAY-HINT     "Text"
   FLAGS            NOLABEL

http     PROTOCOL
         SUMMARIZE
             "$httpData m/^GET|^HTTP|^HEAD|^POST/" :
                 "HTTP $httpData"
             "$httpData m/^[Dd]ate|^[Ss]erver|^[Ll]ast-[Mm]odified/" :
                 "HTTP $httpData"
             "$httpData m/^[Cc]ontent-/" :
                 "HTTP $httpData"
             "$httpData m/^<HTML>/" :
                 "HTTP [HTML document]"
             "$httpData m/^GIF/" :
                 "HTTP [GIF image]"
             "Default" :
                 "HTTP [Data]"
         DESCRIPTION
             "Protocol format for HTTP."
::= { Data=httpData }

http   FLOW
       HEADER { LENGTH=0 }
       CONNECTION { INHERITED }
       PAYLOAD { INCLUDE-HEADER, DATA=Data, LENGTH=256 }
       STATES
           "S0: CHECKCONNECT, GOTO S1
                DEFAULT NEXT S0

            S1: WAIT 2, GOTO S2, NEXT S1
                DEFAULT NEXT S0

            S2: MATCH
                  '\n\r\n'           900 0 0 255 0, NEXT S3
                  '\n\n'             900 0 0 255 0, NEXT S3
                  'POST /tds?'        50 0 0 127 1, CHILD sybaseWebsql
                  '.hts HTTP/1.0'     50 4 0 127 1, CHILD sybaseJdbc
                  'jdbc:sybase:Tds'   50 4 0 127 1, CHILD sybaseTds
                  'PCN-The Poin'     500 4 1 255 0, CHILD pointcast
                  't: BW-C-'         100 4 1 255 0, CHILD backweb
                  DEFAULT NEXT S3

            S3: MATCH
                  '\n\r\n'            50 0 0   0 0, NEXT S3
                  '\n\n'              50 0 0   0 0, NEXT S3
                  'Content-Type:'    800 0 0 255 0, CHILD mime
                  'PCN-The Poin'     500 4 1 255 0, CHILD pointcast
                  't: BW-C-'         100 4 1 255 0, CHILD backweb
                  DEFAULT NEXT S0"

sybaseWebsql    FLOW
                STATE-BASED

sybaseJdbc      FLOW
                STATE-BASED

sybaseTds       FLOW
                STATE-BASED
```

```
pointcast      FLOW
               STATE-BASED

backweb        FLOW
               STATE-BASED

mime           FLOW
               STATE-BASED
               STATES
                 "S0:   MATCH
                         'application' 900 0 0   1 0, CHILD mimeApplication
                         'audio'       900 0 0   1 0, CHILD mimeAudio
                         'image'        50 0 0   1 0, CHILD mimeImage
                         'text'         50 0 0   1 0, CHILD mimeText
                         'video'        50 0 0   1 0, CHILD mimeVideo
                         'x-world'     500 4 1 255 0, CHILD mimeXworld
                    DEFAULT GOTO S0"

mimeApplication FLOW
                STATE-BASED

mimeAudio   FLOW
            STATE-BASED
            STATES
             "S0: MATCH
                    'basic'              100 0 0 1 0, CHILD pdBasicAudio
                    'midi'               100 0 0 1 0, CHILD pdMidi
                    'mpeg'               100 0 0 1 0, CHILD pdMpeg2Audio
                    'vnd.rn-realaudio'   100 0 0 1 0, CHILD pdRealAudio
                    'wav'                100 0 0 1 0, CHILD pdWav
                    'x-aiff'             100 0 0 1 0, CHILD pdAiff
                    'x-midi'             100 0 0 1 0, CHILD pdMidi
                    'x-mpeg'             100 0 0 1 0, CHILD pdMpeg2Audio
                    'x-mpgurl'           100 0 0 1 0, CHILD pdMpeg3Audio
                    'x-pn-realaudio'     100 0 0 1 0, CHILD pdRealAudio
                    'x-wav'              100 0 0 1 0, CHILD pdWav
                  DEFAULT GOTO S0"

mimeImage   FLOW
            STATE-BASED

mimeText    FLOW
            STATE-BASED

mimeVideo   FLOW
            STATE-BASED

mimeXworld  FLOW
            STATE-BASED

pdBasicAudio FLOW
             STATE-BASED

pdMidi       FLOW
             STATE-BASED

pdMpeg2Audio FLOW
             STATE-BASED

pdMpeg3Audio FLOW
             STATE-BASED

pdRealAudio  FLOW
             STATE-BASED

pdWav        FLOW
             STATE-BASED

pdAiff       FLOW
             STATE-BASED
```

As described herein, in order to derive the actual application used to communicate between a client and a server, all of the opening connection packets must be decoded, analyzed and interpreted. There could be several simultaneous and overlapping applications executing over the network that are independent and asynchronous.

Real-time application traffic classification thus includes several major challenges. First is to successfully classify each of the individual packets as they are seen on the network. The contents of the packets must be assembled into a unique flow signature to retrieve future information about the conversational flow. A flexible and intelligent processing system must analyze the content of each and every packet exchanged between the client and server in the network.

Parallel systems must operate together and simultaneously in order to meet the speed requirements of today's client/server networks. In addition, the design must be flexible enough to adapt to future applications developed for client/server networks.

Embodiments of the present invention are preferably completely implemented in application-specific integrated circuits (ASIC) or field programmable gate arrays (FPGA). A packet acquisition device is needed, such as a media access controller (MAC), or a segmentation and reassemble module. Such acquisition device is directly connected to the pattern analysis and recognition engine and is the sole input data stream for all of the packets that are analyzed and classified to the application used.

The packet parsing system preferably comprises two sub-parts, the pattern analysis and recognition engine (PAR), and the field extraction engine (FEE). The pattern analysis and recognition engine interprets each packet that is seen. Individual fields in each packet are analyzed for specific patterns through a process of elimination until a particular pattern for the packet is found.

The recognition patterns are stored in a pattern database that includes a sparsely populated three-dimensional array of patterns and links in the nodes. If a node does not include a link to a deeper level, pattern matching is declared complete. An instruction will be found at that last node in the array, and it is sent to the field extraction engine along with the packet.

The field extraction engine works on the packet contents using the extraction instructions from the pattern analysis and recognition engine. Each of the important packet elements are removed and written into a flow signature generation buffer. Once all the operations requested of the field extraction engine are completed for this packet, the flow signature is set as complete, and a hash is generated to identify this flow signature.

When the parsing system has successfully completed the task of deriving, determining and extracting the required information, the remaining pieces of the packet and the generated flow signature for the packet are passed to the packet analysis system.

All of the extracted packet elements are formulated into flow signatures that are stored in a unified flow signature buffer. Multiple flow signatures from all the packets being analyzed in parallel can be held in the one unified flow signature buffer. While a packet flow signature exists in the unified flow signature buffer, 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.

The first step in the packet analysis process is to lookup the instance in the current database of known packet flow signatures. The lookup/update engine accomplishes this task. Such engine uses the hash and remaining fields of the flow signature from the packet to determine if this packet flow record exists in the flow-entry database of the packet analysis system. Once the lookup processing has been completed the flag stating whether it is found or is new, is set within the unified flow signature buffer structure for this packet flow signature.

After the packet flow signature has been looked up and contents of the current flow signature are in the database, the 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 it's database until there are either no more left or the instruction signifies processing.

Since the sequence of packet exchanges between client and server is crucial in deriving the application component of a conversational flow, the state processor functions must be programmable. Each new application on the network may have different characteristics for identifying the components within packets. The state processor functions take into consideration this variable method of communicating in a client/server network. If during the lookup process for this particular packet flow signature, the flow is required to be inserted into the active database, the flow insertion and deletion engine is initiated. Such engine operates independently from the other two engines within the analysis system. The lookup update engine will determine whether the flow insertion and deletion engine is required to operate for a particular packet flow signature.

Monitor embodiments of the present invention create and maintain classified traffic flows, process statistics for packets and flows, manage the traffic flow-entry database and cache, and perform state-based analysis of traffic flows. In order for the monitor to successfully classify traffic by application, there are several data elements required from each packet to be analyzed. Prior to sending a packet of information to the monitor, all information must be formatted and sent along with the appropriate packet content. The monitor must specifically receive each packets in a conversational flow in the order that they are exchanged between the client and the server. The order is crucial for proper state based classification. More applications running over data networks use complex methods of classifying traffic through the creation of multiple states. The creation of the state based traffic classification causes the need for managing and maintaining learned states from traffic deduced in the network.

In preferred embodiments of the present invention, the flow lookup/update engine, flow insertion and deletion engine, state processor, cache, and unified memory controller all operate in parallel.

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 which analyzes the flow signature of each packet. A memory 324 is used to store the database of flows which are determined and updated by monitor 300. A host processor 1504, which might be any processor, for example, a general purpose processor, is used to analyze the flows in memory 324, these flows obtained via a host interface in the analyzer subsystem, (see Fig. 11). As is conventional, host processor 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 processor 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.

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.

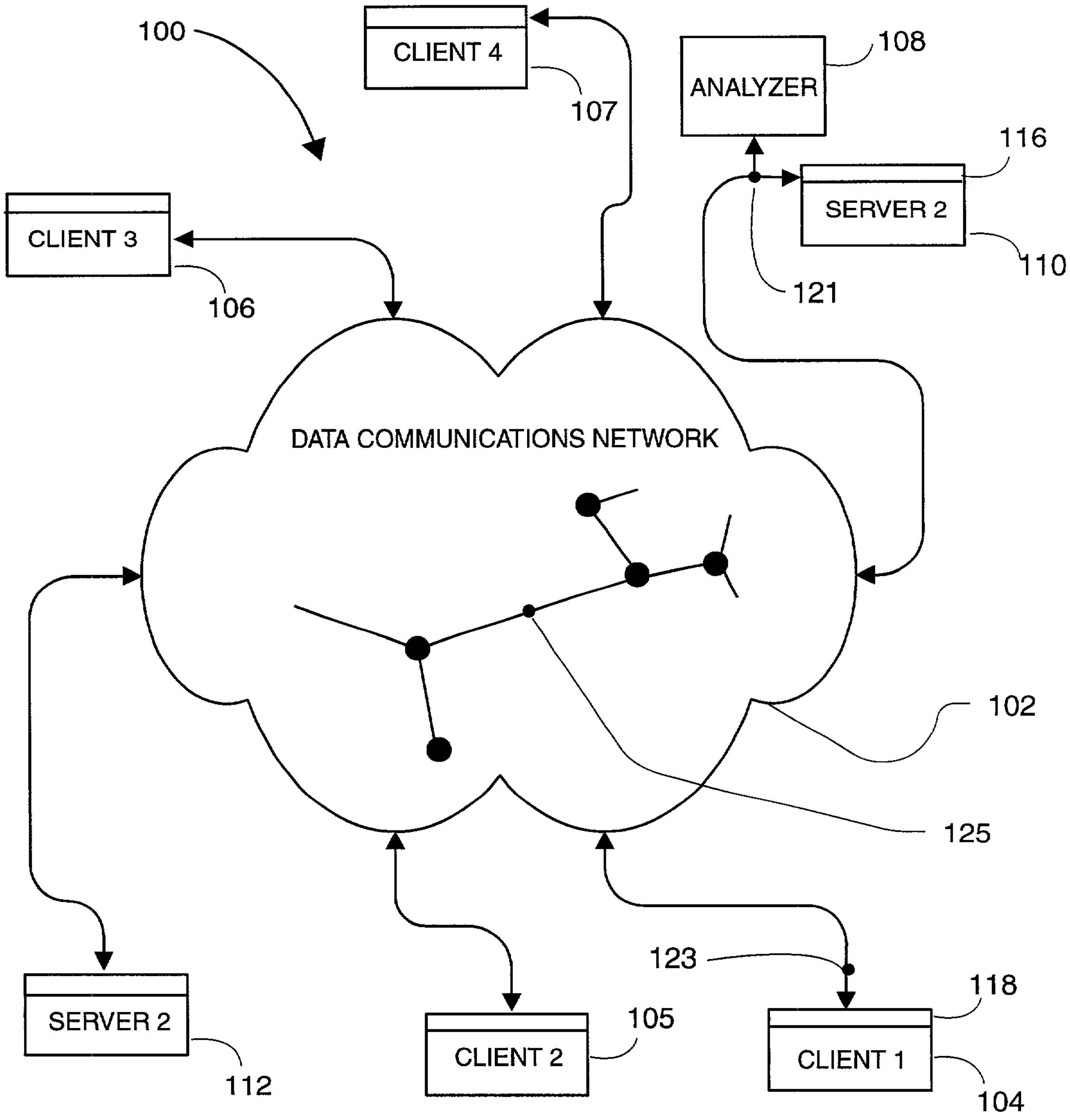
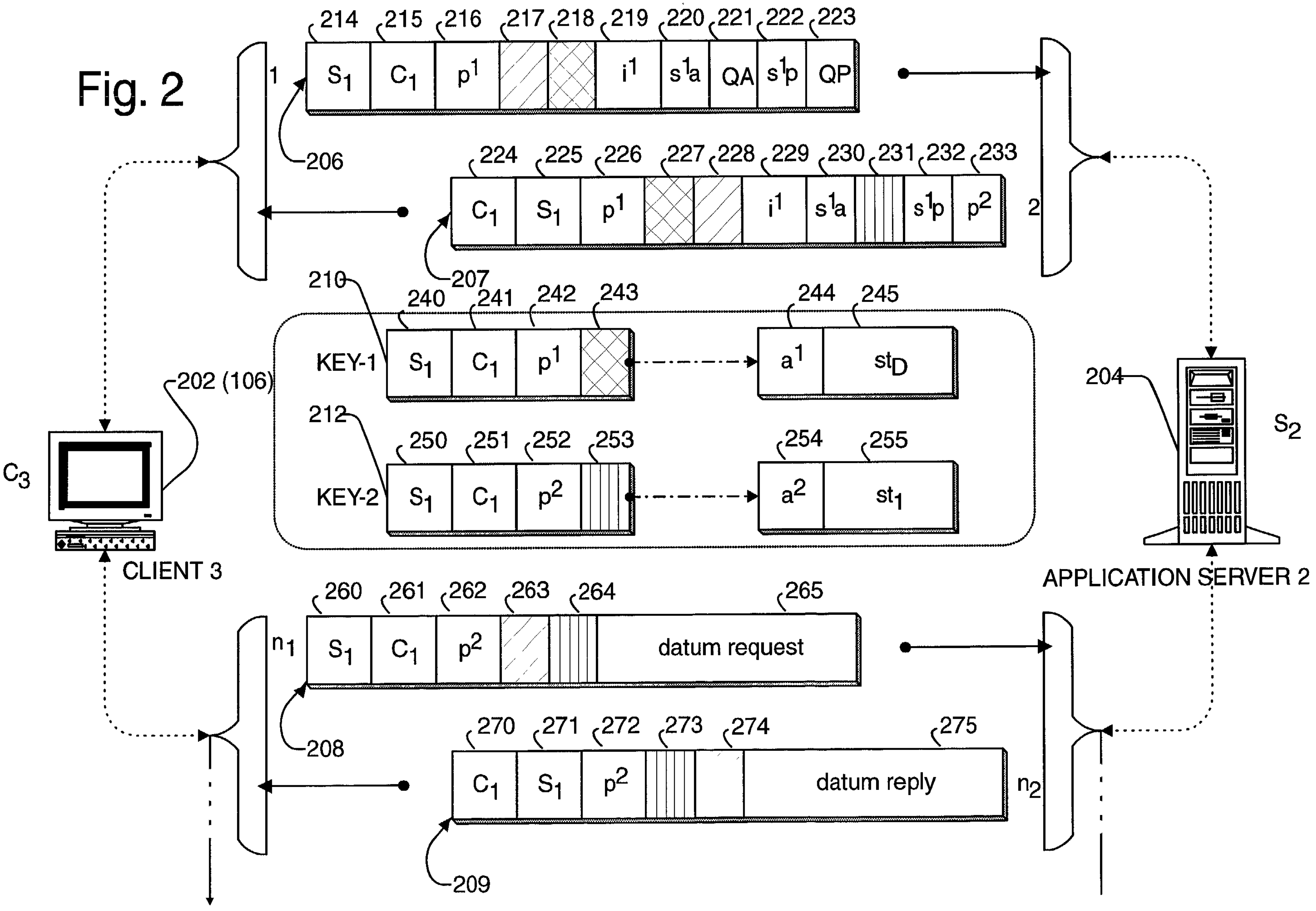

100
CLIENT 4
107
ANALYZER
108
116
SERVER 2
110
121
CLIENT 3
106
DATA COMMUNICATIONS NETWORK
102
125
123
118
SERVER 2
112
CLIENT 2
105
CLIENT 1
104

Fig. 1

Fig. 2


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

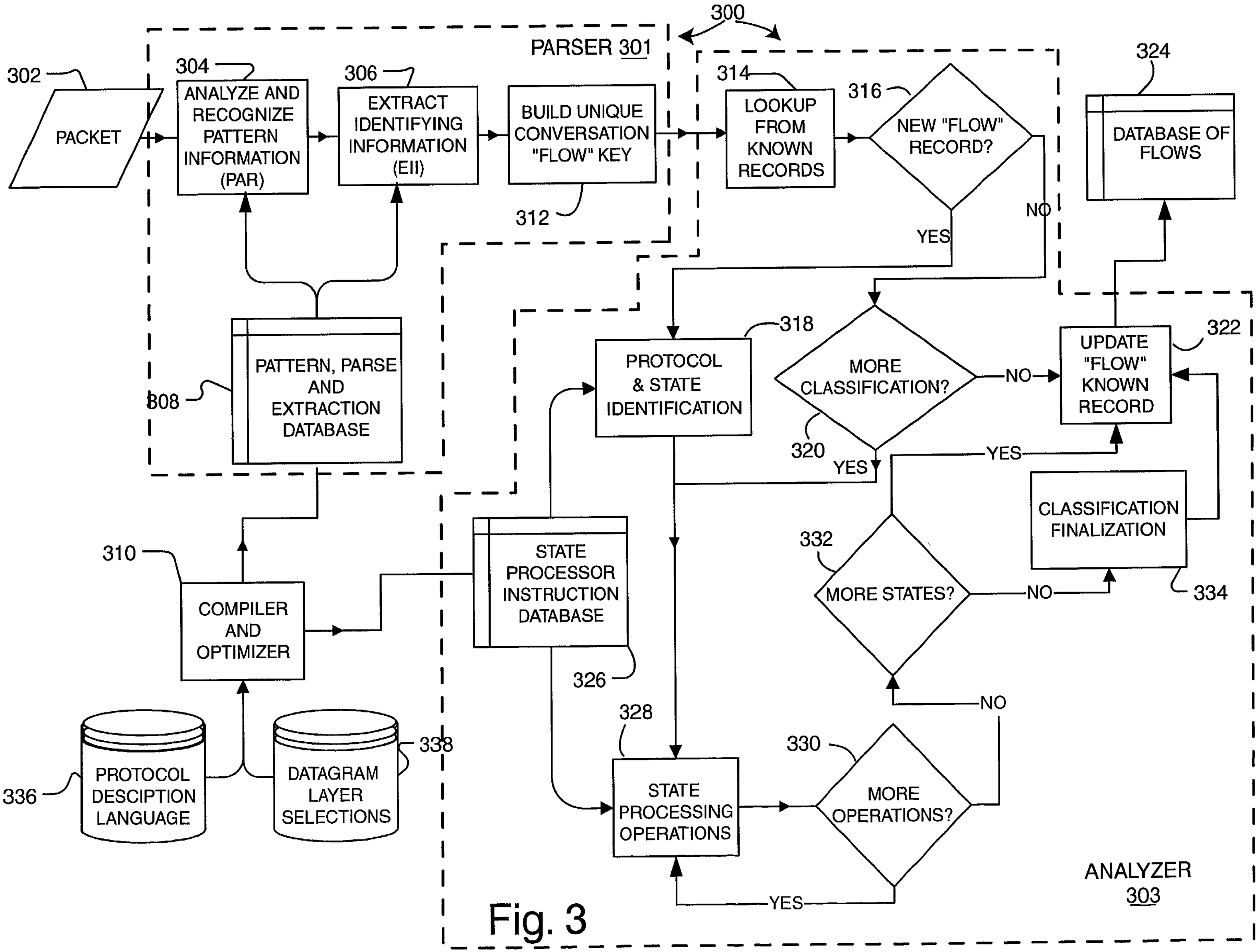

300
PARSER 301
302
PACKET
304
ANALYZE AND RECOGNIZE PATTERN INFORMATION (PAR)
306
EXTRACT IDENTIFYING INFORMATION (EII)
BUILD UNIQUE CONVERSATION "FLOW" KEY
312
308
PATTERN, PARSE AND EXTRACTION DATABASE
314
LOOKUP FROM KNOWN RECORDS
316
NEW "FLOW" RECORD?
YES
NO
324
DATABASE OF FLOWS
318
PROTOCOL &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

Fig. 3

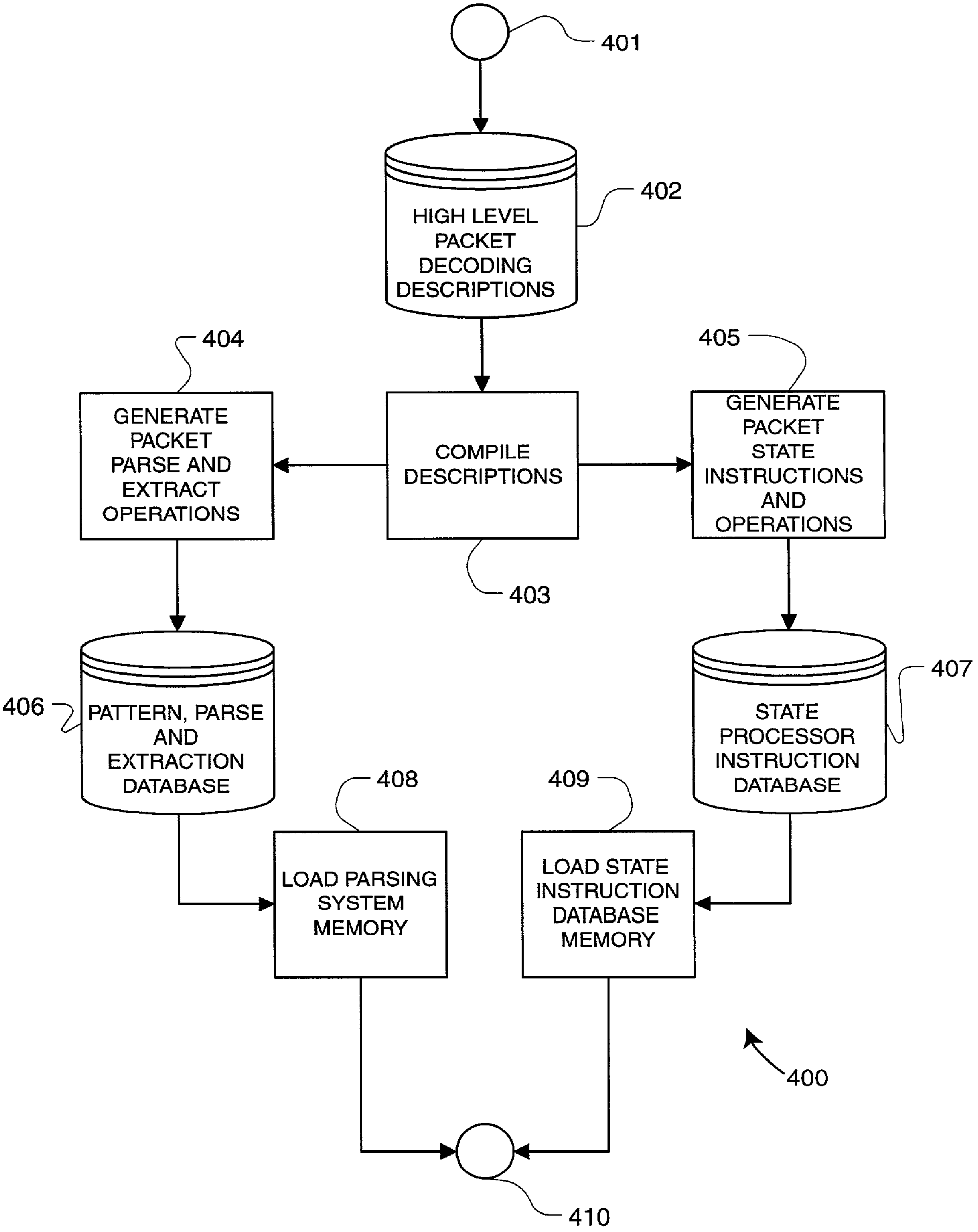

401
402
HIGH LEVEL PACKET DECODING DESCRIPTIONS
404
GENERATE PACKET PARSE AND EXTRACT OPERATIONS
COMPILE DESCRIPTIONS
403
405
GENERATE PACKET STATE INSTRUCTIONS AND OPERATIONS
406
PATTERN, PARSE AND EXTRACTION DATABASE
407
STATE PROCESSOR INSTRUCTION DATABASE
408
LOAD PARSING SYSTEM MEMORY
409
LOAD STATE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

Fig. 4

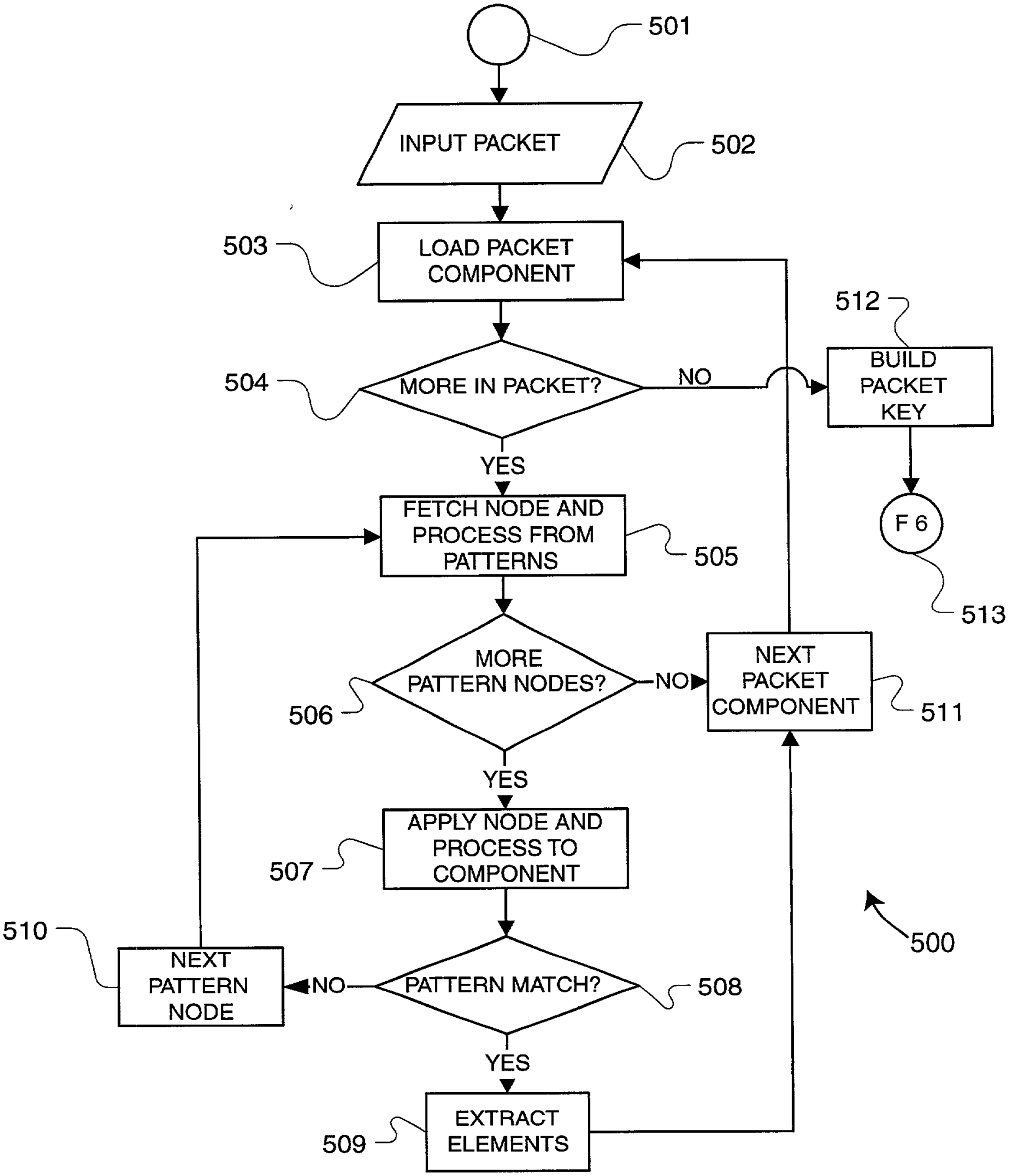

501
INPUT PACKET
502
503
LOAD PACKET COMPONENT
512
504
MORE IN PACKET?
NO
BUILD PACKET KEY
YES
FETCH NODE AND PROCESS FROM PATTERNS
505
F 6
513
MORE PATTERN NODES?
NO
NEXT PACKET COMPONENT
506
511
YES
APPLY NODE AND PROCESS TO COMPONENT
507
500
510
NEXT PATTERN NODE
NO
PATTERN MATCH?
508
YES
EXTRACT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

Fig. 5

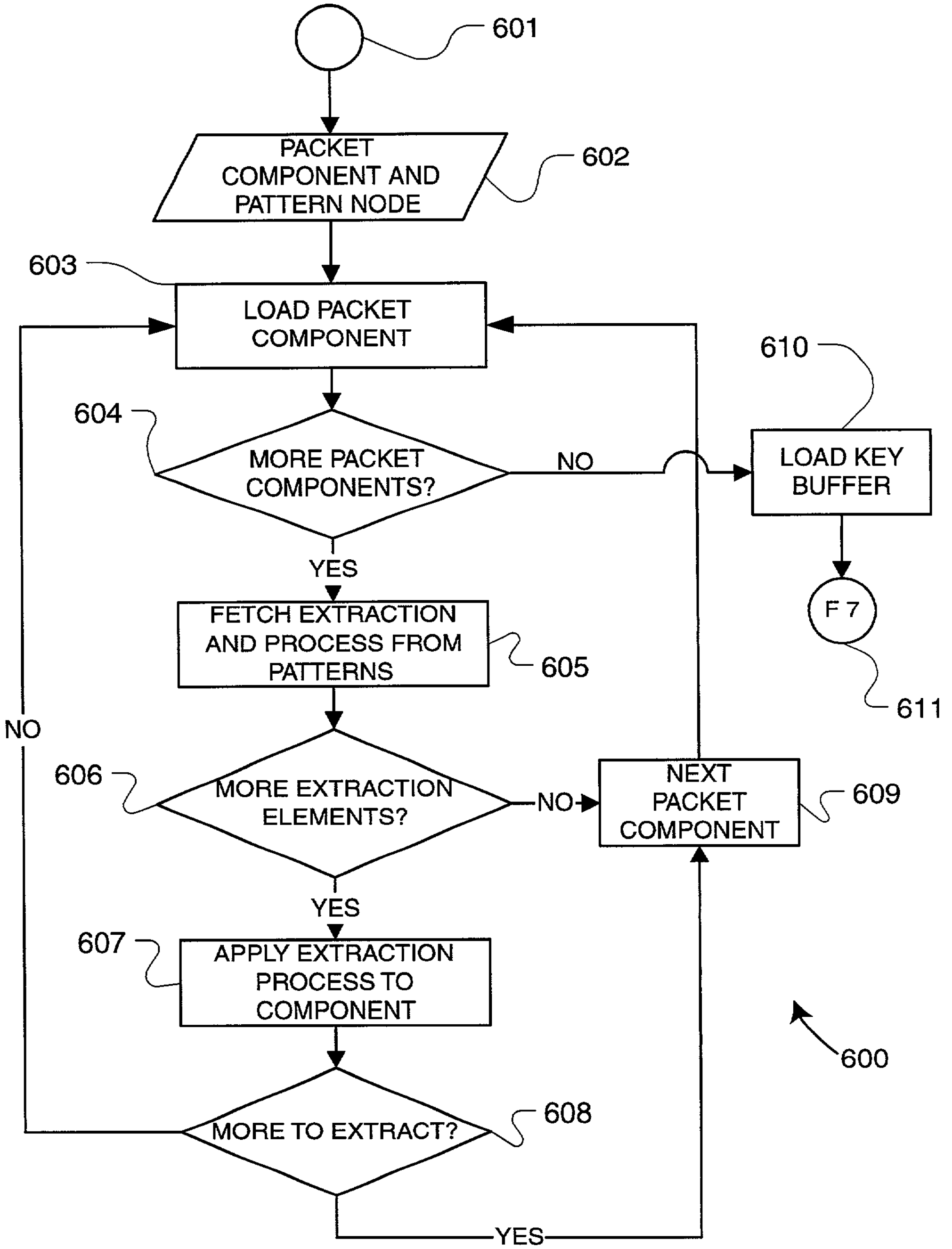

601
PACKET COMPONENT AND PATTERN NODE
602
603
LOAD PACKET COMPONENT
610
604
MORE PACKET COMPONENTS?
NO
LOAD KEY BUFFER
YES
FETCH EXTRACTION AND PROCESS FROM PATTERNS
605
F 7
611
NO
606
MORE EXTRACTION ELEMENTS?
NO
NEXT PACKET COMPONENT
609
YES
607
APPLY EXTRACTION PROCESS TO COMPONENT
600
608
MORE TO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

Fig. 6

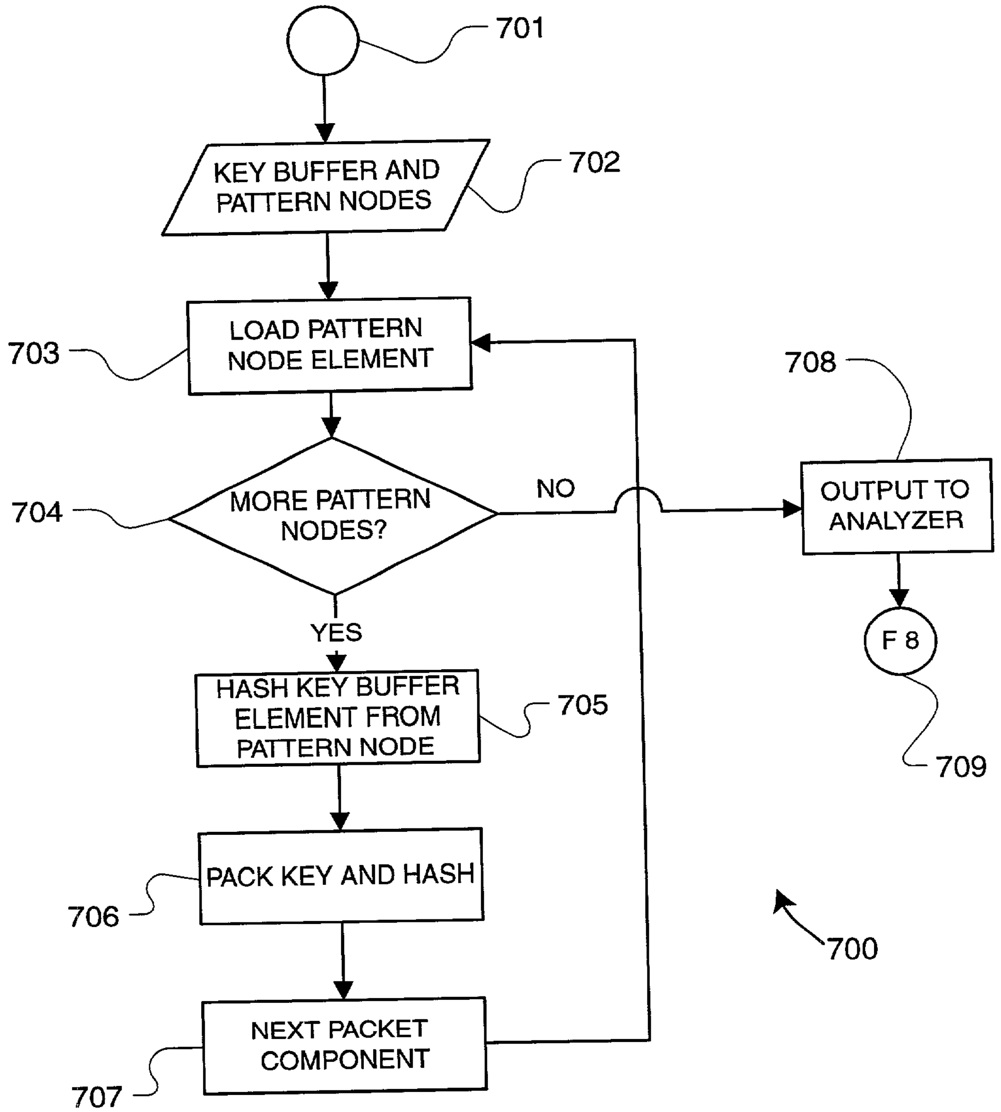
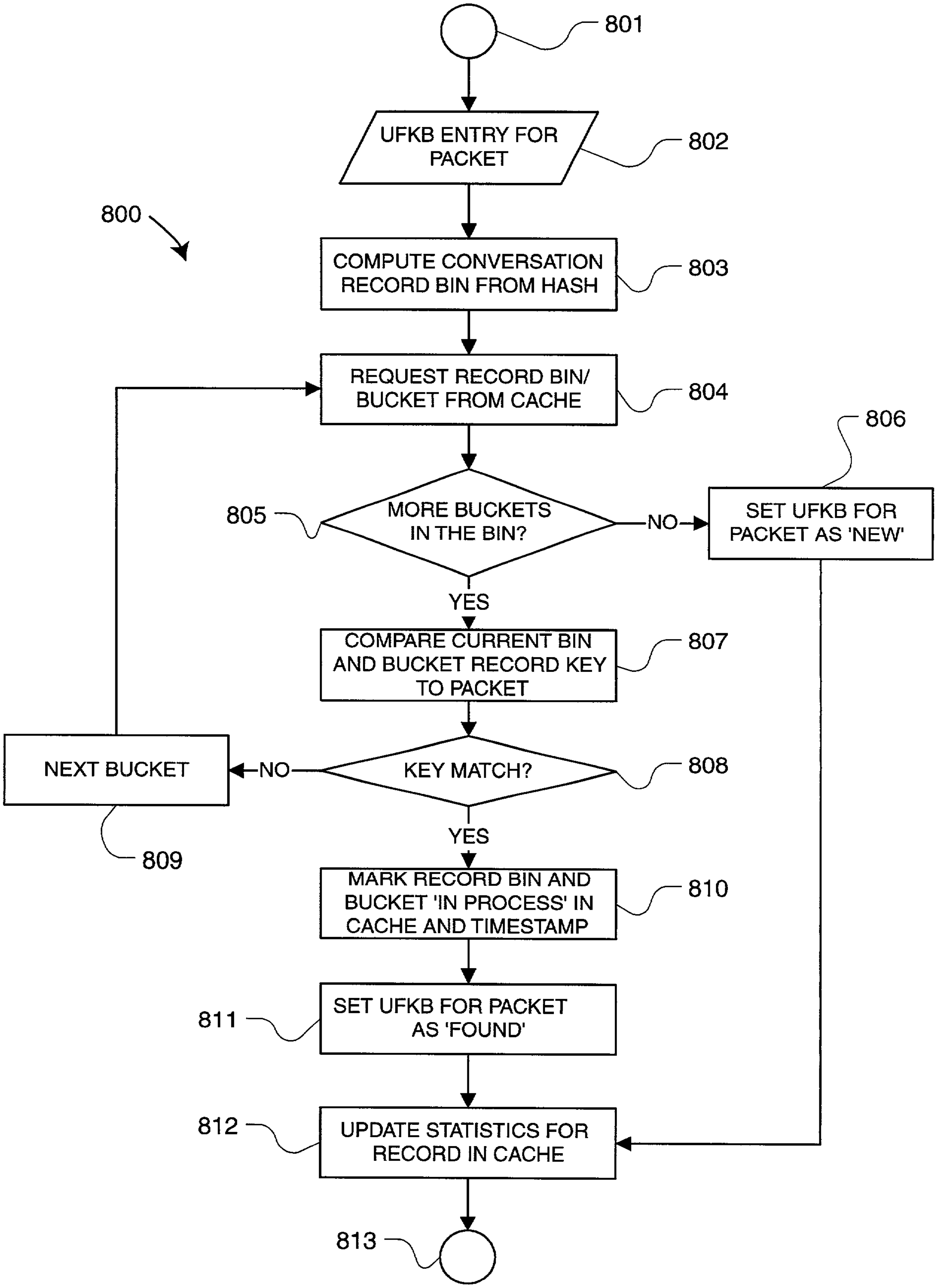

701
KEY BUFFER AND PATTERN NODES
702
LOAD PATTERN NODE ELEMENT
703
708
MORE PATTERN NODES?
704
NO
OUTPUT TO ANALYZER
YES
F 8
HASH KEY BUFFER ELEMENT FROM PATTERN NODE
705
709
PACK KEY AND HASH
706
700
NEXT PACKET COMPONENT
707

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

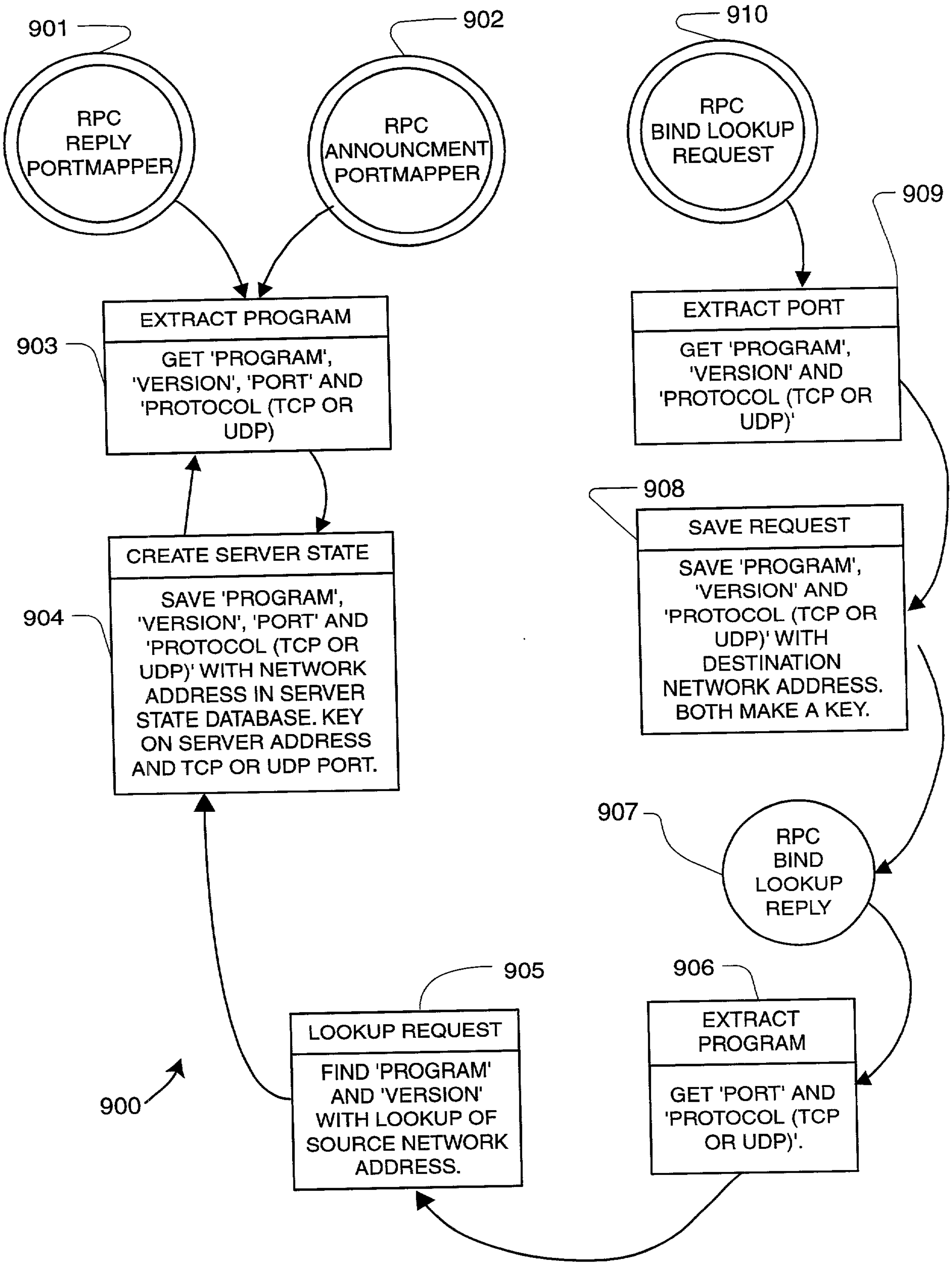

901
902
910
RPC REPLY PORTMAPPER
RPC ANNOUNCMENT PORTMAPPER
RPC BIND LOOKUP REQUEST
909
EXTRACT PROGRAM
903
GET 'PROGRAM', 'VERSION', 'PORT' AND 'PROTOCOL (TCP OR UDP)
EXTRACT PORT
GET 'PROGRAM', 'VERSION' AND 'PROTOCOL (TCP OR UDP)'
908
SAVE REQUEST
SAVE 'PROGRAM', 'VERSION' AND 'PROTOCOL (TCP OR UDP)'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

Fig. 9

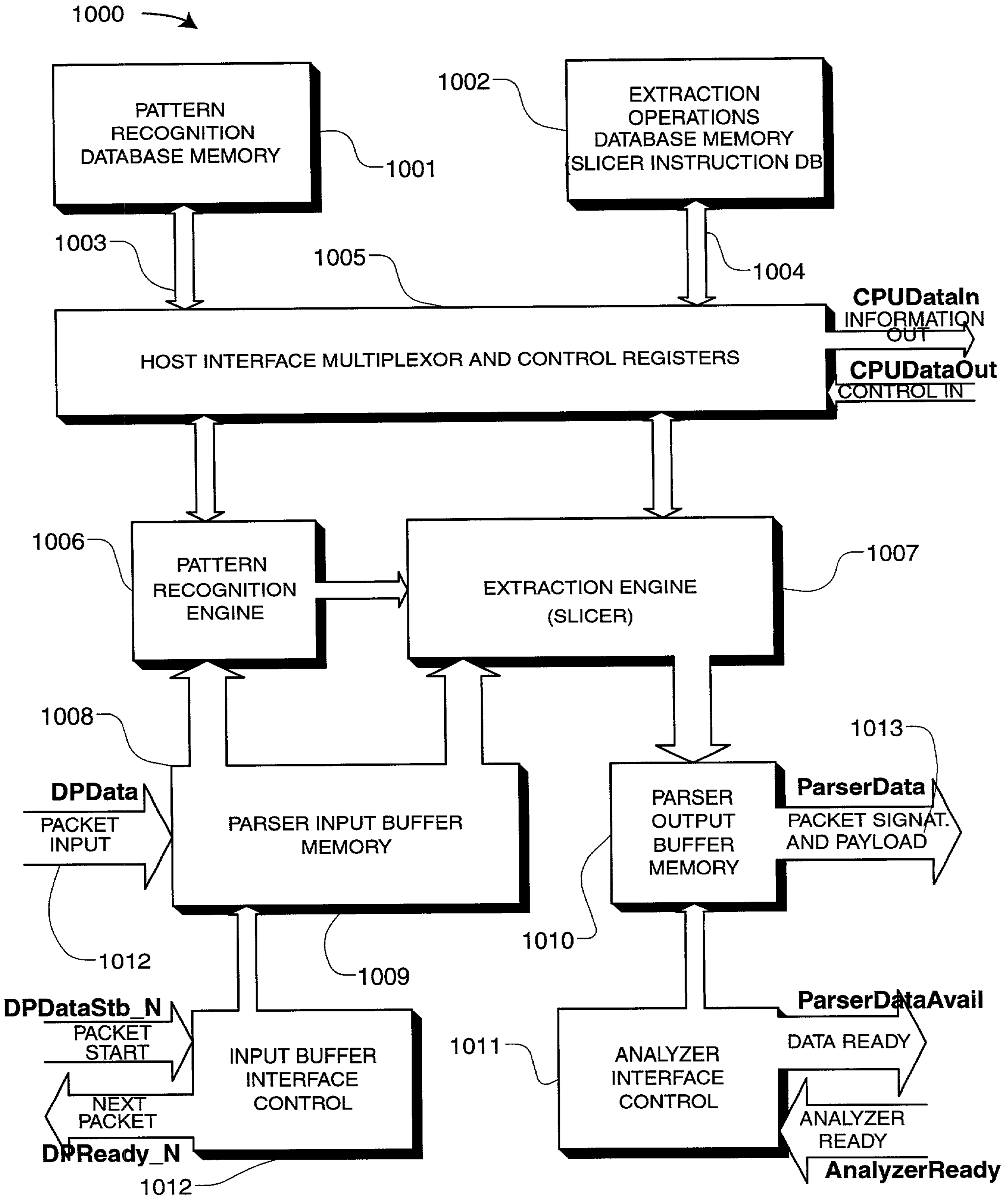
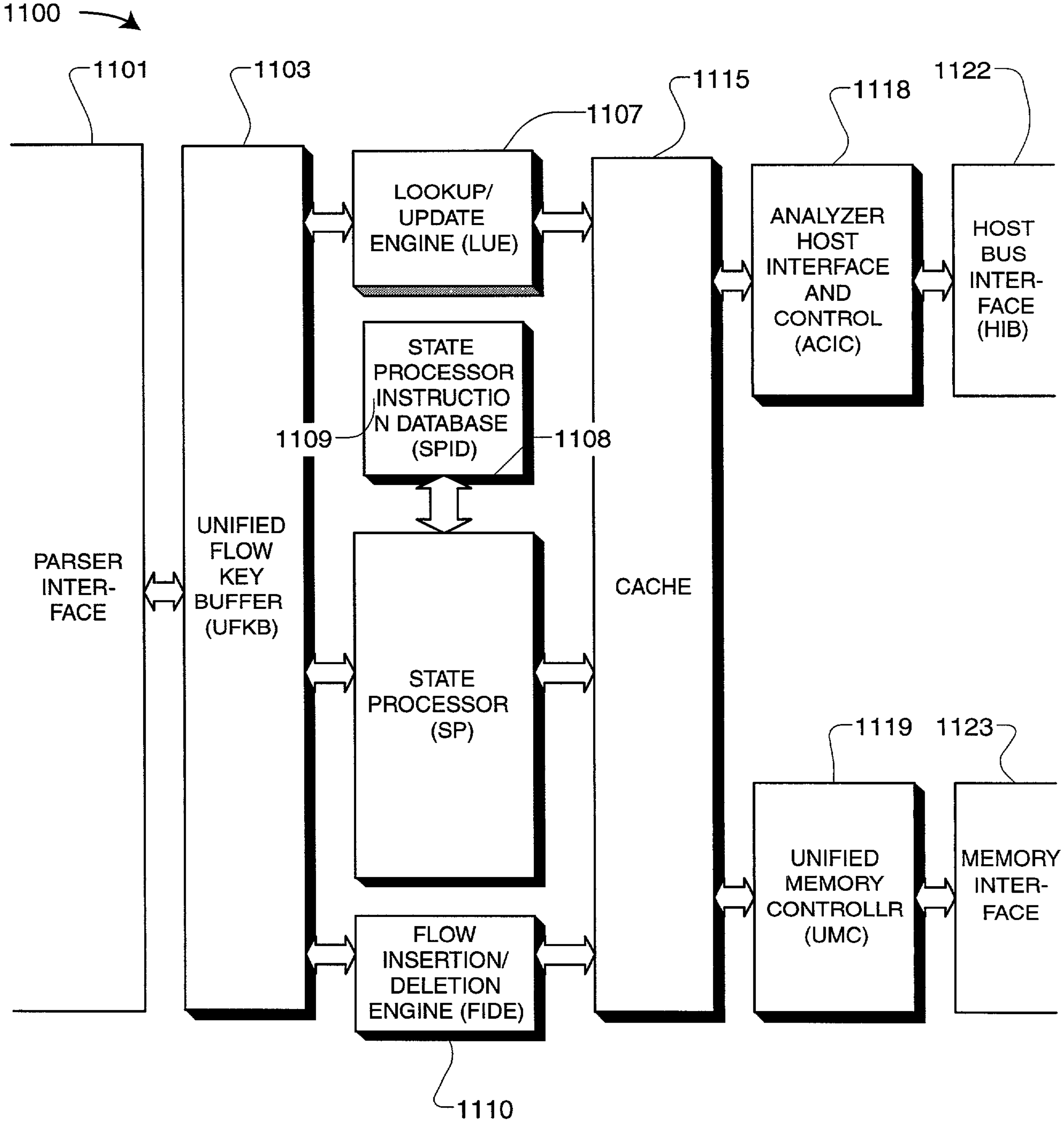

1000
PATTERN RECOGNITION DATABASE MEMORY
1001
1002
EXTRACTION OPERATIONS DATABASE MEMORY (SLICER INSTRUCTION DB
1003
1005
1004
CPUDataIn
INFORMATION OUT
HOST INTERFACE MULTIPLEXOR AND CONTROL REGISTERS
CPUDataOut
CONTROL IN
1006
PATTERN RECOGNITION ENGINE
EXTRACTION ENGINE (SLICER)
1007
1008
1013
DPData
PACKET INPUT
PARSER INPUT BUFFER MEMORY
PARSER OUTPUT BUFFER MEMORY
ParserData
PACKET SIGNAT. AND PAYLOAD
1012
1009
1010
DPDataStb_N
PACKET START
NEXT PACKET
DPReady_N
INPUT BUFFER INTERFACE CONTROL
1011
ANALYZER INTERFACE CONTROL
ParserDataAvail
DATA READY
ANALYZER READY
AnalyzerReady
1012

Fig. 10



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

Fig. 11

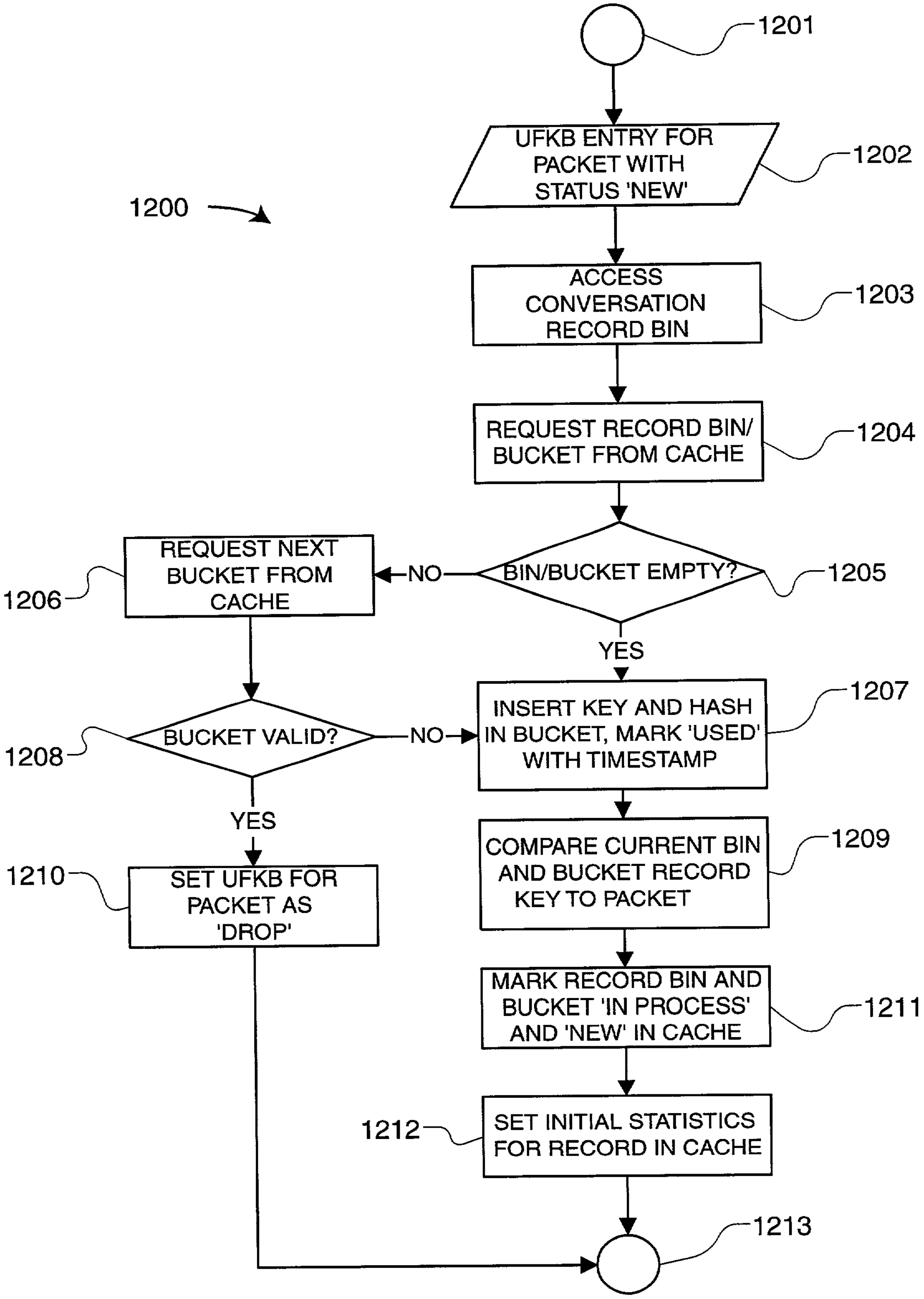

1201
UFKB ENTRY FOR PACKET WITH STATUS 'NEW'
1202
1200
ACCESS CONVERSATION RECORD BIN
1203
REQUEST RECORD BIN/ BUCKET FROM CACHE
1204
REQUEST NEXT BUCKET FROM CACHE
NO
BIN/BUCKET EMPTY?
1205
1206
YES
INSERT KEY AND HASH IN BUCKET, MARK 'USED' WITH TIMESTAMP
1207
1208
BUCKET VALID?
NO
YES
COMPARE CURRENT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

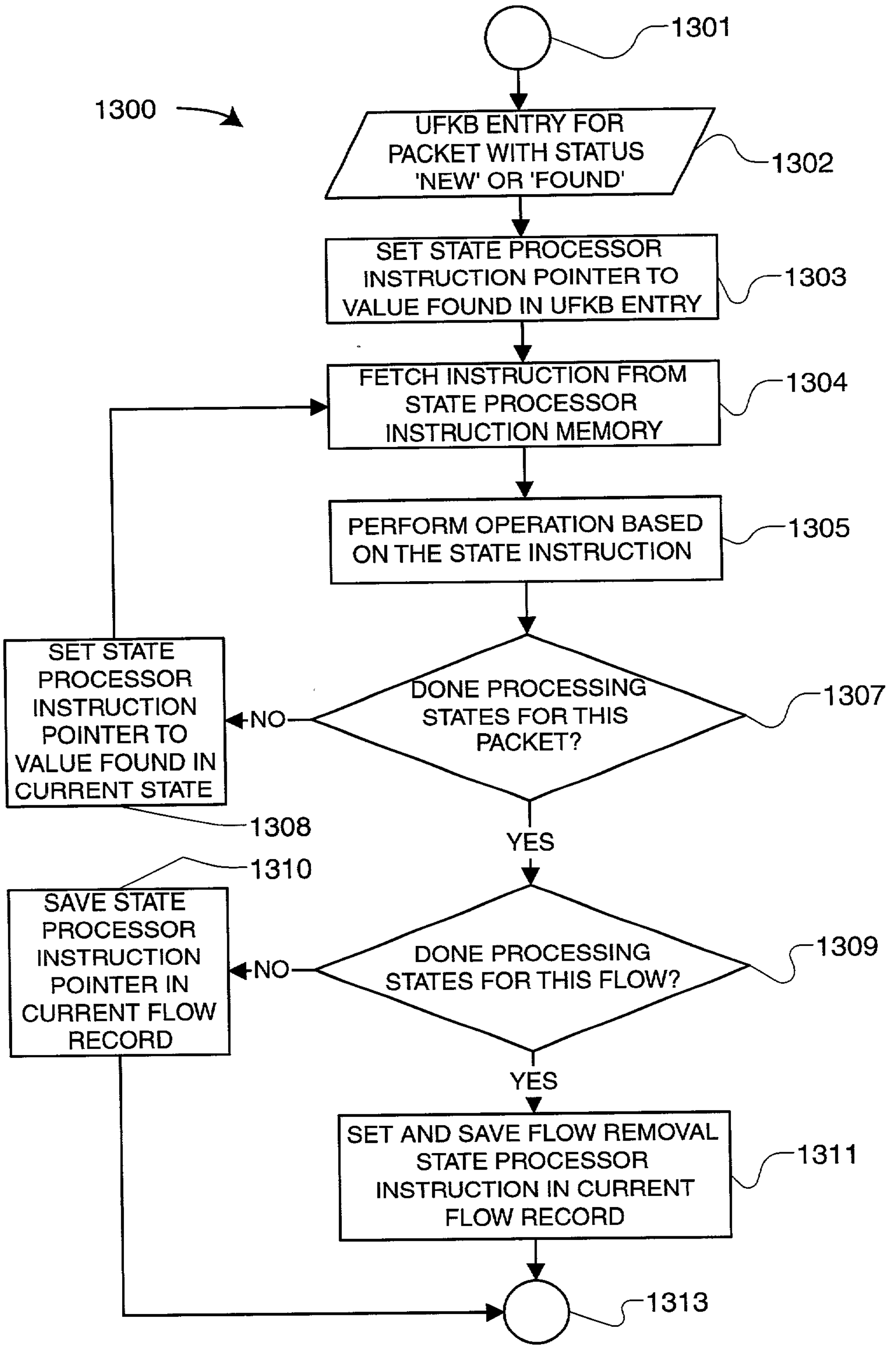

1300
1301
UFKB ENTRY FOR PACKET WITH STATUS 'NEW' OR 'FOUND'
1302
SET STATE PROCESSOR INSTRUCTION POINTER TO VALUE FOUND IN UFKB ENTRY
1303
FETCH INSTRUCTION FROM STATE PROCESSOR INSTRUCTION MEMORY
1304
PERFORM OPERATION BASED ON THE STATE INSTRUCTION
1305
DONE PROCESSING STATES FOR THIS PACKET?
1307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

Fig. 13

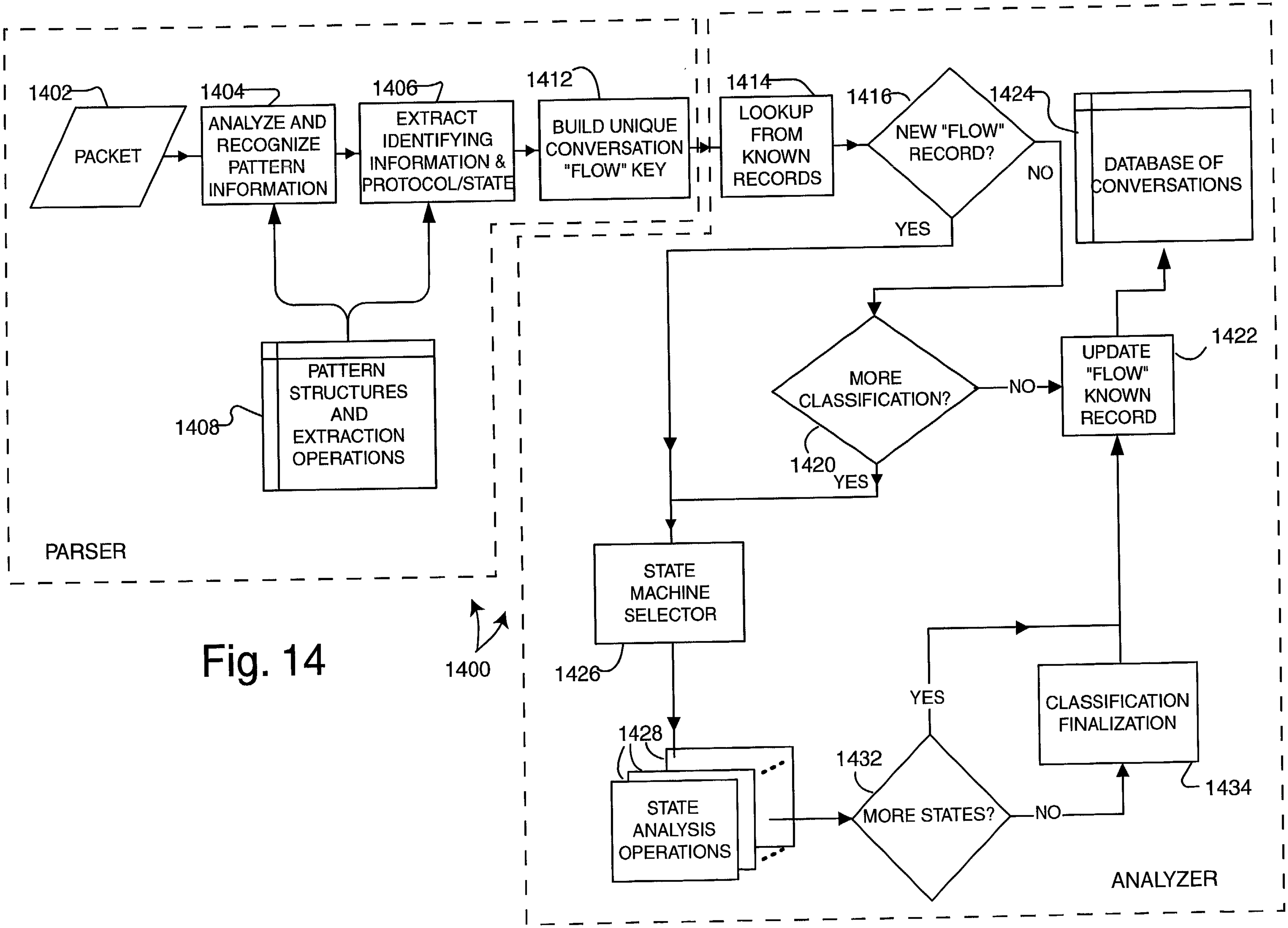

1402
PACKET
1404
ANALYZE AND RECOGNIZE PATTERN INFORMATION
1406
EXTRACT IDENTIFYING INFORMATION & PROTOCOL/STATE
1412
BUILD UNIQUE CONVERSATION "FLOW" KEY
1408
PATTERN STRUCTURES AND EXTRACTION OPERATIONS
PARSER
1414
LOOKUP FROM KNOWN RECORDS
1416
NEW "FLOW" RECORD?
NO
YES
1424
DATABASE OF CONVERSATIONS
MORE CLASSIFICATION?
1420
NO
YES
1422
UPDATE "FLOW" KNOWN RECORD
STATE MACHINE SELECTOR
1426
1428
STATE ANALYSIS OPERATIONS
1432
MORE STATES?
YES
NO
CLASSIFICATION FINALIZATION
1434
ANALYZER
1400

Fig. 14

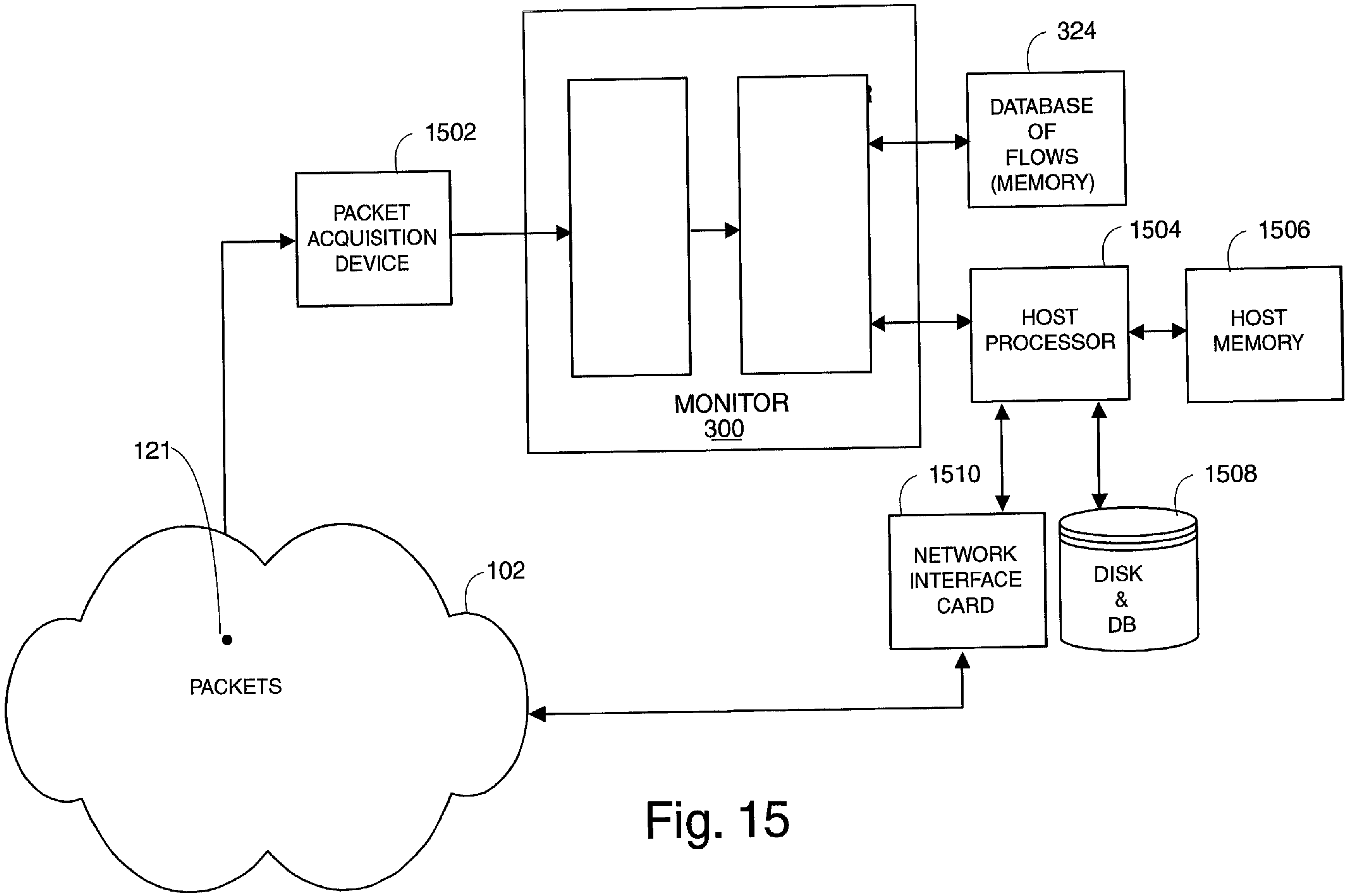

324
DATABASE OF FLOWS (MEMORY)
1502
PACKET ACQUISITION DEVICE
1504
HOST PROCESSOR
1506
HOST MEMORY
MONITOR
300
121
1510
1508
NETWORK INTERFACE CARD
DISK & DB
102
PACKETS

Fig. 15

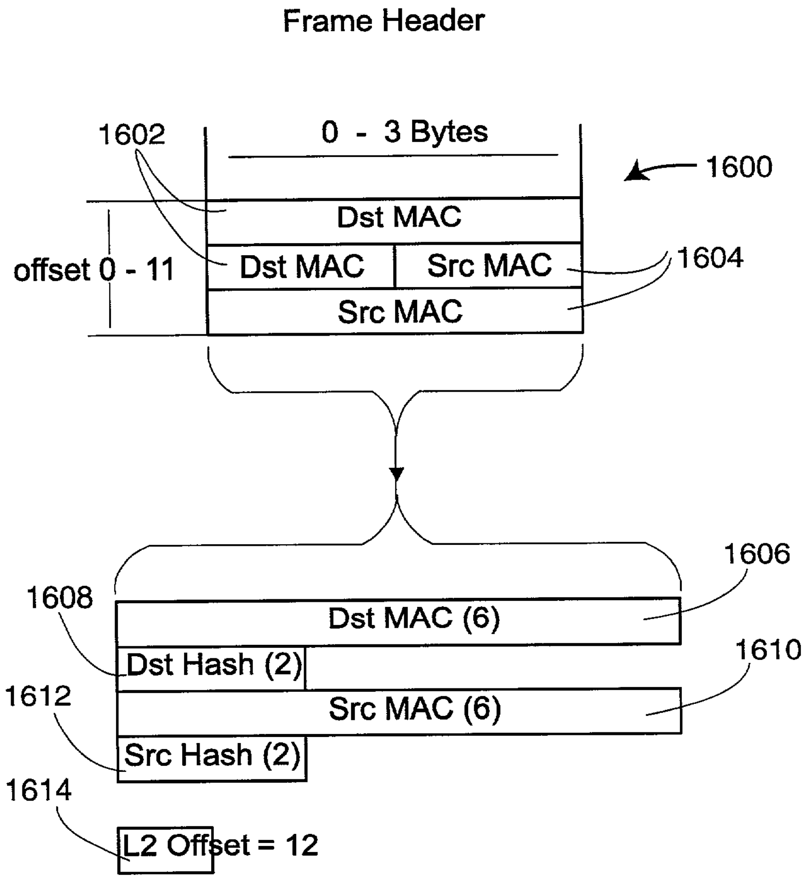
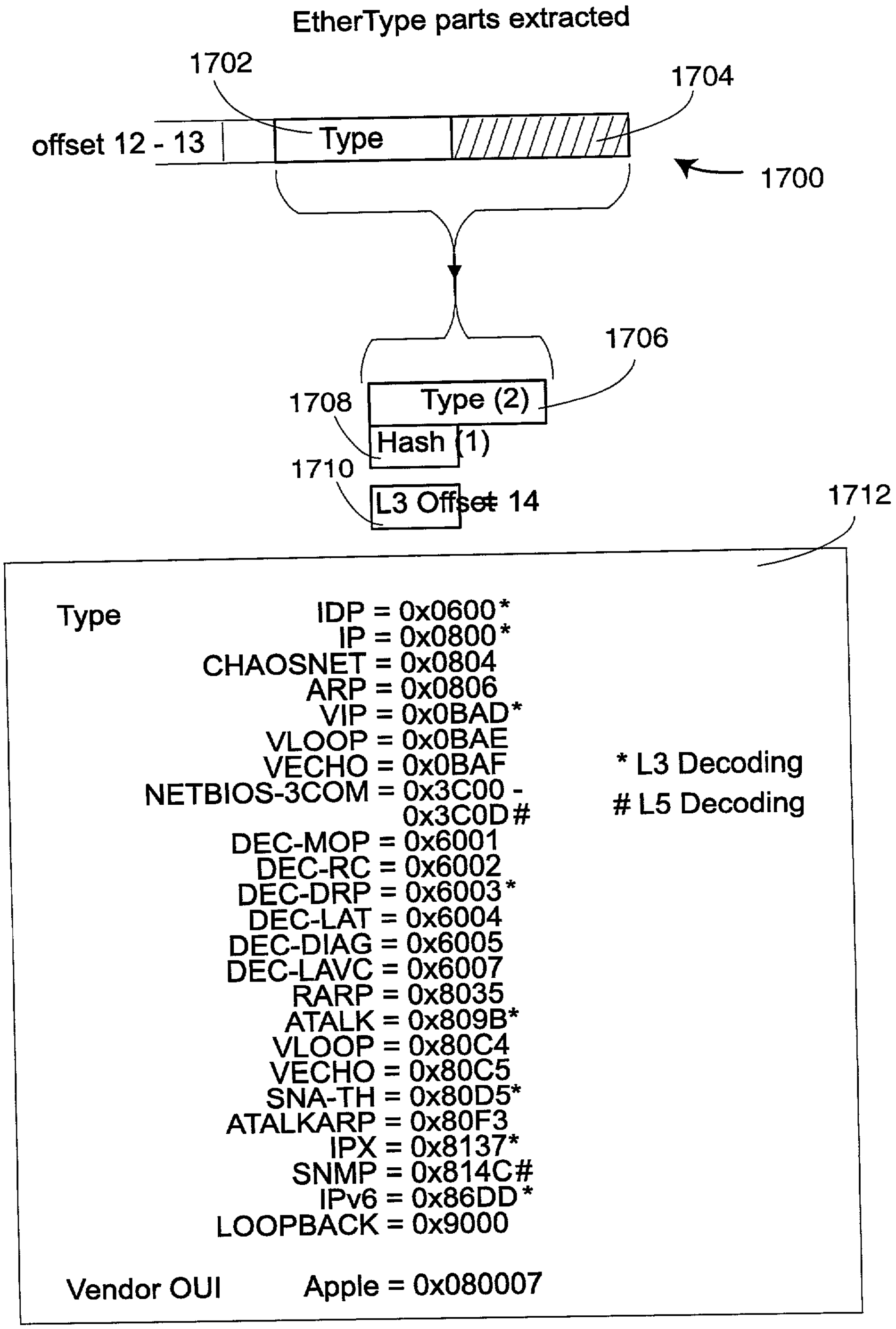

Frame Header
0 - 3 Bytes
1602
1600
Dst MAC
offset 0 - 11
Dst MAC
Src MAC
1604
Src MAC
1606
1608
Dst MAC (6)
Dst Hash (2)
1610
1612
Src MAC (6)
Src Hash (2)
1614
L2 Offset = 12

Fig. 16



Type	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
Vendor OUI	Apple = 0x080007

Fig. 17

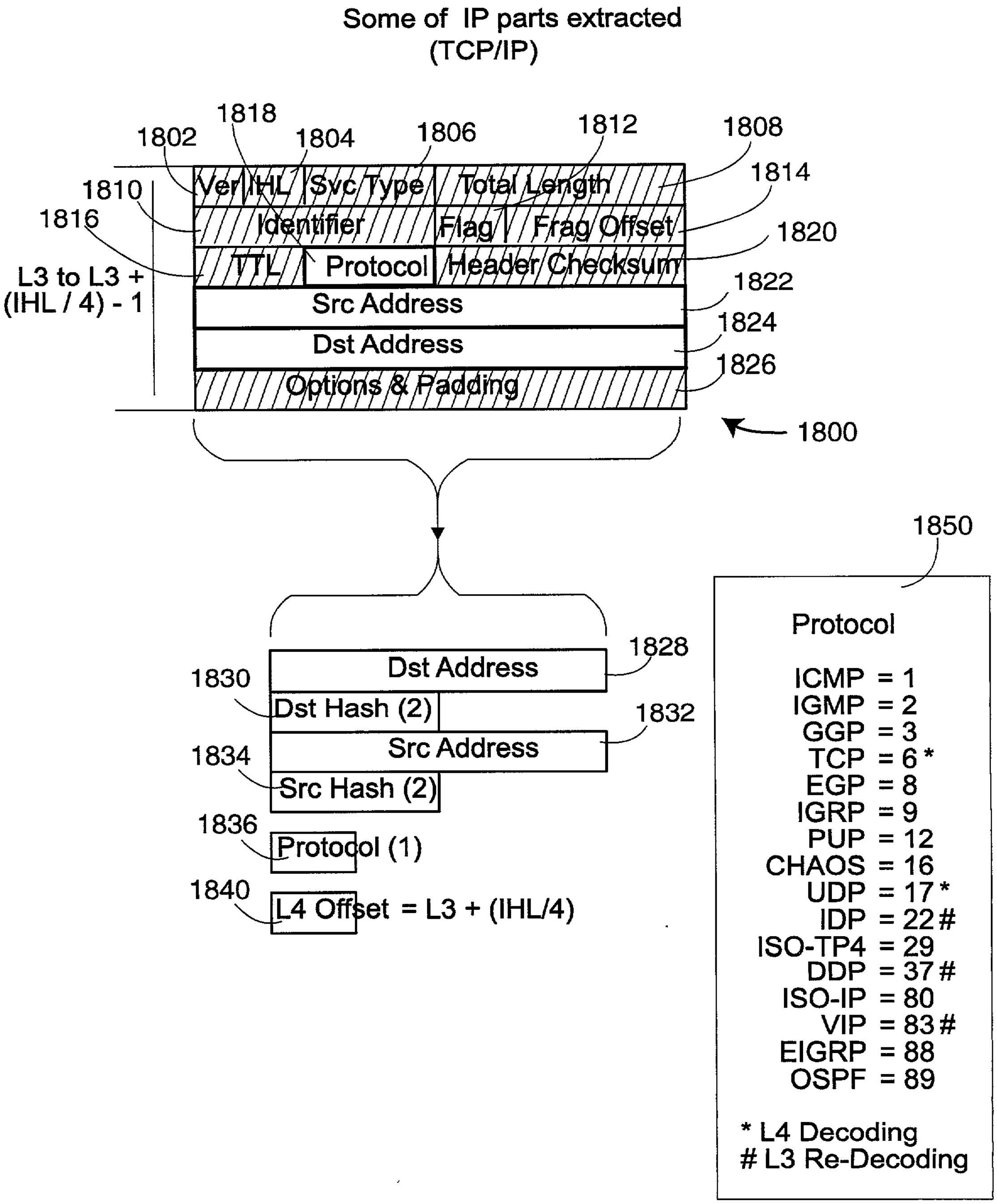

Some of IP parts extracted
(TCP/IP)
1818
1802
1804
1806
1812
1808
1810
1814
1816
1820
Ver
IHL
Svc Type
Total Length
Identifier
Flag
Frag Offset
TTL
Protocol
Header Checksum
L3 to L3 +
(IHL / 4) - 1
Src Address
1822
Dst Address
1824
Options & Padding
1826
1800
1850
Dst Address
1828
1830
Dst Hash (2)
1832
Src Address
1834
Src Hash (2)
1836
Protocol (1)
1840
L4 Offset = L3 + (IHL/4)
Protocol
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. 18

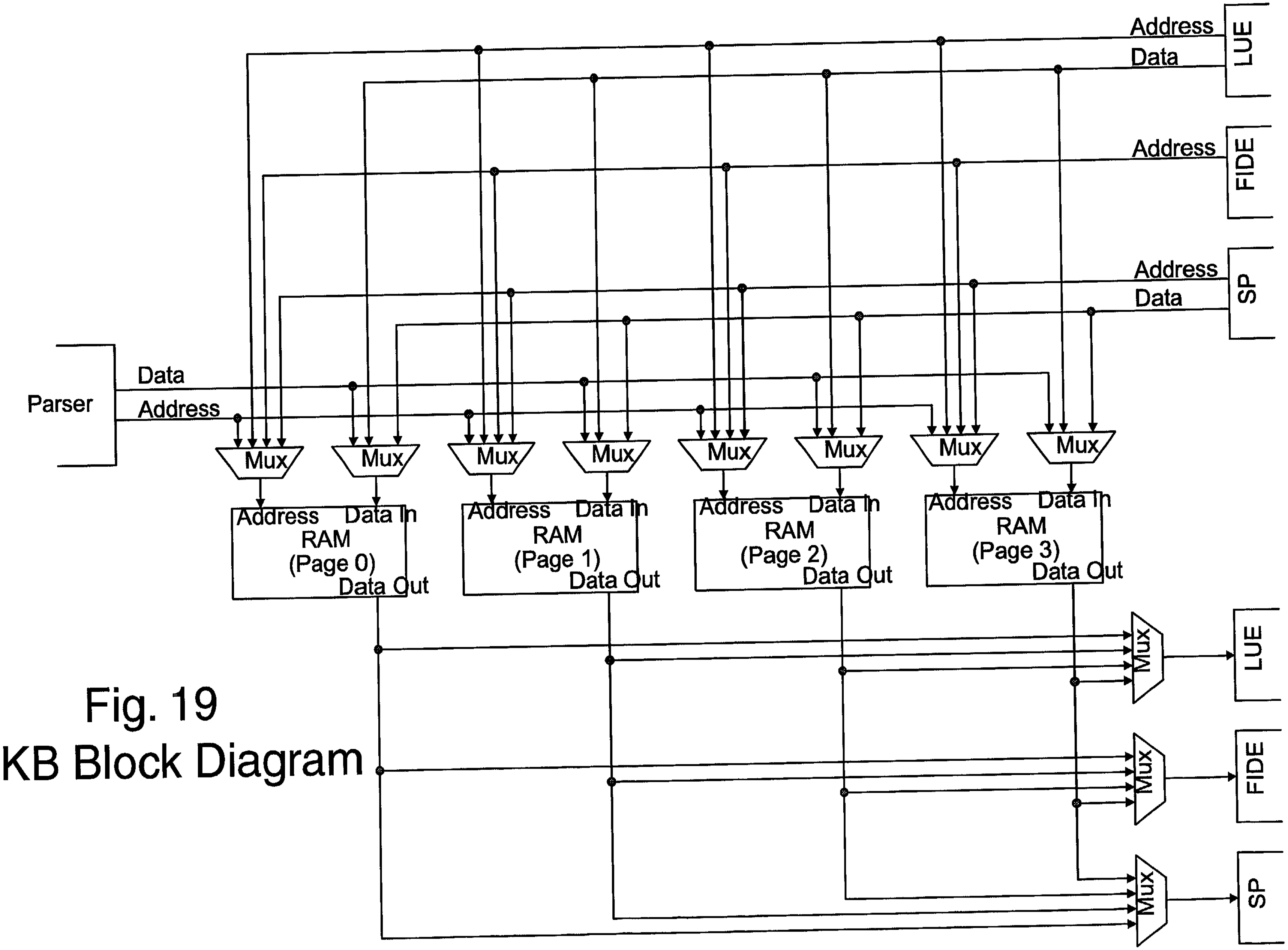


Fig. 19
UFKB Block Diagram

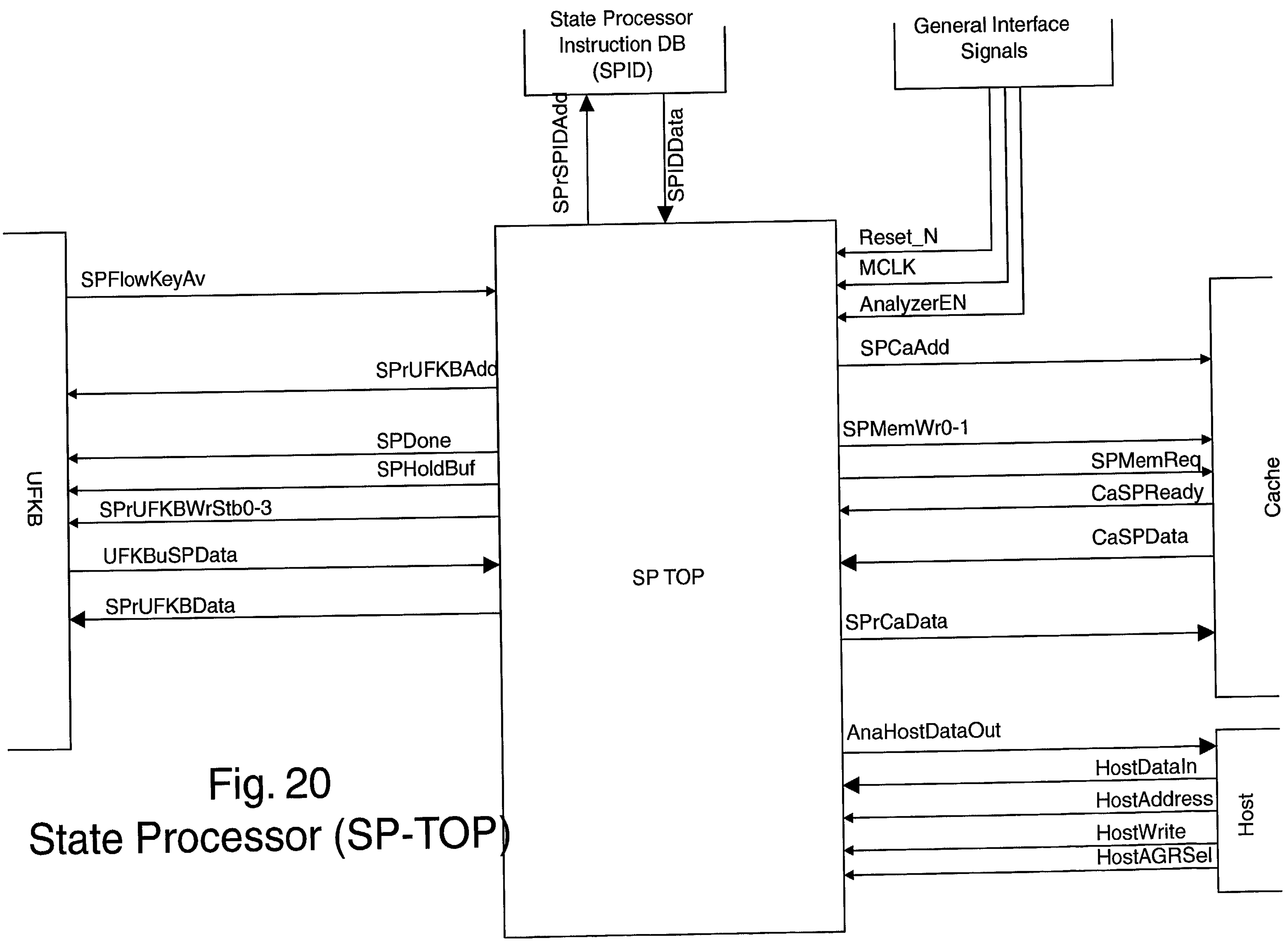
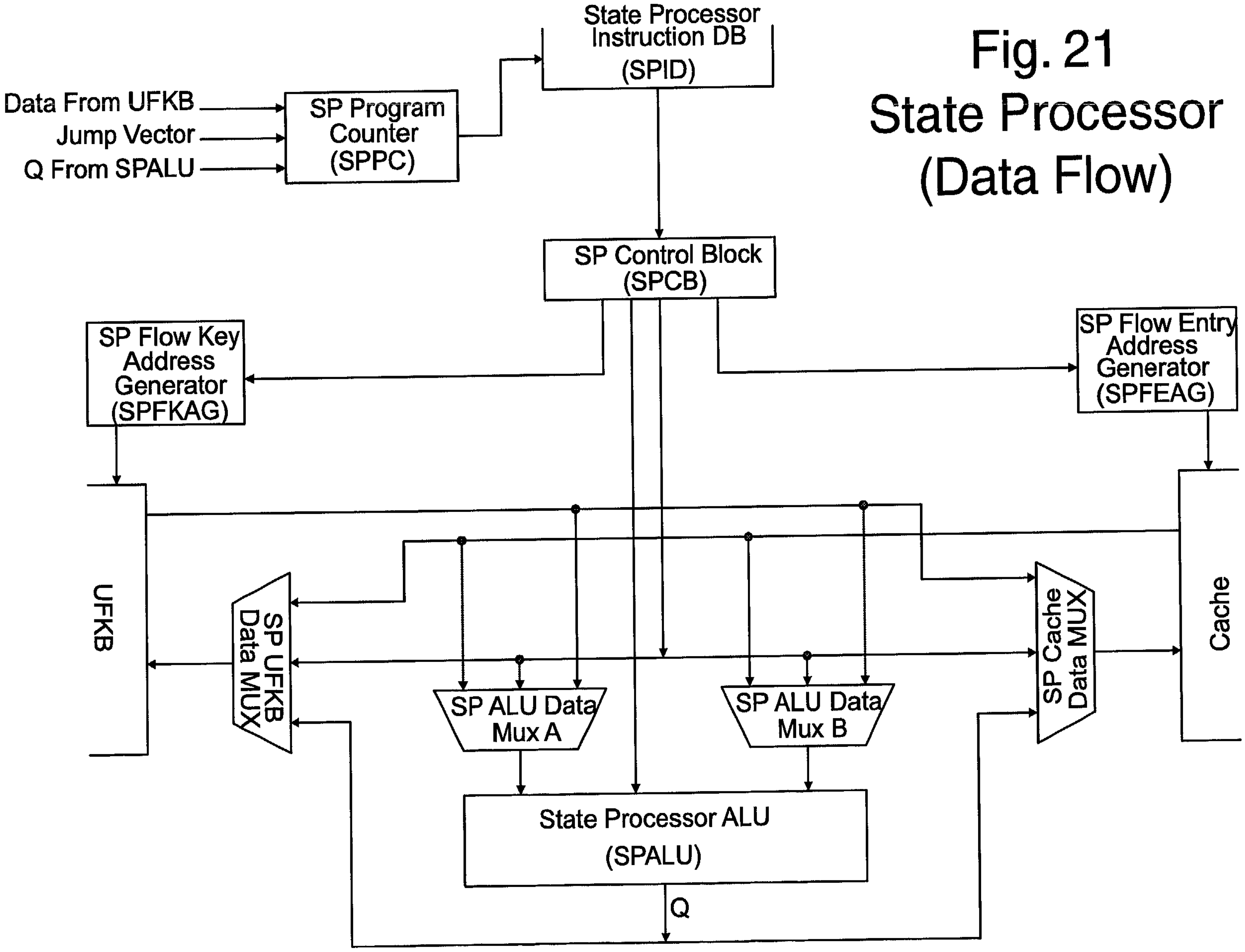


Fig. 20
State Processor (SP-TOP)

Fig. 21
State Processor
(Data Flow)



Data From UFKB
Jump Vector
Q From SPALU
SP Program Counter (SPPC)
State Processor Instruction DB (SPID)
SP Control Block (SPCB)
SP Flow Key Address Generator (SPFKAG)
SP Flow Entry Address Generator (SPFEAG)
UFKB
SP UFKB Data MUX
SP ALU Data Mux A
SP ALU Data Mux B
State Processor ALU (SPALU)
Q
SP Cache Data MUX
Cache

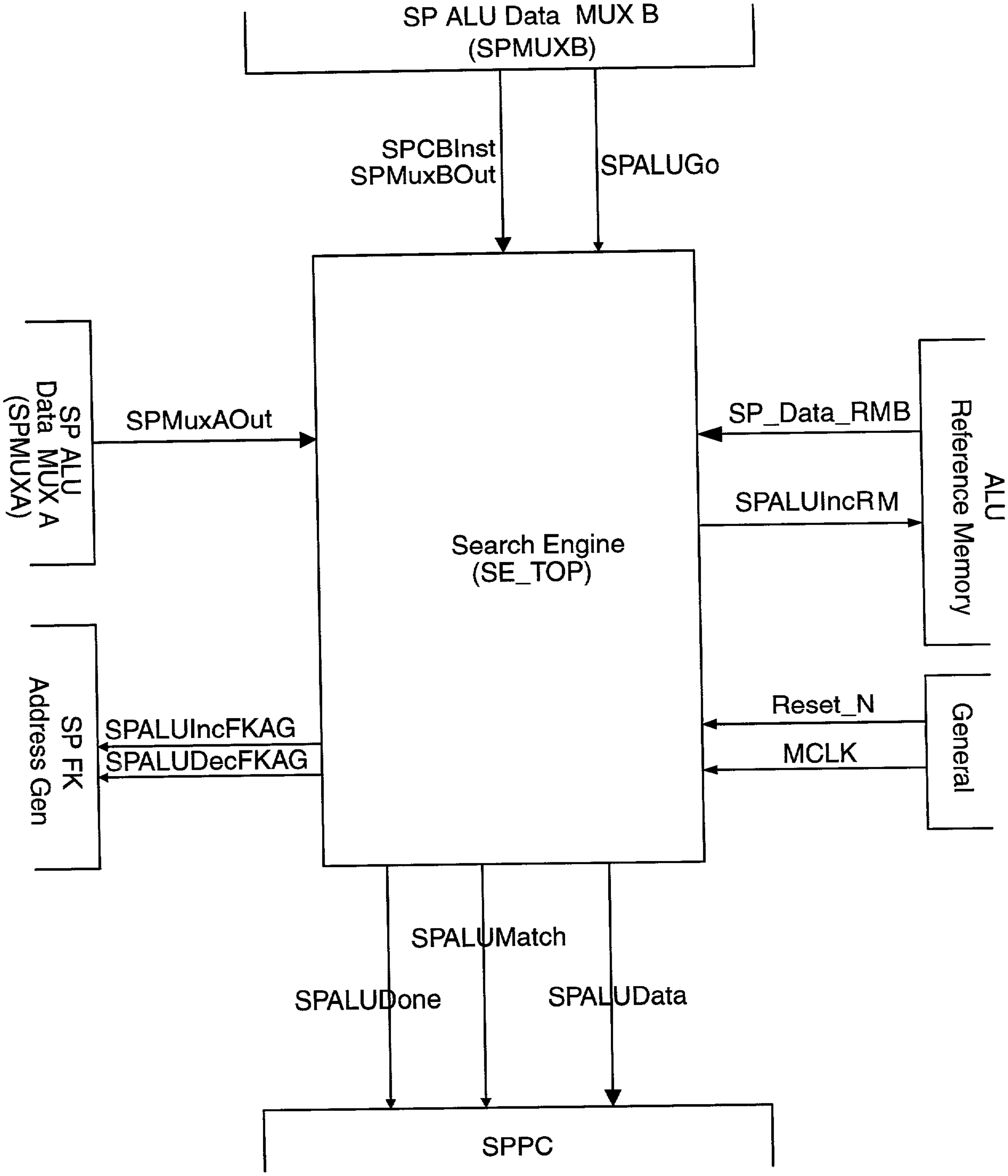
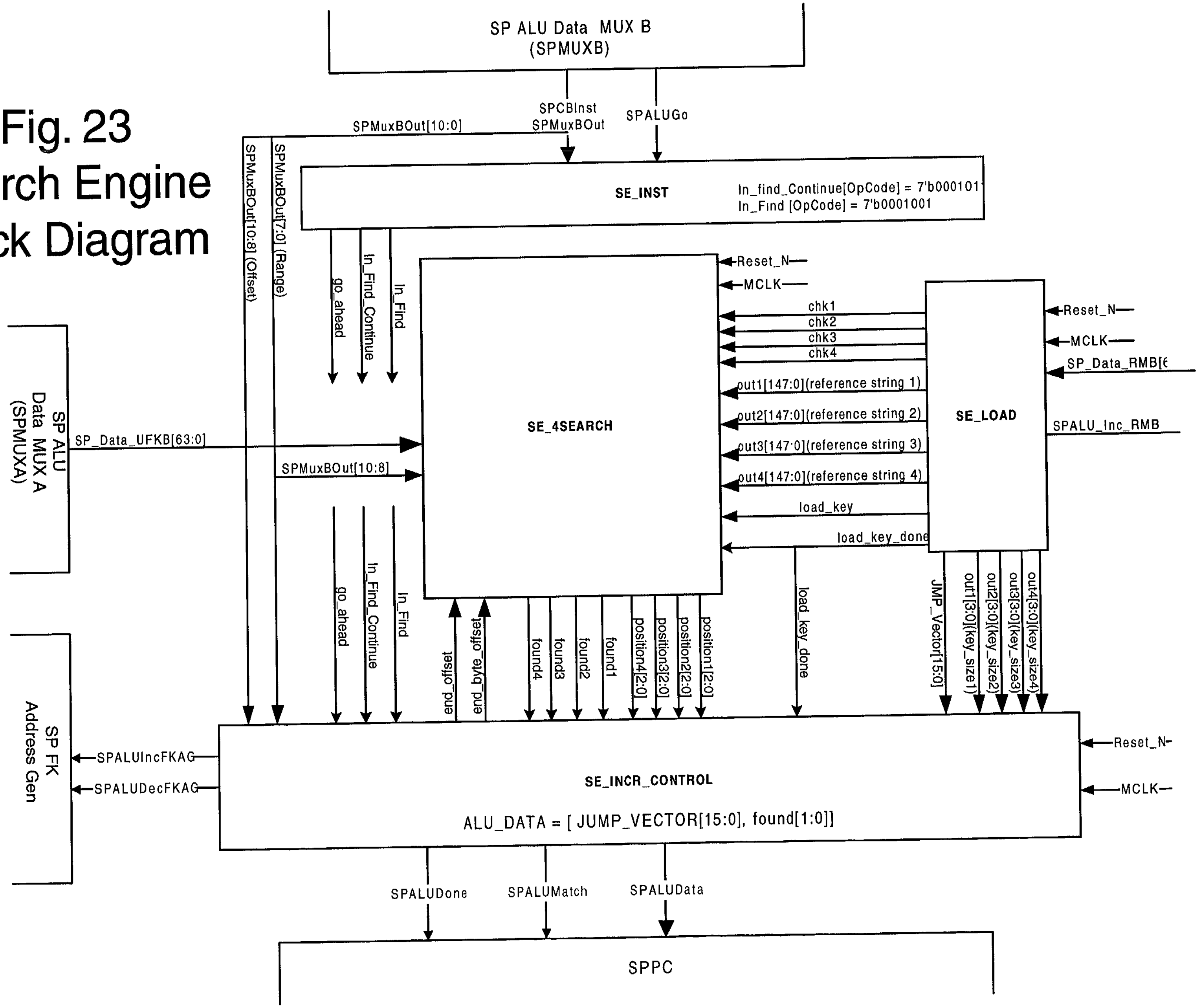


Fig. 22

State Processor ALU
Search Engine TOP Level Diagram
(SPALU_SE)

Fig. 23
Search Engine Block Diagram



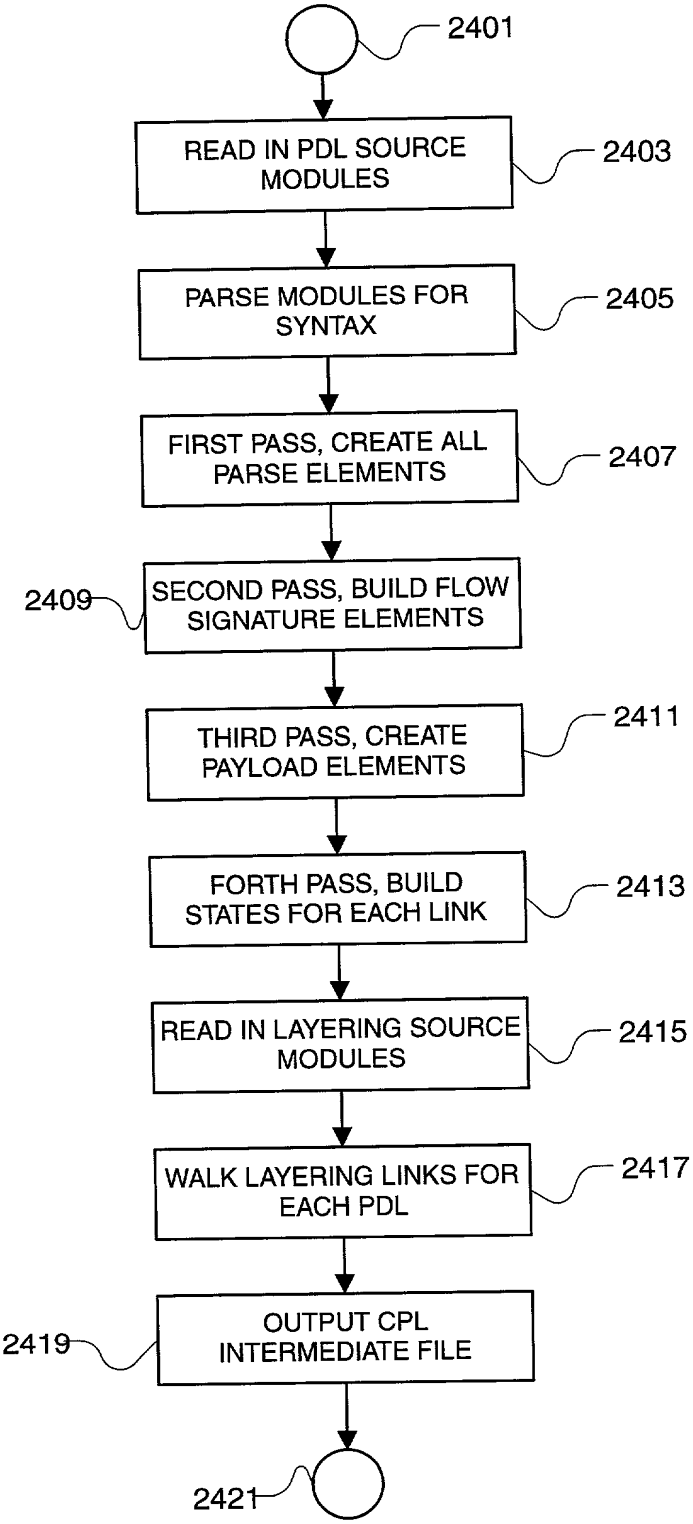

2401
READ IN PDL SOURCE MODULES
2403
PARSE MODULES FOR SYNTAX
2405
FIRST PASS, CREATE ALL PARSE ELEMENTS
2407
2409
SECOND PASS, BUILD FLOW SIGNATURE ELEMENTS
THIRD PASS, CREATE PAYLOAD ELEMENTS
2411
FORTH PASS, BUILD STATES FOR EACH LINK
2413
READ IN LAYERING SOURCE MODULES
2415
WALK LAYERING LINKS FOR EACH PDL
2417
2419
OUTPUT CPL INTERMEDIATE FILE
2421

Fig. 24

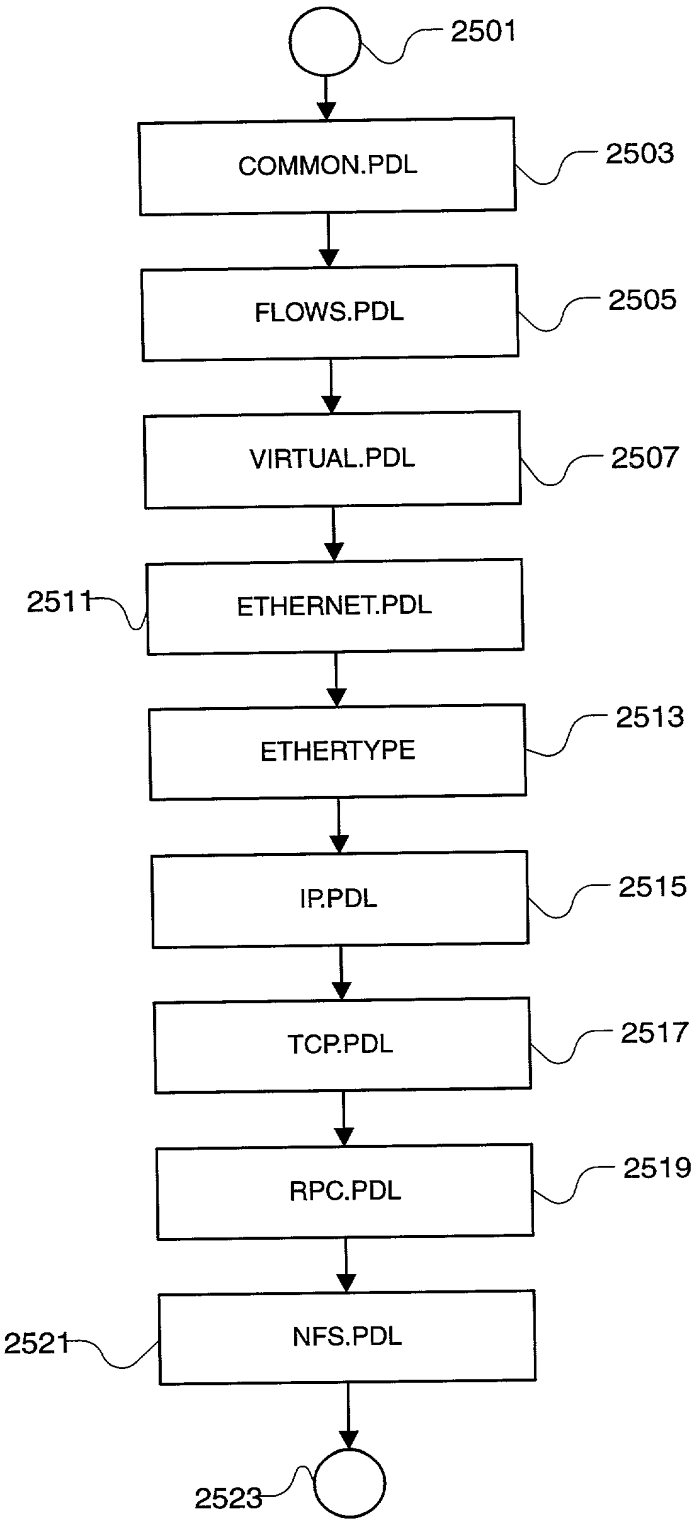

2501
COMMON.PDL
2503
FLOWS.PDL
2505
VIRTUAL.PDL
2507
2511
ETHERNET.PDL
ETHERTYPE
2513
IP.PDL
2515
TCP.PDL
2517
RPC.PDL
2519
2521
NFS.PDL
2523

Fig. 25