IW 7696177

THE UNITED STATES OF AMERICA

TO ALL TO WHOM THESE PRESENTS SHALL COME:

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office

October 17, 2018

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

APPLICATION NUMBER: *09/608,126*
FILING DATE: *June 30, 2000*
PATENT NUMBER: *6,839,751*
ISSUE DATE: *January 04, 2005*

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

G. W. MONTGOMERY
Certifying Officer

PART (3) OF (3) PART(S)

US005507028A

United States Patent [19]

Liu

[11] Patent Number: 5,507,028

[45] Date of Patent: Apr. 9, 1996

[54] **HISTORY BASED BRANCH PREDICTION ACCESSED VIA A HISTORY BASED EARLIER INSTRUCTION ADDRESS**

[75] Inventor: **Lishing Liu**, Pleasantville, N.Y.

[73] Assignee: **International Business Machines Corporation**, Armonk, N.Y.

[21] Appl. No.: **370,342**

[22] Filed: **Jan. 9, 1995**

Related U.S. Application Data

[63] Continuation of Ser. No. 860,631, Mar. 30, 1992, abandoned.

[51] **Int. Cl.**[6] **G06F 9/38**

[52] **U.S. Cl.** **395/375**; 364/261.3; 364/261.7; 364/262.8; 364/DIG. 1

[58] **Field of Search** 395/375

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,763,245	8/1988	Emma et al.	395/375
4,831,517	5/1989	Crouse et al.	395/375
4,984,154	1/1991	Hanatani et al.	395/375
5,168,557	12/1992	Shibuya	395/375
5,175,827	12/1992	Morisada	395/375
5,237,666	8/1993	Suzuki et al.	395/375

Primary Examiner—Kevin J. Teska
Assistant Examiner—Ayni Mohamed
Attorney, Agent, or Firm—Ronald L. Drumheller

[57] **ABSTRACT**

An improved history table is disclosed in which at least some of the entries are stored and accessed based upon the address of an instruction which historically preceeds the branch instruction itself. The access address may be used to determine the location of the entry in the table and/or may be stored in whole or in part in the entry itself. Furthermore, the improved history table may be of any known type including but not limited to branch history table types and decode history table types. The entries in the improved history table preferably are stored and accessed by the address of the preceeding taken branch target and preferably contain a number indicative of the number of instructions between the access address and the address of the branch instruction or its target.

8 Claims, 9 Drawing Sheets

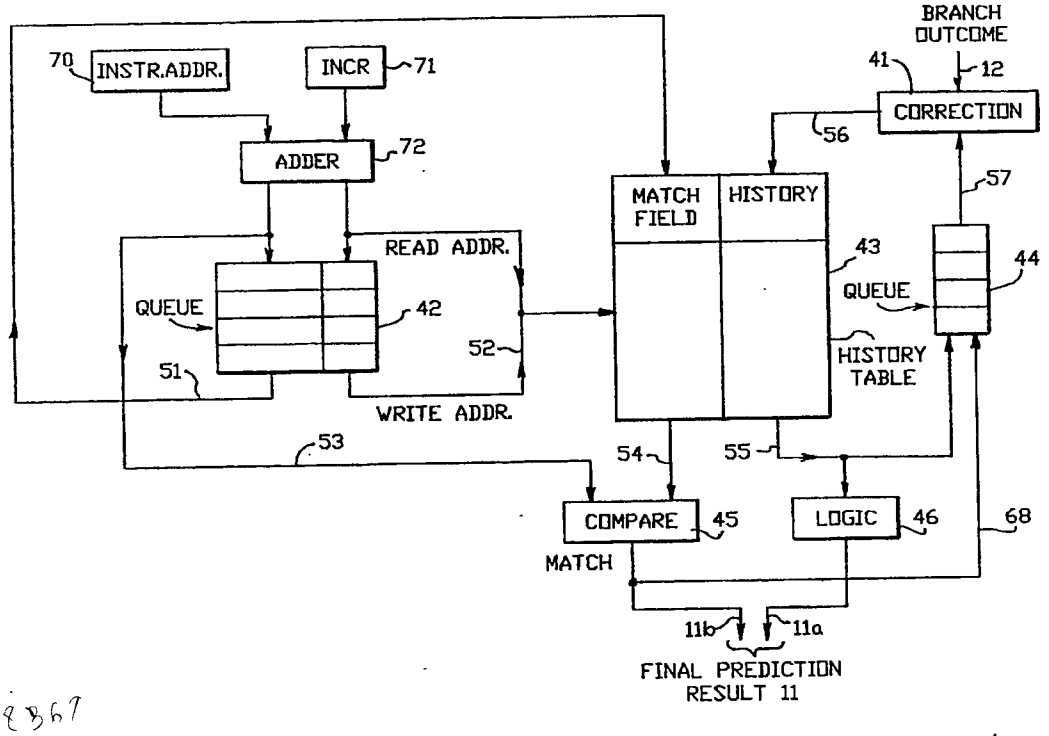


FIG. 2

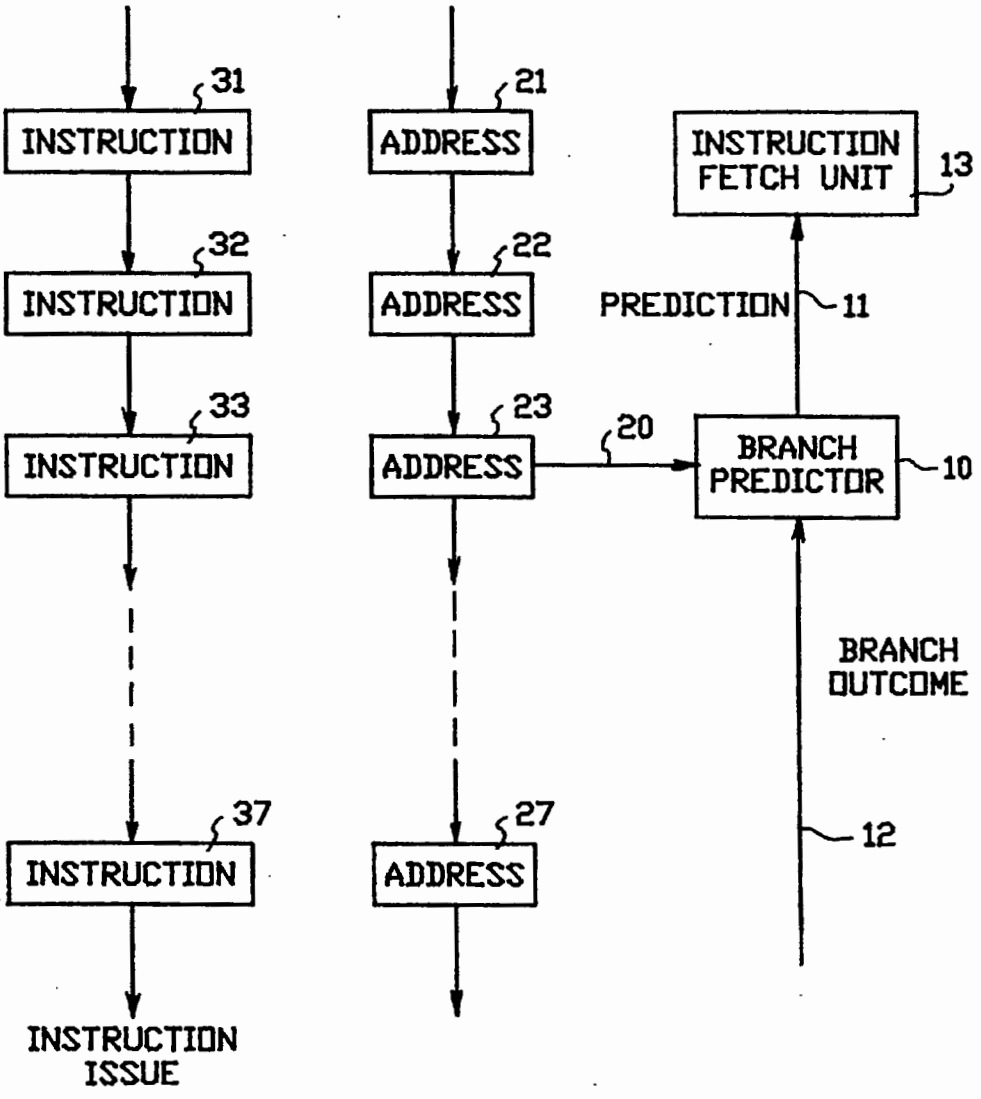
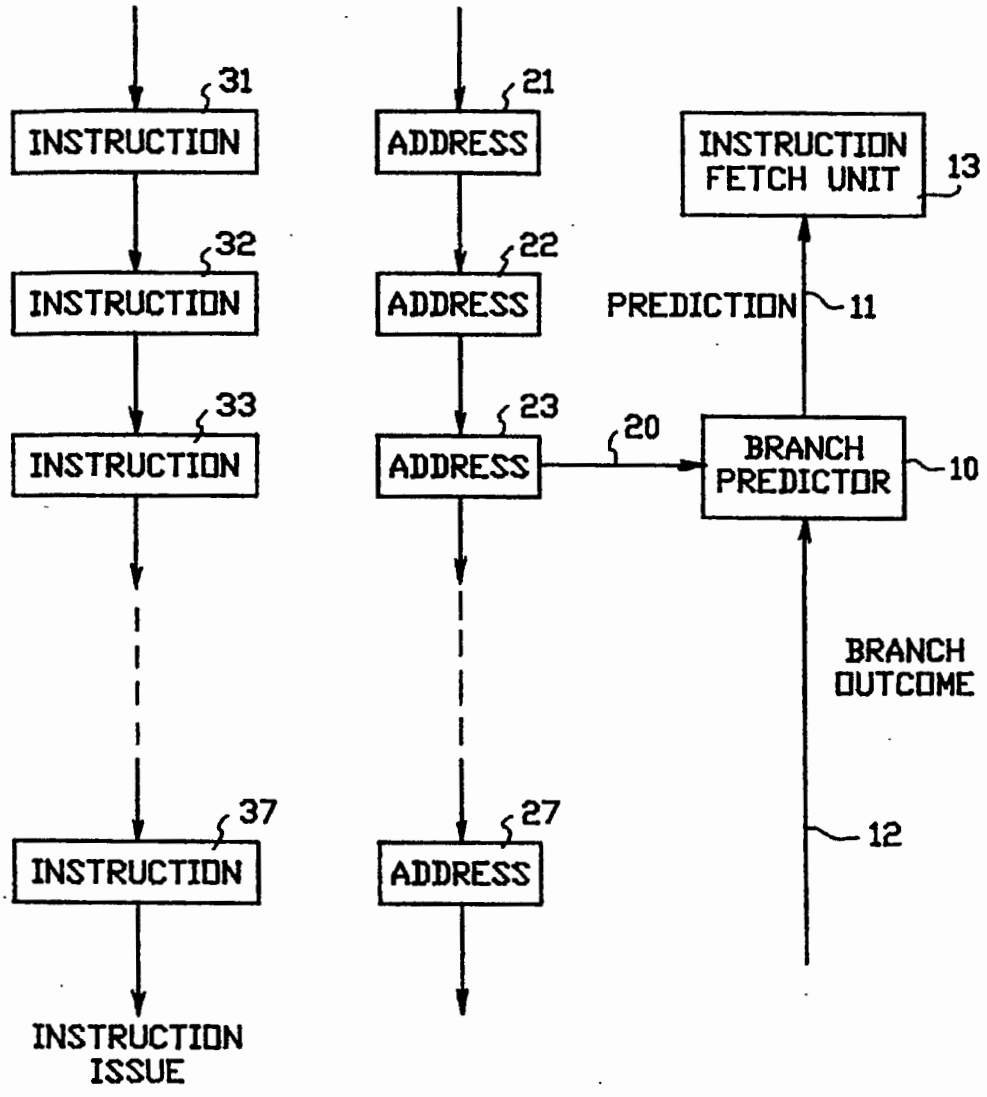

31
INSTRUCTION
32
INSTRUCTION
33
INSTRUCTION
37
INSTRUCTION
INSTRUCTION ISSUE
21
ADDRESS
22
ADDRESS
23
ADDRESS
27
ADDRESS
20
INSTRUCTION FETCH UNIT
13
PREDICTION
11
BRANCH PREDICTOR
10
BRANCH OUTCOME
12

FIG. 2


31
INSTRUCTION
32
INSTRUCTION
33
INSTRUCTION
37
INSTRUCTION
INSTRUCTION ISSUE
21
ADDRESS
22
ADDRESS
23
ADDRESS
27
ADDRESS
20
INSTRUCTION FETCH UNIT
13
PREDICTION
11
BRANCH PREDICTOR
10
BRANCH OUTCOME
12

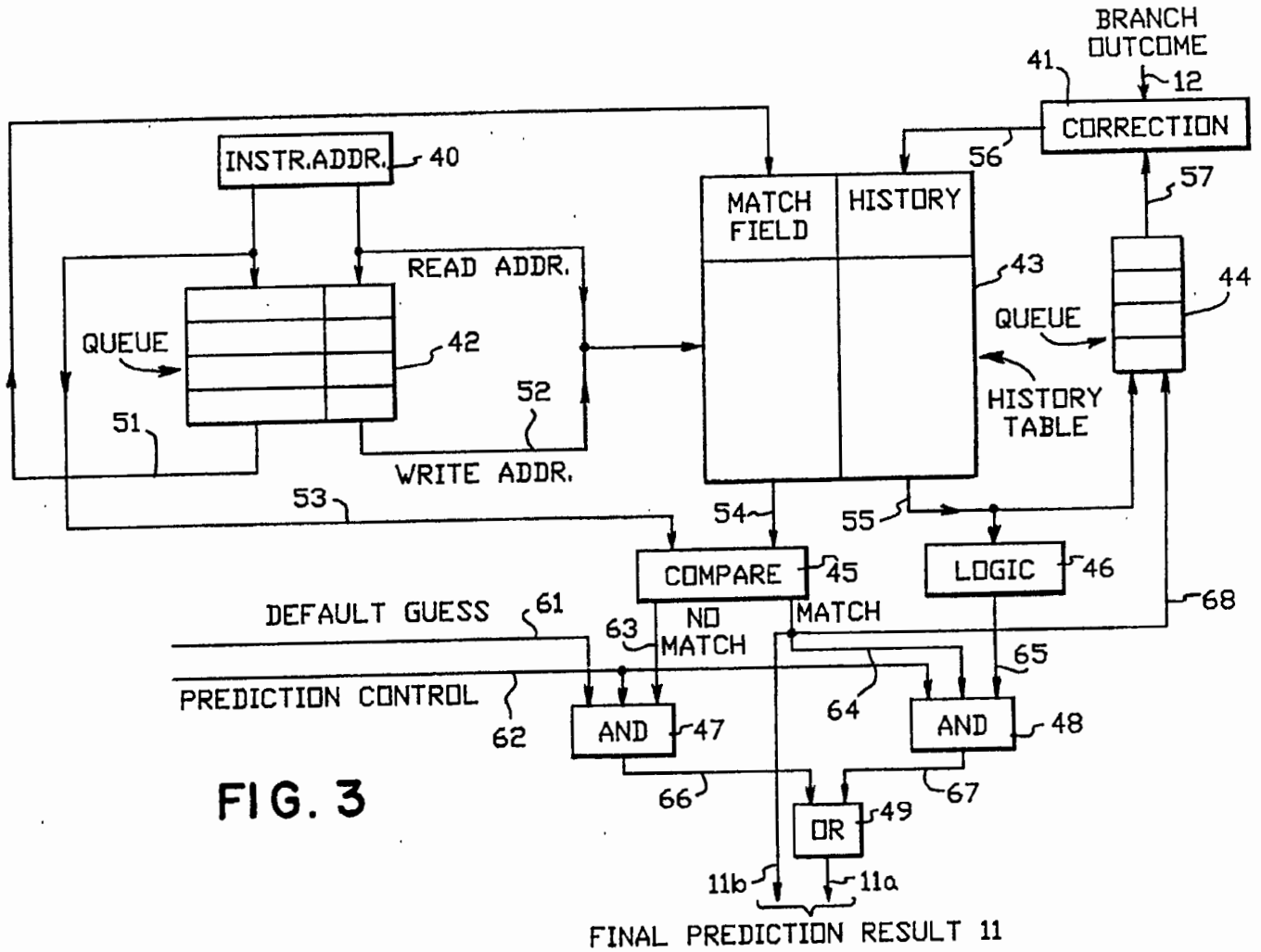

BRANCH OUTCOME
41
12
CORRECTION
56
57
INSTR.ADDR.
40
MATCH FIELD
HISTORY
43
44
READ ADDR.
QUEUE
42
QUEUE
52
HISTORY TABLE
51
WRITE ADDR.
53
54
55
COMPARE
45
LOGIC
46
DEFAULT GUESS
61
NO MATCH
63
MATCH
68
65
PREDICTION CONTROL
62
64
AND
47
AND
48
66
67
OR
49
11b
11a
FINAL PREDICTION RESULT 11

FIG. 3

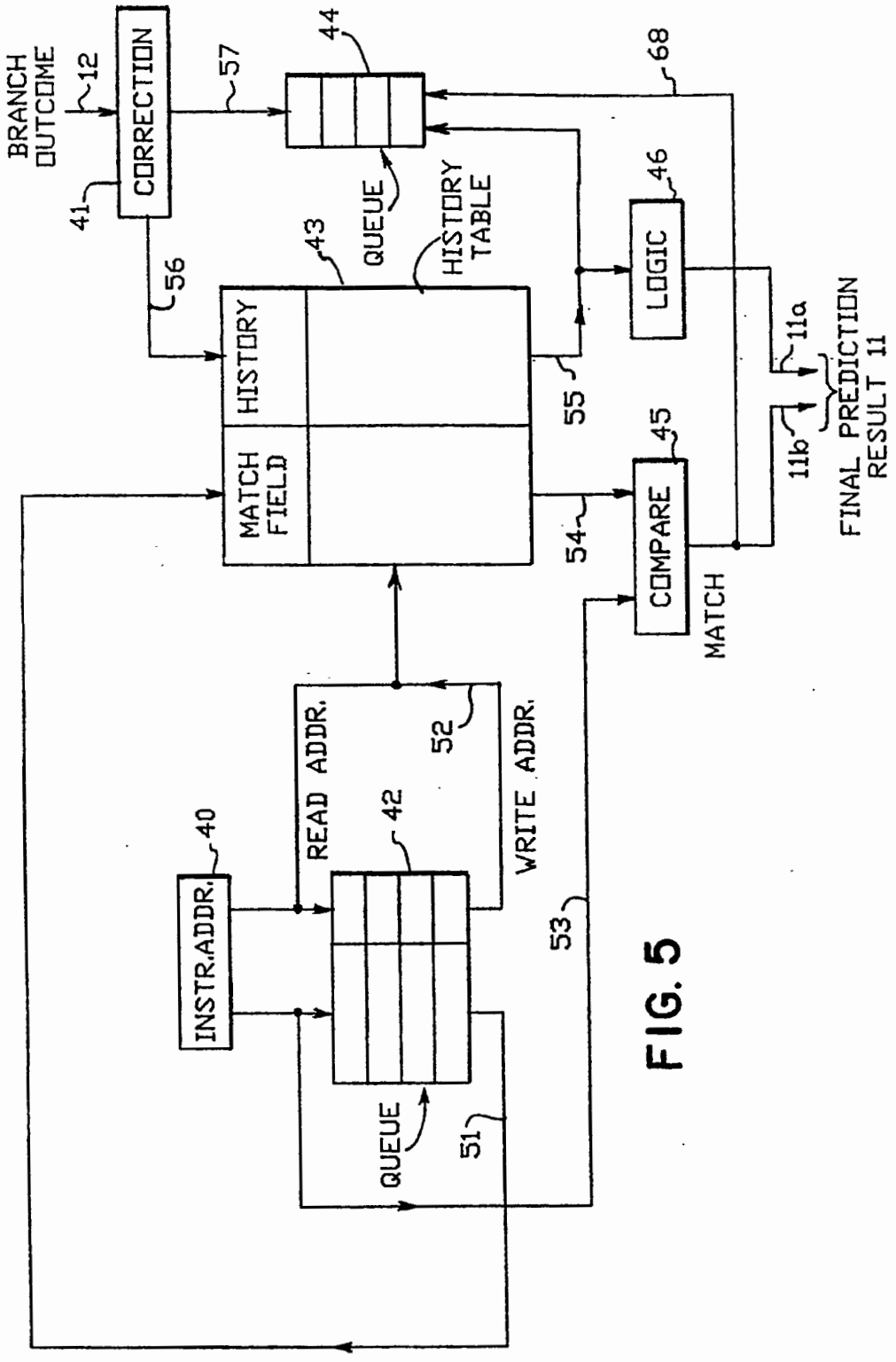

BRANCH OUTCOME
12
CORRECTION
41
57
44
68
QUEUE
HISTORY TABLE
43
46
LOGIC
56
HISTORY
MATCH FIELD
55
11a
11b
FINAL PREDICTION RESULT 11
45
54
COMPARE
MATCH
READ ADDR.
52
WRITE ADDR.
42
40
INSTR.ADDR.
53
QUEUE
51

FIG. 5

FIG. 4

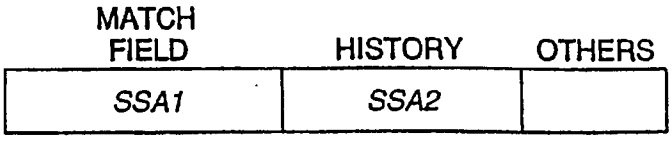


FIG. 6

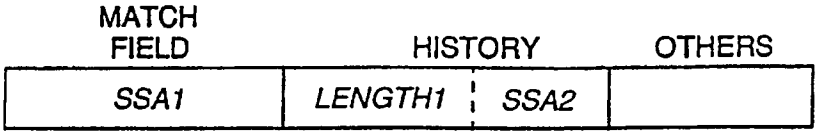


FIG. 7

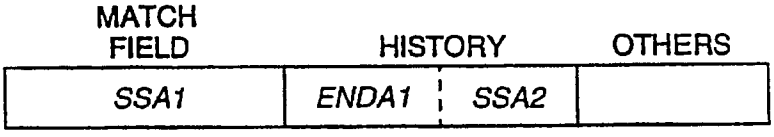


FIG. 8

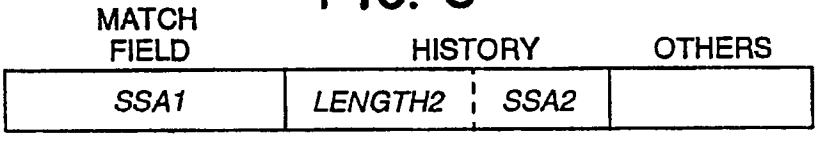


FIG. 9

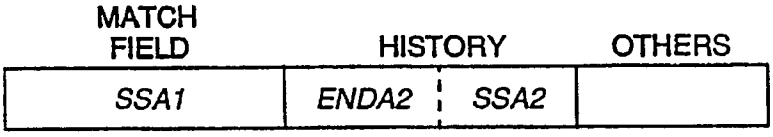
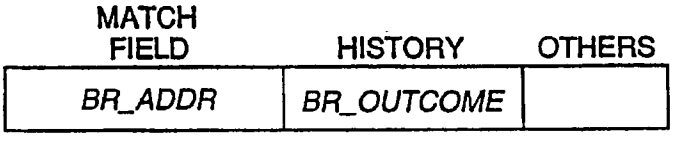


FIG. 10



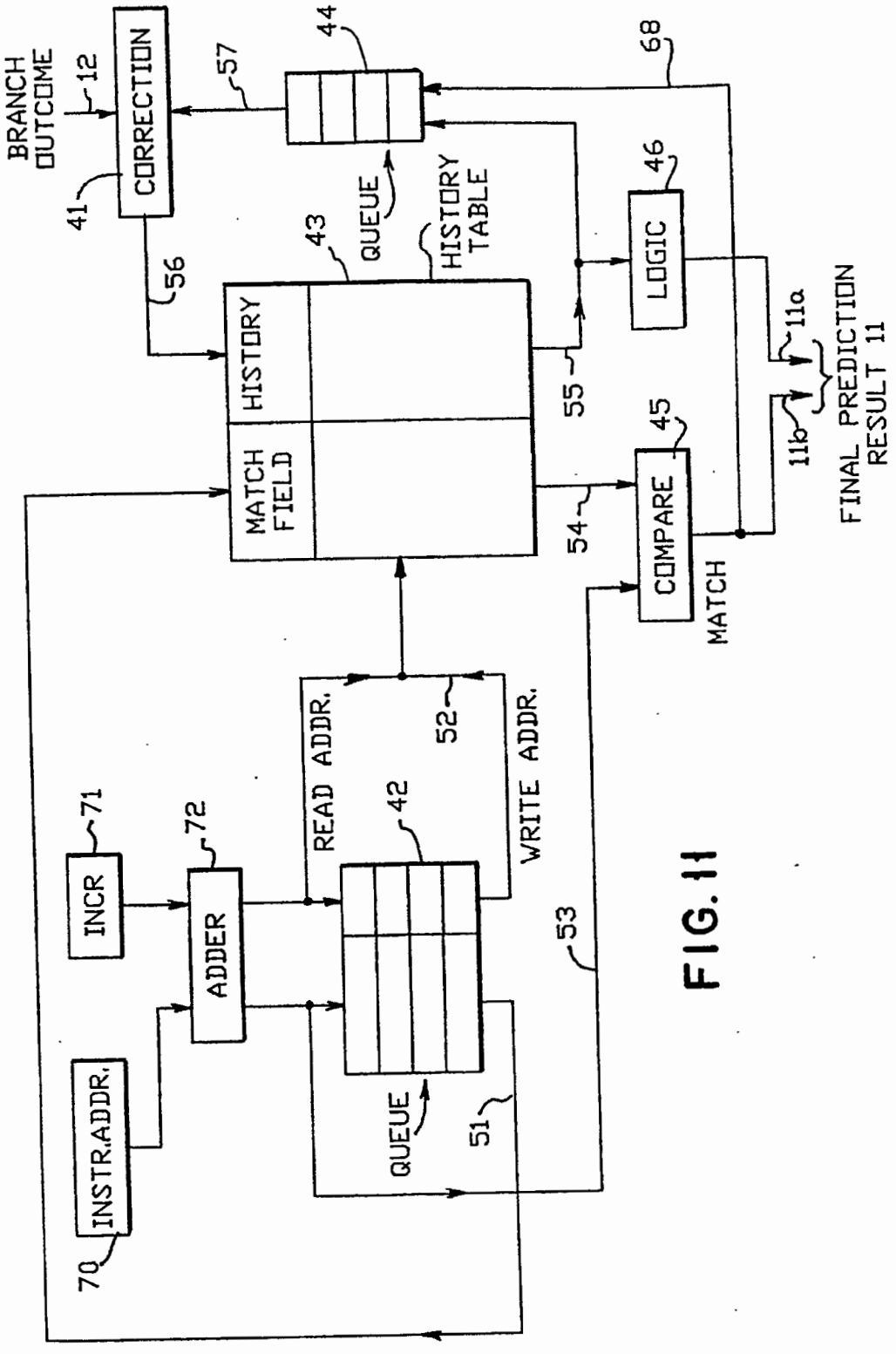

BRANCH OUTCOME
12
CORRECTION
41
57
44
68
QUEUE
HISTORY TABLE
43
46
LOGIC
56
HISTORY
MATCH FIELD
55
11a
11b
FINAL PREDICTION RESULT 11
45
COMPARE
54
MATCH
READ ADDR.
52
WRITE ADDR.
71
72
42
INCR
ADDER
53
INSTR.ADDR.
QUEUE
51
70

FIG. 11

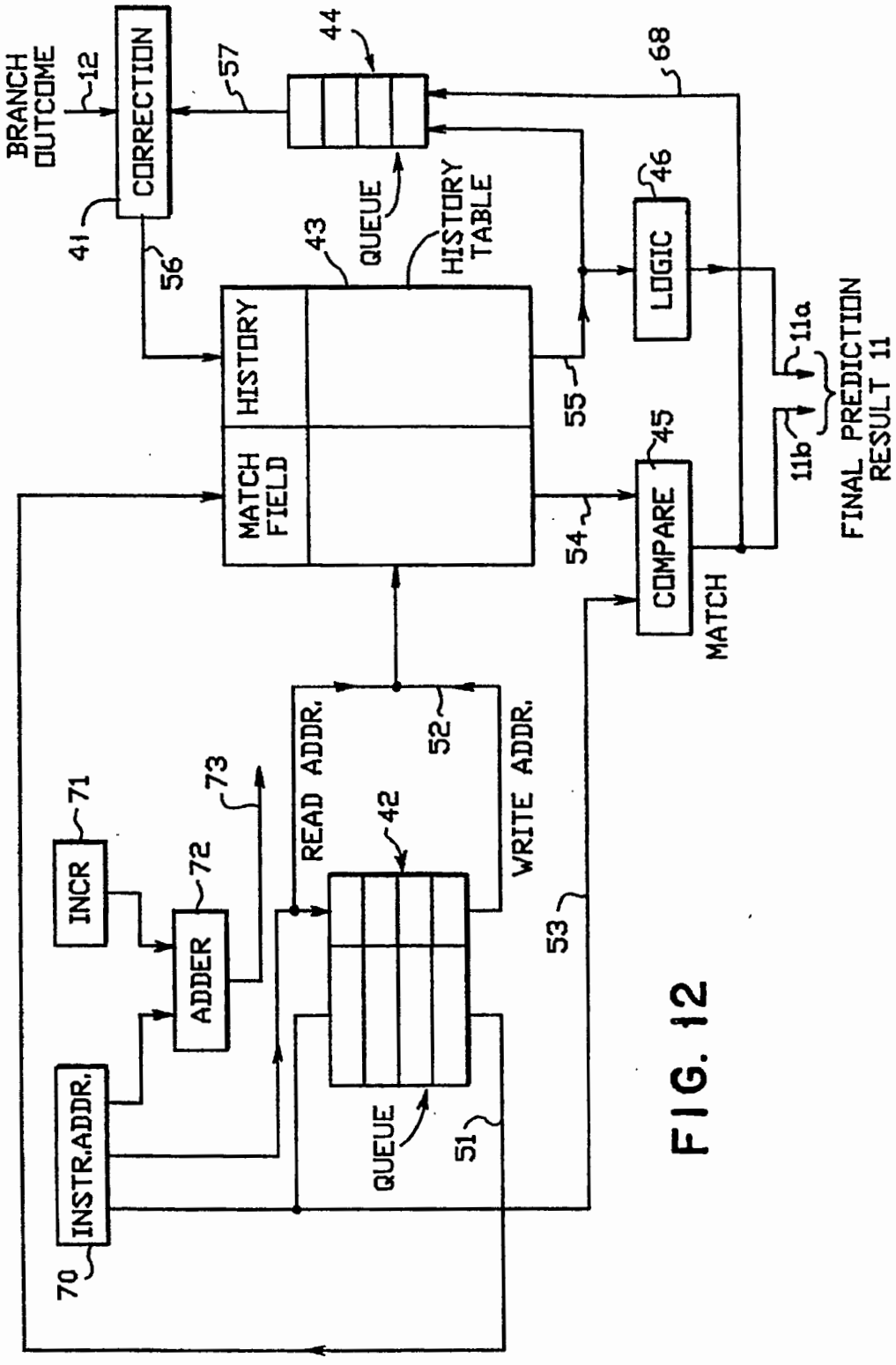

BRANCH OUTCOME
12
CORRECTION
41
57
44
68
56
QUEUE
HISTORY TABLE
43
46
LOGIC
HISTORY
MATCH FIELD
55
11a
11b
FINAL PREDICTION RESULT 11
45
54
COMPARE
MATCH
READ ADDR.
52
WRITE ADDR.
53
INCR
71
73
72
ADDER
42
INSTR.ADDR.
70
QUEUE
51

FIG. 12

FIG. 13

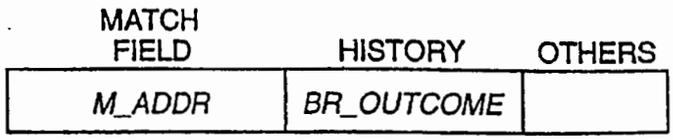


FIG. 14

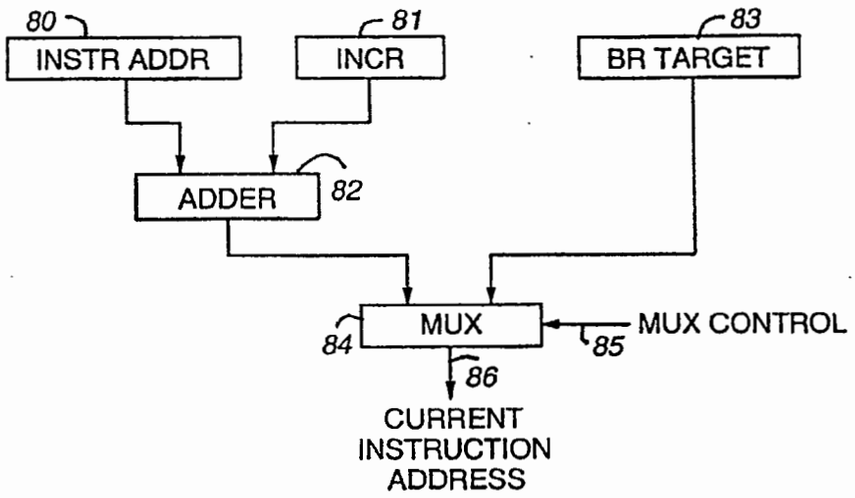


FIG. 15

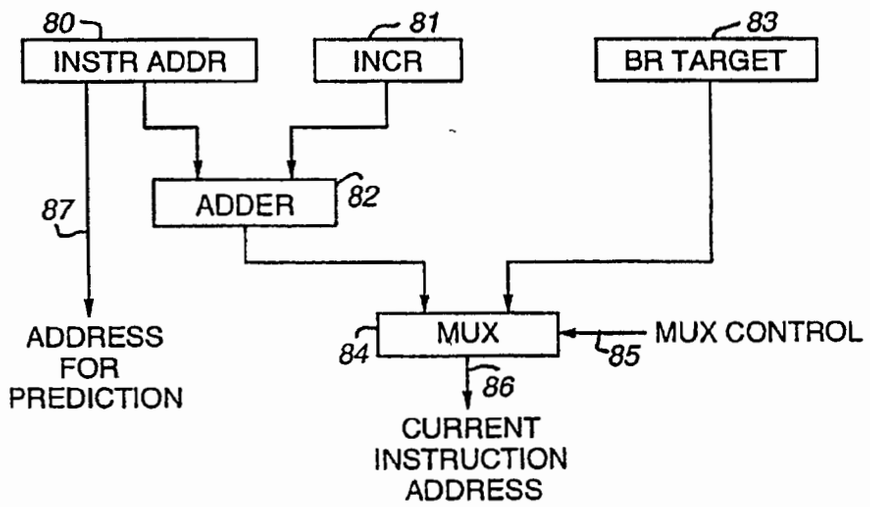


FIG. 16

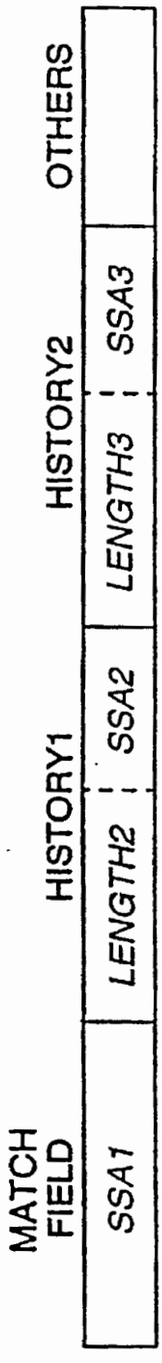
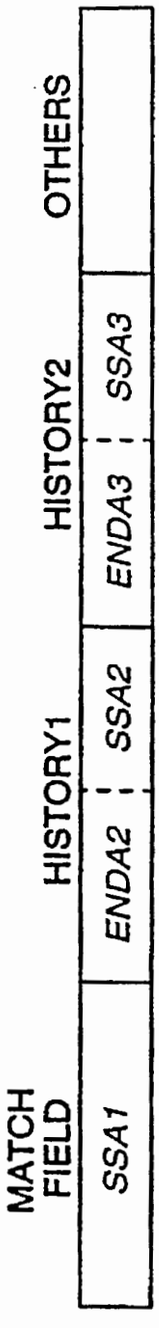

MATCH FIELD
HISTORY1
HISTORY2
OTHERS
SSA1
LENGTH2
SSA2
LENGTH3
SSA3

FIG. 17


MATCH FIELD
HISTORY1
HISTORY2
OTHERS
SSA1
END A2
SSA2
END A3
SSA3

HISTORY BASED BRANCH PREDICTION ACCESSED VIA A HISTORY BASED EARLIER INSTRUCTION ADDRESS

This is a continuation of application Ser. No. 07/860,631, filed Mar. 30, 1992, now abandoned.

FIELD OF THE INVENTION

This invention generally relates to control of instruction flow in a computer system and more particularly to the prediction of outcome of branch instructions using a history based branch prediction table.

BACKGROUND OF THE INVENTION

In typical pipelined processors, the processing of each instruction is divided into successive stages, with each stage of an instruction processing being handled by a specialized unit in a single cycle. Each successively earlier stage in the pipeline of stages is ideally handling simultaneously the successively next instruction. However, when a conditional branch instruction is encountered, there are several cycles of delay between the decoding of the branch instruction and its final execution/resolution, so it is not immediately known which instruction will be the next successive instruction. It is wasteful of the computer resource, however, to wait for the resolution of an instruction before starting with the processing of a next instruction. Therefore, it is recognized that it is advantageous to provide a mechanism for predicting the outcome of a conditional branch instruction in advance of its actual execution in order to provisionally begin processing instructions which will need to be processed if the prediction is correct. When the prediction is correct, the computer system can function without a delay in processing time. There is a time penalty only when a correct prediction cannot be attained ahead of time.

Throughout this application, the following terms and conventions will be used and shall have the indicated meaning. A branch instruction tests a condition specified by the instruction. If the condition is true, then the branch is taken, that is, following instruction execution begins at the target address specified by the branch instruction. If the condition is false, the branch is not taken and instruction execution continues with the instruction sequentially following the branch instruction. There may be branches that are unconditionally taken all the time. Such unconditional branches may simply be viewed as a special form of branches when appropriate.

A number of patents are directed to branch prediction mechanisms. For example, U.S. Pat. No. 4,370,711 to Smith discloses a branch predictor for predicting in advance the result of a conditional branch instruction in a computer system. The principle upon which the system is based is that a conditional branch instruction is likely to be decided in the same way as the instruction's most recent executions.

A simple strategy for handling branches is to suspend pipeline overlap until the branch is fully completed (i.e., resolved as taken or not taken). If taken, the target instruction is fetched from the memory. U.S. Pat. No. 3,325,785 to Stephens sets forth a static branch prediction mechanism. An improved method of this type is to perform static branch prediction by making a fixed choice based on the type of branch and statistical experience as to whether the branch will be taken. When the choice indicates that the branch is predicted to be not taken, normal overlap processing is continued on a conditional basis pending the actual branch outcome. If the choice proves wrong the conditionally initiated instructions are abandoned and the target instruction is fetched. The cycles devoted to the conditional instructions are then lost as well as the cycles to fetch the correct target instruction. However, the latter is often avoided in the prior art by prefetching the target at the time the branch is decoded.

A more sophisticated strategy is embodied in U.S. Pat. No. 3,559,183 to Sussenguth, which patent is assigned to the assignee of the present invention. It is based on the observation that the outcome of most branches, considered individually, tends to repeat. In this strategy, a history table of taken branches is constructed, which is known as a Branch History Table (BHT). Each entry in the table consists of the address of a taken branch followed by the target address of the branch. The table is a hardware construct and so it has a predetermined size. When the table is full, making a new entry requires displacing an older entry. This can be accomplished by a Least-Recently-Used (LRU) policy as in caches. When a branch is resolved as taken during execution, the history information associated with the branch is inserted into or updated in the BHT. Branch prediction and instruction prefetching are accomplished through constant search for the next taken branches in the history table. Upon final resolution/execution of a branch, any incorrect history information associated with the branch will be reset/updated properly. The major benefit of a BHT is to allow a separate branch processing unit to prefetch instructions into the instruction buffer (I-Buffer) ahead of the instruction decode stage. Such instruction prefetching into the I-buffer past predicted taken branches is possible due to the recording of target addresses for taken branches in the BHT. U.S. Pat. No. 4,679,141 to Pomerene et al, which patent is assigned to the assignee of the present invention, improves the BHT design by recording more history information in a hierarchical manner.

U.S. Pat. No. 4,477,872 to Losq et al, which patent is assigned to the assignee of the present invention, proposes a decode time branch prediction mechanism called a Decode History Table (DHT). The DHT mechanism improves the decode time static branch prediction methods of U.S. Pat. No. 3,325,785, to Stephens, by employing a hardware table to record simple histories of conditional branches. In the simplest form a DHT consists of a bit vector of fixed length. For each conditional branch instruction executed, a bit position in the DHT is derived through a fixed hashing algorithm, and the corresponding bit in the DHT records the outcome of the execution, indicating whether the branch was taken or not taken. Similar to U.S. Pat. No. 3,325,785, the DHT method allows overlap processing on a conditional basis past the decode of a conditional branch instruction if the branch is predicted, based on the DHT history, as not taken.

The common technique for the above cited branch prediction methods that are based on the dynamic histories of branches is to first record the previous outcome of branch instructions in a history based dynamic table and to then use such recorded histories for predicting the outcome of subsequently encountered branch instructions. Each branch recorded in such a history based table is recorded with either implicit or explicit information about the address of the recorded branch instruction so that the addresses of later encountered instructions can be correlated against the recorded information (i.e., by using the address of the instruction which is potentially a taken branch instruction in order to access the table for historical branch information). In order for branches to be predicted, the history table is

checked for a relevant entry by correlating the address of the instruction to be predicted against the implicitly or explicitly recorded address information of recorded branch instructions. In the DHT method, the bit position in the history vector is derived through hashing from the address of the conditional branch. In the BHT approach, an instruction is predicted to be a conditional branch which is taken if there is a match of the instruction address with a taken branch address found in an entry in the BHT and the target address recorded in this found entry is predicted as the current branch target address.

Numerous variations and improvements have been proposed in implementing a BHT. For example, in U.S. Pat. No. 4,679,141 to Pomerene et al, a technique is described for recording histories by block (e.g., doubleword) addresses instead of by individual branch instruction addresses. This technique offers advantages in reducing cache fetch traffic and the possibility of identifying the outcome of multiple branches within a single block. However, through more complex tags at each BHT entry, the block recording technique still conceptually operates as in conventional BHT methods in terms of identifying taken branch history by matching the addresses of the currently concerned instructions against the recorded addresses (or more precisely matching a portion of each such address) of the branch instructions recorded in the block.

U.S. Pat. No. 3,940,741 to Horikoshi et al sets forth an information processing device for processing instructions, including branches. A cache-like route memory is provided for storing branch target addresses of a plurality of taken branch instructions and the branch target instructions (code) themselves in corresponding relationship to the branch target addresses. The route memory is referenced by the target address of a branch instruction, and the branch target instruction at the corresponding branch target address is read out. The Horikoshi et al patent utilizes the target address of the branch, which is known upon decoding of the branch, to retrieve target instruction code for decode if the target address is recorded as a previous target for a taken branch in the route memory. Such a mechanism generally requires some delay before the access to the route memory due to the address formation for the branch target.

In practical implementations for branch prediction based on histories, timing is often found to be a critical factor. History table access generally involves address calculations and slower array lookup operations. In order to efficiently search constantly for potentially taken branches in BHT type implementations also involves complexity in the recording of history entries. Considering all of the tasks which need to be accomplished in order to make a prediction and to utilize it to advantage, it is desirable for practical reasons to be able to start the prediction process with respect to a particular instruction of interest as far in advance as possible and also to achieve the prediction as far in advance as possible. Nevertheless, there is no known art that offers the capability of either making a prediction decision or even initiating the prediction decision process with respect to an instruction which is potentially a taken branch instruction prior to identifying the address of that instruction of interest. It would be desirable to be able to predict instruction branches even earlier than the point where the address of an instruction is know which has the potential of being a taken branch instruction, because it would offer an opportunity to implement and use branch prediction with simpler and less costly circuits.

SUMMARY OF THE INVENTION

It is therefore an object of the present invention to provide an alternative approach to the prediction of branch outcome based on histories in order to achieve an earlier prediction.

It is also an object to initiate the process of predicting the outcome of an instruction which is potentially a taken branch instruction prior to the time when the address of that instruction is known.

A further object is to predict a taken branch without first determining the address of the branch instruction.

It is another object to provide a history based branch prediction table which can be maintained and accessed using an instruction address which historically preceeds the branch instruction.

Another object is to record in a history based branch prediction table a number indicative of the number of instructions by which such a preceeding instruction historically preceeds the recorded branch instruction.

It is a further object to record the address of such a preceeding instruction in a history based branch prediction table.

These and further objects have been achieved by this invention with an improved history table in which at least some of the entries are stored and accessed based upon the address of an instruction which historically preceeds the branch instruction itself. The access address may be used to determine the location of the entry in the table and/or may be stored in whole or in part in the entry itself. Furthermore, the improved history table may be of any known type including but not limited to branch history table types and decode history table types. The entries in the improved history table preferably are stored and accessed by the address of the preceeding taken branch target and preferably contain a number indicative of the number of instructions between the access address and the address of the branch instruction or its target.

Theory of Operation

Instructions are executed in a computer according to a certain logical sequence. Each instruction resides in the memory at a specific address. Two successive instructions in the memory may be executed in sequential or non-sequential ordering. When two instructions are sequentially adjacent to each other the address of the second instruction is exactly the address of the first instruction incremented by the length of the first instruction code. Non-sequential flow of instructions may occur during execution by various causes, among which branch instruction executions are the most frequent. Instructions in a typical computer system may be classified into two categories: breaker and non-breaker. A non-breaker instruction is the type that will cause the sequentially following instruction to be executed next, unless an exception condition (e.g., divide-by-zero) occurs. A breaker instruction is the type that can cause natural jumps to a non-sequentially related instruction to be executed next. Taken branch instructions are typical breaker type instructions. For the simplicity of description of the basic concept of the present invention only branch instructions will be considered for breakers.

Without an exception condition, the instruction stream executed is a sequence of sequential blocks (S-blocks), with each sequential block consisting of zero or more non-breakers followed by a branch instruction at the end. The branch instruction of an S-block may be taken or not taken during execution. Similarly, the instruction stream executed is a sequence of sequential, segments (S-segments), with each sequential segment consisting of one or more successive S-blocks such that only the last S-block has a taken

branch at the end. FIG. 1 depicts such a pattern of instruction flow during program execution. Only two S-segments (SS1 and SS2) are illustrated. SS1 consists of two S-blocks, SB11 and SB12. The branch instruction B11 at the end of SB11 is not taken, while the branch instruction B12 of SB12 is taken with the beginning of SS2 as the branch target. Similarly SS2 consists of three successive S-blocks SB21, SB22 and SB23. The ending branches B21 and B22 for SB21 and SB22, respectively, are not taken. The ending branch B23 of SB23 is taken with the beginning of another S-segment as the target.

Considering any conventional history based branch prediction method, the instruction stream is constantly examined for the next branch to be predicted. Upon locating such a branch instruction the branch predictor uses the branch address (in whatever representation) to look for an applicable history in the history table. For example, when conventional BHT design is applied to the S-segment S11 in FIG. 1, the S-segment S11 is scanned through for a taken branch in the history table. If B12 is the taken branch in the history table the S-segment S11 can be rather long and involves multiple cycles for the scan, even when block recording technique is used. Only upon locating the taken branch B12 in the history table prediction can then be carried out. Such conventional approach often leads to complex encoding of the history table in order to locate the relevant taken branches in a timely manner.

The present invention is based on the observation that the main purpose of conventional branch prediction methods is to resolve the flow of the instruction stream early in order to bring instructions into the instruction buffer soon enough for decoding. Branch addresses have been used for history table look-up for the natural reason that branches are normally the cause of non-sequential instruction flow. I have observed that predicting the outcome of a taken branch can be done without first locating the branch. For example, it is possible to create all S-segment history table (SSHT) to record the flow patterns among successive S-segments. With an SSHT, the instruction flow prediction can be achieved without locating and predicting individual branches. This observation has been further generalized into the concept of predicting instruction flow (as dominated by taken branches) based on addresses of instructions prior to the concerned branches. Use of such a technique for early resolution of branch predictions not only allows more timely resolution of instruction flow control but also offers flexibility for simpler implementations.

BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing and other objects, aspects and advantages of the invention will be better understood from the following detailed description of a preferred embodiment of the invention with reference to the drawings, in which:

FIG. 1 is a block diagram showing the instruction flow patterns between successive sequential blocks and sequential segments;

FIG. 2 is a block diagram showing the structure of pipelining with branch prediction;

FIG. 3 is a block diagram showing the general structure of a history-based branch predictor;

FIG. 4 shows the format of history table entries for prediction based on S-segments;

FIG. 5 is a block diagram showing a structure for the predictor of S-segment flow;

FIG. 6 shows an alternative format of history table entries for S-segment flow prediction, with length information recorded for the first S-segment;

FIG. 7 shows an alternative format of history table entries for S-segment flow prediction, with end address recorded for the first S-segment;

FIG. 8 sbows an alternative format of history table entries for S-segment flow prediction, with length information recorded for the second S-segment;

FIG. 9 shows an alternative format of history table entries for S-segment flow prediction, with end address recorded for the second S-segment;

FIG. 10 shows the format of history table entries for a conventional DHT type branch prediction mechanism;

FIG. 11 is a block diagram showing a branch predictor in which an address adder is used to form the precise branch address;

FIG. 12 is a block diagram showing an alternative design of the predictor shown in FIG. 11 which bypasses the address adder for history table access;

FIG. 13 shows an alternative to the history table entry format shown in FIG. 10 with a more general address at the MATCH FIELD;

FIG. 14 shows an alternative method of address generation from the method illustrated in FIG. 11;

FIG. 15 illustrates the method of carrying out branch prediction always based on the address of the previous instruction processed;

FIG. 16 shows a generalization of the history table entry format shown in FIG. 8, with the successive flow between more than two S-segments recorded; and

FIG. 17 shows a generalization of the history table entry format shown in FIG. 9, with the successive flow between more than two S-segments recorded.

DETAILED DESCRIPTION OF A PREFERRED EMBODIMENT OF THE INVENTION

Referring now to the drawings, and more particularly to FIG. 2, there is illustrated a typical computer pipeline structure with a branch prediction mechanism. The branch predictor 10 performs the branch prediction and branch history management operations. Instructions 31–37 are processed according to a certain logical sequence for a particular pipeline design. Addresses 21–27 represent the corresponding addresses of instruction stream 31–37. The branch predictor 10 receives an input instruction address on line 20 from address 23 in the instruction address pipeline. The prediction output 11 from the branch predictor 10 is normally used to determine the subsequent instructions to be fetched into instruction buffer. The branch predictor 10 also receives from input signal line 12 information that allows it to determine the correctness of prediction and the adjustment of history information. The issuing of an instruction 37 for actual processing (e.g., for decode or for a later step) is usually one or more cycles past the branch prediction stage in the pipeline.

The function of the branch predictor 10 is highly dependent upon the particular branch prediction design employed. For instance, with respect to the DHT mechanism employed in IBM 3090 systems, the branch predictor 10 is activated during the decode stage of an instruction in the pipeline with the output 11 being as simple as a single bit indicating the predicted outcome as taken or not taken for the instruction being decoded. In a more sophisticated BHT mechanism, the

branch predictor **10** can be activated for an instruction several cycles earlier than the actual decode of the instruction with the prediction output **11** indicating whether the instruction is a taken branch or not and the target instruction address if the instruction is guessed as a taken branch. Although not a necessity, the output **11** of the branch predictor is normally passed to an instruction fetch unit **13** for prefetching the predicted subsequent instructions. The branch predictor **10** in prior art branch prediction schemes for early branch resolution conceptually guesses the outcome of a branch instruction **33** according to the branch address **23** input from line **20**.

Referring now to FIG. **3**, which is a block diagram representation of the branch predictor **10** with a history table **43** utilized for branch prediction. An instruction address **40** from a register is the input instruction address for which prediction steps are to be taken. For simplicity of description of the invention, the history table **43** may be a 1-dimensional table with a fixed number of entries. The selection of an entry in the history table **43** for a given instruction address **40** is via proper hashing of certain address bits, Each entry in the history table **43** generally consists of two portions: MATCH FIELD and HISTORY. The MATCH FIELD portion is used for matching with input address bits in order to determine whether the history information recorded in the HISTORY portion of the entry is applicable. In the simplest case the MATCH FIELD portion can be null with the address match results considered always successful. The address queue **42** is used to hold address indicators for instructions with unfinished branch prediction status, The left portion of each entry of the address queue **42** contains address identifiers that are compatible with the MATCH FIELD in the history table **43**. The right portion of each entry contains a tag allowing the identification of the entry selection for the associated instruction address. The address queue **42** is implemented as a first-in-first-out queue, The write address **52** signal is used for updating the history table **43**. Upon update to the history table **43** the left portion of the oldest entry in the address queue **42** supplies the new value **51** for the MATCH FIELD. The result queue **44** is used for queuing predictions that have not yet been verified as correct,

The compare logic **45** is used for determining a match between the MATCH FIELD value **54** of the selected entry of the history table **43** and the correspondence in the instruction address **40**. The output HISTORY value **55** at the selected entry of the history table **43** is input to both the result queue **44** and logic **46**. The logic **46** simply readjusts the HISTORY value received into a proper format. When a match is found at the compare logic **45**, MATCH line **64** is raised (with value **1**) and the AND logic **48** provides the prediction result to the OR logic **49** via line **67**. If a match is not found at the compare logic **45**, the NO MATCH line **63** is raised instead and causes a default guess result to be passed to the OR logic **49** via line **66**. The output **11** from the branch predictor consists of two portions **11***a* and **11***b*. The output line **11***a* from the OR logic **49** gives the final result of the prediction based on the contents of the instruction address **40**. The compare unit **45** provides the MATCH signal to the output line **11***b*. In certain implementations the output line **11***b* can be eliminated. For example, in the IBM 3090 implementation of a DHT, the MATCH FIELD is null and the MATCH signal **11***b* is conceptually on always and hence the compare unit **45** and the output line **11***b* can be ignored. The correction logic **41** takes as input the actual branch outcome **12** from the branch execution unit and via signal line **57** the earlier prediction from the first-in-first-out result from queue **44**. By matching the branch prediction with the actual branch outcome the correction logic **41** performs the function of verifying correctness of active branch predictions and triggering necessary adjustments to the HISTORY portion of a relevant entry in the history table **43**. The two queues **42** and **44** are controlled such that the history table **43** can be updated consistently with the corresponding prediction verified by the correction unit **41**. In an implementation that allows only one access (Read or Write) per cycle to the history table **43**, priority needs to be provided between concurrent accesses. For example, update to the history table **43** due to signal **56** from correction logic **41** may delay the read access for a new prediction. Such implementation choices are not critical to the present invention and will not be discussed in further detail. The MATCH signal from the compare unit **45** is also sent to the result queue **44** via signal line **68**. It is assumed that each entry of the result queue **44** has a history hit bit (HH-bit) with the value (**0** or **1**) received from the signal line **68** upon prediction.

Upon history reset as triggered by signal **56** from the correction unit **41**, all the subsequent instruction flow predictions may be regarded as abandoned. As a result, upon a history reset condition, both the address queue **42** and the result queue **44** will be emptied. Also as a direct consequence, prefetched instructions in the instruction buffer must be flushed as well.

In many implementations, the instruction address need not always trigger prediction through the history table Eliminating unnecessary prediction actions can reduce the size requirements for the address queue **42** and the result queue **44**. Certain unspecified controls are assumed for controlling the activation and deactivation of the prediction mode.

The general concept of the present invention may be realized in various different ways. In order to illustrate this, various techniques will be described now with different types of designs utilizing different kinds of histories for prediction of instruction flow.

Instruction flow prediction will be described now using an S-Segment History Table (SSHT). In a conventional BHT approach, the MATCH FIELD portion of an entry in the history table **43** identifies the address of a previously taken branch and the HISTORY portion records the target instruction address to which the branch was taken. In the SSHT approach the history table **43** records the flow history of S-segment instead. FIG. **4** depicts the format of an entry in the history table **43** in which SSA1 and SSA2 represent the addresses of two S-segments such that a previous execution of the program previously flowed from the S-segment (SS1) beginning at SSA1 to the S-segment (SS2) beginning at SSA2 consecutively. Therefore, when the flow history entry depicted in FIG. **4** gets recorded in the history table **43**, an execution flow to the first instruction of SS2 (at. SSA2) directly from the last (branch) instruction of SS1 has just occurred.

Referring now to FIG. **5**, which shows a modified version of the block diagram of FIG. **3**, for illustrating an SSHT implementation. The compare unit **45** simply outputs the MATCH signal **11***b* as part of the final prediction result **11**. The default guess **61** in FIG. **3** is discarded. Assuming that the prediction mode is activated, there are two possibilities. When the MATCH line is raised (=1) in the output **11***b* of the predictor, an S-segment flow history is found with the instruction address **40** as the first S-segment and the address of the next S-segment indicated in the prediction output **11***a*.

On the other hand, when the MATCH line is low (=0) in the output line **11***b*, such an S-segment flow history is missing in the history table **43**. In the SSHT implementation, the branch prediction for instruction address **40** can be activated only if the beginning of an S-segment is located, either through the S-segment chain as searched from the SSHT history table **43** or when a taken branch actually occurs during execution. When the S-segment chain through history pairs <SSA1,SSA2> of the history table **43** reaches a point where a next flow of S-segments is missing, there can be several different ways of proceeding. One simple approach to handling missing history in SSHT **43** is to hold the prediction activation until desired flow information is determined from a relevant taken branch. The correction logic **41** is invoked only when a breaker (e.g., taken branch) is found in execution. In principle, when the instruction to be executed at the moment is in an S-segment at beginning address SSA1, the oldest entry in the address queue **42** contains information for that S-segment and the oldest entry of the result queue **44** contains the corresponding earlier prediction result. The correction logic **41** is invoked when a breaker of the currently active S-segment is detected during execution, for which the correction logic **41** checks whether the oldest entry in the result queue **44** contains correct S-segment flow information. If at the oldest entry of the result queue the history-hit flag HH-bit is on (=1) and the target S-segment address matches the target for the currently executed breaker, the prediction is regarded as successful. Otherwise, either when the HH-bit is off (=0) or when the target S-segment address is wrong, the earlier prediction is regarded as unsuccessful. Upon detection of unsuccessful prediction, the correction logic **41** triggers history update operations by updating or inserting the relevant entry into the history table **43**. The new history information gets the MATCH FIELD value from line **51** from the address queue **42** and the HISTORY value from line **56** from the correction unit

S-segment prediction methods utilizing an SSHT have been described above. A major purpose for prediction of instruction flow is to facilitate prefetching of instructions into the instruction buffer. Still referring to FIG. **5**, the output signals **11***a* and **11***b* are passed to a separate instruction prefetch unit. When the MATCH signal **11***b* is on (=1) the output **11***a* contains the address SSA2 of the S-segment which is predicted to follow the currently predicted S-segment at address SSA1. The instruction prefetch control still needs information on how long the S-segment at SSA1 is in order to determine when to start prefetching instructions for the S-segment at SSA2. There are various solutions for providing such length information to the instruction prefetch control. The most natural approach is to provide the information through the history table **43**. An enhancement to the SSHT entry format shown in FIG. **4** is illustrated in FIG. **6**, in which the HISTORY portion also contains a new tag LENGTH1. LENGTH1 describes the length of the S-segment at address SSA1. Upon a history-hit, as indicated by MATCH=1 in the output line **11***b* of FIG. **5**, the output line **11***a* also provides the LENGTH1 information to the instruction prefetch control. With the LENGTH1 information the instruction prefetch control is able to determine how far sequential instructions for the S-segment at address SSA1 needs to be prefetched into the instruction buffer before the instruction prefetch for the S-segment at SSA2 should start. The result queue **44** in FIG. **5** also records the [LENGTH1, SSA2] prediction information. Upon locating a breaker during execution, the correction logic **41** calculates the actual length for the S-segment at SSA1 and determines whether the LENGTH1 prediction is correct. If not, proper instruction prefetch corrections need to be carried out in addition to the update of history information in the SSHT history table **41**. There are various ways the instruction prefetch unit may handle history miss conditions (i.e., when the MATCH signal is off in output line **11***b* from the predictor). A straightforward approach is to carry out limited sequential instruction prefetching for the S-segment at address SSA1. For example, two consecutive doublewords may be prefetched into the instruction buffer for the current S-segment and the rest of instruction fetch might wait till more information becomes available. The history tag LENGTH1 may be represented in various ways as well, for which the most straightforward is to record the number of bytes. In many computer systems, instructions are all of equal length (e.g., a 4-byte word), in which case LENGTH1 only needs to record the number of instruction words involved. In IBM/390 architecture each instruction is of a length which is a multiple (1, 2 or 3) of 2-byte halfwords, for which LENGTH1 only needs to record the number of halfwords involved. In practice it is desirable to utilize few bits for LENGTH1 recording. From simulation studies it has been observed that a great majority of S-segments are rather short (e.g., ≦64 bytes). When the length of an S-segment is beyond the capacity of recording in the LENGTH1 tag, an overflow condition might be flagged at the LENGTH1 tag in the SSHT history table **43**. Upon a LENGTH1 overflow the instruction prefetch control might simply treat the length of the corresponding S-segment as infinite, which will not be unreasonable in practice due to the rarity of long S-segments. Another possible alternative is to break up a long S-segment into multiple short partitions for recording as different entries in the SSHT history table

A possible alternative to the recording of LENGTH1 information in FIG. **6** is to precisely identify the ending address of an S-segment instead. FIG. **7** describes such an alternative format for history table entry, in which an address tag ENDA1 is used for identifying the end of the S-segment beginning at address SSA1. The instruction prefetch control and the correction unit **41** might easily utilize the ENDA1 information for instruction prefetching and prediction verification.

In the above configuration, the LENGTH1 tag in FIG. **6** and the ENDA1 tag in FIG. **7** are both for identifying the first S-segment (beginning at address SSA1). It is also straightforward to modified the illustrated predictor constructs of FIG. **5** so that these tags are associated with the target S-segment instead. That is, the LENGTH1 tag could be replaced with a LENGTH2 tag that describes the length for the target S-segment beginning at address SSA2. Similarly, the ENDA1 tag may be replaced with a ENDA2 tag that describes the ending address of the target S-segment beginning at address SSA2.

In the above described SSHT prediction mechanisms, the instruction flow predictions are based on S-segment granularity. The principles can be generalized into other granule sizes, as long as the prediction operations can be carried out in a definitive manner with reasonable accuracies. For example the history of instruction flow can be constructed based on S-block granules instead. Prediction of S-block flow may be carried out in the manner similar to conventional BHT mechanisms. For example, it is not necessary to record sequential flow of S-blocks upon a miss of S-block flow in the history table. Sequential flow may be assumed and the next S-block may be looked for in the history table.

The above described SSHT prediction mechanism illustrates a way of realizing the concept of predicting instruction

flow prior to identifying a branch address. Now another form of application of the concept will be described. In conventional history based branch prediction methods, the address of a branch is the basis for history look up. FIG. **10** describes a general format for history table entry for conventional methods. The MATCH FIELD contains BR_ADDR, which normally consists of a subset of the address bits of the branch, that is used for matching with the branch being predicted. The HISTORY portion contains the history BR_OUTCOME for the actual outcome of the branch upon a previous execution. In DHT type methods, the BR_OUTCOME can be as simple as a 1-bit tag indicating whether the associated conditional branch was taken or not taken.

Consider the branch predictor platform in FIG. **5**. In many branch prediction methods, instruction addresses are constantly passed to the branch predictor. For example in the DHT approach, the decode of each instruction will trigger a possible branch prediction using the address of the currently decoded instruction as the instruction address **40** for the branch predictor. The prediction output **11***a* is considered active only when the MATCH signal is on (=1) in the output line **11***b*. An instruction can be reached (e.g., for decode) generally in two ways: a) falling through sequentially from the previous instruction; and b) being the target from another taken branch instruction. In most computer systems, a program counter is used for holding the address of a currently active instruction. The address of the sequentially next instruction can be derived by adding to the current program counter the length increment of the first instruction. FIG. **11** modifies the block diagram of FIG. **5** to reflect the process of deriving a new instruction address in some implementations. In FIG. **11** the instruction address **40** input of FIG. **5** is replaced with three components: an instruction address **70**, a length increment register INCR **71**, and an address adder ADDER **72**. At the beginning of each cycle, ADDER **72** sums up the contents of INSTR.ADDR. **70** and INCR **71** to output the next instruction address for branch prediction. In certain more complex implementations more multiplexing operations might be involved in deriving the instruction address for branch prediction. The INCR **71** register content may either be 0 (e.g., for a taken branch target) or>0 (e.g., for a sequentially fall-through instruction).

In high performance computer designs, ADDER **72** might introduce a heavy burden on the critical path timing for branch prediction, which involves array accessing for the history table. FIG. **12** is a modification of FIG. **11** for reducing the timing burden of branch prediction. ADDER **72** is still used for generating the new instruction address from INSTR.ADDR. **70** and INCR **71**, since the address is generally needed for various operations other than branch prediction. The input to the branch predictor itself is, however, not from the output of ADDER **72**. Instead the INSTR.ADDR. **70** itself supplies the input address for branch prediction. In this arrangement the prediction of branch outcome can be based on input address for an instruction that is sequentially preceding the actual instruction being predicted. The output **73** of ADDER **72** is passed to other units that require the precise address of the current instruction. This implementation clearly reduces the path timing of branch prediction by at least the amount needed for ADDER **72**. The accuracy with this modification should be about the same as the one in FIG. **11** since it is highly repeatable how an instruction is reached during program execution. In this illustration it is not necessary to include length information in the entry for the history table **43**, since the register INCR **71** itself already provides the increment information for the previous instruction.

FIG. **13** describes a modification to the format of the history table entry of FIG. **10**. The MATCH FIELD now contains M_ADDR, which identifies a more general address that can be preceding the actual branch address under consideration.

In some implementations the portion of FIG. **11** for instruction address generation, including INSTR.ADDR. **70**, INCR **71** and ADDER **72**, can be carried out by an alternative method described in FIG. **14**. In FIG. **14** an additional BR TARGET register **83** and a multiplexer MUX **84** are included. INSTR ADDR register **80** now always contains the address of the instruction last processed (e.g., decoded). BR TARGET **83** contains the address of target instruction if the last processed instruction is a taken branch. The multiplexer MUX **84** is used to select an address from BR TARGET **83** or from ADDER **82**, depending upon whether the last instruction processed results in sequential flow. The MUX CONTROL **85** is a signal controlling the selection (e.g., from the instruction decoder). The output **86** from MUX **84** provides the final instruction address to be processed currently. When the last instruction processed is a taken branch (or any kind of execution that results in a non-sequential jump of instruction flow) BR TARGET register **83** will provide the current instruction address. When the instruction flow from the last processed instruction is sequential the address of current instruction is formed by ADDER **82** as illustrated before. FIG. **15** describes a modification to FIG. **14** for the purpose of early timing for branch prediction. As described for FIG. **12** the branch predictor **10** can use address not belong to the currently processed instruction. In FIG. **15** the INSTR ADDR register **80**, which always contains the address of the last processed instruction, is passed for branch prediction via signal line **87**. What should be noted is that in FIG. **15** the branch prediction can be acted upon the target of a taken branch based on the address of the taken branch itself.

The concept and techniques described in this preferred embodiment can be applied with different variations. For example, it is well-known that the matching of history entry through branch address can utilize only portion of the address bits. Hence, the MATCH FIELD portion of a history table entry described in this embodiment may cover only partial address bit information. More generally, the MATCH FIELD may cover information definitively derivable from relevant instruction status. For instance, the MATCH FIELD may contain a value hashed from an instruction address with other definitive execution status (e.g., the Segment Table Origin of IBM/390 architecture, which identifies the process address space of a particular program). The recording of length information of a S-segment may be in terms of the number of instructions or the number of branches covered by the S-segment, as long as certain pre-scanning mechanism is provided by the instruction fetch control to identify the relevant information through the prefetched instruction stream.

It is possible to enhance the history table with additional information when beneficial. For example, it is possible to include in the SSHT history table entry of FIG. **8** an additional field TARGET CODE, which stores the first instruction code at the target S-segment beginning at address SSA2. FIG. **16** and FIG. **17** contain extensions to the history information described in FIG. **6** and FIG. **7**, with the flow patterns of three successive S-segments at each history table entry. Maintaining such explicitly the flow of more than two S-segments, however, requires much higher complexity in hardware design and should be exercised with caution.

While the invention has been described in terms of a preferred embodiment, those skilled in the art will recognize

...t the invention can be practiced with modification within ...s spirit and scope of the appended claims.

Having thus described my invention, what I claim as new ...d desire to secure by Letters Patent is as follows:

1. Apparatus for prefetching instructions for execution, comprising:

a history based branch prediction table for storing information about previously executed segments of sequentially addressed instructions, said segments terminating with a taken branch instruction;

said table storing for each of said segments a target address for said each segment, said target address being an address of a next executed instruction following said each segment when said each segment was previously executed;

said table also storing for said each segment, an address tag identifying a first instruction of said each segment and information to determine a length of said each segment, said length being variable from segment to segment;

means for holding in an instruction buffer a prefetched sequence of instructions for execution, said prefetched sequence including a last instruction having a last instruction address;

address comparing means for comparing said last instruction address to said address tags stored in said table in order to detect a matching address tag stored in said table; and

prefetching means responsive to detection of a matching address tag in said table for prefetching storage said segment of instructions identified by said matching address tag, adding said prefetched segment to said prefetched sequence, and making said target address of said prefetched segment said last instruction address of said prefetched sequence.

2. Apparatus as defined in claim 1 wherein at least some of said address tags stored in said table correspond to a target address stored in said table.

3. Apparatus as defined in claim 2 wherein all of said address tags stored in said table correspond to a target address stored in said table.

4. Apparatus as defined in claim 1 wherein some of said segments terminating with a taken branch instruction include at least one additional branch instruction in addition to said terminating taken branch instruction.

5. Apparatus as defined in claim 4 wherein said additional branch instructions were not taken when said segments containing said additional branch instructions were previously executed.

6. Apparatus as defined in claim 1 wherein at least some of said segments have a size corresponding to at least three instructions.

7. Apparatus as defined in claim 1 wherein each said segment terminating with a taken branch instruction has for a stored target address a target address of said each taken branch instruction.

8. Apparatus as defined in claim 7 wherein all of said segments terminate with a taken branch instruction.

* * * * *

US005586254A

United States Patent [19]

Kondo et al.

[11] Patent Number: 5,586,254

[45] Date of Patent: Dec. 17, 1996

[54] **SYSTEM FOR MANAGING AND OPERATING A NETWORK BY PHYSICALLY IMAGING THE NETWORK**

[75] Inventors: **Mariko Kondo**; **Teruo Nakamura**; **Yumiko Mori**; **Toshiyuki Tsutsumi**, all of Yokohama, Japan

[73] Assignee: **Hitachi Software Engineering Co., Ltd.**, Kanagawa-ken, Japan

[21] Appl. No.: **18,430**

[22] Filed: **Feb. 16, 1993**

[30] **Foreign Application Priority Data**

Feb. 13, 1992 [JP] Japan 4-026405
Sep. 18, 1992 [JP] Japan 4-249890

[51] **Int. Cl.**[6] **G06F 15/40**; G06F 15/66; G06F 13/94

[52] **U.S. Cl.** **395/200.1**; 395/615; 340/825.03; 364/228; 364/229.4; 364/927.99

[58] **Field of Search** 395/51, 200, 600, 395/200.1; 340/825.03

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,833,625	5/1989	Fisher et al.	364/518
4,964,088	11/1990	McAuliffe et al.	364/200
5,175,800	12/1992	Galis et al.	395/51
5,202,985	4/1993	Goyal	395/600
5,307,484	4/1994	Baker et al.	395/600

FOREIGN PATENT DOCUMENTS

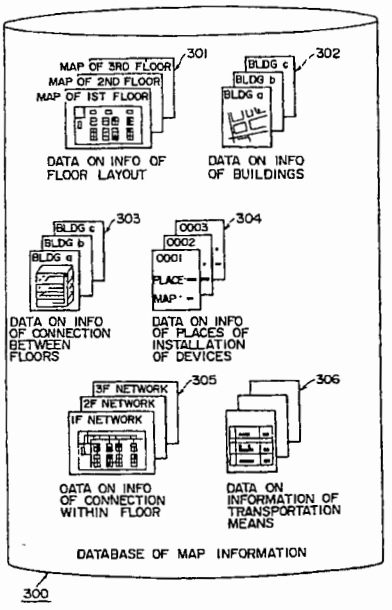
61-180340	8/1986	Japan .
63-117532	11/1986	Japan .
63-226772	9/1988	Japan .
63-279643	11/1988	Japan .
1-78053	3/1989	Japan .
1-218236	8/1989	Japan .
2-18651	1/1990	Japan .
2-305140	12/1990	Japan .
3-101539	4/1991	Japan .
3-973300	4/1991	Japan .
3-195230	8/1991	Japan .

Primary Examiner—Thomas C. Lee
Assistant Examiner—Rehana Perveen Krick
Attorney, Agent, or Firm—Fay, Sharpe, Beall, Fagan, Minnich & McKee

[57] **ABSTRACT**

A system for operating and managing the network equipment is so adapted as to operate and manage a network in which plural computers and network devices are connected to each other. The system is provided with database storing data corresponding to the computers and the network devices and with means for preparing a network specification drawing which satisfies conditions required by the user from the data, for checking the physical data as to whether the network specification satisfies the physical data, for checking the logical data as to whether the network specification satisfies the logical data, and for displaying the network specification drawing in a two-dimensional or three-dimensional manner on the basis of the data stored in the database. The system for operating and managing the network equipment can reduce and simplify management business for network managers as well as management business for managing materials and products by managers managing the materials and products. Further, the system can take necessary measures in case of a fault or a failure of the network and save a resource by sharing the computer resources and the data in an appropriate way.

17 Claims, 93 Drawing Sheets



US005805816A

United States Patent [19]

Picazo, Jr. et al.

[11] Patent Number: 5,805,816

[45] Date of Patent: Sep. 8, 1998

[54] NETWORK PACKET SWITCH USING SHARED MEMORY FOR REPEATING AND BRIDGING PACKETS AT MEDIA RATE

[75] Inventors: **Jose J. Picazo, Jr.**, San Jose; **Paul Kakul Lee**, Union City; **Robert P. Zager**, San Jose, all of Calif.

[73] Assignee: **Compaq Computer Corp.**, Houston, Tex.

[21] Appl. No.: **788,429**

[22] Filed: **Jan. 28, 1997**

Related U.S. Application Data

[62] Division of Ser. No. 694,491, Aug. 7, 1996, which is a continuation of Ser. No. 498,116, Jul. 5, 1995, which is a continuation-in-part of Ser. No. 881,931, May 12, 1992, Pat. No. 5,432,907.

[51] **Int. Cl.**[6] **G06F 13/00**

[52] **U.S. Cl.** **395/200.53**; 395/200.64; 395/200.79; 395/200.8; 370/401; 370/230; 370/315; 370/351

[58] **Field of Search** 395/200.53, 200.64, 395/200.79, 200.8; 370/230, 315, 351, 401

[56] **References Cited**

U.S. PATENT DOCUMENTS

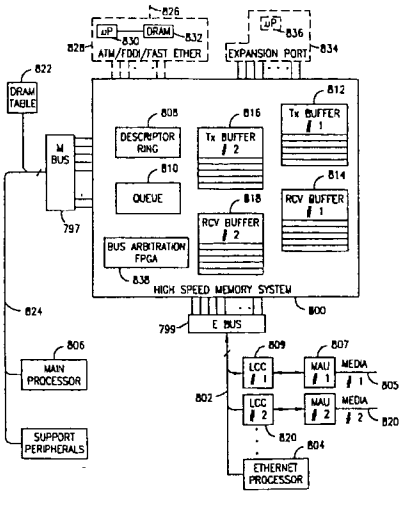
4,641,307	2/1987	Russell	370/445
4,715,030	12/1987	Koch et al.	370/85
5,133,062	7/1992	Joshi et al.	395/500
5,210,749	5/1993	Firoozmand	370/463
5,264,742	11/1993	Sourgen	307/465
5,274,631	12/1993	Bhardwaj	370/401
5,299,313	3/1994	Petersen et al.	395/200.64
5,303,302	4/1994	Burrows	380/49
5,339,313	8/1994	Ben-Michael et al.	370/230
5,361,372	11/1994	Rege et al.	395/200.64
5,434,863	7/1995	Onishi et al.	395/200.64
5,440,546	8/1995	Bianchini, Jr. et al.	370/60
5,440,690	8/1995	Rage et al.	395/200.8
5,457,681	10/1995	Gaddis et al.	370/56
5,477,547	12/1995	Sugiyama	370/85
5,521,913	5/1996	Gridley	370/58.2
5,560,029	9/1996	Papadopoulos et al.	395/800.25

Primary Examiner—Christopher B. Shin
Attorney, Agent, or Firm—Jenkens & Gilchrist

[57] **ABSTRACT**

A hub circuit with an integrated bridge circuit carried out in software including a switch for bypassing the bridge process such that the two bridged networks effectively become one network. An in-band management process in software is disclosed which receives and executes network management commands received as data packets from the LANs coupled to the integrated hub/bridge. Also, hardware and software to implement an isolate mode where data packets which would ordinarily be transferred by the bridge process are not transferred except in-band management packets are transferred to the in-band management process regardless of which network from which they arrived. Also disclosed, a packet switching machine having shared high-speed memory with multiple ports, one port coupled to a plurality of LAN controller chips coupled to individual LAN segments and an Ethernet microprocessor that sets up and manages a receive buffer for storing received packets and transferring pointers thereto to a main processor. The main processor is coupled to another port of the memory and analyzes received packets for bridging to other LAN segments or forwarding to an SNMP agent. The main microprocessor and the Ethernet processor coordinate to manage the utilization of storage locations in the shared memory. Another port is coupled to an uplink interface to higher speed backbone media such as FDDI, ATM etc. Speeds up to media rate are achieved by only moving pointers to packets around in memory as opposed to the data of the packets itself. A double password security feature is also implemented in some embodiments to prevent accidental or intentional tampering with system configuration settings.

6 Claims, 13 Drawing Sheets



US005822542A

United States Patent [19]

Smith et al.

[11] Patent Number: 5,822,542

[45] Date of Patent: Oct. 13, 1998

[54] ELECTRONIC AND STRUCTURAL COMPONENTS OF AN INTELLIGENT VIDEO INFORMATION MANAGEMENT APPARATUS

[75] Inventors: **Gordon W. Smith**, San Marcos; **Charles Park Wilson**, Santee; **David James Ousley**, San Diego; **Chris Harvey Pedersen, Jr.**, Santee; **Sherwin Sheng-shu Wang**; **David Ross MacCormack**, both of San Diego, all of Calif.

[73] Assignee: **Sensormatic Electronics Corporation**

[21] Appl. No.: **729,620**

[22] Filed: **Oct. 31, 1996**

[51] **Int. Cl.**[6] **H04N 1/413**

[52] **U.S. Cl.** **395/200.77**; 348/317; 348/700; 348/715

[58] **Field of Search** 395/200.09, 114, 395/894, 200.77; 382/236; 348/317, 700, 715; 360/97 01

[56] **References Cited**

U.S. PATENT DOCUMENTS

3,988,533	10/1976	Mick et al.	178/6.8
5,109,278	4/1992	Erickson et al.	358/108
5,202,759	4/1993	Laycock	358/108
5,493,329	2/1996	Ohguchi	348/17

OTHER PUBLICATIONS

Geutebrück, "MultiScop Video Disc Recorder," (brochure). No Date.

Robot (A Sensormatic Company), "Multivision Optima II," (brochure), 1995.

Robot Research, Inc. (A Sensormatic Company), *Multivision Optima II Multiplexers, Installation and Operation Manual*, 1995.

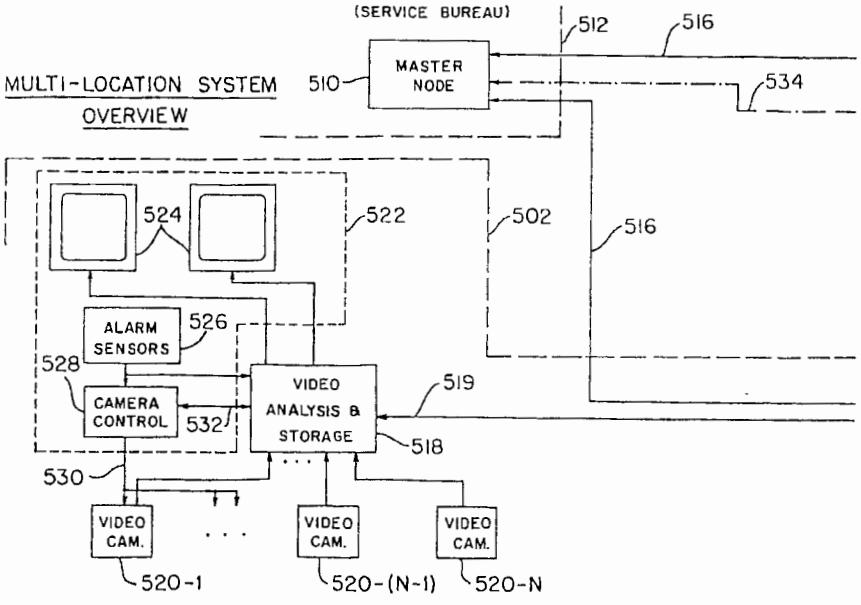
Primary Examiner—Emanuel Todd Voeltz
Assistant Examiner—Thomas Peeso
Attorney, Agent, or Firm—Robin, Blecker & Daley

[57] **ABSTRACT**

A structure for supporting a plurality of recording medium drive units includes a base member supporting a floppy disk drive and a DAT drive, an intermediate member supported on the base member and supporting two hard disk drives, and a top member supported on the intermediate member and supporting one or two hard disk drives.

The recording medium drive unit support structure is mounted within a housing, within which are also housed a motherboard, a second printed circuit board and a third printed circuit board. The motherboard has mounted thereon a microprocessor for controlling storage of video data on at least one of the hard disks. The second printed circuit board has integrated circuits mounted thereon for receiving plural streams of video information and for selecting for storage fields of video information included in the streams of video information. The third printed circuit board has mounted thereon a first digital signal processing integrated circuit (DSP-IC) for applying a data compression to the streams of video information, a second DSP-IC for controlling scaling and overlay mixing processes applied to the streams of video information, and a third DSP-IC for applying a moving image content analysis algorithm to the streams of video information. The first DSP-IC exchanges data with the microprocessor and transmits command messages to, and receives status messages from, the second and third DSP-IC's.

44 Claims, 158 Drawing Sheets



US006112238A

United States Patent [19]

Boyd et al.

[11] Patent Number: 6,112,238

[45] Date of Patent: Aug. 29, 2000

[54] SYSTEM AND METHOD FOR ANALYZING REMOTE TRAFFIC DATA IN A DISTRIBUTED COMPUTING ENVIRONMENT

[75] Inventors: **William Glen Boyd**; **Elijahu Shapira**, both of Portland, Oreg.

[73] Assignee: **Webtrends Corporation**, Portland, Oreg.

[21] Appl. No.: **08/801,707**

[22] Filed: **Feb. 14, 1997**

[51] **Int. Cl.**[7] **G06F 11/34**

[52] **U.S. Cl.** **709/224**; 709/218

[58] **Field of Search** 395/200.54, 200.53; 709/224, 223, 218

[56] **References Cited**

U S. PATENT DOCUMENTS

5,675,510	10/1997	Coffey et al.	709/224
5,689,416	11/1997	Shimizu et al.	364/185
5,727,129	3/1998	Barrett et al.	706/10
5,732,218	3/1998	Bland et al.	709/224
5,796,952	8/1998	Davis et al.	709/224
5,878,223	3/1999	Becker et al.	709/223

OTHER PUBLICATIONS

WebTrends™ Essential Reporting for your Web Server, Installation and User Guide, Jan. 1996 Edition, by e.g Software, Inc., 62 page manual.

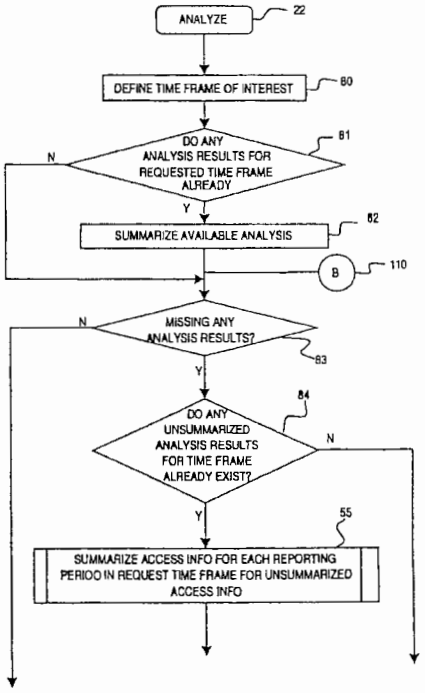
Primary Examiner—Kenneth Coulter

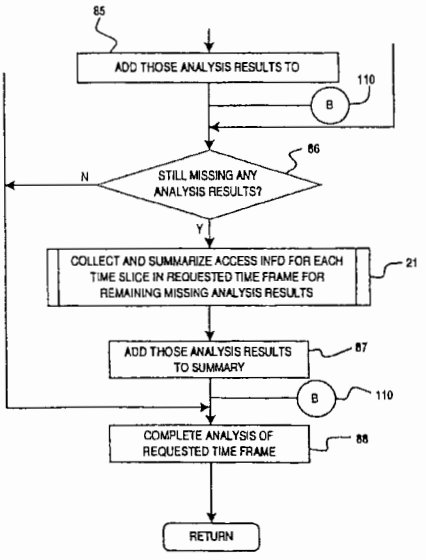
Attorney, Agent, or Firm—Marger Johnson & McCollom P.C

[57] **ABSTRACT**

A system, method and storage medium embodying computer-readable code for analyzing traffic data in a distributed computing environment are described. The distributed computing environment includes a plurality of interconnected systems operatively coupled to a server, a source of traffic data hits and one or more results tables categorized by an associated data type. Each results table includes a plurality of records. The server is configured to exchange data packets with each interconnected system. Each traffic data hit corresponds to a data packet exchanged between the server and one such interconnected system. Each traffic data hit is collected from the traffic data hits source as access information into one such record in at least one results table according to the data type associated with the one such results table. Each of the records in the results table corresponds to a different type of access information for the data type associated with the results table. The access information collected into the results tables during a time slice is summarized periodically into analysis results. The time slice corresponds to a discrete reporting period. The access information is analyzed from the results tables in the analysis results to form analysis summaries according to the data types associated with the results tables.

20 Claims, 12 Drawing Sheets





US006262983B1

(12) **United States Patent**
Yoshizawa et al.

(10) **Patent No.: US 6,262,983 B1**
(45) **Date of Patent: Jul. 17, 2001**

(54) **PROGRAMMABLE NETWORK**

(75) Inventors: **Satoshi Yoshizawa**, Saratoga, CA (US); **Toshiaki Suzuki**, Kokubunji (JP); **Mitsuru Ikezawa**, Asaka (JP); **Itaru Mimura**, Sayama (JP); **Tatsuya Kameyama**, Hachioji (JP)

(73) Assignee: **Hitachi, LTD**, Tokyo (JP)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/391,404**

(22) Filed: **Sep. 8, 1999**

(30) **Foreign Application Priority Data**

Sep. 8, 1998 (JP) 10-254228

(51) **Int. Cl.**[7] **H04L 12/28**
(52) **U.S. Cl.** **370/389**
(58) **Field of Search** 370/351, 389, 370/390, 392, 394, 428

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,473,609 * 12/1995 Chaney 370/312
5,563,648 * 10/1996 Menand et al. 348/13
5,619,501 * 4/1997 Tamer et al. 370/392
5,666,293 * 9/1997 Metz et al. 709/220
6,172,990 * 1/2001 Deb et al. 370/474
6,185,568 * 2/2001 Douceur et al. 707/10

* cited by examiner

Primary Examiner—David R. Vincent
(74) *Attorney, Agent, or Firm*—Antonelli, Terry, Stout & Kraus, LLP

(57) **ABSTRACT**

At each network node, a packet classification unit makes a judgement to transfer only a packet necessary to be processed by software to a packet processing program processor and transfer other packets directly to a routing processor. Processing history information indicating the process history executed at each network node on a network route is transferred to the other network nodes so that other network nodes can store the processing history information in respective processing history repository table. Each node refers to this table and further transfers only the packet necessary to be processed by software to the packet processing processor.

10 Claims, 12 Drawing Sheets

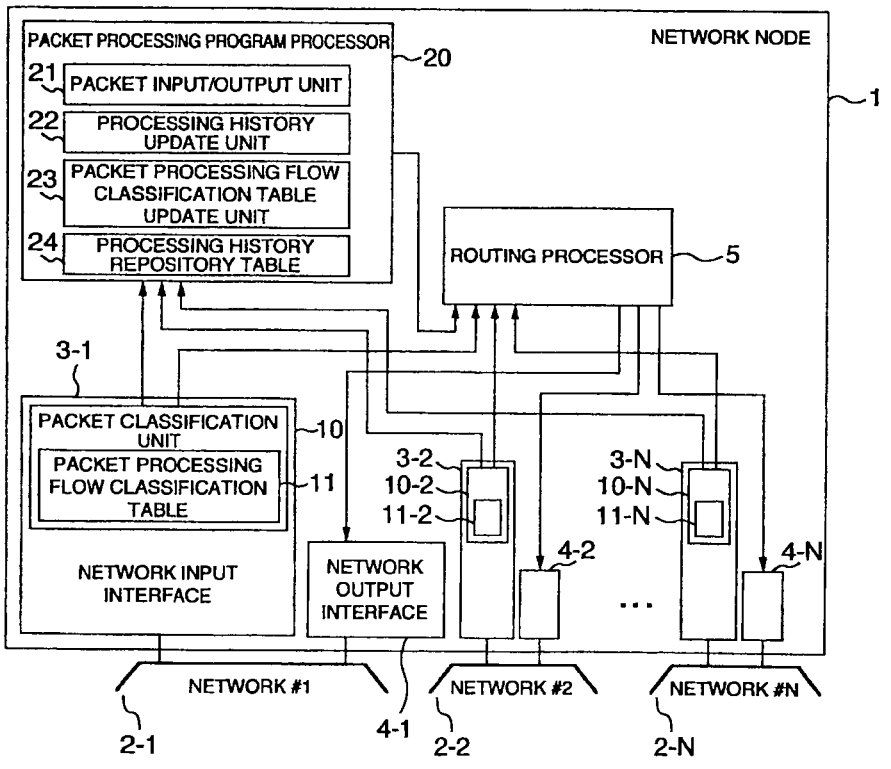


FIG.1

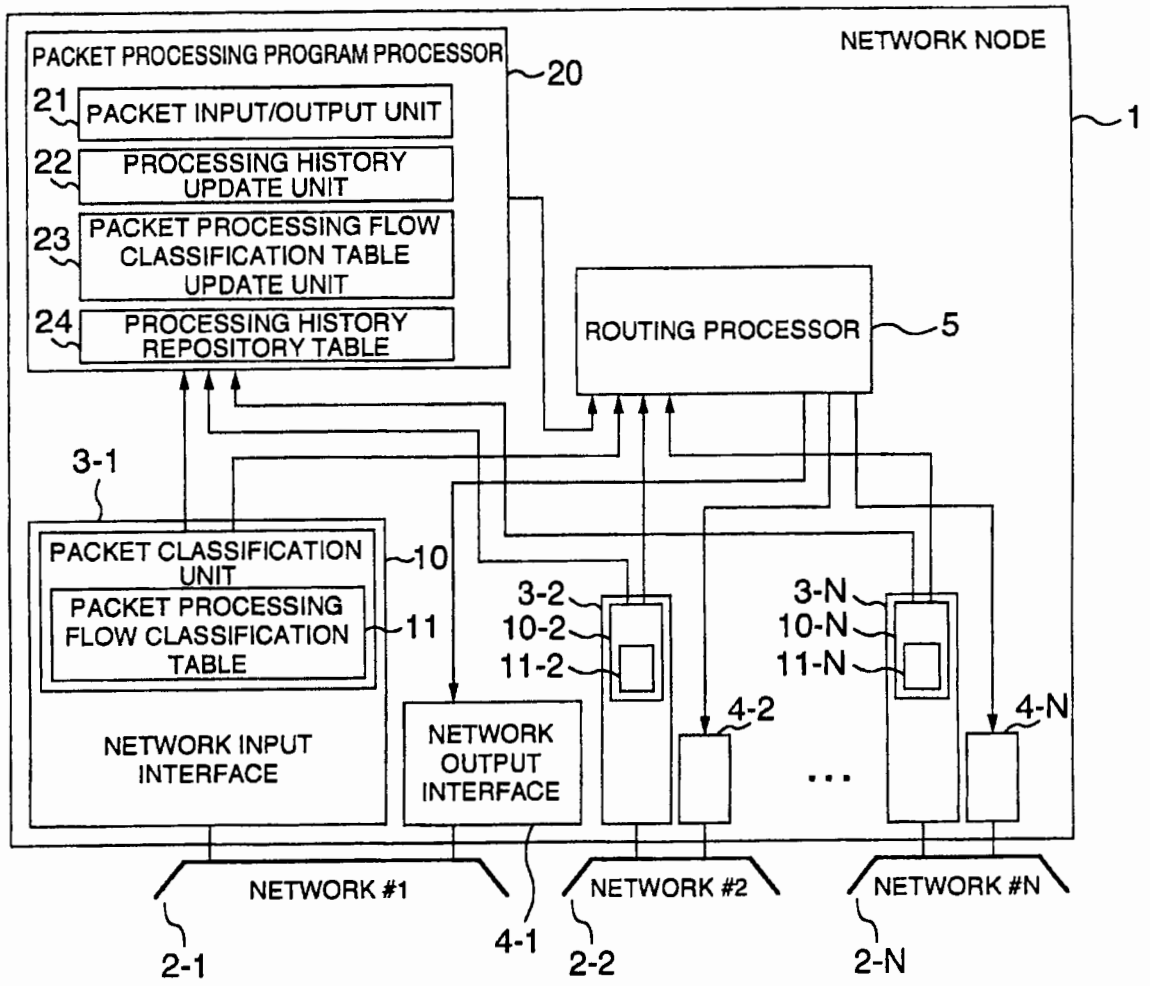

NETWORK NODE
1
PACKET PROCESSING PROGRAM PROCESSOR
20
21
PACKET INPUT/OUTPUT UNIT
22
PROCESSING HISTORY UPDATE UNIT
23
PACKET PROCESSING FLOW CLASSIFICATION TABLE UPDATE UNIT
24
PROCESSING HISTORY REPOSITORY TABLE
ROUTING PROCESSOR
5
3-1
PACKET CLASSIFICATION UNIT
10
PACKET PROCESSING FLOW CLASSIFICATION TABLE
11
NETWORK INPUT INTERFACE
NETWORK OUTPUT INTERFACE
3-2
10-2
11-2
4-2
3-N
10-N
11-N
4-N
NETWORK #1
NETWORK #2
NETWORK #N
2-1
4-1
2-2
2-N

FIG.2

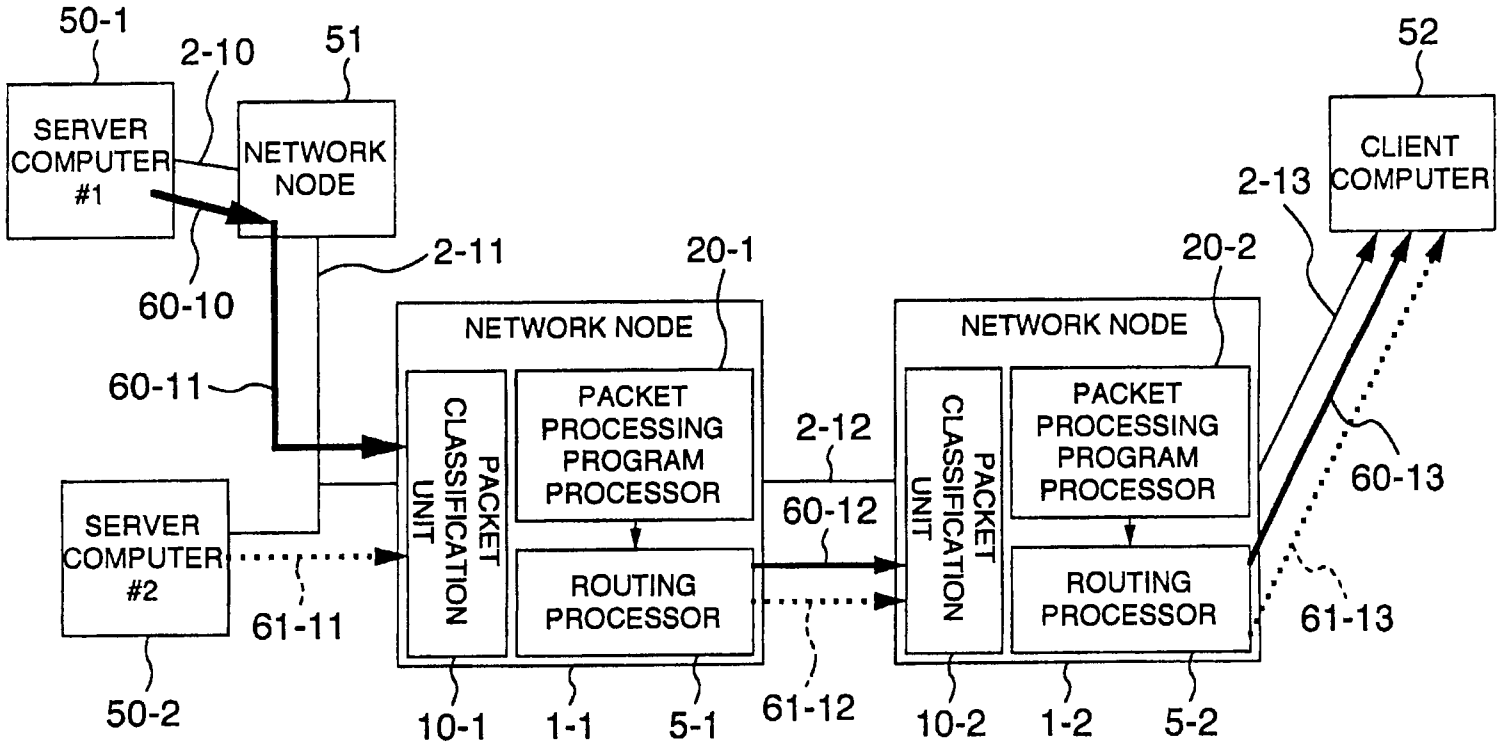

50-1
2-10
51
52
SERVER COMPUTER #1
NETWORK NODE
CLIENT COMPUTER
2-13
2-11
20-1
20-2
60-10
60-11
NETWORK NODE
NETWORK NODE
PACKET CLASSIFICATION UNIT
PACKET PROCESSING PROGRAM PROCESSOR
2-12
PACKET CLASSIFICATION UNIT
PACKET PROCESSING PROGRAM PROCESSOR
60-13
60-12
SERVER COMPUTER #2
ROUTING PROCESSOR
ROUTING PROCESSOR
61-13
61-11
50-2
10-1
1-1
5-1
61-12
10-2
1-2
5-2

FIG.6

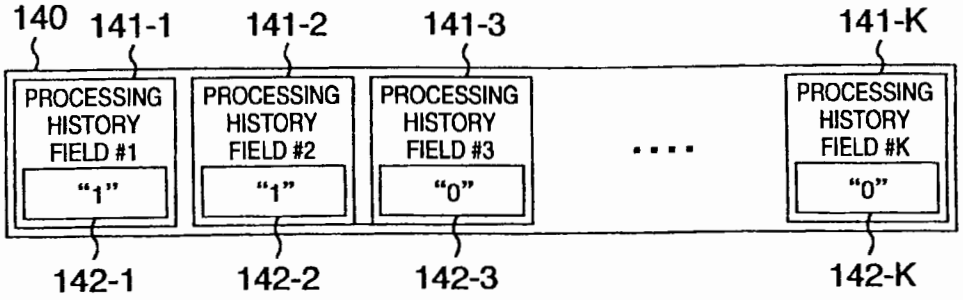

140
141-1
141-2
141-3
141-K
PROCESSING HISTORY FIELD #1
"1"
PROCESSING HISTORY FIELD #2
"1"
PROCESSING HISTORY FIELD #3
"0"
PROCESSING HISTORY FIELD #K
"0"
142-1
142-2
142-3
142-K

FIG.7

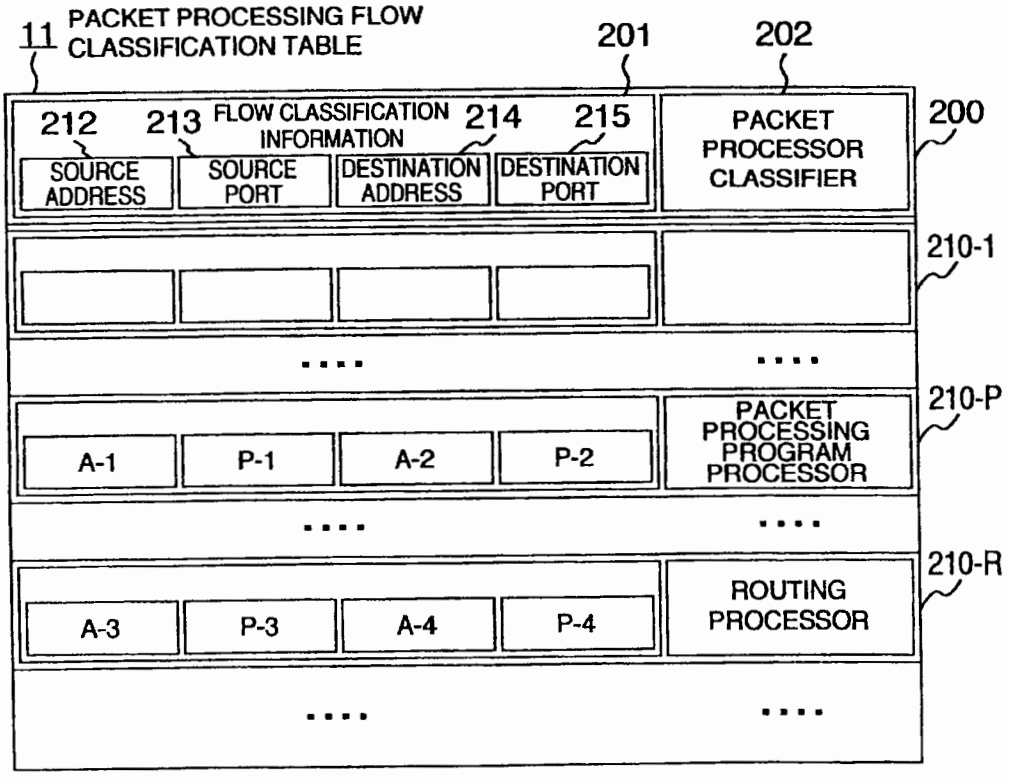

11
PACKET PROCESSING FLOW CLASSIFICATION TABLE
201
202
212
213
FLOW CLASSIFICATION INFORMATION
214
215
SOURCE ADDRESS
SOURCE PORT
DESTINATION ADDRESS
DESTINATION PORT
PACKET PROCESSOR CLASSIFIER
200
210-1
A-1
P-1
A-2
P-2
PACKET PROCESSING PROGRAM PROCESSOR
210-P
A-3
P-3
A-4
P-4
ROUTING PROCESSOR
210-R

FIG.8

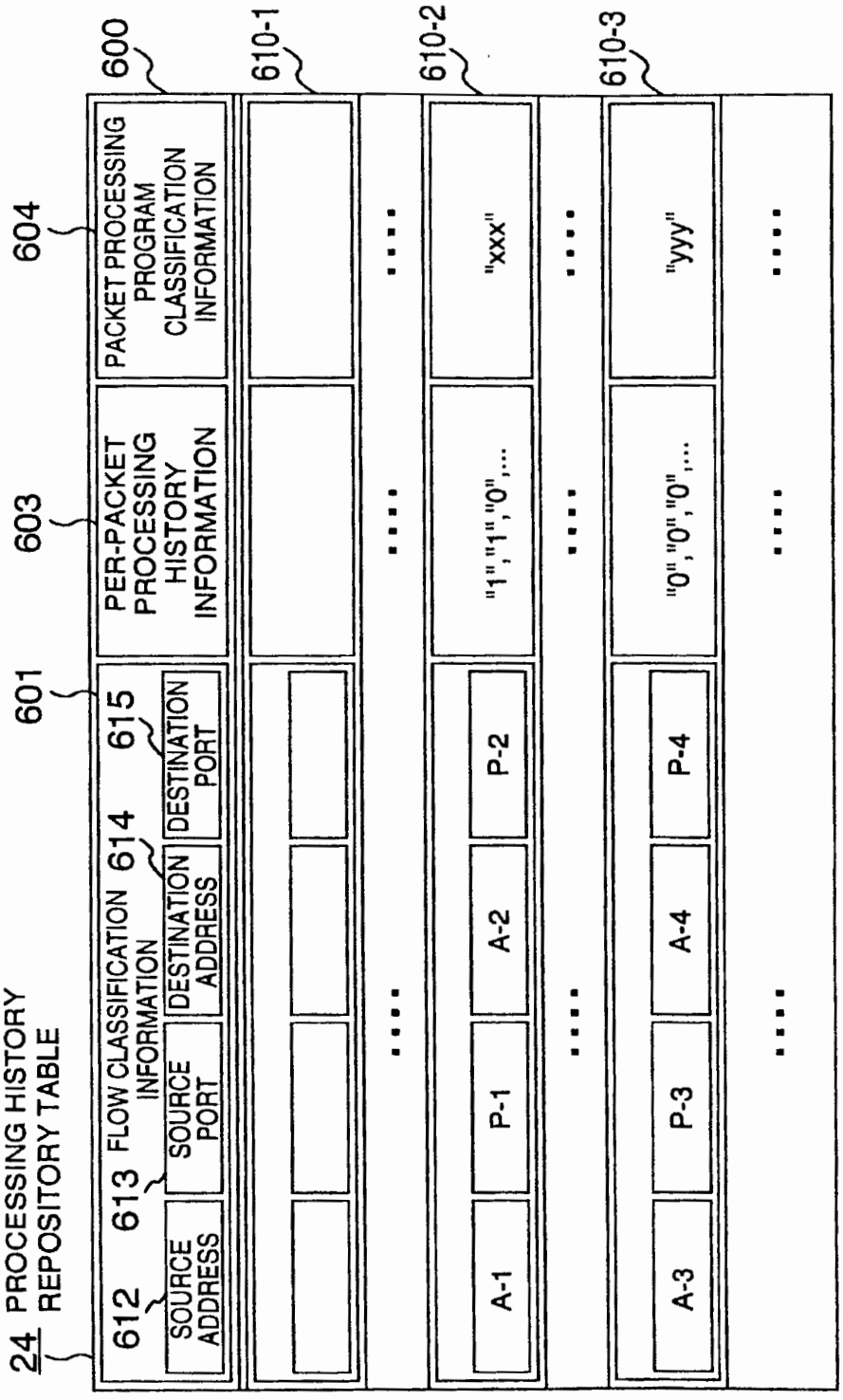

24 PROCESSING HISTORY REPOSITORY TABLE
601
612
613
FLOW CLASSIFICATION INFORMATION
614
615
SOURCE ADDRESS
SOURCE PORT
DESTINATION ADDRESS
DESTINATION PORT
603
PER-PACKET PROCESSING HISTORY INFORMATION
604
PACKET PROCESSING PROGRAM CLASSIFICATION INFORMATION
600
610-1
610-2
610-3
A-1
P-1
A-2
P-2
"1","1","0",...
"xxx"
A-3
P-3
A-4
P-4
"0","0","0",...
"yyy"

FIG.9

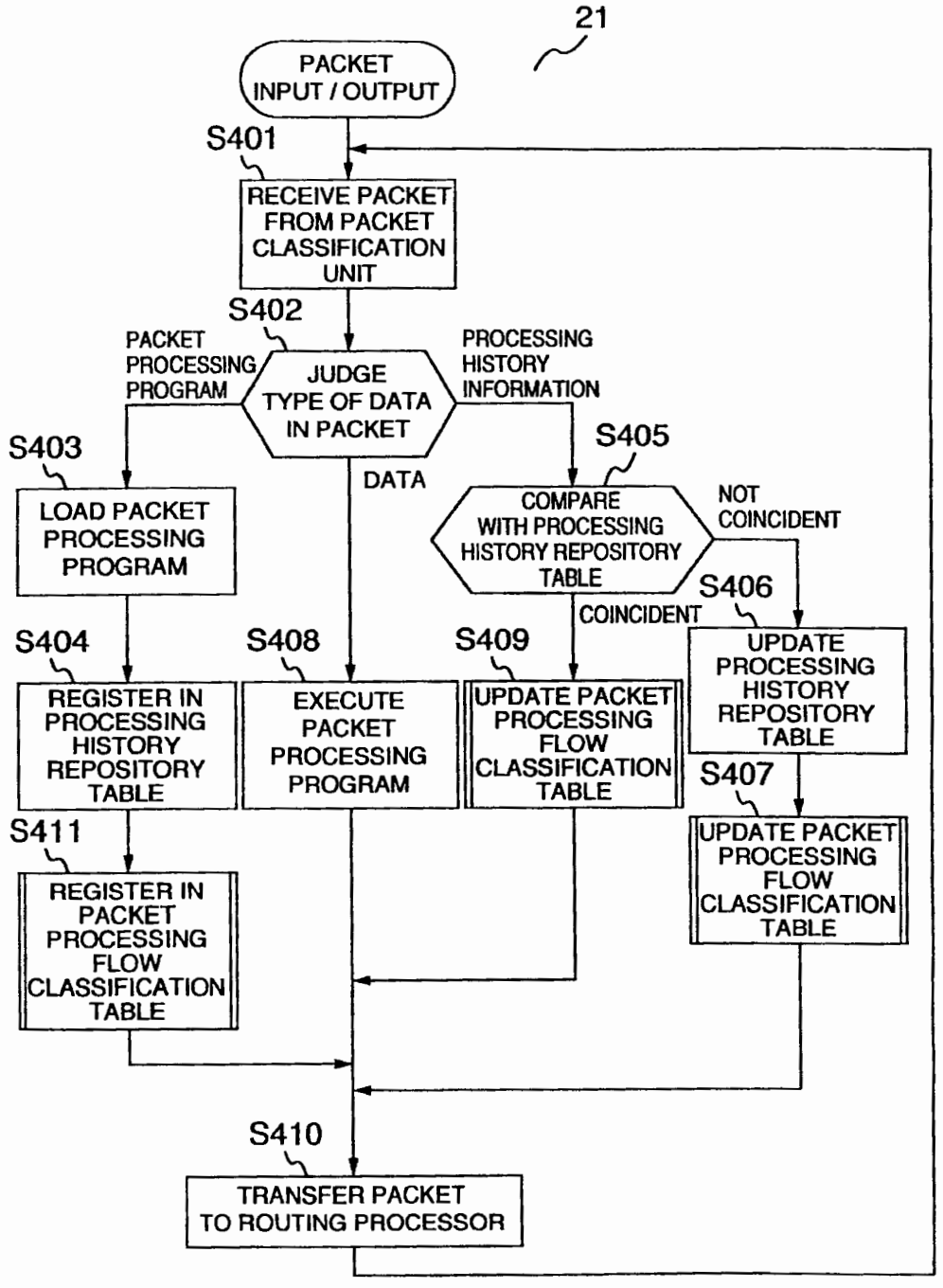

21
PACKET INPUT / OUTPUT
S401
RECEIVE PACKET FROM PACKET CLASSIFICATION UNIT
S402
JUDGE TYPE OF DATA IN PACKET
PACKET PROCESSING PROGRAM
PROCESSING HISTORY INFORMATION
DATA
S403
LOAD PACKET PROCESSING PROGRAM
S404
REGISTER IN PROCESSING HISTORY REPOSITORY TABLE
S405
COMPARE WITH PROCESSING HISTORY REPOSITORY TABLE
NOT COINCIDENT
COINCIDENT
S406
UPDATE PROCESSING HISTORY REPOSITORY TABLE
S407
UPDATE PACKET PROCESSING FLOW CLASSIFICATION TABLE
S408
EXECUTE PACKET PROCESSING PROGRAM
S409
UPDATE PACKET PROCESSING FLOW CLASSIFICATION TABLE
S410
TRANSFER PACKET TO ROUTING PROCESSOR
S411
REGISTER IN PACKET PROCESSING FLOW CLASSIFICATION TABLE

FIG.10

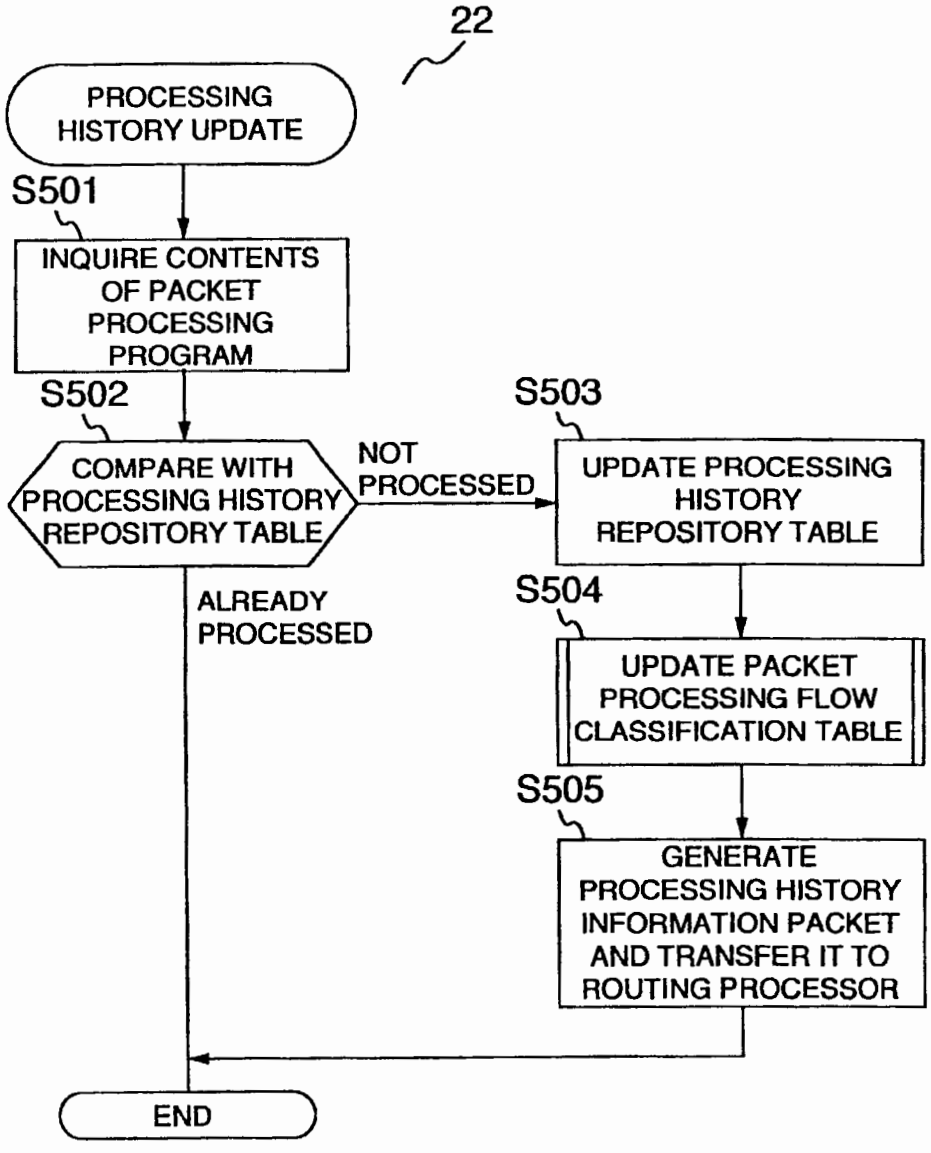

22
PROCESSING HISTORY UPDATE
S501
INQUIRE CONTENTS OF PACKET PROCESSING PROGRAM
S502
COMPARE WITH PROCESSING HISTORY REPOSITORY TABLE
NOT PROCESSED
S503
UPDATE PROCESSING HISTORY REPOSITORY TABLE
ALREADY PROCESSED
S504
UPDATE PACKET PROCESSING FLOW CLASSIFICATION TABLE
S505
GENERATE PROCESSING HISTORY INFORMATION PACKET AND TRANSFER IT TO ROUTING PROCESSOR
END

FIG.11

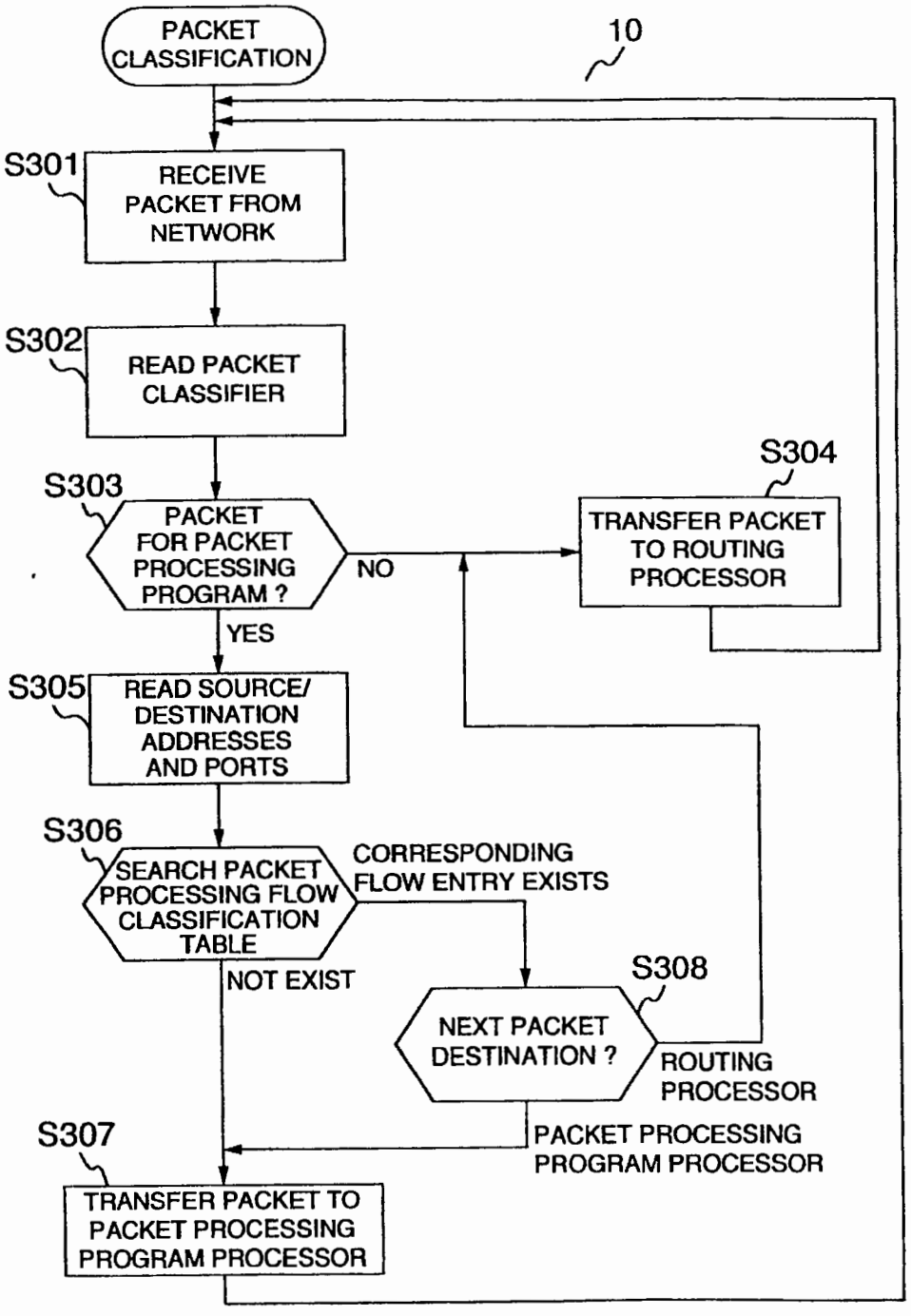

PACKET CLASSIFICATION
10
S301
RECEIVE PACKET FROM NETWORK
S302
READ PACKET CLASSIFIER
S303
PACKET FOR PACKET PROCESSING PROGRAM ?
NO
S304
TRANSFER PACKET TO ROUTING PROCESSOR
YES
S305
READ SOURCE/ DESTINATION ADDRESSES AND PORTS
S306
SEARCH PACKET PROCESSING FLOW CLASSIFICATION TABLE
CORRESPONDING FLOW ENTRY EXISTS
NOT EXIST
S308
NEXT PACKET DESTINATION ?
ROUTING PROCESSOR
PACKET PROCESSING PROGRAM PROCESSOR
S307
TRANSFER PACKET TO PACKET PROCESSING PROGRAM PROCESSOR

FIG.12

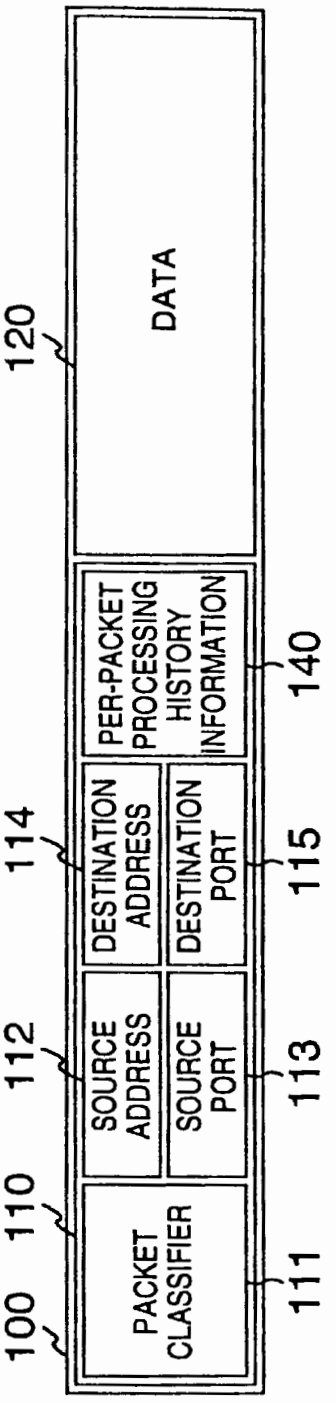

100
110
112
114
120
PACKET CLASSIFIER
SOURCE ADDRESS
DESTINATION ADDRESS
PER-PACKET PROCESSING HISTORY INFORMATION
DATA
111
113
115
140
SOURCE PORT
DESTINATION PORT

FIG.13

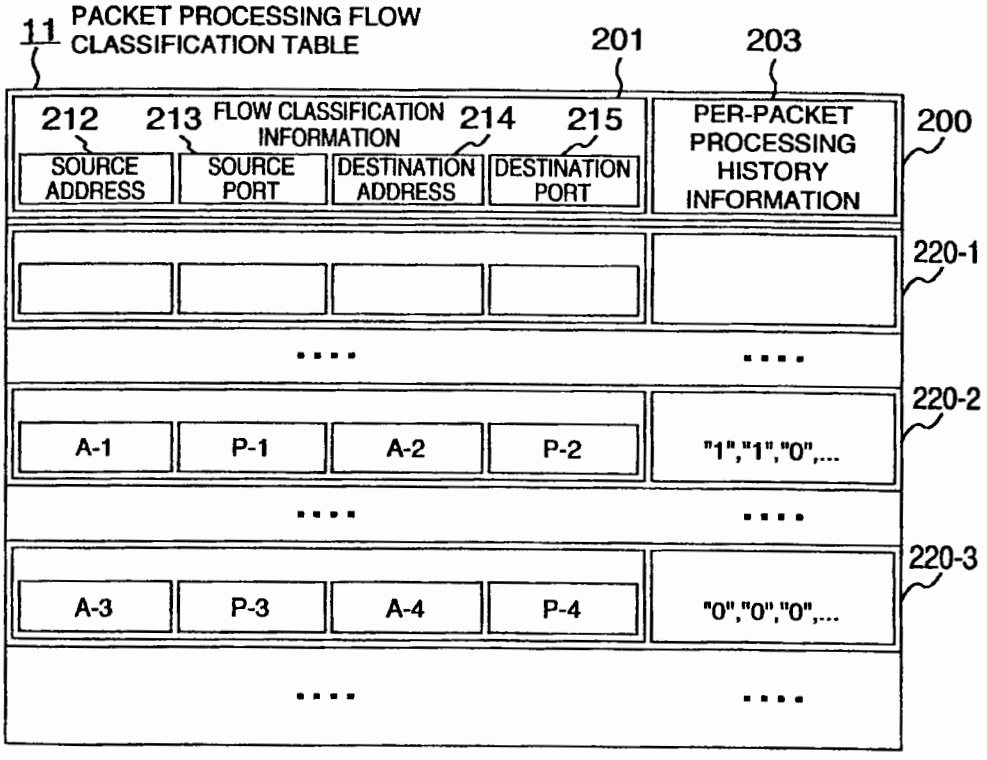

11
PACKET PROCESSING FLOW CLASSIFICATION TABLE
201
203
212
213
FLOW CLASSIFICATION INFORMATION
214
215
SOURCE ADDRESS
SOURCE PORT
DESTINATION ADDRESS
DESTINATION PORT
PER-PACKET PROCESSING HISTORY INFORMATION
200
220-1
220-2
A-1
P-1
A-2
P-2
"1","1","0",...
220-3
A-3
P-3
A-4
P-4
"0","0","0",...

FIG.14

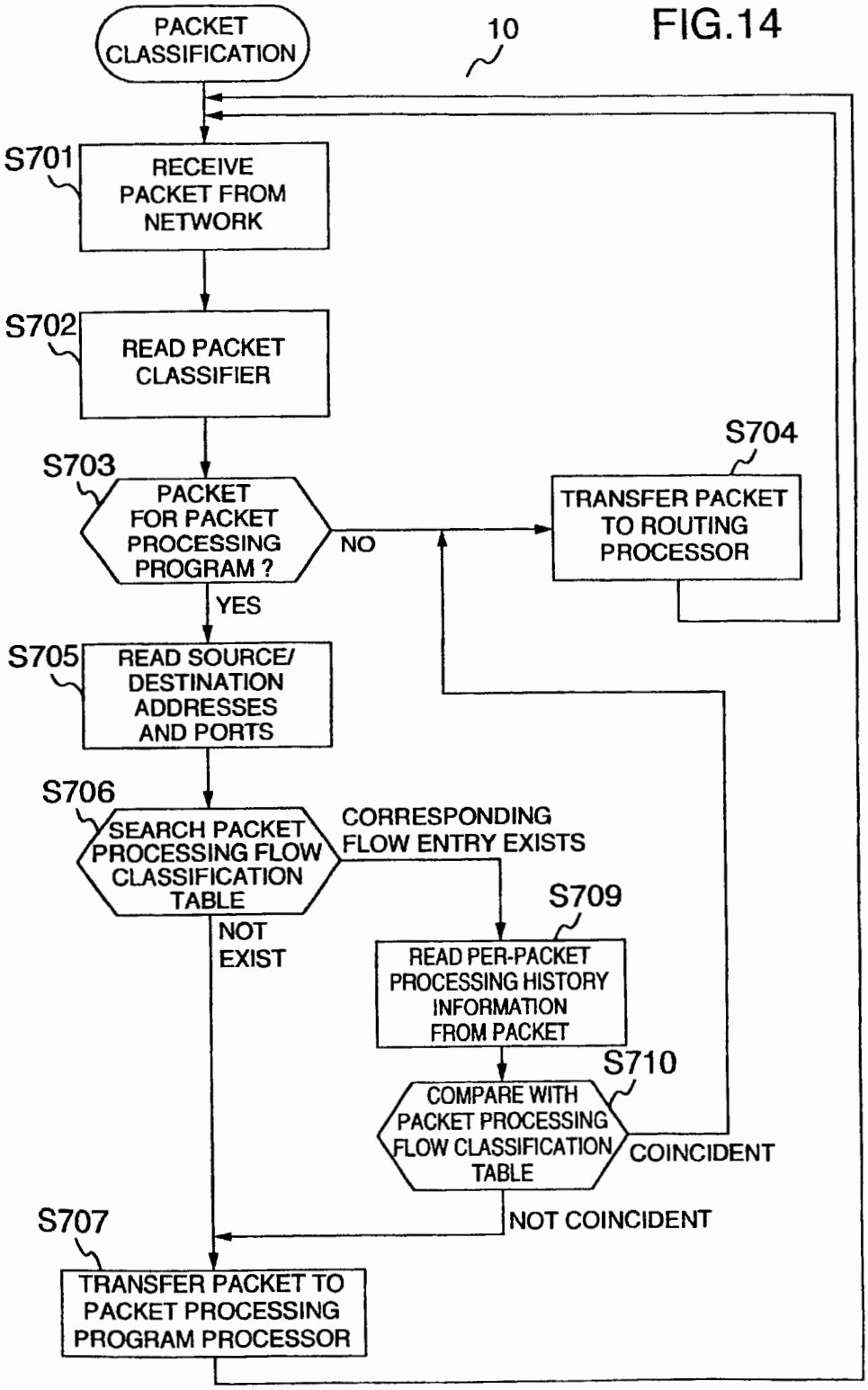
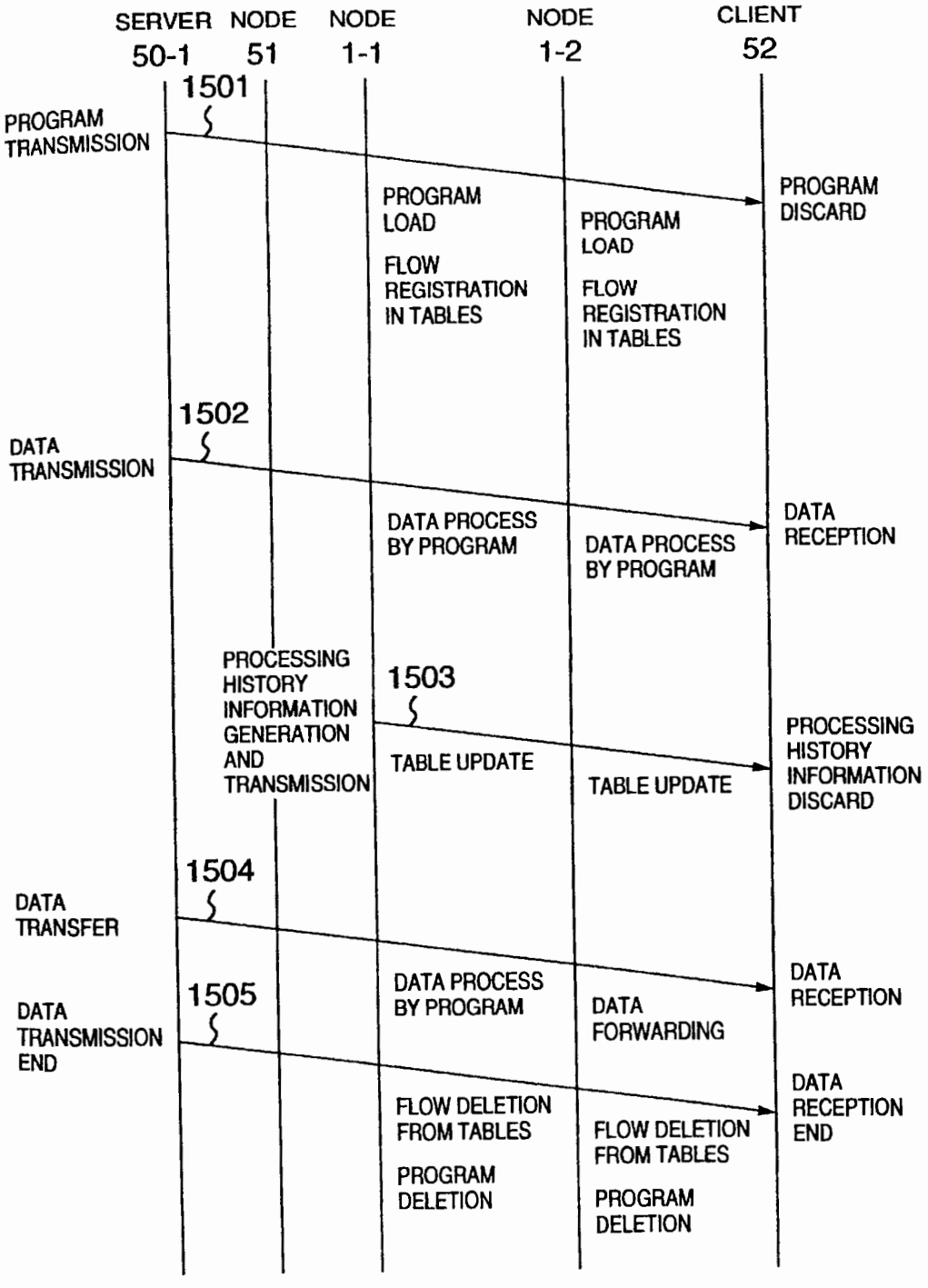

PACKET CLASSIFICATION
10
S701
RECEIVE PACKET FROM NETWORK
S702
READ PACKET CLASSIFIER
S703
PACKET FOR PACKET PROCESSING PROGRAM ?
NO
S704
TRANSFER PACKET TO ROUTING PROCESSOR
YES
S705
READ SOURCE/ DESTINATION ADDRESSES AND PORTS
S706
SEARCH PACKET PROCESSING FLOW CLASSIFICATION TABLE
CORRESPONDING FLOW ENTRY EXISTS
NOT EXIST
S709
READ PER-PACKET PROCESSING HISTORY INFORMATION FROM PACKET
S710
COMPARE WITH PACKET PROCESSING FLOW CLASSIFICATION TABLE
COINCIDENT
NOT COINCIDENT
S707
TRANSFER PACKET TO PACKET PROCESSING PROGRAM PROCESSOR

FIG.15


SERVER 50-1
NODE 51
NODE 1-1
NODE 1-2
CLIENT 52
1501
PROGRAM TRANSMISSION
PROGRAM LOAD
FLOW REGISTRATION IN TABLES
PROGRAM LOAD
FLOW REGISTRATION IN TABLES
PROGRAM DISCARD
1502
DATA TRANSMISSION
DATA PROCESS BY PROGRAM
DATA PROCESS BY PROGRAM
DATA RECEPTION
PROCESSING HISTORY INFORMATION GENERATION AND TRANSMISSION
1503
TABLE UPDATE
TABLE UPDATE
PROCESSING HISTORY INFORMATION DISCARD
1504
DATA TRANSFER
DATA PROCESS BY PROGRAM
DATA FORWARDING
DATA RECEPTION
1505
DATA TRANSMISSION END
FLOW DELETION FROM TABLES
PROGRAM DELETION
FLOW DELETION FROM TABLES
PROGRAM DELETION
DATA RECEPTION END

PROGRAMMABLE NETWORK

BACKGROUND OF THE INVENTION

The present invention relates to a programmable network in which in a network system interconnecting a plurality of computers via a network, a program is loaded to a network node on a route of a flow constituted of packets, and the node executes the program for each packet.

For a network system in which data is processed and then transmitted, a method is known by which a video is compressed and packetized for each wavelet band, as disclosed in JP-A-7-15609. The invention provides a video transfer method, a video transmitter, a video receiver and an video transfer apparatus, in which a video is transferred in accordance with a network bandwidth on a video reception side and a decoding capability of the decoder, even if which capabilities are inferior to a network bandwidth on a video transmission side and an encoding capability of the encoder. The video transmitter packetizes video data for each wavelet band, adds a predetermined classifier to the packet, and then transmits it. If video data transmitted by ATM (Asynchronous Transfer Mode) network, a priority order is added to a cell header to transmit a cell. On the network which transfers video data, a cell having a higher priority order is transferred with a priority over other cells when the network is congested. The video receiver checks the classifier of the received video data, selects only necessary video data and decodes it in accordance with the decoding capability of the decoder to reproduce the video data.

A programmable network is known as described in "A Survey of Active Network Research" in "IEEE Communications Magazine", January issue of 1997, at pp. 80–86. In the programmable network, each network node constituting the network executes a packet processing program for each packet. For example, the above-described transfer with a priority order by ATM is realized by software using a program loaded at each network node. This system can be realized on the network having a QoS (Quality of Service) control function of ATM, e.g., on an IP (Internet Protocol) network such as the Internet.

For the IP network, techniques are also known by which a routing process for controlling a route of packets via network nodes can be speeded up by using a dedicated processor.

In the programmable network system according to the conventional techniques described above, all packets are processed by software using a program loaded at each network node. Therefore, although packets to be processed by software and packets not to be processed are both input to a network node, software processing is performed for both types of the packets so that a process efficiency is not good.

SUMMARY OF THE INVENTION

It is an object of the present invention to provide a programmable network for processing a packet by a packet processing program at each program node. Only the packet necessary to be processed is processed and the packet unnecessary to be processed is transferred to a routing processor, so that a transmission speed can be improved and the throughput of the programmable network can be improved.

It is another object of the present invention to provide a method of loading a program in associated program nodes of a network dynamically and efficiently.

In order to achieve the above object, a network node constituting a network comprises: a program processor for executing a packet processing program to each packet corresponding to a flow; a routing processor for performing a routing process for an input packet; and a packet classification unit for analyzing the input packet to transfer a packet belonging to the flow to the program processor and transfer other packets to the routing processor. The network node has a function of receiving the packet processing program via the network and making it executable.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a block diagram showing the structure of a network node of a programmable network system according to this invention.

FIG. **2** is a block diagram explaining the structure and operation of the programmable network system according to the invention.

FIG. **3** is a diagram showing the format of a packet **100** according to a first embodiment.

FIG. **4** is a diagram showing the format of the packet **100** shown in FIG. **3** whose packet payload **120** is a packet processing program.

FIG. **5** is a diagram showing the format of the packet **100** shown in FIG. **3** whose packet payload **120** is packet history information **140**.

FIG. **6** is a diagram showing the format of the packet history information according to an embodiment.

FIG. **7** is a diagram showing the format of a packet processing flow classification table **11** according to a first embodiment.

FIG. **8** is a diagram showing the format of a processing history repository table.

FIG. **9** is a flow chart illustrating the operation to be executed by a packet input/output unit **21** of a packet program processor **20** according to the first embodiment.

FIG. **10** is a flow chart illustrating the operation to be executed by a processing history update unit **22** of a packet program processor **20** according to the first embodiment.

FIG. **11** is a flow chart illustrating the operation to be executed by a packet classification unit **10** according to the first embodiment.

FIG. **12** is a diagram showing the format of the packet **100** according to a second embodiment.

FIG. **13** is a diagram showing the format of a packet processing classification table **11** according to the second embodiment.

FIG. **14** is a flow chart illustrating the operation to be executed by the packet classification unit **10** according to the second embodiment.

FIG. **15** is a sequence diagram illustrating a data transmission procedure according to this invention.

DETAILED DESCRIPTION OF THE EMBODIMENTS

(1) Structure and Outline of System

FIG. **2** is a block diagram explaining an example of the structure and operation of a programmable network system.

As shown in FIG. **2**, the programmable network system of this embodiment is constituted of networks **2**, server computers **50**, network nodes **1**, **51**, and client computers **52**.

The network may be the Internet, LAN or the like. The network node is a network apparatus such as a router and a gateway. The numbers of server computers, client computers and network nodes are not limited to this embodiment.

If there are a plurality of network nodes **1**, they are represented by **1-1**, **1-2**, . . . with branch numbers. The same notation is also applied to the server computer **50** and network **2**.

It is assumed that a server computer **50-1** transmits a flow **60** representative of a series of data and a server computer **50-2** transmits a flow **61**, respectively to the networks **2**.

A plurality of network nodes are provided in order to interconnect networks and it is assumed that conventional network nodes **51** and programmable network nodes **1-1**, **1-2** are used in a mixed state.

The programmable network node has a packet classification unit **10**, a packet processing program processor **20** and a routing processor **5**.

The server computer **50-1** transmits in advance a packet processing program to the packet processing program processor **20** at the network node **1-1**, **1-2** to make the packet processing program executable so as to execute a software process for a data packet to be transmitted.

It is assumed herein that the packet processing program processes a data packet belonging to the flow **60** and does not process a data packet belonging to the flow **61**.

At the network node **1-1**, **1-2**, the packet classification unit **10** controls a next destination of each data packet belonging to the flow **60** from the server computer **50-1** to the client computer **52**, in order to allow each packet processing program processor **20** to execute the packet processing program (refer to bold lines **60-10** to **60-13** in FIG. **2**).

The packet belonging to the flow **61** from the server computer **50-2** to the client computer **52** is not necessary to be processed by the packet processing program. Therefore, at the network node **1-1**, **1-2**, the packet classification unit **10** controls a next destination of each data packet so that the data packet is processed directly by the routing processor **5** without being transferred to the packet processing program processor **20** (refer to broken lines **61-11** to **61-13** in FIG. **2**).

(2) Programmable Network Node: First Embodiment

Next, of the network nodes described above, the network node of the type having the packet processing program processor **20** will be detailed as to its structure.

As shown in FIG. **1**, the network node **1** of the embodiment is constituted of a network input interface **3**, a network output interface **4**, a routing processor **5** and a packet processing program processor **20**. The network input interfaces **3** (**3**-i, i=1, . . . , N) and network output interfaces **4** (**4**-i, i=1, . . . , N) are connected to N networks **2** (**2**-i, i=1, . . . , N) at their inputs and outputs.

Each network input interface **3** is provided with a packet classification unit **10** which has a packet processing flow classification table **11**. The packet processing flow classification table **11** stores information for judging whether a packet belonging to each flow is to be transferred to the packet processing program processor **20** or directly to the routing processor **5**.

The packet processing program processor **20** is made of a CPU and a memory and includes a packet input/output unit **21**, a processing history update unit **22**, a packet processing flow classification update unit **23**, and a processing history repository table **24**. The packet processing flow classification update unit **23** has a function of setting information to the packet processing flow classification table **11**. The units **21**, **22** and **23** are realized by software which is executed by the CPU. The packet processing program is stored in the memory and executed when necessary.

The routing processor **5** controls a route of each packet and is a processor dedicated to a routing process.

This embodiment will be described by taking as an example that the server computer **50** transmits hierarchically encoded video data **60-10**. The server computer **50** transmits compressed data by using a packet different for each frequency band. The programmable network node discards a packet containing high frequency band data in response to a request, and transfers the other packets to the next network node or client computer **52**. Prior to data transmission, the server computer **50** transmits to the network nodes, as the packet processing program, a software program for checking the frequency band of a data packet and judging whether the data packet is discarded or transferred. The packet processing program has a function of determining the frequency band of a data packet to be discarded at the network node and storing this information, in response to a request from the server computer **50** or client computer **52** and/or in response to a check result of the status of networks to be connected. It is assumed herein that the packet processing program has a function of determining a discard packet frequency band and notifying it to an external software program, in response to a request from the external software program.

The format of a packet to be received at the programmable network node is shown in FIG. **3**.

As shown in FIG. **3**, the packet **100** is constituted of: a packet header **110** for storing destination information and the like of the packet; and a packet data field **120** for storing data. The packet header **110** is constituted of a packet classifier **111** for classifying a protocol type for the packet, a source address **112** and a source port **113** for identifying a source computer, and a destination address **114** and a destination port **115** for identifying a destination computer. For example, in the case of a widely used IP (Internet Protocol), the source address **112** and destination address **114** correspond to IP addresses, and the source port **113** and destination port **115** correspond to port numbers of TCP/UDP (Transmission Control Protocol)/(User Datagram Protocol). A flow can be discriminated by using these source/destination addresses and ports.

As shown in FIG. **4**, there is a packet **100** whose data **120** is a packet processing program **130**. FIG. **4** shows the format of a packet which is used when the server computer **50** transmits the packet processing program to the network node **1-1**, **1-2** as shown in FIG. **2**. The packet processing program **130** may be any type such as an object module executable at the network node **1**, a script written by text data, byte codes written by Java language, or the like. If the packet processing program **130** has a length unable to be written in one packet **100**, it is possible to transmit it by dividing to a plurality of packets **100**. The packet shown in FIG. **4** may include a program module name of the packet processing program **130**.

There is another example of a packet **100** whose data **120** corresponds to processing history information. This format of a packet is used when the network node **1-1**, **1-2** transmits the processing history information as shown in FIG. **2**. Also in this case, if the processing history information **140** has a length unable to be written in one packet **100**, it is possible to transmit it by dividing to a plurality of packets **100**.

For example, the processing history information **140** may have the format such as shown in FIG. **6**. In this example, the processing history information **140** is constituted of: a bit

field (processing history field) **141** for storing "1" or "0" indicating a processing execution history of each frequency band; and its bit information **142**. If the hierarchically encoded video data described earlier is to be transmitted, the frequency band is partitioned into K stages in such a manner that a transfer of a packet storing the lowest frequency band is allocated to **141-1** and a transfer of a packet storing the highest frequency band is allocated to **141-K**. For example, an initial value "1" is written in each field **142**, and "0" is written in the field **142** corresponding to the packet at the frequency band discarded by the packet processing program processor **20**. Although the processing history information **140** shown in FIG. **6** is realized by using the bit field, it may be realized by a script written by text data or any other type.

An example of the format of the packet processing flow classification table **11** is shown in FIG. **7**. The packet processing flow classification table **11** is constituted of a flow classification information field **201** and a packet processor classifier field **202**.

The flow classification information field **201** is constituted of a source address **212**, a source port **213**, a destination address **214** and a destination port **215**. These addresses and ports **212** to **215** correspond to the addresses and ports **112** to **115** in the packet header **110** shown in FIG. **3**. The packet processor classifier field **202** stores a classifier indicating whether the packet **100** is transferred to the packet processing program processor **20** or to the routing processor **5**.

The packet processing flow classification table **11** is constituted of a plurality of entries **210**. An entry **210-P** indicates that the packet constituting the flow represented by the addresses and ports **212** to **215** is transferred to the packet processing program processor **20**, whereas an entry **210-R** indicates that the packet is transferred to the routing processor **5**. These entries **210** are set by the packet processing flow classification table update unit **23** of the packet processing program processor **20**, and referred by the packet classification unit **10** of the network input interface **3**.

An example of the format of the processing history repository table **24** is shown in FIG. **8**. In this example, the processing history repository table **24** is constituted of a flow classification information field **601**, a per-packet processing history information field **603** and a packet processing program classification information field **604**. The flow classification information field **610** is constituted of a source address **612**, a source port **613**, a destination address **614** and a destination port **615**. The addresses and ports **612** to **615** correspond to the addresses and ports **112** to **115** shown in FIG. **3**.

The per-packet processing history information **603** stores the processing history information **140** shown in FIG. **6**. The timing when the per-packet processing history information **603** is updated is either the timing when the packet **100** whose data is the processing history information **140** shown in FIG. **5** is transmitted to the network node, or the timing when the processing history update unit **22** updates the per-packet processing history **603** in accordance with the process contents of the packet processing program to be executed at this network node.

The packet processing program classification information **604** stores information of the packet processing program which processes the packet **100**, e.g., a program name. The timing when this packet processing program classification information **604** is updated is the timing when the packet **100** whose data is the packet processing program **130** shown in FIG. **3** reaches this network node.

The processing history repository table **24** is constituted of a plurality of entries **610**. An entry **610-2** indicates that the packet constituting the flow represented by the addresses and ports **612** to **615** is processed by a packet processing program "xxx", and an entry **610-3** indicates that it is precessed by a packet processing program "yyy".

First, the operation to be executed by the packet input/output unit **21** of the packet processing program processor **20** will be described with reference to the flow chart of FIG. **9**.

The packet input/output unit **21** receives a packet **100** from the packet classification unit **10** (S401), and judges the type of data stored in the packet payload **120** (S402).

The data type may be judged from the packet classifier **111** in the packet header or it may be judged by providing a field representative of the data type at a predetermined position in the packet payload **120**.

If the packet processing program **130** is stored in the packet payload **120**, the program module name is read and the packet processing program is loaded in the main memory so as to make it executable (S403).

Entries are registered in the processing history repository table **24** shown in FIG. **8** to set values to the fields of the flow classification information **601** and packet processing program information **604** (S404). The setting in the example of the entry **610-2** shown in FIG. **8** means that a program having the program module name "xxx" is executed for the flow belonging to the source address of A-1, the source port of P-1, the destination address of A-2 and the destination port of P-2.

Similar entries (**210-P**) are set to the packet processing flow classification table **11** shown in FIG. **7** by requesting to the packet processing flow classification table update unit **23** (S411).

The packet **100** containing the program is transferred to the routing processor **5** (S410).

If it is judged that the processing history information is stored in the packet data field **120**, this processing history information is compared with the processing history information stored in the processing history repository table **24** (5405).

If the comparison result shows a coincidence, it means that this processing history information has already been stored so that the processing history repository table **24** is not necessary to be updated, and the flow advances to the next step.

Since the process requested to this packet has already been executed at the previous node in the case, the packet processing flow classification table update unit **23** is activated to search the entry having the same flow classification information **201** as that of this packet, and if the value of the packet processor classifier field **202** of the packet processing flow classification table **11** is "packet processing program processor", then it is changed to "routing processor" (S409).

The packet including the processing history information is transferred to the routing processor **5** (5410).

If not the same, the processing history repository table **24** is updated in accordance with the supplied processing history information (S406). The entries in the flow classification information designated by the packet are updated.

The packet processing flow classification table update unit **23** is activated to update the packet processing flow classification table **11** (S407).

Updating is performed by analyzing the contents of the processing history information **140** contained in the packet

100. This will be detailed by using as an example the entry 610-2 of the processing history repository table 24.

It is assumed that the processing history information 140 of ("1", "1", "1", . . .) is supplied by the packet belonging to the flow of this entry. This case means that the packet becomes not to be processed by the packet processing program at the previous node, because the third frequency band was changed from "0" to "1". In order to process the packet at this node, if the packet processor classifier field 202 at the entry having the same flow classification information 201 of the packet processing flow classification table 11 as that of this packet, has the value of "routing processor", it is changed to the value of "packet processing program processor".

Alternately, it is assumed that the processing history information 140 of ("1", "0", "0", . . .) is supplied by the packet belonging to the flow of this entry. This case means that the packet becomes to be processed by the packet processing program at the previous node, because the second frequency band was changed from "1" to "0". In this case, it is not necessary to process the packet of the second frequency band. Therefore, if the packet processor classifier field 202 at the entry having the same flow classification information 201 of the packet processing flow classification table 11 as that of this packet, has the value of "packet processing program processor", it is changed to the value of "routine processor".

Then, the packet 100 including the processing history is transferred to the routing processor 5 (S410). In this manner, the processing history information is transferred to the next node.

If the data 120 is neither the packet processing program 130 such as shown in FIG. 4 nor the processing history information such as shown in FIG. 5, this data is the data to be processed by the packet processing program. In this case, the packet processing program is executed to process the packet 100 (S408), and thereafter the packet 100 is transferred to the routing processor 5 (S410).

After the last step 410 is completed, the flow returns to the step 401 whereat the next packet 100 from the packet classification unit 10 is waited for.

Next, the operation to be executed by the processing history update unit 22 of the packet processing program processor 20 will be described with reference to the flow chart shown in FIG. 10.

The timing when this operation starts is the timing when the traffic status of the network 2 to which the packet is transmitted changes or the timing when a notice is received from the client computer 52 or server computer 50, or the operation may start periodically at every predetermined time.

First, the processing history update unit 22 supplies the current processing history information to the packet processing program and inquires the process contents (S501). In response to this, the packet processing program checks the traffic of the network and determines how the packet is processed by the packet processing program, the determined process contents being notified to the processing history update unit 22.

The operation will be described, also in this case, by taking as an example the entry 610-2 of the processing history repository table 24.

It is assumed that the packet processing program judges that the packet of the third frequency band is processed in order to reduce the traffic of the network. In this case, the entry of the per-packet processing history information 603 shown in FIG. 8 indicates that the packet of the third frequency band has already been processed by the packet processing program. Therefore, the process is terminated without performing any operation.

Alternatively, it is assumed that the packet processing program judges that the packet of the second frequency band is processed. In this case, the entry of the per-packet processing history information 603 shown in FIG. 8 indicates that the packet of the second frequency band is not still processed by the packet processing program.

In this case, therefore, the processing history update unit 22 changes the value in the per-packet processing history information 603 of the entry 610-2 of the processing history repository table 24 to ("1", "0", "0", . . .) (S503).

In order to process the packet at this node, if the packet processor classifier field 202 at the entry having the same flow classification information 201 of the packet processing flow classification table 11 as that of this packet, has the value of "routing processor", it is changed to the value of "packet processing program processor" (S504).

Next, a processing history information packet 100 having the processing history information of ("1", "0", "0", . . .) is created and transferred to the routing processor 5 in order to supply this processing history information to the next node (S505).

In this manner, the process is terminated.

Lastly, the operation to be executed by the packet classification unit 10 will be described with reference to the flow chart shown in FIG. 11.

First, the packet classification unit 10 receives a packet 100 from the network 2 (S301). Next, the packet classification unit 10 reads the packet classifier 111 indicating the packet type (S302) to judge whether the packet is to be processed by the packet processing program 130 (S303). The packet type to be processed by the packet processing program is determined in advance. Alternatively, the packet type may be determined by referring to a correspondence table which is prepared in the packet processing classification table 11 and indicates a correspondence between each packet classifier and a flag indicating whether or not the packet is to be processed by the packet processing program.

If it is judged that the packet 100 is not the target packet to be processed by the packet processing program 130, the packet 100 is transferred to the routing processor 5 (S304) to thereafter return to the step S301 to wait for the next packet.

If it is judged that the packet 100 is the target packet to be processed by the packet processing program 130, the source address 112, source port 113, destination address 114 and destination port 115 are read from the packet header field 110 (S305), and the flow classification information 201 of the packet processing flow classification table 11 is searched to find the corresponding entry 210 (S306).

If the corresponding entry 210 does not exist, the packet 100 is transferred to the packet processing program processor 20 (S307) to thereafter return to the step S301 and wait for the next packet.

If the corresponding entry 210 exists and the packet is the data packet, the contents of the packet processor classifier 202 at that entry are checked. If the contents indicate "packet processing program processor", the packet is transferred to the packet processing program processor 20, whereas if the contents indicate "routing processor", the packet is transferred to the routing processor 5. If the corresponding entry

exists and the packet is not the data packet, the packet is transferred to the packet processing program processor **20** (S307).

The contents of the packet processor classifier **202** at this entry are set in accordance with the algorithms described with reference to FIGS. **9** and **10**.

(3) Transmission of Data

A series of data transmission sequences will be described with reference to FIG. **15**.

A server **50-1**, nodes **51**, **1-1**, and **1-2** and a client **52** shown in FIG. **15** correspond to those elements shown in FIG. **2** and having the identical reference numerals. An example of an operation of transmitting data from the server **50-1** to client **52** will be described.

First, the server **50-1** transmits the packet processing program to the client **52** by using a packet having the format shown in FIG. **4** (**1501**). The conventional type network node **51** transfers the received packet to the next destination. The programmable network nodes **1-1** and **1-2** load the packet processing program contained in the packet in the main memories and register the source/destination addresses and ports in the processing history repository table **24** and packet processing flow classification table **11**, and thereafter transfer the packet to the next destination. The client **52** discards the packet containing the program.

Next, the server **50-1** transmits data as the packet having the format shown in FIG. **3** to the client (**1502**). The conventional type network node **51** transfers the received packet to the next destination. The programmable network nodes **1-1** and **1-2** process the data in the received packet by using the already loaded packet processing program and transfer the packet to the next destination. The client **52** receives the data processed at the nodes **1-1** and **1-2**.

It is assumed that the network between the nodes **1-1** and **1-2** is congested. In this case, the node **1-1** changes the process contents in cooperation with the packet processing program, and updates the processing history repository table **24**, and if necessary, the packet processing flow classification table **11**. Then, the node **1-1** generates a packet having the format shown in FIG. **5** and containing the processing history information, by using the source/destination addresses and ports in the flow classification information, and transmits the packet to the next destination (**1503**). Upon reception of the processing history information packet, the node **102** updates, if necessary, the processing history repository table **24** and the packet processing flow classification table **11**. The client **52** discards the packet processing information.

The server **50-1** transmits again data (**1504**). In the example shown in FIG. **15**, the node **1-2** stops processing the data by the program and transfers the data to the client **52**. This is because the packet processing flow classification table **11** was updated with the packet processing history at **1503**.

The server **50-1** transmits the last data or a packet notifying a data transmission end to the client **52** (**1505**). After the last data was transferred to the next destination, the nodes **1-1** and **1-2** delete the packet processing program in the memory, and delete the corresponding entry of the flow in the processing history repository table **24** and the packet processing flow classification table **11**.

For example, TCP is used as the protocol for the-above-described sequences.

(4) Programmable Network Node: Second Embodiment

The second embodiment will be described.

As shown in FIG. **12**, although a packet **100** of the second embodiment is generally the same as that of the first embodiment, per-packet processing history information **140** is added to the last of the packet header **110**. This per-packet processing history information **140** is updated when the packet is processed by the packet processing program processor, to store the history.

As different from the first embodiment, a packet processing flow classification table **11** of the second embodiment shown in FIG. **13** is different from that of the first embodiment in that the last field is per-packet processing history information **203**. This means that the packet belonging to the flow classification information **201** is processed in accordance with contents of the per-packet processing history information **203**. The next destination is therefore determined through matching the information **203** with the per-packet processing history information **140** in the packet **100**. The other constituent elements are similar to those of the first embodiment shown in FIG. **7**.

FIG. **14** is a flow chart illustrating the operation to be executed by a packet classification unit **10** of the second embodiment.

First, the packet classification unit **10** receives a packet **100** from the network **2** (S701). Next, the packet classification unit **10** reads the packet classifier **111** indicating the packet type (S702) to judge whether the packet is to be processed by the packet processing program **130** (S703).

If it is judged that the packet **100** is not the target packet to be processed by the packet processing program **130**, the packet **100** is transferred to the routing processor **5** (S704) to thereafter return to the step S701 to wait for the next packet.

If it is judged that the packet **100** is the target packet to be processed by the packet processing program **130**, the source address **112**, source port **113**, destination address **114** and destination port **115** are read from the packet header field **110** (S705), and the flow classification information **201** of the packet processing flow classification table **11** is searched to find the corresponding entry **220** (S706).

If the corresponding entry **220** does not exist, the packet **100** is transferred to the packet processing program processor **20** (S707) to thereafter return to the step S701 and wait for the next packet.

It is to be noted that the processes described above are the same as the first embodiment.

If the corresponding entry **220** exists, the per-packet processing history **140** of the packet **100** is read (S710) and compared with the per-packet processing history **203** in the packet processing flow classification table **11** (S710). If coincident, it means that the packet has already been processed. Therefore, the packet is transferred to the routing processor **5** (S704).

If not coincident, the packet is transferred to the packet processing program processor **20** and processed thereat. In the example shown in FIG. **14**, if the packet processing information is not coincident, it is presumed that the packet is required to be processed. Therefore, the packet is transferred to the packet processing program processor **20**.

In this embodiment, the processing history repository table **24** and its table operation process are unnecessary. The algorithm can therefore be simplified and the structure of a network node can also be simplified.

(5) Modifications

In the example shown in FIG. **15**, the program transmission and the data transmission are continuously and consecutively executed. During the program transmission, a flow is registered in the packet processing flow classification table **11** and processing history repository table **24**, and after the data transmission, the flow and program are deleted from the tables. However, the program transmission and the data transmission may be executed at different timings. Registration and deletion of a flow to and from the tables and deletion of the program may be performed by the following method. Namely, the server transmits a packet containing an instruction command for such processes to a programmable network node which receives and executes the instruction command.

In the above embodiments, a bitmap indicating a packet processing status for each frequency band is used as the packet processing history. However, the packet processing history may be various formats in accordance with the process contents of the packet processing program. Therefore, the packet processing history may be realized by a script written by text data or any other type.

In the first embodiment shown in FIG. **1**, the packet classification units **10** are provided in respective network input interface **3**. However, only one packet classification unit **10** may be provided in the routing processor **5**. In this case, the network input interface **3** transfers all received packets to the routing processor **5**. The routing processor **5** judges using its packet classification unit **10** whether the received packet is to be transferred to the packet processing program processor **20**, and transfers only necessary packets to the packet processing program processor **20** and the other packets directly to the network output interface **4**.

What is claimed is:

1. A network including network nodes, wherein at least one of the network nodes comprises:

program executing means for executing a packet processing program relative to each packet belonging to a flow;

routing means for performing a routing process for an input packet; and

packet classification means for analyzing the input packet to transfer a packet belonging to the flow to said program executing means and transfer other packets to said routing means.

2. A network according to claim **1**, wherein said program executing means of the network node receives the packet processing program via the network and makes the received packet processing program executable.

3. A network according to claim **1**, wherein said packet classification means of the network node includes a correspondence table between each flow and one of the program executing means and the routing means to transfer the packet in accordance with the table.

4. A network according to claim **1**, wherein said network node further comprises:

processing history repository means for storing packet processing history of a packet process already executed by another network node on a route of the flow; and

means for notifying the packet processing history of the packet process to another network node,

wherein upon arrival of the packet at the subject network node, said packet classification means refers to said processing history repository means;

transfers the packet to said program executing means only if the packet belonging to the flow has not been applied with a process to be executed at the subject network node; and

transfers the packet to said routing means if the packet has already been applied with the process.

5. A network according to claim **1**, wherein each packet used by the network has a field for storing packet processing history of a packet process executed by the network node on a route of the flow, and the network node updates the field when executing the packet processing program, and

upon arrival of the packet at the subject network node, said packet classification means refers to the field;

transfers the packet to said program executing means only if the packet belonging to the flow has not been applied with a process to be executed at the subject network node; and

transfers the packet to said routing means if the packet has already been applied with the process.

6. A network node constituting a network, comprising:

program executing means for executing a packet processing program relative to each packet belonging to a flow;

routing means for performing a routing process for an input packet; and

packet classification means for analyzing the input packet to transfer a packet belonging to the flow to said program executing means and transfer other packets to said routing means.

7. A network node according to claim **6**, wherein said program executing means receives the packet processing program via the network and makes the received packet processing program executable.

8. A network node according to claim **6**, wherein said packet classification means includes a correspondence table between each flow and one of the packet executing means and the routing means to transfer the packet in accordance with the table.

9. A network node according to claim **6**, further comprises:

processing history repository means for storing packet processing history of a packet process already executed by another network node on a route of the flow; and

means for notifying the packet processing history of the packet process to another network node,

wherein upon arrival of the packet at the subject network node, said packet classification means refers to said processing history repository means;

transfers the packet to said program executing means only if the packet belonging to the flow has not been applied with a process to be executed at the network node; and

transfers the packet to said routing means if the packet has already been applied with the process. program.

10. A storage medium for storing program codes realizing a method of processing a packet at a network node constituting a network, the packet processing method comprising the steps of:

receiving a packet processing program via the network and storing a correspondence relation between a flow and the packet processing program.

making executable in advance the packet processing program relative to each packet belonging to the flow; and

analyzing an input packet, inputting the input packet belonging to the flow to the packet processing program, and executing a routing process for the input packet not belonging to the flow.

* * * * *

US006359976B1

(12) **United States Patent**
Kalyanpur et al.

(10) **Patent No.: US 6,359,976 B1**
(45) **Date of Patent: Mar. 19, 2002**

(54) **SYSTEM AND METHOD FOR MONITORING SERVICE QUALITY IN A COMMUNICATIONS NETWORK**

(75) Inventors: **Gaurang S. Kalyanpur**, Allen; **Chad Daniel Harper**, **Grant Michael Brehm**, both of McKinney; **Chunchun Jonina Chan**, Plano, all of TX (US)

(73) Assignee: **Inet Technologies, Inc.**, Richardson, TX (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/395,801**

(22) Filed: **Sep. 14, 1999**

Related U.S. Application Data

(63) Continuation-in-part of application No. 09/093,955, filed on Jun. 8, 1998, and a continuation-in-part of application No. 09/093,824, filed on Jun. 8, 1998, now Pat. No. 6,249,675.

(51) **Int. Cl.**[7] **H04M 1/24**; H04M 15/00

(52) **U.S. Cl.** **379/134**; 379/32.01; 379/32.02; 379/133; 379/111; 379/112.01; 379/114.01

(58) **Field of Search** 379/34, 112, 113, 379/114, 133, 134, 139, 140, 210, 32.01, 32.02, 111, 112.01, 112.06, 112.07, 112.08, 114.01, 114.14, 114.28, 242, 243, 244

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,008,929	A	4/1991	Olsen et al.	379/112
5,333,183	A	7/1994	Herbert	379/112
5,426,688	A	6/1995	Anand	379/5
5,438,570	A	8/1995	Karras et al.	370/94.2
5,448,624	A	9/1995	Hardy et al.	379/67
5,457,729	A	10/1995	Hamann et al.	379/2
5,473,596	A	12/1995	Garafola et al.	370/13
5,488,648	A	1/1996	Womble	375/13
5,521,902	A	5/1996	Ferguson	370/13
5,539,804	A	7/1996	Hong et al.	379/33
5,550,914	A	8/1996	Clarke et al.	379/230

(List continued on next page.)

FOREIGN PATENT DOCUMENTS

EP	0541145 A1	10/1992	H04M/3/36
WO	WO 97/05749 A3	2/1970	H04M/15/00
WO	WO 95/33352	12/1995	H04Q/7/34
WO	WO 97/05749 A2	2/1997	
WO	WO98/47275	10/1998	H04M/7/00

OTHER PUBLICATIONS

International Search Report (PCT/US 00/25070) dated Dec. 14, 2000.

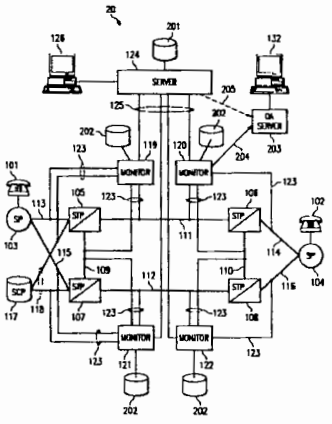
George Pavlou et al., Intelligent Remote Monitoring, Oct. 16, 1995.

Primary Examiner—Binh Tieu
Assistant Examiner—Quoc D. Tran
(74) *Attorney, Agent, or Firm*—Fulbright & Jaworski LLP

(57) **ABSTRACT**

A system and method for monitoring service quality using Call Detail Records (CDR) in a communications network, such as a Signaling System No. 7 (SS7) network, is disclosed. Network monitors capture substantially all signaling units in the SS7 network generate a complete record for all calls, transactions and other communications over the network. Users configure CDR profiles that are used to filter the records. A CDR application filters the records by parsing out signaling unit components that have been selected by the user in the CDR profile. The selected message components are then formatted into a CDR record, which is sent to an external system that generates certain statistics for the message records and stores the statistics to a database. A report application recalls the statistics from the database and presents statistics in a reporting format configured by the user. The reports indicate the statistical performance of network providers for selected called or calling telephone numbers or for selected services. The CDRs and statistics are available to a user either in real-time or in response to a query of historical CDR data. The network quality monitoring system is separate and independent from the network monitoring equipment.

10 Claims, 4 Drawing Sheets



U.S. PATENT DOCUMENTS

5,550,984	A		8/1996	Gelb 395/200.17
5,579,371	A	*	11/1996	Aridas et al. 379/34
5,590,171	A		12/1996	Howe et al. 379/33
5,592,530	A		1/1997	Brockman et al. 379/34
5,675,635	A	*	10/1997	Vos et al. 379/113
5,680,437	A		10/1997	Segal 379/10
5,680,442	A		10/1997	Bartholomew et al. 379/67
5,694,451	A		12/1997	Arinell 379/34
5,699,348	A		12/1997	Baidon et al. 370/242
5,699,412	A		12/1997	Polcyn 379/89
5,703,939	A		12/1997	Bushnell 379/113
5,706,286	A		1/1998	Reiman et al. 370/401
5,712,908	A		1/1998	Brinkman et al. 379/119
5,729,597	A		3/1998	Bhusri 379/115
5,737,399	A	*	4/1998	Witzman et al. 379/112
5,757,895	A		5/1998	Aridas et al. 379/136
5,793,771	A		8/1998	Darland et al. 370/467
5,799,073	A		8/1998	Fleischer, III et al. 379/113
5,822,401	A		10/1998	Cave et al. 379/34
5,825,769	A	*	10/1998	O'Reilly et al. 370/360
5,828,729	A		10/1998	Clermont et al. 379/34
5,854,824	A		12/1998	Bengal et al. 379/34
5,854,835	A		12/1998	Montgomery et al. 379/119
5,867,558	A		2/1999	Swanson 379/34
5,875,238	A		2/1999	Glitho et al. 375/116
5,881,132	A		3/1999	O'Brien et al. 379/35
5,883,948	A		3/1999	Dunn 379/210
5,892,812	A		4/1999	Pester, III 379/34
5,912,954	A		6/1999	Whited et al. 379/115
5,920,613	A		7/1999	Alcott et al. 379/114
5,999,604	A	*	12/1999	Walter 379/133
6,028,914	A		2/2000	Lin et al. 379/14

* cited by examiner

FIG. 1

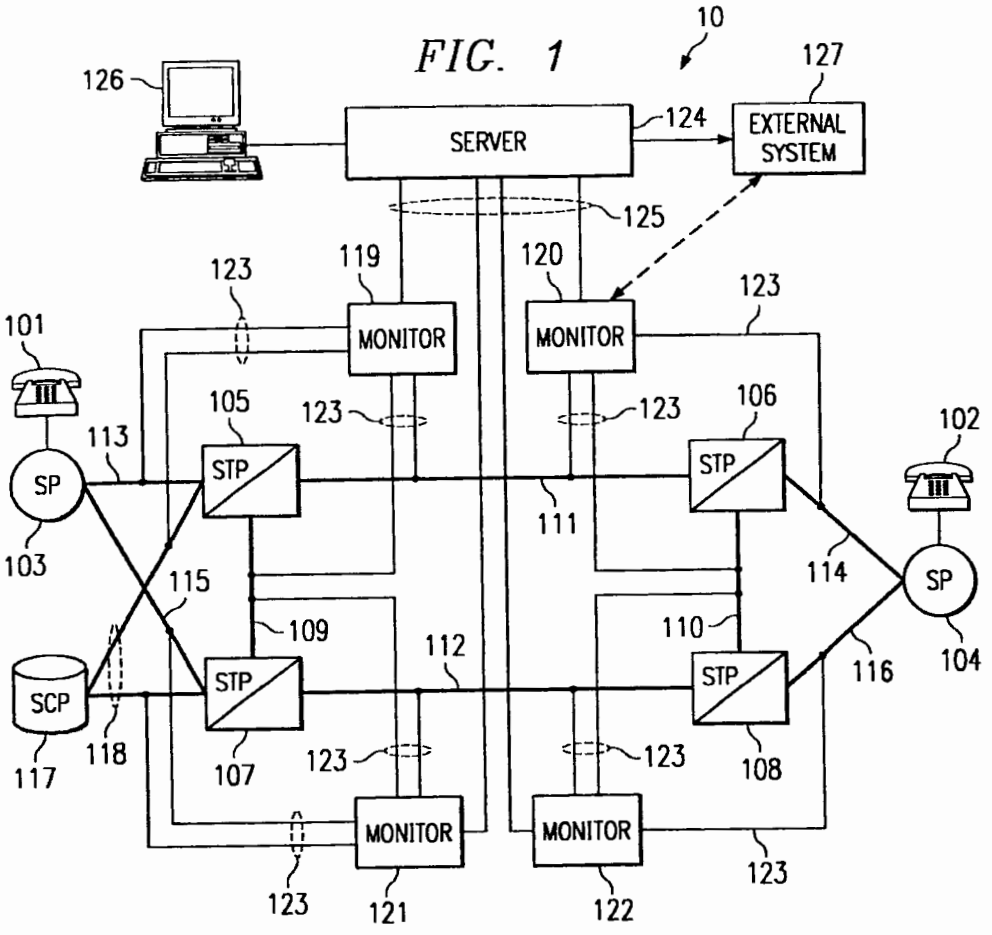
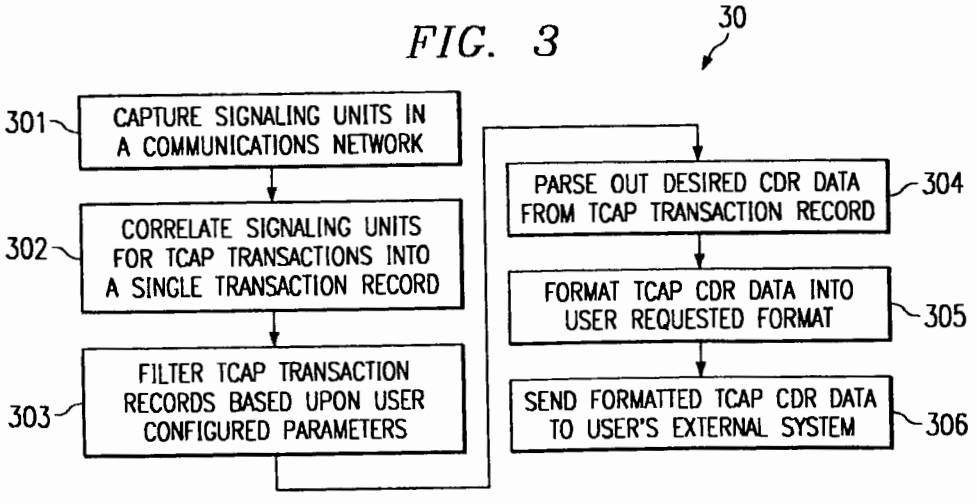

10
126
SERVER
124
EXTERNAL SYSTEM
127
125
123
119
120
MONITOR
MONITOR
101
105
106
102
SP
STP
STP
SP
103
111
113
114
115
109
110
112
SCP
STP
STP
116
104
117
118
107
108
123
MONITOR
MONITOR
121
122

FIG. 3


30
301
CAPTURE SIGNALING UNITS IN A COMMUNICATIONS NETWORK
302
CORRELATE SIGNALING UNITS FOR TCAP TRANSACTIONS INTO A SINGLE TRANSACTION RECORD
303
FILTER TCAP TRANSACTION RECORDS BASED UPON USER CONFIGURED PARAMETERS
304
PARSE OUT DESIRED CDR DATA FROM TCAP TRANSACTION RECORD
305
FORMAT TCAP CDR DATA INTO USER REQUESTED FORMAT
306
SEND FORMATTED TCAP CDR DATA TO USER'S EXTERNAL SYSTEM

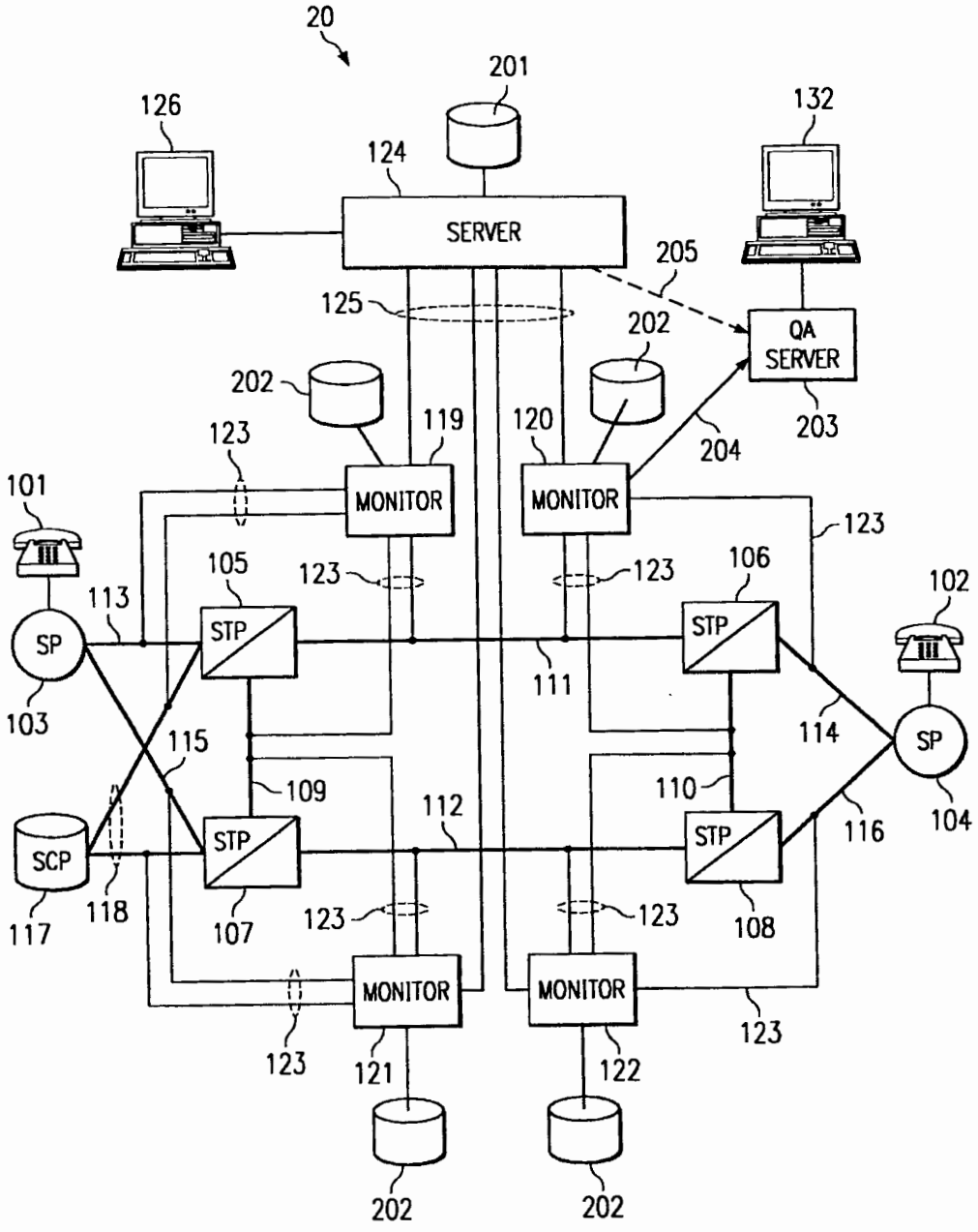

20
201
126
132
124
SERVER
205
125
202
QA SERVER
202
203
123
119
120
204
101
MONITOR
MONITOR
123
105
123
123
106
102
113
SP
STP
STP
111
103
114
SP
115
109
110
112
SCP
STP
STP
116
104
118
107
123
123
108
MONITOR
MONITOR
123
123
121
122
202
202

FIG. 2

FIG. 4

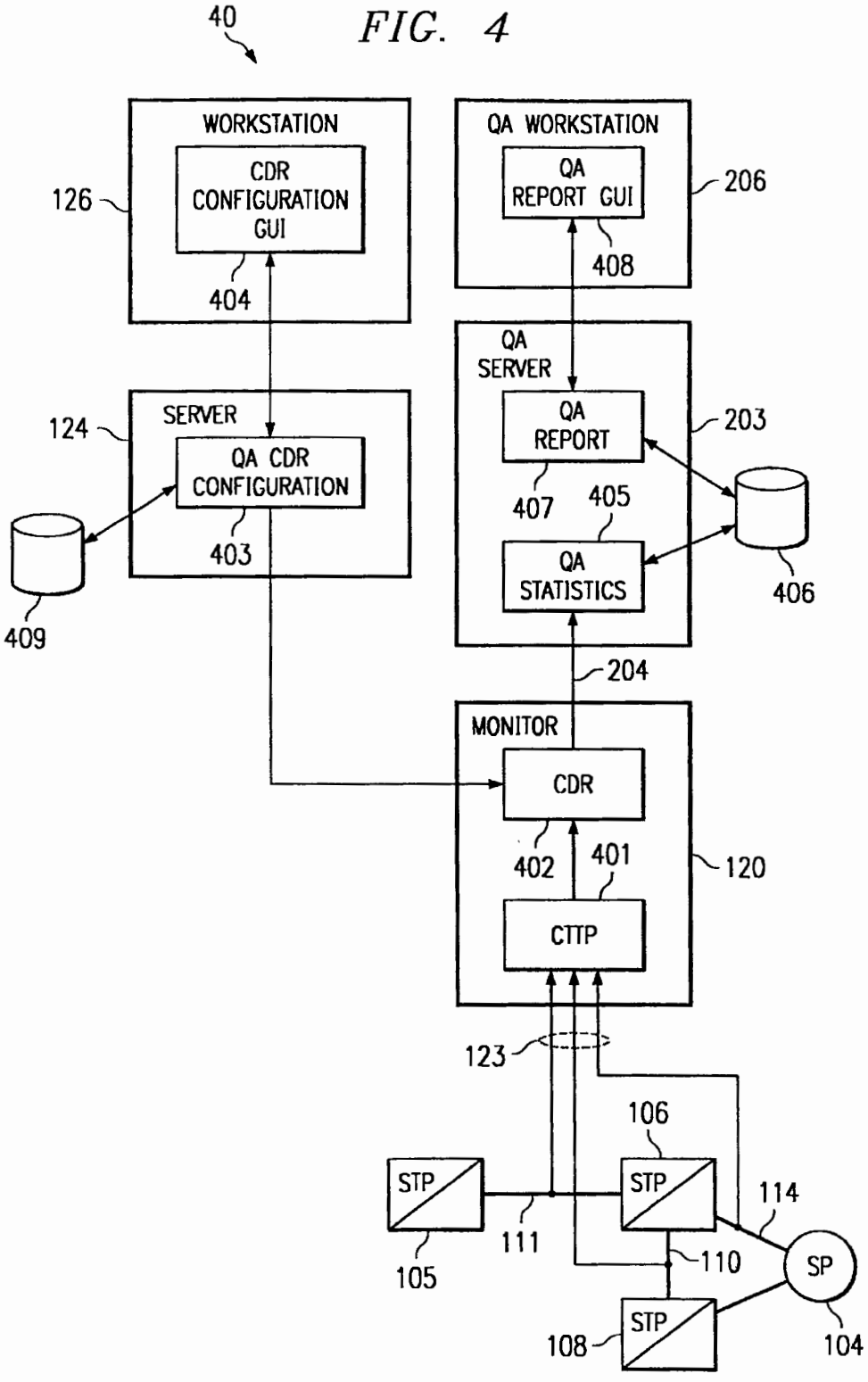
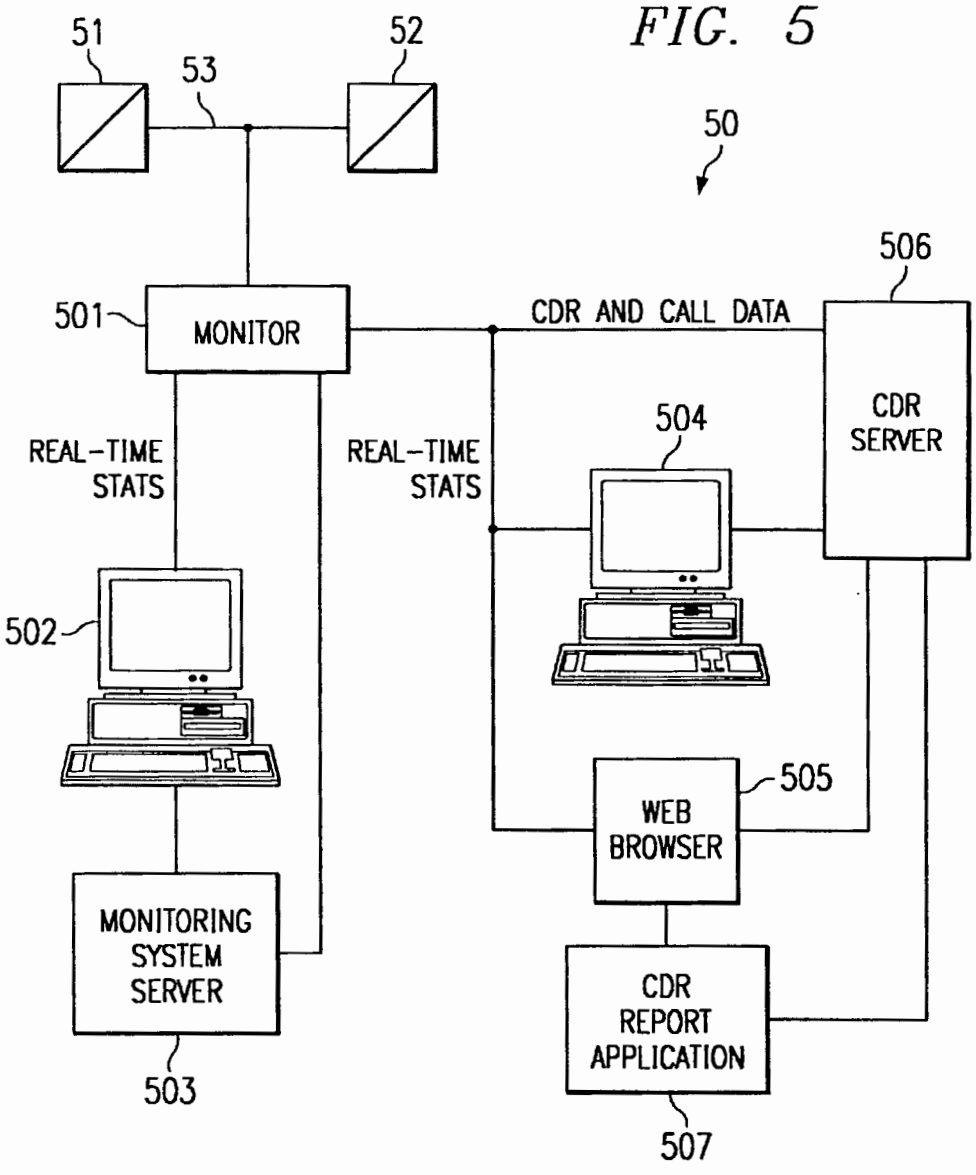

40
WORKSTATION
CDR CONFIGURATION GUI
126
404
QA WORKSTATION
QA REPORT GUI
206
408
QA SERVER
QA REPORT
203
407
405
QA STATISTICS
406
SERVER
124
QA CDR CONFIGURATION
403
409
204
MONITOR
CDR
402
401
120
CTTP
123
106
STP
STP
114
111
110
SP
105
STP
108
104

FIG. 5


51
53
52
50
506
501
MONITOR
CDR AND CALL DATA
CDR SERVER
504
REAL-TIME STATS
REAL-TIME STATS
502
505
WEB BROWSER
MONITORING SYSTEM SERVER
CDR REPORT APPLICATION
503
507

detail records (CDR) are received from a network monitoring system. The monitoring system generates CDRs for calls, transactions, and/or other communication on a network. For example, the monitoring systems may have network monitors that capture communication messages and message signal units (MSU) from links in a communications network. The links may be between the originating, intermediate, and terminating nodes, switches or end offices. The messages or MSUs are captured and used to generate call detail records. A network of signal monitoring units may capture and correlate all messages for a particular call, transaction or other communication. Preferably, the monitors have a plurality of processors for processing the captured messages or MSUs. The processors may run any of a number of message or record processing applications.

Typically, CDR profiles are used to determine which messages or other data should be included in the CDRs. The CDR profile comprises particular parameters that are used to identify relevant calls, transactions or other communications. After a transaction record is selected, specific information is extracted to create a CDR record. Users define both the CDR profile, which is used to select relevant records, and the CDR format, which defines how the CDR data will be sent to the user. The CDR data is sent to a quality monitoring system in a formatted CDR stream. The CDR data may be used to monitor network quality in real-time. Additionally, the CDR data may be stored so that historical network analysis may be performed. The CDRs may also be processed by other applications, such as billing or fraud applications.

Typically, the CDR is generated when a call is completed. The CDR includes information such as the originating network, terminating network, and length of trunk usage for the call. Since the identity of the originating service provider and the duration of the call or transaction are contained in the CDR, a CDR billing application may be used for generating interconnection revenue for reciprocal compensation. External applications may use the CDR data to generate bills or track SS7 bandwidth use. The CDR data can be ported to a customer's external application, where the call can be rated and a bill or invoice can be generated for the transaction or call.

In the present invention, a quality assurance application provides an integrated platform for message tracking on a per customer and/or a per service provider basis. The tracked messages may be part of one of a number of message protocols, such as Integrated Services Digital Network—User Part (ISUP), Telephone User Part (TUP), Network User Part (TUP), Transaction Capabilities Application Part (TCAP), Advanced Intelligent Network (AIN) or Integrated Network Application Part (INAP) calls or transactions. The quality assurance application is useful for larger networks or for evaluating service quality of application-layer services, such as FNAP, Global System for Mobile Communications (GSM), AIN, IS-41 and 800/LIDB/CLASS.

The system disclosed herein comprises a number of monitors which are capable of non-intrusively monitoring all of the links in a communication network, such as an SS7 network. CDR data is initially collected from the various SS7 links. The monitors that are connected to the links store the data in a binary format. The binary data is then continuously sent to the central server where it is stored to disk. This application can be used in conjunction with the monitor's server, or customers may choose to deploy a dedicated CDR server, separate from the system. The server also correlates partial CDRs that have been collected from different "legs" of each individual call to formulate a complete CDR. At the server, CDRs are formatted from binary into ASCII-formatted records based on a CDR format that is selected by the user. The size and processing power of the server are scaled based on the number of CDRs, the network-wide call rate, and the bandwidth capacity of the customer's transport network. The formatted CDR binary streams are sent to the user's billing system using any standard or customized File Transfer Protocol (FTP). Additional data formatting may be performed in the customer's external billing system.

In order to generate CDRs, users create profiles that tell the monitor system how to collect SS7 information from the signaling links. The profiles contain all of the information required to generate CDRs. Multiple profiles can be created to be used simultaneously on the system. The profiles may include parameters such as the calling party number, called party number, mobile identification number (MIN), point codes, and application type. Essentially, any component of a transaction signaling unit may be used as a filter parameter.

In one embodiment, the quality assurance application runs on a server that is external to the network monitoring system. The monitoring system provides data to the external server in the form of Call Detail Records (CDRs). The quality assurance application tracks the quality of service that is provided to customers on a particular communications network. The present invention allows customers, service providers and others to monitor how a service is performing not only within the network infrastructure, but also how well that service is working on a call-by-call, customer-by-customer basis. Additionally, the present invention allows service providers to efficiently manage network services without requiring an increased support staff.

The quality assurance application runs on an independent server and processes CDRs that are received from the monitoring system. In an exemplary embodiment, individual monitoring units exchange and correlate messages into call or transaction records. The monitoring unit then filters the records using a CDR profile to determine which records, and which messages, should be combined to form the CDR. The monitoring units then transmit the CDRs directly to the external server. In an alternate embodiment, some other entity in the monitoring system such as a central server, may generate and forward CDRs to the independent quality assurance application.

A CDR collection process on a CDR server collects all of the legs of a transaction, call or other communication and correlates the individual leg information into a single CDR. The CDR is then put in the required format. The CDR consists of data from multiple message that are related to a single transaction, call or message.

The CDR server, which may or may not be a dedicated server, acts as a client and initiates the connection to an external system on a predefined port number. A configuration file on the CDR server designates an IP address and port where all the formatted CDRs are to be directed. All CDRs will be streamed to the external system and no application level protocol will be followed. The underlying protocol will be TCP/IP. All MSUs related to a single transaction are packetized in a single CDR and a CDR will be generated and sent for every transaction. The CDRs are destroyed as soon as they are successfully transmitted from the CDR server to the external system. No acknowledgment is expected from the external system for CDR receipt. For each unique pointcode in a profile the server spawns a TCP/IP connection to an external system to send CDRs. If the pointcode is repeated in multiple profiles, only one connection shall be

established. A configuration file lists the pointcode to connection/port number mapping. If a pointcode is not listed in the configuration file, CDRs generated for this pointcode will be destroyed immediately. A log is kept to track when the connections are established or down and to track the numbers of CDRs sent and dropped hourly and daily.

The configuration file mapping method may also be defined to provide a CDR profile to connection mapping. In this case, all CDRs generated by a profile are sent to the same destination.

The CDR server may store CDRs on a local disk using a predefined file naming convention so that all CDRs for a profile are stored in one file. New files are created for a defined interval and, as soon as the file is closed, an external system can retrieve the file using FTP, or some other protocol.

The quality assurance application provides service quality analysis tools and reports. The application generates historical statistic reporting for circuit-based services or for application-layer services. The statistics are maintained in a database which can be accessed to generate quality of service reports. When used to monitor service on an SS7 network, the present invention provides users with the capability to select from a number of parameters which can be used to filter call, transaction or other communication records. Filters may be based upon called and calling numbers, or groups of digits within the called or calling numbers. Additionally, application types and point codes may be used as filter parameters.

The quality assurance application maintains statistics for all ISUP/TUP circuit-based calls. Statistics are maintained by called number, calling number and translated number. Users may generate reports for the statistical information by accessing the database through a workstation. The reports may be customized using various indices, such as by called, calling, or translated number.

Additional statistics may be monitored and other reports may be created for other communications networks or protocols. For example, TCAP statistics may be monitored and reports may be generated by service as well as by called, calling and translated number. Statistics for other application layer services could also be monitored. Such as for INAP, GSM, AIN, IS-41 and 800/LIDB/CLASS services.

Communications network monitoring equipment which may be used in conjunction with the present invention is disclosed in U.S. Pat. No. 5,592,530, entitled TELEPHONE SWITCH DUAL MONITORS; and in the above-referenced pending patent applications the disclosures of which have been incorporated by reference herein. Additionally, the present invention may be used with any network monitoring equipment or other equipment that generates call detail records. Such network monitoring equipment may include hardware and/or software that is integral to a communications network node. Alternatively, the monitoring equipment may be external hardware and/or software that detects call, transaction or other messages passing over communications links between network nodes. The systems and methods disclosed herein are capable of receiving and processing call detail records from any source. As used herein, the term call detail record refers to any record or message that comprises data related to a call, transaction or other communication on a network.

It is a feature of the present invention to track performance statistics for a communications network. The invention provides statistical reports that allow users to determine the reason for call failures and to identify portions of the network which are not operating properly.

It is another feature of the present invention to allow customers, service providers and third parties with the ability to monitor a the quality of service on a particular communications network. Customers can use the statistical reports to determine their service provider's quality of service. The present system can also be used by customers to determine if the customers' systems are providing adequate service. For example, call centers can use the statistical data to determine whether additional agents are needed to answer calls that have been dropped due to busy lines. Service providers may use the information to monitor the service provided by their network and to identify failure points on the network. Service providers can also monitor the quality of service provided by other service providers on other networks.

It is an additional feature of the invention to generate statistical reports for called, calling or translated numbers or for services. Additionally, users can designate particular link sets to be used for the statistical report generation. As a result, only those monitors capturing messages from the designated link sets will send CDRs to the quality of service application.

It is another feature of the present invention to provide statistical reports in real-time on a network-wide basis for both calls and transactions. Historical reports may also be created from CDR data that is stored to a database.

The foregoing has outlined rather broadly the features and technical advantages of the present invention in order that the detailed description of the invention that follows may be better understood. Additional features and advantages of the invention will be described hereinafter which form the subject of the claims of the invention. It should be appreciated by those skilled in the art that the conception and the specific embodiment disclosed may be readily utilized as a basis for modifying or designing other structures for carrying out the same purposes of the present invention. It should also be realized by those skilled in the art that such equivalent constructions do not depart from the spirit and scope of the invention as set forth in the appended claims.

BRIEF DESCRIPTION OF THE DRAWINGS

For a more complete understanding of the present invention, and the advantages thereof, reference is now made to the following descriptions taken in conjunction with the accompanying drawings, in which:

FIG. **1** is a block diagram of an exemplary communications network and monitoring system;

FIG. **2** is a block diagram of exemplary quality assurance system connected to a network monitoring system;

FIG. **3** is a flowchart that illustrates the creation of CDRs in accordance with one embodiment of the present invention;

FIG. **4** is a block diagram that illustrates one embodiment of a quality assurance system; and

FIG. **5** is a block diagram of a network quality monitoring system that is independent of the network monitoring system.

DETAILED DESCRIPTION

FIG. **1** is a block diagram of communications network **10** in which telephones **101** and **102** communicate via a signaling network, such as an SS7 network. It will be understood that telephones **101, 102** are used for illustration purposes only and that any voice or data communications device may be connected to the network. In the embodiment

illustrated in FIG. **1**, telephones **101** and **102** are connected to end offices **103** and **104**, which may be Signaling Points (SP), as shown, or SSPs. End offices **103** and **104** are linked to each other through a signaling network comprised of STPs **105–108**, which are connected via links **109–112**. SPs **103** and **104** are connected to STPs **105–108** via links **113–116**. Calls, transactions and other signals or messages between end office **103** and end office **104** may take any of a number of paths across links **109–116** and through STPs **105–108**.

Typically, a series of signals that are related to one call or transaction will traverse across the same path through network **10** from one end office to another. For example, for a particular transaction, all signaling units sent from SP **103** may cross links **113**, **111**, and **114** through STPs **105** and **106**. However, network problems or system failures may cause different signals for the same transaction to take different paths. It is also typical that signals traversing the system in the reverse direction may use a different path through the network. For example, for the same transaction illustrated above, signals from SP **104** may traverse links **116**, **112**, and **115** through STPs **108** and **107**. Therefore, a single link or network element may not see all the messages or signals for one complete transaction or call.

In certain circumstances, such as for an 800 number call or for a call to an exchange or number that has been ported to a different switch, message may be sent to SCP **117** to perform various database look-up functions. Signals or messages are exchanged with SCP **117** via links **118**. In other embodiments, there may be additional components in network **10**, such as Service Nodes (SN) or Intelligent Peripherals (IP), which would require additional signal paths.

In network **10**, monitors **119–122** are individually paired with STPs **105–108**. Each monitor **119–122** is coupled to every link for a particular STP by connections **123**, which may be embodied as a branch or tee off of links **109–116**. This allows monitors **119–122** to capture or detect every signaling unit that is sent to, or from, each STP **105–108**. As described in U.S. Pat. No. 5,592,530 and application Ser. No. 09/057,940, monitors **119–122** are coupled via an inter-monitor communications link (not shown) which allows monitors **119–122** to transfer captured signaling units and messages among themselves. Typically, the first monitor to detect a signaling unit for a call or transaction is designated as a controlling or anchor monitor. The other monitors then send any later detected signaling units for the same transaction or call to the anchor monitor. The anchor monitors correlates all of the messages from a particular transaction or call into a single record. Usually, each signaling unit is identified as belonging to a particular transaction by the Transaction Identifier (TID).

Monitors **119–122** are connected to server **124** via connection **125**, which may be a Wide Area Network (WAN) or any other data network connection. Once a call or transaction record is complete, the record is then sent to server **124** for further processing. Monitors may determine that a record is complete when an end message is detected for that particular call or transaction. Workstation **126** is connected to server **124** and may be directly connected to monitor **119–122**. Workstation **126** provides network service providers or other users with access to retrieve data or to configure server **124** or monitors **119–122**.

Monitors **119–122** detect and correlate TCAP messages from network **10**. These messages are used to generated binary call detail records (CDRs) which are streamed to server **124** over WAN **125**. Server **124** formats each binary TCAP CDR stream into a format selected by a user or customer and forwards the formatted TCAP CDR data to the customer's external system **127**. The TCAP CDR data may be sent to external system **127** either from server **124** or directly from the network monitors, such as from monitor **120** as illustrated.

The user sets up a profile on workstation **126**, such as a SUN workstation. The user may interact with workstation **126** via a Graphical User Interface (GUI) to configure the CDR profile. The profile is a filter having certain criteria configured by the user. System **127** may be comprised of a server or other processor which is capable of using the CDR data to rate and bill transactions on network **10**. External system **127** may be a quality of service application that processes CDRs and generates historical and/or real-time reports on network quality.

Typically, the CDRs are created at transaction termination. The binary CDRs are sent via Transmission Control Protocol/Internet Protocol (TCP/IP) to the server listed in the CDR profile for further processing. A collection process on server **124** then collects all legs of a transaction and correlates the data into a single CDR. The CDR consists of multiple signaling units which are related to a single transaction, for example, the entire TCAP dialogue, including prearranged ends and time-outs. Each CDR is assigned a unique sequence number during the CDR collection process. The CDR is then formatted as defined by the user and sent to external system **127**. In one embodiment, server **124** acts as a client and initiates a connection to the external system on a predefined port number.

Monitors **119–122** are capable of monitoring a multitude of SS7 links at one time. A unique identifier, or CDR sequence number, is generated for every CDR and uniqueness is guaranteed system wide. The CDR application can be configured in a sampling mode with the sampling rate determined on a per profile basis. The maximum sampling rate is decided based on the monitor system sizing. Preferably, the sampling rate can be selected from 1% to 100% in increments of 1%. In the preferred embodiment, monitors **119–122** contain software that delivers the signaling units captured from the SS7 network to a CDR filtering process for evaluation. Server **124** is responsible for tracking all CDR configurations set up by the user and for downloading CDR configurations to monitors **119–122** as necessary. Depending upon the configuration selected by the user, monitors **119–122** determine if a message has passed the filter criteria. If a message does pass the criteria, it is sent to a tracking task located on monitors **119–122** and then to server **124**. In the event a message does not match the characteristics defined by a user, the message will be discarded.

Server **124** may be a single server or it may be embodied as two or more servers having separate functions. For example, one server may act as a central information point for all entities of the monitoring system and another server may control CDR processes. Any entity needing common information can obtain that information from a monitoring system server database. The database control on the monitoring system server includes configurations for all monitor applications. In this embodiment, the CDR configuration information can be stored on the monitoring system server in a configuration file. At the start of a CDR generation session, the CDR configuration file is downloaded to specific monitors over network **125**.

Either server **124**, or a separate CDR server, maintains another CDR configuration file to provide mapping of CDR

profile names to virtual connections. This file lists CDR profile names and the corresponding connection identifications on which external system is expecting the CDRs for that profile. The CDR configuration file also comprises a mapping of the virtual connection identifiers to their connection names. In the preferred embodiment, multiple CDR profiles can be mapped to a single virtual connection identifier, but a single CDR profile cannot be mapped to multiple connection identifiers.

Server **124**, or the CDR server, performs CDR processing. The CDR process collects all binary CDRs from monitors **119–122** and format the CDRS. The formatted CDR is then sent via TCP/IP to an external system. Each profile in the configuration file can instruct monitors **119–122** to send binary CDRs to different servers or workstations **126**. However, it is mandatory that a CDR collection process should be running and listening on the assigned IP address and port.

External system **127** shall act as a server and listen on a pre-defined port number for incoming CDRs. Server **124** shall act as a client and initiate a connection to the external system on a predefined port number. Server **124** is capable of spawning multiple connections based on the configuration file and the number of configurations are configurable. Server **124** is also capable of communicating with multiple external servers. Server **124** can send formatted CDRs that have been generated using different profiles to different servers.

The formatted CDRs may be queued in a list to be sent to external system **127**. If there is a loss of communication on a port, up to **512** CDRs shall be stored per connection. When the CDR queue is full, the CDRs will be deleted on a First-In-First-Out (FIFO) basis. On start-up, after a communication failure, any pre-existing CDRs shall be sent to the external system before any of the new CDRs are sent. A connection acceptance message from external system **127** contains the last sequence number received. Server **124** reads the sequence number and sends the next available CDR. In some situations, the first CDR transmitted after communication re-establishment may not be the CDR external system **127** was expecting. Thus, there is a potential for data loss if the connection is down for a long time.

The following messages may be logged to a daily file on a per connection basis:

Connection Established;

Connection Down;

Number of CDRs Sent (per hour);

Number of CDRs Dropped (per hour); and

Daily Total Number of CDRs Sent and Dropped.

Local time and date are indicated on each message and the logical connection name is included on each line.

As discussed above, workstation **126** has a GUI configuration interface that enables users to select the signaling groups and point codes to be used in configuring the CDRs. The GUI allows users to add, modify, or delete CDR profiles. The CDR configurations are active until they are deleted. Once a profile is activated, the user is notified. The CDR profiles indicate the address and port for the external system **127**, which is collecting the CDRs. All CDRs generated by a profile are sent to the destination IP address via TCP/IP.

Filters may be selected for called, calling and translated numbers based on the selected protocol, such as INAP, 800, or IS-41. The CDR configuration supports the use of wildcards for point codes or system nodes. Additionally, wildcards are supported for phone numbers. Wildcards allow the user to configure profiles which encompass all of the point codes, network nodes, or telephone numbers having a common series of numbers, such as a common area code or exchange. For example, the wildcard telephone number "1-NPA-*" for a called number can be used to filter out all records for calls or transactions to a telephone number in the "NPA" area code.

TABLE 1 is a list of the parameters that customers can use to create CDR profiles. The CDR profiles tell monitors **119–122** how to collect SS7 information from the signaling links. The profiles contain all of the information that is necessary to generate CDRs. A single customer can generate multiple profiles and each profile can include different parameters.

TABLE 1 is a list of the parameters that can be used to create CDR profiles.

TABLE 1

Call State that Triggers the CDR Generation
Address Complete
Answer
Call Termination
Application Type
ANSI ISUP
ITU ISUP
ITU TUP
ITU NUP
IS-41
CLASS
LIDB
AIN
INAP
National Variants
Toll Free/800
Point Codes
OPC
DPC
Calling Party Numbers
Called Party Numbers
Translated Numbers
Dialed Digits
Destination Digits
Mobile Identification Number (MIN)
Routing Numbers
Account Numbers
Electronic Serial Number
Location Routing Number

TABLE 2 lists the fields of a preferred CDR format and the definitions of the field contents.

TABLE 2

Length of Entire CDR	
Length of Fixed Fields	Indicates the length of the fixed CDR fields. The value is the number of bytes after the "Length of Fixed" field to the "User Field Length" field.
CDR Sequence Number	Numeric value that uniquely identifies the call record within the monitoring system. Uniqueness is guaranteed system wide. The system also uses this number to indicate the delivery monitor and its process ID.
CDR Condition Indicator	Indicates various conditions within a call/transaction.
Date/Time of Transaction Start	GMT time when a transaction begins.
Date/Time of Transaction End	GMT time, when a transaction end message is encountered.
CIC	Carrier Identification Code
OPC	Network indicator, protocol as well the origination pointcode of the call.
DPC	Network indicator, protocol as well the destination pointcode of the call.
Abort Reason	Abort cause of the transaction.

coupled to a database (not shown) that is capable of storing filtered TCAP transaction records or formatted TCAP CDRs.

FIG. **4** illustrates system **40** in which CDR applications are running on the components of a monitoring network and quality assurance applications are running on QA server **203** and workstation **206**. Components of FIG. **4** are numbered to correspond with like components of FIG. **1** and **2**. Monitor **120** is capable of monitoring several hundred SS7 links at one time. Monitor links **123** capture messages from network links, such as **111**, **110** and **114**, in the SS7 network. The messages are provided to a call/transaction processing application, such as Call/Transaction Tracking Processor (CTTP) **401**. Monitor **120** comprises a number of versatile processors which may be assigned to process and correlate calls, transactions, or other messages. One or more of these processors run CTTP application **401** depending upon the volume of message traffic received from the SS7 network. As discussed above, monitor **120** communicates with other monitors, such as **119**, **121** and **122**, and exchanges messages pertaining to the calls and transactions that are being monitored.

Monitor **120** also comprises CDR application **402**, which runs on another processor. CDR application **402** receives correlated message records from CTTP application **401** and filters the records using a CDR profile. Ideally, CDR application **402** receives complete records for each call and transaction from CTTP application **401**. However, depending upon the state of a particular call or transaction, partial records may be provided. CDR application **402** collects messages for call legs and generates a Call Detail Record. The CDR contains summary information of the statistics for each call. Application **402** generates a binary CDR that is sent to QA server **203** via Transmission Control Protocol/Internet Protocol (TCP/IP) for further processing. There may be one or more QA servers coupled to the monitoring network or to individual monitors. Monitor **120** sends the CDR data to the QA server listed in a QA CDR profile.

Typically, the CDRs are not stored on monitor **120**. The binary CDR data is streamed to QA server **203** as soon as it is created. A unique identifier is created for each CDR so that QA server can distinguish among the CDRs that are received from various monitors. Messages that are received out of sequence by CTTP application **401** are sent to CDR application **202**, which attaches the out of sequence message to the CDR data stream.

Monitoring system server **124** is responsible for tracking all CDR configurations that have been set up by users. QA CDR configuration application **403** cooperates with CDR application **404** on workstation **126** to provide a user interface to configure the QA CDR profiles. Server **124** stores the QA CDR profiles as files in memory **409**. The profiles are downloaded to monitors **119–122** as necessary so that the monitors will have the proper configuration to process the correlated message records.

Users configure the QA CDR profiles, and other monitoring system parameters, using workstation **126**. CDR configuration application **404**, which may be a Graphical User Interface (GUI), allows users to configure CDR profiles for storage on server **124**. Information provided by users on workstation **126** is stored as a configuration file in database **409**. Sever **124** downloads the configuration file data to specific monitors **119–122** over Simple Network Management Protocol (SNMP). Users may modify the CDR profile configurations. Changes to old configurations are relayed to the appropriate monitors **119–122**.

QA server **203** is preferably a dedicated server for the quality assurance application because of the high volume of data associated with the call and transaction records. CDR data streams from monitors, such as data on link **204** from monitor **120**, is processed by QA statistics application **405**. Database **406** holds the CDR data for QA server **203**. QA statistics application **405** collects CDRs from monitors **119–122** and stores the data to database **406**. This data is then later recalled by QA report application **407**, which reports statistics on the data when requested by users. QA workstation **206** provides the user interface to QA report application **407** through QA report GUI **208**. Users configure the desired parameters for the statistical reports via QA report GUI **208**. QA report application **407** then recalls and formats the stored data from database **406**.

Depending upon the user's system, databases **406** and **409** may be an integral part of servers **124** and **203**, or the databases **406**, **409**, may be embodied as separate storage devices.

The amount of data stored and the message traffic volume are the key determinants of the size and processing power of QA server **203**. Processing capabilities can be adjusted on a per user basis. The minimum configuration of the preferred embodiment is a server having 150 GB storage and 1 GB memory. A redundant server having equivalent capacity may also be used. Workstation **206** provides users with a GUI interface to configure statistic reports.

QA server **203** collects CDRs from monitors **119–122** and extracts statistical information to be stored in database **406**. CDRs for calls in an SS7 network are available upon call completion. QA statistics application **405** accumulates the messages statistics completion of the call or transaction and adds the statistics to database **406** at intervals based upon the origination time of the call or transaction. The statistics are continually collected and stored to database **406**, but they are reported only upon user request.

The format used to store the statistics data in database **406** is highly configurable and may be adapted for any storage configuration that the user may desire. For example, in one embodiment, separate data entries are made for each hour in a daily table in database **20**. Thus, if server **203** and database **406** are configured to hold a week's worth of statistical data, then seven daily tables, each having 24 intervals, are stored on database **406**. Each daily table is stored for seven days. Daily tables are summarized into weekly tables at the end of seven days. Weekly tables have seven intervals, each interval representing a summarized daily table. Weekly tables are stored for 90 days, at which point they are summarized into monthly tables having 28–31 intervals. Monthly tables are stored locally on database **406** as long as space permits. The aging and summarizing process can be customized by users to comply with individual requirements.

Table 5 is a list of statistics that are stored to database **406** for each CDR profile.

TABLE 5

Number of Call Attempts
Number of Call Attempts Answered
Number of User Busy Calls
Number of Ring No Answer (RNA) Calls
Number of Normal Release Calls
Number of Abnormal Release Calls
Number of Unallocated Number Calls
Number of Address Incomplete Calls
Number of Transaction Aborts
Number of Congested Transactions
Number of Congested Calls
Number of Circuit Unavailable Calls
Number of Failed Transactions
Number of Failed Calls

TABLE 5-continued

Number of Undefined Release Cause Failed Calls
Number of Destination Out of Order Failed Calls
Average Call Set-Up Time
Average Call Hold Time
Average Answer Seizure Ratio
User Defined

The user can define specific statistics, such as release causes, that are to be stored for a particular CDR profile.

Table 6 is a list of aggregations that can be used to group the above statistics for reports to be generated by QA report application **407**.

TABLE 6

Calling Numbers
Called Numbers
Translated Numbers
Called Numbers, then by Calling Numbers
Translated Numbers, then by Calling Numbers
Called Numbers, then by Translated Numbers
Services
Services, then by Calling Numbers
Services, then by Called Numbers

In Table 2 it will be understood that called, calling or translated numbers may be either a complete telephone number or a partial telephone number. For example, under the North American Numbering Plan, reports may be created for full telephone numbers (i.e. 1-NPA-NXX-XXXX). Alternatively, wildcards can be used at the end of the grouping telephone number so that statistics are reported for all calls or transactions directed to a particular area code (i.e. 1-NPA) or a particular exchange code (i.e. 1-NPA-NXX).

Users can also configure QA report application **407**, via QA report GUI **208**, to create their own query parameters. Queries can be stored in database **406** and stored queries can be modified. Reports from QA report application **407** may be displayed to the user on QA workstation **206**. Alternatively, reports may be printed, directed to an electronic mail address, stored to a database file, or exported to an ASCII file. Users can configure weekly, monthly, or other periodic reports which are sent at intervals to specific users. Such periodic reports may be assigned to QA report application **407** to be run automatically.

Dynamic behavioral statistics may also be generated by QA report application **407**. Users can select to have the statistics of Table 1 reported as to the highest and/or lowest values. For example, a report may comprise the 16 highest called numbers, or the 16 services that are used the least. Behavioral statistics are retrieved using a Structured Query Language (SQL) query. Triggers can be configure to update a user's display according to changes in database **406**. Once a group or aggregation of statistics is displayed, users can refine the report to obtain more specific data, such as a specific area code and exchange.

Users may track statistical events by designating a statistics to be displayed based upon a first occurrence, a occurrence that is more than some delta away from a certain value, or rising/falling thresholds. When triggered, events may be displayed to the user, or stored to a log file.

Users may also designate specific link sets or network nodes to be used for the statistical reports. Only those monitors that are coupled to the relevant links and nodes will receive the CDR profile data and only those monitors will send CDRs to QA server **203** for that profile.

Real-time statistics are also available from QA report application **407**. Statistics are then updated after call or transaction completion and CDR generation. Displayed reports may be in the form of peg counts, bar graphs, or trend curves. Users may also configure reports based upon a sample of the calls or transactions or based upon a sample of the CDRs. The sampling rate may be selected using CDR generation GUI **404** on the user workstation.

It will be understood that workstation **126** and **206** may be separate components as described herein, or one workstation may be used to run both CDR configuration GUI **404** and QA report GUI **408**.

It will also be understood that the QA server can accept CDR data from any source, not only from the monitoring system. For example, a switch or end office may generate CDRs and provide the data directly to the QA server for further processing. The QA server has a modularized front end which allows it to receive data from any source.

Reporting and measuring of the CDR data allows users to define any number of digit combinations. A leading digit summary is defined as an aggregation of CDRs associated with a selected prefix. This parameter returns a composite result on numbers having the selected prefix, instead of providing results for each discrete number within the number range. For example, an entry of 972 returns a single measurement for all calls with the 972 prefix. An entry of 972-578-0000 returns two measurements; a measurement for 972-578-0000 as well as a measurement in the 972 leading digit summary aggregation entry.

In a preferred embodiment, a service quality monitoring system provides a stand-alone CDR processing system that is separate from, and external to, a network monitoring system. The service quality monitoring system provides performance monitoring statistics based on CDRs that are acquired from a network surveillance system. The CDR data can be analyzed through a real-time interface or through a historical reporting tool. The network quality monitoring system is capable of monitoring any communications network, such as an SS7 network, and provides information about the network to a user or operator. The system integrates data from CDRs on a per customer and/or per dialed service basis and provides historical performance data that helps to ensure that optimum service quality is provided to network customers. The service quality application receives CDRs that are streamed from the network monitoring system and processes the CDRs by filtering, analyzing and storing the CDR information. CDRs are stored in a database that is accessed by the historical reporting tool to generate user-requested reports. The received CDRs are also filtered and analyzed as they are received to generate real-time reports and/or alarms for the user or operator.

The quality monitoring system software may run on one or more system components. A dedicated CDR server may be required if a high volume of historical data is expected The data on the CDR server can be stored in an associated database. The CDR server includes one or more processors that are responsible for collecting and processing CDRs and for generating performance monitoring statistics. The CDR server also provides data to users and operators upon request. The amount of data to be stored and the volume of traffic expected determine the processing power that is required in the CDR server.

Users access the historical reports and real-time data through a PC client. The service quality monitoring system may also include a web-reporting tool and a web server that are used to provide platform independent access to the historical data. As a result, the historical database and real-time data may be accessed remotely, for example, via a private computer network, such as a Local Area Network

(LAN) or Wide Area Network (WAN), or via a global computer network, such as the Internet or World Wide Web.

A stand-alone, independent service quality monitoring system is illustrated in FIG. **5**. Communication network nodes **51** and **52** may be any two nodes that are connected by link **53**. Nodes **51** and **52** are each typically connected to additional nodes by other links which are not shown in FIG. **5** to simplify the drawing. Calls, transactions and/or other communication messages are exchanged between nodes **51** and **52** over link **53**. Individual ones of these messages are associated with any number of unrelated calls, transactions and other communications across a network. Monitor **501** captures these messages as described herein and correlates related messages into communications detail records, such as call or transaction records. Monitor **501** may capture messages from one or more links. A number of such monitors **501** that are coupled to all or most of the links in a communication network are used to capture substantially all the messages in the network. Monitors **501** may be linked to each other and they may exchange data via an inter-monitor bus (not shown) as described in the related applications.

Monitoring unit **501** is coupled to monitoring system server **503**, which maintains system configuration information. Monitor **501** may send CDRs and other message data to server **503** for further processing or storage. Network operators access monitors **501** and monitoring system server **503** via workstation **502**. Operators use workstation **502** to configure the monitoring system and to access CDR or other data. Workstation **502** may also be used to configure and receive alarms and real-time performance statistics.

Server **502** acts as the central information point for all entities in the network monitoring system. Any entity, such as monitor **501**, that needs common information can obtain that information from a database on server **502**. Server **502** maintains the configuration information for all the applications that are provided by monitors **501**. Network information that is provided by the user or operator is stored in a configuration file on server **502**. A configuration process downloads configurations to specific monitors **501**, for example, using SNMP. The configuration process also accepts new configurations and receives changes to old configurations from workstation **502** or external client **504**. New or updated information is immediately relayed to the appropriate monitors **501**.

Monitor **501** may be configured to send CDR data to external equipment, such as a CDR server **506**. CDR server **506** may be an external network service quality application. External workstation **504** and web browser **505** allow users to access the CDR information and performance statistics remotely. Monitor **501** collects message data for call and transaction legs from a tracking process. Then monitor **501** correlates associated messages and generates a communication detail record or CDR. The CDR contains summary information of the call, transaction or other communication. Monitor **501** then generates a binary CDR stream that is sent via Transmission Control Protocol/Internet Protocol (TCP/IP) to devices that are listed in a configuration profile. The configuration profile may use pointcodes to designate external devices or the CDR profile may be global. A unique identifier is generated for every CDR to guarantee system-wide uniqueness. The binary CDR is sent to the external device, such as to CDR server **506**, as soon as it is created. The CDRs can also be stored on monitor **501** and sent at regular intervals or at a preset time. Typically, the CDR information is sent as call or transaction legs are closed.

Server **503** may send a configuration profile to monitor **501** directing the monitor to send CDRs to server **506** for storage in a historical statistics database. Monitor **501** can also be configured to deliver real-time performance monitoring statistics to an application running on workstation **504**. Workstation **504** can access historical CDR statistics from server **506** in addition to receiving real-time data.

Attempt-based statistics are collected and stored in the historical database at server **506**. Service quality applications can generate performance statistics based on the digits in the stored CDRs. The digits may include all of, or part of, the calling number, dialed number or translated number. Historical data can be provided for indexes that are digit-based or based on pointcode by simplex digit. Attempt-based statistics are collected and stored in a historical database for CDRs based on complex digit combinations. The digit combinations for user provisioned calling numbers may be based on, for example, calling number by dialed number. For user provisioned dialed number, the statistics may be collected and stored based on, for example, dialed number by calling number, dialed number by translated number, or translated number by calling number. In the preferred embodiment, historical data for a complex index will be CDR based only and not pointcode keyed.

The service quality monitoring system also may include a web-reporting tool. Web browser client **505** is used to provide platform independent access to both real-time and historical CDR data. The historical database is accessible from any standard web browser via the web-reporting tool **505**. This allows the user to execute and view reports remotely via the web interface.

Completed CDRs are available at call-completion. Call statistics for CDRs are accumulated after delivery of the CDR to CDR server **506**. Pointcode by simplex digit statistics are accumulated on monitors **501** and retrieved by CDR server **506** on a user-defined interval basis. In a preferred embodiment, all data is accumulated based on call origination time. Statistics applicable to a current configuration are collected continuously. Real-time data is typically pointcode keyed on calling party and dialed number.

Table 7 illustrates the type of statistics that can be generated from the historical CDR data on CDR server **506**.

TABLE 7

Total Call Attempts
Total Calls Answered
Total Calls with Address Complete
Failed Calls
Release Cause
Abnormal Release
User Busy
Normal Release
Circuit Unavailable
Network Congestion
Network Failures
Average Setup Time
Average Hold Time
Average Conversation Time

Table 8 illustrates the type of information that may be used to configure a profile for the service quality monitoring system.

TABLE 8

Provisioning index type	Calling numbers
	Dialed numbers
Number Provisioning	Telephone numbers to be tracked
	Complete number
	Partial number

TABLE 8-continued

	Partial number with trailing wildcard
	Alias name
Results Required, Real-Time	Pointcode by calling number performance
	Pointcode by Dialed number performance
Results Required, Historical	Calling number by Dialed number performance
	Pointcode by Calling number performance
	Calling number performance on digits only
	Pointcode by Dialed number performance
	Dialed number performance on digits only
	Dialed number by translated number performance
	Dialed number by calling number performance
	Translated number by calling number performance
Filtering of pointcodes	
Filtering of signaling groups	
Application type used within the network	
Number of digits for secondary number indexes	

Operators can create their own queries or modify existing queries. Reports that are generated from the historical database can be viewed, printed or saved using a standard PC. Using web-based or global access, a report can be generated through any standard web-browser. Users can also configure hourly, daily, weekly or monthly reports that are automatically generated. These prescheduled reports can be sent to specific users via electronic mail.

Preconfigured reports are also available to the users. These reports depend upon the index tracking options, such as calling party or dialed number, that are available to the operator. The reporting tool also allows users to create new reports using the existing data in the database. For example, a report could be created for the ratio results of a comparison of any measurements. Other reporting features include formatting, filtering and other options.

Users can format Reports using thresholds, sorting and graphing parameters. Reports can be configured to highlight data that exceeds a user defined threshold. Data results can be sorted in ascending or descending order for any parameter. Users can graphically display report data on workstation **502** or **504**.

Users can configure the reports so that the CDR data is filtered to show selected parameters only. Threshold filtering can also be used to create reports that show data that exceeds a user defined threshold. Time range filtering can be used to shows data within a user-selected period of time. Digits filtering is used to show CDR data for numbers that have a user-defined digits prefix. Digit level filtering aggregates data row totals using a user-defined digit level. Reports can also be configured to provide a snapshot of the service performance data and to provide different levels of summary totals for various report types.

Number alias filtering displays data that is defined by a user-selected number alias. A series of numbers or a group of numbers can be assigned to a number alias or a service name that is defined by the user. Reports can then be generated using the alias name. The statistics for all of the numbers that belong to a selected number alias or service name will be aggregated together. In a preferred embodiment, a report selection macro in the form of a dialog is provided for the user to select a report to open based on report type, filter type (digit, alias, alias/digit), and aggregation level. Standard reports and graphs including 'ranking' reports to show the 'top **10**' disabled or dialing numbers for a specific measurement. The reporting tool supports user-defined exclusion entries, for example, by providing aggregate measurements for all dialed numbers in a selected NPA-NXX except for a specific number or numbers.

Reporting and measuring of the CDR data allows users to define any number of digit combinations. A leading digit summary is defined as an aggregation of CDRs associated with a selected prefix. This parameter returns a composite result on numbers having the selected prefix, instead of providing results for each discrete number within the number range. For example, an entry of 972 returns a single measurement for all calls with the 972 prefix. An entry of 972-578-0000 returns two measurements; a measurement for 972-578-0000 as well as a measurement in the 972 leading digit summary aggregation entry.

The quality monitoring system manages the CDR profiles so that they do not generate several CDRs for the same call leg. The system also correlates multiple related CDRs to produce a single CDR per event. The CDR processing system is also capable of processing translated numbers. When the call legs are delivered to the CDR analysis system, then the calling, dialed number and translated number fields are populated.

Users can set up a statistical event alarm on any real-time profile so that statistical event alarms are displayed when alarm conditions are detected.

In a preferred embodiment, the minimum granularity is five minutes of data. The granularity can also be configured to be traffic dependent. All data is entered in appropriate hourly, daily, weekly, and monthly tables. The quantity of hourly, daily, weekly, and monthly tables stored in the database is dependent upon the amount of hard disk space procured with the CDR server hardware **506**. The system is capable of using additional on-line CDR storage.

The quality monitoring system application can be configured to sample only a percentage of CDRs to populate the historical database. Users can select an appropriate sampling rate through the CDR configuration profile set-up. The system application hardware is configured as per the customers sampling requirements. All of the quality statistics may be stored for the sampled set of CDRs.

The CDR analysis system also has analysis tools to provide users with the ability to drill down and analyze data at specific geographic, node, link, call, transaction or message levels.

Although the invention has been described with respect to an SS7 system, it will be understood that the present invention may be adapted to monitor the quality of service provided on any communications network.

The present invention and its advantages have been described in detail herein, however, it should be understood that various changes, substitutions and alterations can be made herein without departing from the spirit and scope of the invention as defined by the appended claims.

What is claimed is:

1. A method for monitoring the service quality in a communications network, wherein a communication monitoring system generates communication detail records for communications on said network, the method comprising the steps of:

capturing, at monitoring units, messages from one or more communication links in the communication network, wherein each message corresponds to a specific one of a plurality of calls or transactions, and

wherein a plurality of messages are associated with each of said calls or transactions;

identifying a first-detected message for a particular call or transaction;

designating a monitoring unit that detected said first-detected message as an anchor monitor for the particular call or transaction;

forwarding any other messages associated with the particular call or transaction to the anchor monitor;

correlating, at the anchor monitor, all of the messages associated with the particular call or transaction into a call detail record for the particular call or transaction, wherein the call detail record comprises all of the messages associated with the particular call or transaction that have been captured by all monitors from all links in the communications network;

transmitting said call detail records directly from said anchor monitor to a quality of service application;

filtering said call detail records to generate a historical report of service quality on said network, wherein said call detail records are filtered according to a user configured profile to generate said historical reports; and

filtering said call detail records as they are received to generate real-time service quality reports.

2. The method of claim **1** further comprising the step of:

storing the call detail records at said anchor monitor upon a loss of communication between said anchor monitor and said quality of service application.

3. The method of claim **1** wherein said call detail records are Transaction Control Application Part Call Detail Records (TCAP CDR).

4. The method of claim **1** further comprising the step of:

allowing a user to modify said configuration profile via a global computer network.

5. The method of claim **1** further comprising the steps of:

monitoring said real-time service quality reports for alarm conditions; and

notifying a user when an alarm condition is detected.

6. The method of claim **5** wherein said real-time service quality reports are monitored using a user configured profile to detect said alarm conditions.

7. A system for monitoring service quality in a communications network having associated network monitoring equipment for monitoring communications across said network, said system comprising:

communications network monitoring devices coupled to communication links in the communication network, each of the monitoring device comprising:

means for capturing messages from said communication links, wherein each message corresponds to a specific one of a plurality of calls or transactions, and wherein a plurality of messages are associated with each of said calls or transactions;

means for identifying a first-detected message for a particular call or transaction;

means for designating one of said monitoring devices as an anchor monitor for the particular call or transaction, if that monitor detected the first-detected message;

means for forwarding other messages associated with the particular call or transaction to the anchor monitor;

means for correlating all of the messages associated with the particular call or transaction into a call detail record for the particular call or transaction, wherein the call detail record comprises all of the messages associated with the particular call or transaction that have been captured by all monitors from all links in the communications network;

means for transmitting said call detail records directly from said anchor monitor to a service quality application processor that is separate from said communications network monitoring devices;

a database associated with said processor for storing said communication detail records;

means for filtering said communication detail records using a user defined profile to identify selected ones of said communication detail records;

means for generating a report of historical network quality using said selected ones of said communication detail records; and

means for filtering said communications detail records to identify communication detail records to be displayed to a user in real-time.

8. The system of claim **7** further comprising:

means to display said historical reports to said user; and

means to display said call detail records to said user in real-time.

9. The system of claim **8** wherein said historical display means and said real-time display means are the same device.

10. The system of claim **8** wherein said display means provide a graphical user interface for said users.

* * * * *

US006389468B1

(12) **United States Patent**
Muller et al.

(10) **Patent No.: US 6,389,468 B1**
(45) **Date of Patent: May 14, 2002**

(54) **METHOD AND APPARATUS FOR DISTRIBUTING NETWORK TRAFFIC PROCESSING ON A MULTIPROCESSOR COMPUTER**

(75) Inventors: **Shimon Muller**, Sunnyvale; **Denton E. Gentry, Jr.**, Fremont, both of CA (US)

(73) Assignee: **Sun Microsystems, Inc.**, Palo Alto, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/259,445**

(22) Filed: **Mar. 1, 1999**

(51) **Int. Cl.**[7] **G06F 15/173**; G06F 15/16

(52) **U.S. Cl.** **709/226**, 709/235

(58) **Field of Search** 709/224, 249, 709/250, 226, 229, 235

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,414,704	A	5/1995	Spinney	370/60
5,583,940	A	12/1996	Vidrascu et al.	380/49
5,684,954	A	11/1997	Kaiserswerth et al.	395/200.2
5,748,905	A	5/1998	Hauser et al.	395/200.79
5,758,089	A	5/1998	Gentry et al.	395/200.64
5,778,180	A	7/1998	Gentry et al.	395/200.42
5,778,414	A	7/1998	Winter et al.	711/5
5,793,954	A	8/1998	Baker et al.	395/200.8
5,870,394	A	2/1999	Oprea	370/392
6,014,699	A *	1/2000	Ratcliff et al.	709/224

FOREIGN PATENT DOCUMENTS

EP	0 447 725	9/1991	G06F/15/16
EP	0 573 739	12/1993	H04L/12/56
EP	0 853 411	7/1998	H04L/29/06
EP	0865180	9/1998	H04L/12/56
WO	WO 95/14269	5/1995	G06F/7/08
WO	WO 97/28505	8/1997	G06F/13/14
WO	WO 99/00737	1/1999	G06F/13/00
WO	WO 99/00945	1/1999	H04L/12/46
WO	WO 99/00948	1/1999	H04L/12/56
WO	WO 99/00949	1/1999	H04L/12/56

OTHER PUBLICATIONS

T.Suda. "Measuring the Performance of Parallel Message Based Process Architectures". INFOCOM 95. Fourteenth Annual Joint Conference. IEEE Computer and Communications Societies. Apr 2–6, 1995 ISBN: 0–8186–6990–x. Page(s) 624–633.*

D.J. Yates. "Networking Support For Large Scale Multiprocessor Server". (HPCS'95), 1995 Third IEEE workshop. pp. 153–157. Aug. 23–25, 1995.*

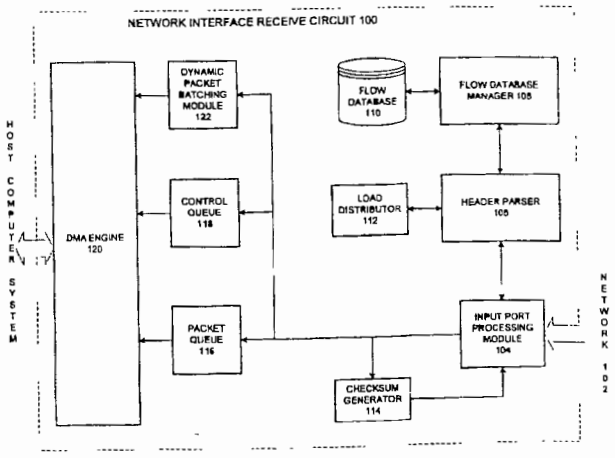
(List continued on next page.)

Primary Examiner—Glenton B. Burgess
Assistant Examiner—Kenneth W. Fields
(74) *Attorney, Agent, or Firm*—Park, Vaughan & Fleming LLP

(57) **ABSTRACT**

A system and method are provided for distributing or sharing the processing of network traffic (e.g., through a protocol stack on a host computer system) received at a multiprocessor computer system. A packet formatted according to one or more communication protocols is received from a network entity at a network interface circuit of a multiprocessor computer. A header portion of the packet is parsed to retrieve information stored in one or more protocol headers, such as source and destination identifiers or a virtual communication connection identifier. In one embodiment, a source identifier and a destination identifier are combined to form a flow key that is subjected to a hash function. The modulus of the result of the hash function over the number of processors in the multiprocessor computer is then calculated. In another embodiment a modulus operation is performed on the packet's virtual communication connection identifier. The result of the modulus operation identifies a processor to which the packet is submitted for processing.

48 Claims, 49 Drawing Sheets



US006405251B1

(12) **United States Patent**
Bullard et al.

(10) **Patent No.: US 6,405,251 B1**
(45) **Date of Patent: Jun. 11, 2002**

(54) **ENHANCEMENT OF NETWORK ACCOUNTING RECORDS**

(75) Inventors: **William Carter Carroll Bullard**, New York, NY (US); **Kevin Farrell**, Windham, NH (US); **Steven Ball**, Sandown, NH (US); **Daniel O. Mahoney, II**, Rollinsford, NH (US)

(73) Assignee: **Nortel Networks Limited**, Quebec (CA)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/276,201**

(22) Filed: **Mar. 25, 1999**

(51) **Int. Cl.**[7] **G06F 13/00**
(52) **U.S. Cl.** **709/224**
(58) **Field of Search** 709/200, 210, 709/223, 224

(56) **References Cited**

U.S. PATENT DOCUMENTS

3,463,222	A	8/1969	Grames	165/10
4,396,058	A	8/1983	Kurschner et al.	165/8
4,449,573	A	5/1984	Pettersson et al.	165/10
4,744,410	A	5/1988	Groves	165/10

OTHER PUBLICATIONS

XACCT Usage Overview, XACCT Technologies, 1997.
HP and Cisco Deliver Internet Usage Platform and Billing and Analysis Solutions (http://www.hp.com/smartinternet/press/prapr28.html), Hewlett Packard Company, 1998.
Article, Quadri, et al., Internet Usage Platform White Paper (http://www.hp.com/smartinternet/solutions/usagewp), Hewlett Packard Company, 1998.
Article, Strategies for Managing IP Data (http://www.-hp.com/smartinternet/press/not.html), Hewlett Packard Company Undated.
Article, Nattkemper, HP and Cisco Deliver Internet Usage and Billing Solutions (http://www.interex.org/hpworldnews/hpW806.html), Hewlett Packard Company, Jun. 1, 1999(?).
HP Smart Internet Billing Solution (http://hpcc925.external.hp.com/smartinternet/solutions/usagebilling.html), Hewlett Packard Company, 1998.
HP Smart Internet Usage Analysis Solution (http://www.hp.com/smartinternet/solutions/usageanalysis.html), Hewlett Packard Company, 1998.
Press Release, New cisco IOS NetFlow Software and Utilities Boost Service Provider Revenues and Service Management Capabilities (http://www.cisco.com/warp/public/cc/cisco/mkt/gen/pr/archive/cros pr.htm), Cisco Systems, Inc., Jul. 1, 1997.
Documentation, NewFlow FlowCollector 2.0 (http://www.cisco.com/univercd/cc/td/doc/product/rtrmgmt/nfc/nfc 2 0/index.htm), Cisco Systems, Inc., 1998.

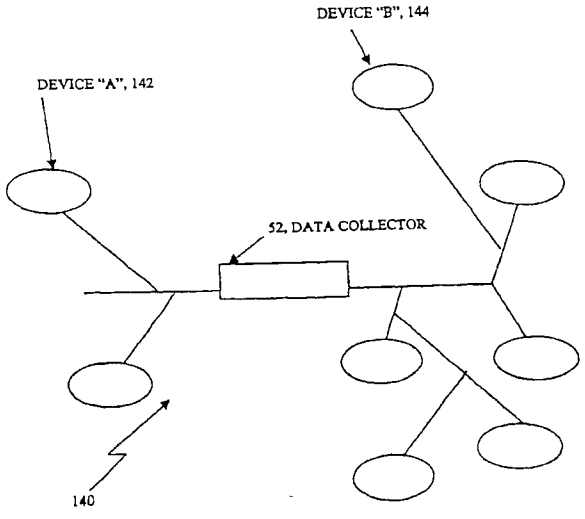
(List continued on next page.)

Primary Examiner—Robert B. Harrell
(74) *Attorney, Agent, or Firm*—Fish & Richardson P.C.

(57) **ABSTRACT**

A system for collecting and aggregating data from network entities for a data consuming application is described. The system includes a data collector layer to receive network flow information from the network entities and to produce records based on the information. The system also includes a flow aggregation layer fed from the data collection layer and coupled to a storage device. The flow aggregation layer receiving records produced by the data collector layer and aggregates received records. The system can also include an equipment interface layer coupled to the data collector layer and a distribution layer to obtain selected information stored in the storage device and to distribute the select information to a requesting, data consuming application.

22 Claims, 36 Drawing Sheets



US006404857B1

(12) **United States Patent**
Blair et al.

(10) **Patent No.: US 6,404,857 B1**
(45) **Date of Patent: Jun. 11, 2002**

(54) **SIGNAL MONITORING APPARATUS FOR ANALYZING COMMUNICATIONS**

(75) Inventors: **Christopher Douglas Blair**, Sussex; **Roger Louis Keenan**, West Sussex, both of (GB)

(73) Assignee: **Eyretel Limited**, West Sussex (GB)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/500,800**

(22) Filed: **Feb. 10, 2000**

Related U.S. Application Data

(62) Division of application No. 08/936,428, filed on Sep. 24, 1997, now abandoned.

(30) **Foreign Application Priority Data**

Sep. 26, 1996 (GB) 9620082

(51) **Int. Cl.**[7] **H04M 1/64**; H04M 15/00

(52) **U.S. Cl.** **379/67.1**; 379/85; 379/88.04; 379/135

(58) **Field of Search** 379/135, 67.1, 379/85, 88.09

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,567,512	A		1/1986	Abraham
4,924,488	A	*	5/1990	Kosich 379/135
4,969,136	A		11/1990	Chamberlin et al.
4,975,896	A		12/1990	D'Agosto, III et al.
5,260,943	A		11/1993	Comroe et al
5,274,572	A		12/1993	O'Neill et al.
5,390,243	A		2/1995	Casselman et al.
5,535,261	A	*	7/1996	Brown et al. 379/67.1
5,696,811	A	*	12/1997	Maloney et al. 379/85
5,818,907	A	*	10/1998	Maloney et al. 379/85
5,946,375	A	*	8/1999	Pattison et al. 379/85
6,035,017	A	*	3/2000	Fenton et al. 379/88.04
6,058,163	A	*	5/2000	Pattison et al. 379/85

FOREIGN PATENT DOCUMENTS

EP	0 510 412	10/1992
GB	2 257 872	1/1993

OTHER PUBLICATIONS

So–Lin Yen et al., "Intelligent MTS Monitoring System", 10/94, pp. 185–187, Scientific and Research Center for Criminal Investigation, Taiwan, Republic of China.

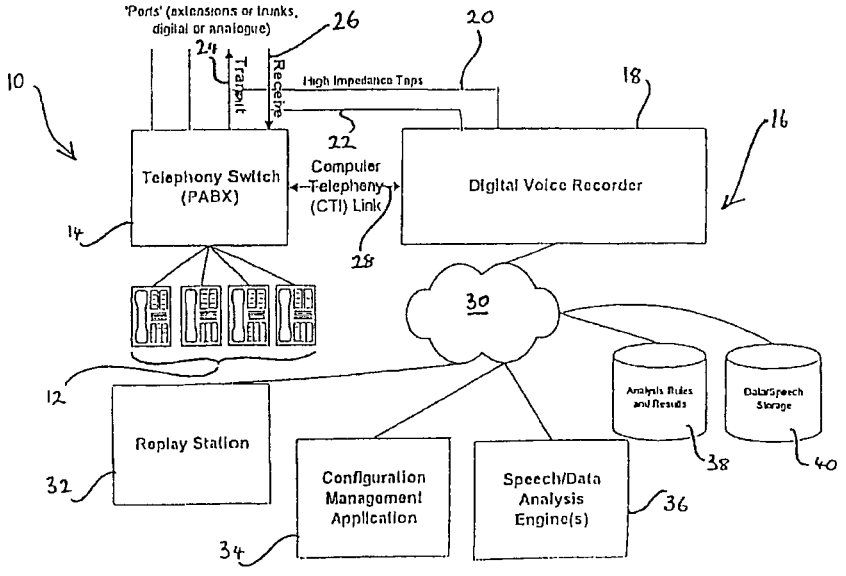
* cited by examiner

Primary Examiner—William Cumming
(74) *Attorney, Agent, or Firm*—Young & Thompson

(57) **ABSTRACT**

A signal monitoring apparatus and method involving devices for monitoring signals representing communications traffic, devices for identifying at least one predetermined parameter by analyzing the context of the at least one monitoring signal, a device for recording the occurrence of the identified parameter, a device for identifying the traffic stream associated with the identified parameter, a device for analyzing the recorded data relating to the occurrence, and a device, responsive to the analysis of the recorded data, for controlling the handling of communications traffic within the apparatus.

31 Claims, 2 Drawing Sheets



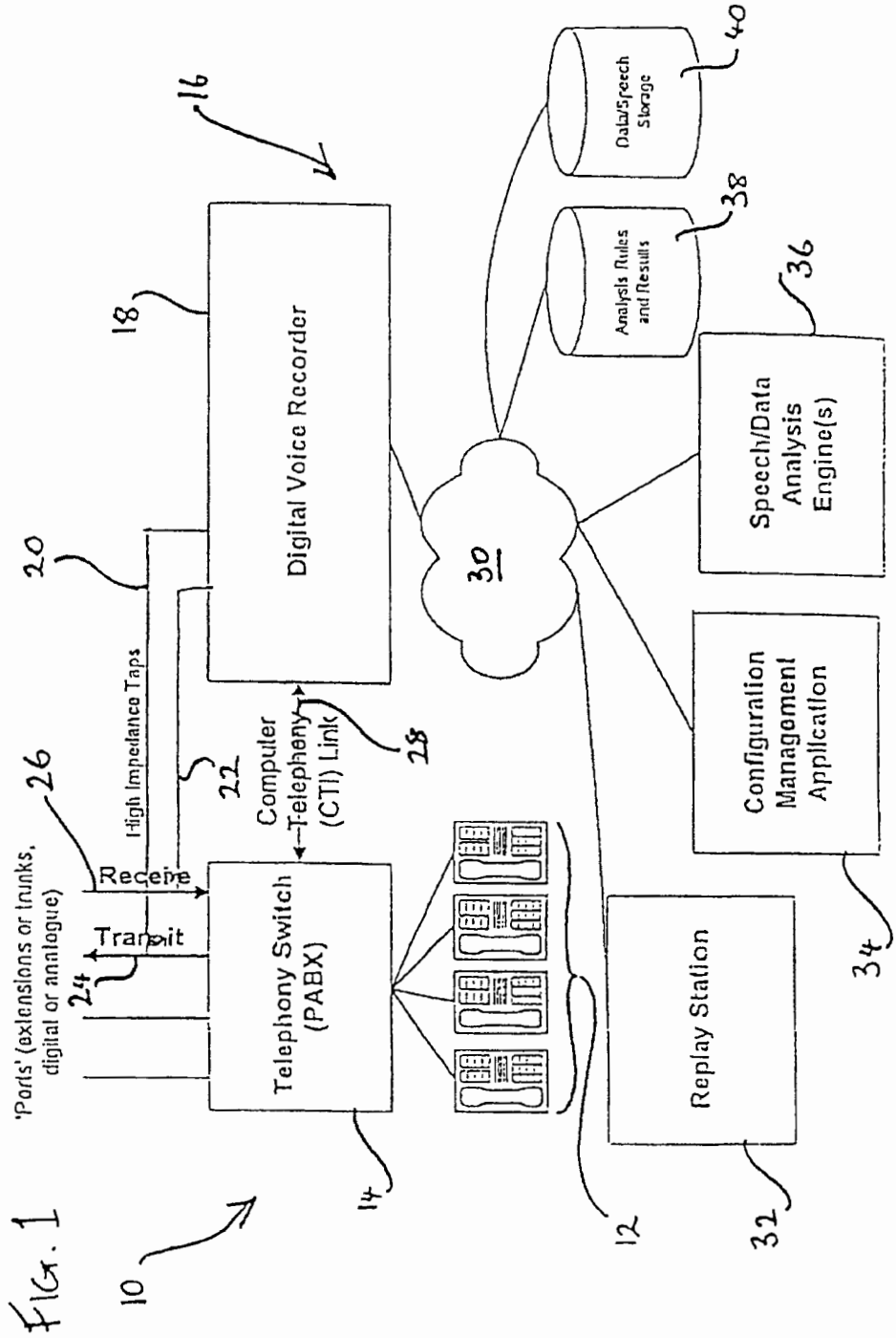

Data/Speech Storage
40
Analysis Rules and Results
38
36
Speech/Data Analysis Engine(s)
16
18
Digital Voice Recorder
20
30
Configuration Management Application
34
High Impedance Taps
22
Computer-Telephony (CTI) Link
28
26
Receive
Transmit
24
'Ports' (extensions or trunks, digital or analogue)
Telephony Switch (PABX)
14
12
Replay Station
32
10

Fig. 1

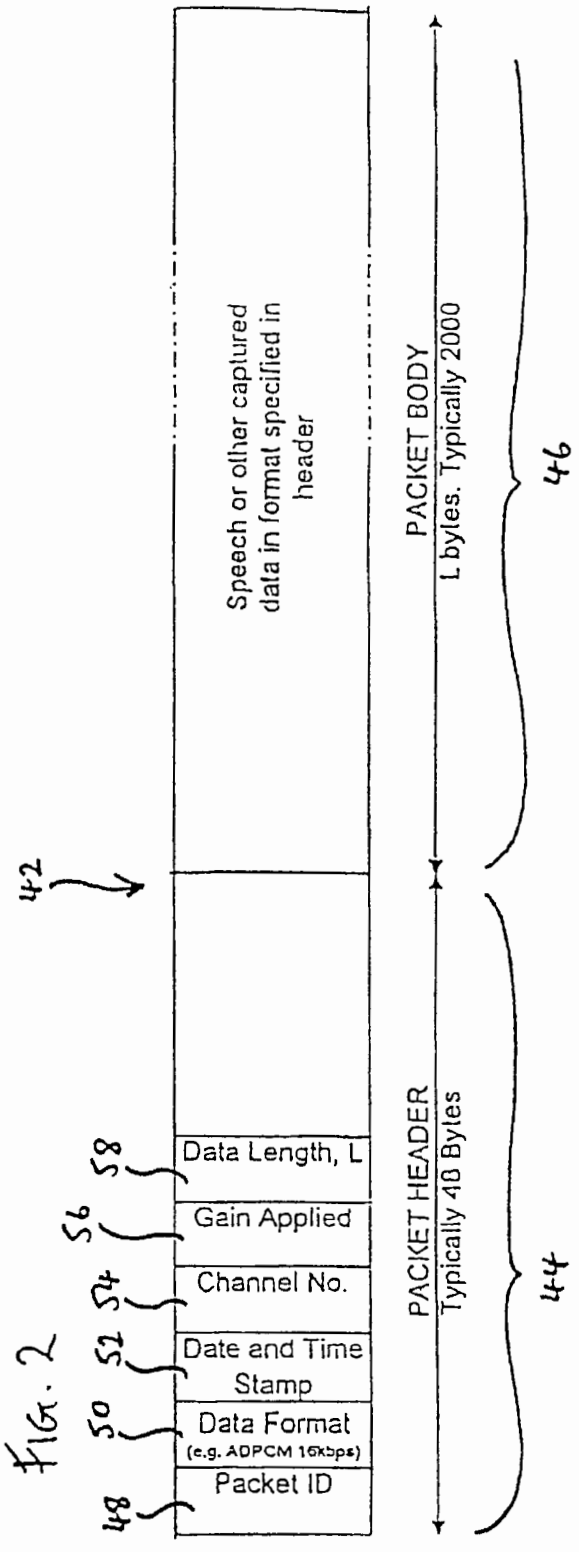

Speech or other captured data in format specified in header
Data Length, L
Gain Applied
Channel No.
Date and Time Stamp
Data Format
(e.g. ADPCM 16kbps)
Packet ID
PACKET BODY
L bytes. Typically 2000
PACKET HEADER
Typically 48 Bytes
42
46
44
58
56
54
52
50
48

Fig. 2

SIGNAL MONITORING APPARATUS FOR ANALYZING COMMUNICATIONS

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is a division of application Ser. No. 08/936,428, filed Sep. 24, 1997 now abandoned.

BACKGROUND OF THE INVENTION

The present invention relates to signal monitoring apparatus and in particular, but not exclusively to telecommunications monitoring apparatus which may be arranged for monitoring a plurality of telephone conversations.

DESCRIPTION OF THE RELATED ART

Telecommunications networks are increasingly being used for the access of information and for carrying out commercial and/or financial transactions. In order to safeguard such use of the networks, it has become appropriate to record the two-way telecommunications traffic, wether voice traffic or data traffic, that arises as such transactions are carried out. The recording of such traffic is intended particularly to safeguard against abusive and fraudulent use of the telecommunications network for such purposes.

More recently, so-called "call-centers" have been established at which operative personnel are established to deal with enquiries and transactions required of the commercial entity having established the call-center. An example of the increasing use of such call-centers is the increasing use of "telephone banking" services and the telephone ordering of retail goods.

Although the telecommunications traffic handled by such call-centers is monitored in an attempt to preserve the integrity of the call-center, the manner in which such communications networks, and their related call-centers, are monitored are disadvantageously limited having regard to the data/information that can be provided concerning the traffic arising in association with the call-center.

For example, in large call-centers, it is difficult for supervisors to establish with any confidence that they have accurately, and effectively, monitored the quality of all their staff's work so as to establish, for example, how well their staff are handling customers' enquiries and/or transaction requirements, or how well their staff are seeking to market/publicise a particular product etc.

SUMMARY OF THE INVENTION

The present invention seeks to provide for telecommunications monitoring apparatus having advantages over known such apparatus.

According to one aspect of the present invention there is provided signal monitoring apparatus comprising

means for monitoring signals representing communications traffic;

means for identifying at least one predetermined parameter by analysing the content of at least one monitored signal;

means for recording the occurrence of the identified parameter;

means for identifying the traffic stream associated with the identified parameter;

means for analysing the recorded data relating to the said occurrence; and

means, responsive to the analysis of the said recorded data, for controlling the handling of communications traffic within the apparatus.

Preferably, the means for controlling the handling of the communications traffic serves to identify at least one section of traffic relative to another.

Also, the means for controlling may serve to influence further monitoring actions within the apparatus.

Advantageously, the analysed contents of the at least one signal comprise the interaction between at least two signals of traffic representing an at least two-way conversation. In particular, the at least two interacting signals relate to portions of interruption or stiltedness within the traffic.

Preferably, the means for monitoring signals can include means for recording signals.

Preferably, the means for recording the occurrence of the parameter comprises means for providing, in real time, a possibly instantaneous indication of said occurrence, and/or comprises means for storing, permanently or otherwise, information relating to said occurrence.

Dependent upon the particular parameter, or parameters, relevant to a call-center provider, the present invention advantageously allows for the improved monitoring of traffic so as to identify which one(s) of a possible plurality of data or voice interactions might warrant further investigation whilst also allowing for statistical trends to be recorded and analysed.

The apparatus is advantageously arranged for monitoring speech signals and indeed any form of telecommunication traffic.

For example, by analysing a range of parameters of the signals representing traffic such as speech, data or video, patterns, trends and anomalies within a plurality of interactions can be readily identified and these can then be used for example, to influence future automated analysis, and rank or grade the conversations and/or highlight conversations likely to be worthy of detailed investigation or playback by the call-center provider. The means for monitoring the telecommunications signals may be advantageously arranged to monitor a plurality of separate two-way voice, data or video conversations, and this makes the apparatus particularly advantageous for use within a call-center.

The means for monitoring the telecommunications signals advantageously arranged to monitor the signals digitally by any one variety of appropriate means which typically involve the use of high impedance taps into the network and which have little, or no, effect on the actual network.

It should of course be appreciated that the invention can be arranged for monitoring telecommunications signals transmitted over any appropriate medium, for example a hard-wired network comprising twisted pair or co-axial lines or indeed a telecommunications medium employing radio waves.

In cases where the monitored signal is not already in digital form, the apparatus can advantageously include analogue/digital conversion means for operating on the signal produced by the aforesaid means for monitoring the telecommunications signals.

It should also be appreciated that the present invention can comprise means for achieving passive monitoring of a telecommunications network or call-centre etc.

The means for identifying the at least one predetermined parameter advantageously includes a Digital Signal Processor which can be arranged to operate in accordance with any appropriate algorithm. Preferably, the signal processing required by the means for identifying the at least one parameter can advantageously be arranged to be provided by spare capacity arising in the Digital Signal Processors found

within the apparatus and primarily arranged for controlling the monitoring, compression and/or recording of signals.

As mentioned above, the particular parameters arranged to be identified by the apparatus can be selected from those that are considered appropriate to the requirements of, for example, the call-centre provider.

However, for further illustration, the following is a non-exhaustive list of parameters that could be identified in accordance with the present invention and assuming that the telecommunications traffic concerned comprises a plurality of two-way telephone interactions such as conversations:

- non-voice elements within predominantly voice-related interactions for example dialling, Interactive Voice Response Systems, and recorded speech such as interactive voice response prompts, computer synthesized speech or background noise such as line noise;
- the relationship between transmissions in each direction, for example the delay occurring, or the overlap between, transmissions in opposite directions;
- the amplitude envelope of the signals, so as to determine caller anger or episodes of shouting;
- the frequency spectrum of the signal in various frequency bands;
- advanced parameters characterizing the actual speaker which may advantageously be used in speech authentication;
- measures of the speed of interaction, for example for determining the ratio of word to inter-word pauses;
- the language used by the speaker(s);
- the sex of the speaker(s);
- the presence or absence of particular words, for example word spotting using advanced speech recognition techniques;
- the frequency and content of prosody including pauses, repetitions, stutters and nonsensical utterances in the conversation;
- vibration or tremor within a voice; and
- the confidence/accuracy with which words are recognized by the receiving party to the conversation so as to advantageously identify changes in speech patterns arising from a caller.

Parameters such as the following, and having no direct relationship to each call's content, can also be monitored:

- date, time, duration and direction of call:
- externally generated "tagging" information for transferred calls or calls to particular customers;

As will be appreciated, the importance of each of the above parameters and the way in which they can be combined to highlight particular good, or bad, caller interactions can be readily defined by the call-center provider.

Advantageously, the apparatus can be arranged so as to afford each of the parameters concerned a particular weighting, or relative value.

The apparatus may of course also be arranged to identify the nature of the data monitored, for example whether speech, facsimile, modem or video etc. and the rate at which the signals are monitored can also be recorded and adjusted within the apparatus.

According to a further feature of the invention, the means for identifying the at least one parameter can be arranged to operate in real time or, alternatively, the telecommunications signals can be recorded so as to be monitored by the means for identifying at least one parameter at some later stage.

Advantageously, the means for recording the actual occurrence of the identified parameter(s) can be arranged to identify an absolute value for such occurrences within the communications network and/or call-centre as a whole or, alternatively, the aforementioned recording can be carried out on a per-conversation or a per-caller/operative basis.

The means for recording the occurrence of the identified parameter(s) can advantageously be associated means for analysing the results of the information recorded so as to identify patterns, trends and anomalies within the telecommunications network and/or call-center.

Advantageously, the means for recording the occurrence of the identified parameter(s) can, in association with the means for identifying the predetermined parameter and the means for monitoring the telecommunications signals, be arranged to record the aforementioned occurrence in each of the two directions of traffic separately.

Preferably, the means for identifying the source of the two-way traffic includes means for receiving an identifier tagged on to the traffic so as to identify its source, i.e. the particular operative within the call-centre or the actual caller. Alternatively, means can be provided within the telecommunications monitoring apparatus for determining the terminal number, i.e. the telephone number, of the operative and/or the caller.

The aforementioned identification can also be achieved by way of data and/or speech recognition.

It should also be appreciated that the present invention can include means for providing an output indicative of the required identification of the at least one predetermined parameter. Such output can be arranged to drive audio and/or visual output means so that the call-centre provider can readily identify that a particular parameter has been identified and in which particular conversation the parameter has occurred. Alternatively, or in addition, the occurrence of the parameter can be recorded, on any appropriate medium for later analysis.

Of course, the mere single occurrence of a parameter need not establish an output from such output means and the apparatus can be arranged such that an output is only provided once a decision rule associated with such parameter(s) has been satisfied. Such a decision rule can be arranged such that it depends on present and/or past values of the parameter under consideration and/or other parameters.

Further, once a particular conversation has been identified as exhibiting a particular predetermined parameter, or satisfying a decision rule associated with such parameters, the apparatus can be arranged to allow ready access to the telecommunications "line" upon which the conversation is occurring so that the conversation can be interrupted or suspended as required.

As mentioned previously, the apparatus can be arranged to function in real time or, alternatively, the apparatus can include recording means arranged particularly to record the telecommunications traffic for later monitoring and analysis.

Preferably, the apparatus includes means for reconstructing the signals of the telecommunications traffic to their original form so as, for example, to replay the actual speech as it was delivered to the telecommunications network and/or call-center.

The apparatus can therefore advantageously recall the level of amplification, or attenuation, applied to the signal so as to allow for the subsequent analysis of the originating signal with its original amplitude envelope.

Further, the apparatus may include feedback means arranged to control the means for monitoring the telecommunications signals responsive to an output from means being provided to identify the source of the conversation in

which the parameter has been identified, or the decision rule associated with the parameter has been exceeded.

A further embodiment of the present invention comprises an implementation in which means for recording and analysing the monitored signals are built into the actual system providing the transmission of the original signals so that the invention can advantageously take the form of an add-in card to an Automatic Call Distribution System or any other telecommunications system.

Also, it will be appreciated that the present invention can be advantageously arranged so as to be incorporated into a call-center and indeed the present invention can provide for such a call-center including apparatus as defined above.

In accordance with another aspect of the present invention, there is provided a method of monitoring signals representing communications traffic, and comprising the steps of:

identifying at least one predetermined parameter associated with a monitored signal:

recording the occurrence of the identified parameter: and

identifying the traffic stream in which the parameter was identified.

The invention is therefore particularly advantageous in allowing the monitoring of respective parts of an at least two-way conversation and which may include the of analysis of the interaction of those parts.

Of course, the method of the present invention can advantageously be arranged to operate in accordance with the further apparatus features defined above.

The invention is described further hereinafter, by way of example only, with reference to the accompanying drawings in which:

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a block diagram of a typical recording and analysis system embodying the present invention; and

FIG. **2** is a diagram illustrating a typical data packetisation format employed within the present invention.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

As mentioned above, the apparatus can advantageously form part of a call-centre in which a plurality of telephone conversations can be monitored so as to provide the call-centre operator with information relating to the "quality" of the service provided by the call-center operatives. Of course, the definition of "quality" will vary according to the requirements of the particular call-centre and, more importantly, the requirements of the customers to that call-centre but typical examples are how well the call-centre operatives handle customers telephone calls, or how well an Interactive Voice Response System serves customers calling for, for example, product details.

The system generally comprises apparatus for the passive monitoring of voice or data signals, algorithms for the analysis of the monitored signals and, apparatus for the storage and reporting of the results of the analysis.

Optional features can include apparatus for recording the actual monitored signals particularly if real time operation is not required, and means for reconstructing the monitored signals into their original form so as to allow for, for example, replay of the speech signal.

FIG. **1** is a block diagram of a recording and analysis system for use in association with a call-centre **10** which includes an exchange switch **14** from which four telephone terminals **12** extend: each of which is used by one of four call-centre operatives handling customer enquiries/transactions via the exchange switch **14**.

The monitoring apparatus **16** embodying the present invention, comprises a digital voice recorder **18** which is arranged to monitor the two-way conversation traffic associated with the exchange switch **14** by way of high impedance taps **20, 22** which are connected respectively to signal lines **24, 26** associated with the exchange switch **14**. As will be appreciated by the arrows employed for the signal lines **24, 26**, the high impedance tap **20** is arranged to monitor outgoing voice signals from the call-centre **10** whereas the high impedance tap **22** is arranged to monitor incoming signals to the call-center **10**. The voice traffic on the lines **24, 26** therefore form a two-way conversation between a call-centre operative using one of the terminals **12** and a customer (not illustrated).

The monitoring apparatus **16** embodying the present invention further includes a computer telephone link **28** whereby data traffic appearing at the exchange switch **14** can be monitored as required.

The digital voice recorder **18** is connected to a network connection **30** which can be in the form of a wide area network (WAN), a local area network (LAN) or an internal bus of a central processing unit of a computer.

Also connected to the network connection **30** is a replay station **32**, a configuration management application station **34**, a station **36** providing speech and/or data analysis engine(s) and also storage means comprising a first storage means **38** for the relevant analysis rules and the results obtained and a second storage means **40** for storage of the data and/or speech monitor.

FIG. **2** illustrates the typical format of a data packet **42** used in accordance with the present invention and which comprises a packet header **44** of typically 48 bytes and a packet header **46** of typically of 2000 bytes.

The packet header is formatted so as to include the packet identification **48**, the data format **50**, a date and time stamp **52**, the relevant channel number within which the data arises **54**, the gain applied to the signal **56** and the data length **58**.

The speech, or other data captured in accordance with the apparatus of the present invention, is found within the packet body **46** and within the format specified within the packet header **44**.

The high impedance taps **20, 22** offer little or no effect on the transmission lines **24, 26** and, if not in digital form, the monitored signal is converted into digital form. For example, when the monitored signal comprises a speech signal, the signal is typically converted to a pulse code modulated (PCM) signal or is compressed as an Adaptive Differential PCM (ADPCM) signal.

Further, where signals are transmitted at a constant rate, the time of the start of the recordings is identified, for example by voltage or activity detection, i.e. so-called "vox" level detection, and the time is recorded. With asynchronous data signals, the start time of a data burst, and optionally the intervals between characters, may be recorded in addition to the data characters themselves.

The purpose of this is to allow a computer system to model the original signal to appropriate values of time, frequency and amplitude so as to allow the subsequent identification of one or more of the various parameters arising in association with the signal. The digital information describing the original signals is then analysed at station **36**, in real time or later, to determine the required set of metrics, i.e. parameters, appropriate to the particular application.

A particular feature of the system is in recording the two directions of data transmission separately so allowing further analysis of information sent in each direction independently. In analogue telephone systems, this may be achieved by use of a four-wire (as opposed to two-wire) circuit whilst in digital systems, it is the norm to have the two directions of transmission separated onto separate wire pairs. In the data world, the source of each data packet is typically stored alongside the contents of the data packet.

A further feature of the system is in recording the level of amplification or attenuation applied to the original signal. This may vary during the monitoring of even a single interaction (e.g. through the use of Automatic Gain Control Circuitry). This allows the subsequent reconstruction and analysis of the original signal amplitude.

Another feature of the system is that monitored data may be "tagged" with additional information such as customer account numbers by an external system (e.g. the delivery of additional call information via a call logging port or computer telephony integration (CTI) port).

The importance of each of the parameters and the way in which they can be combined to highlight particularly good or bad interactions is defined by the user of the system. One or more such analysis profiles can be held in the system. These profiles determine the weighting given to each of the above parameters.

The profiles are normally used to rank a large number of monitored conversations and to identify trends, extremes, anomalies and norms. "Drill-down" techniques are used to permit the user to examine the individual call parameters that result in an aggregate or average score and, further, allow the user to select individual conversations to be replayed to confirm or reject the hypothesis presented by the automated analysis.

A particular variant that can be employed in any embodiment of the present invention uses feedback from the user's own scoring of the replayed calls to modify its own analysis algorithms. This may be achieved using neural network techniques or similar giving a system that learns from the user's own view of the quality of recordings.

A variant of the system uses its own and/or the scoring/ranking information to determine its further patterns of operation i.e.

- determining which recorded calls to retain for future analysis,
- determining which agents/lines to monitor and how often, and
- determining which of the monitored signals to analyse and to what depth.

In many systems it is impractical to analyse all attributes of all calls hence a sampling algorithm may be defined to determine which calls will be analysed. Further, one or more of the parties can be identified (e.g. by calling-line identifier for the external party or by agent log-on identifiers for the internal party). This allows analysis of the call parameters over a number of calls handled by the same agent or coming from the same customer.

The system can use spare capacity on the digital signal processors (DSPs) that control the monitoring, compression or recording of the monitored signals to provide some or all of the analysis required. This allows analysis to proceed more rapidly during those periods when fewer calls are being monitored.

Spare CPU capacity on a PC at an agent's desk could be used to analyse the speech. This would comprise a secondary tap into the speech path being recorded as well as using "free" CPU cycles. Such an arrangement advantageously allows for the separation of the two parties, e.g. by tapping the headset/handset connection at the desk. This allows parameters relating to each party to be stored even if the main recording point can only see a mixed signal.

A further variant of the system is an implementation in which the systems recording and analysing the monitored signals are built into the system providing the transmission of the original signals (e.g. as an add-in card to an Automatic Call Distribution (ACD) system).

The apparatus illustrated is particularly useful for identifying the following parameters:

- degree of interruption (i.e. overlap between agent talking and customer talking);
- comments made during music or on-hold periods;
- delays experienced by customers (i.e. the period from the end of their speech to an agent's response);
- caller/agent talk ratios, i.e. which agents might be talking too much.

However, it should be appreciated that the invention could be adapted to identify parameters such as:

- "relaxed/stressed" profile of a caller or agent (i.e. by determining changes in volume, speed and tone of speech)
- frequency of keywords heard (separately from agents and from callers) e.g. are agents remembering to ask follow-up questions about a certain product/ service etc; or how often do customers swear at each agent? Or how often do agents swear at customers?
- frequency of repeat calls. A combination of line, ID and caller ID can be provided to eliminate different people calling from single switchboard/business number
- languages used by callers?
- abnormal speech patterns of agents. For example if the speech recognition applied to an agent is consistently and unusually inaccurate for, say, half an hour, the agent should be checked for: drug abuse, excessive tiredness, drunkenness, stress, rush to get away etc.

It will be appreciated that the illustrated and indeed any embodiments of the present invention can be set up as follows.

The Digital Trunk Lines (e.g. T1/E1) can be monitored trunk side and the recorded speech tagged with the direction of speech. A MediaStar Voice Recorder chassis can be provided typically with one or two E1/T1 cards plus a number of DSP cards for the more intense speech processing requirements.

Much of its work can be done overnight and in time, some could be done by the DSPs in the MediaStar's own cards. It is also necessary to remove or at least recognise, periods of music, on-hold periods, IVR rather than real agents speaking etc. thus, bundling with Computer Integrated Telephony Services such as Telephony Services API (TSAPI) in many cases is appropriate.

Analysis and parameter identification as described above can then be conducted. However, as noted, if it is not possible to analyse all speech initially, analysis of a recorded signal can be conducted.

In any case the monitoring apparatus may be arranged to only search initially for a few keywords although re-play can be conducted so as to look for other keywords.

It should be appreciated that the invention is not restricted to the details of the foregoing embodiment. For example, any appropriate form of telecommunications network, or signal transmission media, can be monitored by apparatus according to this invention and the particular parameters identified can be selected, and varied, as required.

What is claimed is:

1. A signal monitoring system for monitoring and analyzing communications passing through a monitoring point, the system comprising:

a digital voice recorder (18) for monitoring two-way conversation traffic streams passing through the monitoring point, said digital voice recorder having connections (20) for being operatively attached to the monitoring point;

a digital processor (30) connected to said digital voice recorder for identifying at least one predetermined parameter by analyzing the voice communication content of at least one monitored signal taken from the traffic streams;

a recorder (38) attached to said digital processor for recording occurrences of the predetermined parameter;

a traffic stream identifier (36) for identifying the traffic stream associated with the predetermined parameter;

a data analyzer (36) connected to said digital processor for analyzing the recorded data relating to the occurrences; and

a communication traffic controller (34) operatively connected to said data analyzer and, operating responsive to the analysis of the recorded data, for controlling the handling of communications traffic within said monitoring system,

wherein said at least one predetermined parameter is an amplitude envelope of the voice communication content of the at least one monitored signal, and

said digital processor further identifies episodes of anger or shouting by analyzing amplitude envelope.

2. A signal monitoring system for monitoring and analyzing communications passing through a monitoring point, the system comprising:

a digital voice recorder (18) for monitoring two-way conversation traffic streams passing through the monitoring point, said digital voice recorder having connections (20) for being operatively attached to the monitoring point;

a digital processor (30) connected to said digital voice recorder for identifying at least one predetermined parameter by analyzing the voice communication content of at least one monitored signal taken from the traffic streams;

a recorder (38) attached to said digital processor for recording occurrences of the predetermined parameter;

a traffic stream identifier (36) for identifying the traffic stream associated with the predetermined parameter;

a data analyzer (36) connected to said digital processor for analyzing the recorded data relating to the occurrences; and

a communication traffic controller (34) operatively connected to said data analyzer and, operating responsive to the analysis of the recorded data, for controlling the handling of communications traffic within said monitoring system,

wherein said at least one predetermined parameter is a prosody of the voice communication content of the at least one monitored signal, and

the prosody content comprises a frequency and content of pauses, repetitions, stutters, and nonsensical utterances.

3. A signal monitoring system for monitoring and analyzing communications passing through a monitoring point, the system comprising:

a digital voice recorder (18) for monitoring two-way conversation traffic streams passing through the monitoring point, said digital voice recorder having connections (20) for being operatively attached to the monitoring point;

a digital processor (30) connected to said digital voice recorder for identifying at least one predetermined parameter by analyzing the voice communication content of at least one monitored signal taken from the traffic streams;

a recorder (38) attached to said digital processor for recording occurrences of the predetermined parameter;

a traffic stream identifier (36) for identifying the traffic stream associated with the predetermined parameter;

a data analyzer (36) connected to said digital processor for analyzing the recorded data relating to the occurrences; and

a communication traffic controller (34) operatively connected to said data analyzer and, operating responsive to the analysis of the recorded data, for controlling the handling of communications traffic within said monitoring system,

wherein said digital processor is a Digital Signal Processor (30) arranged to operate in accordance with an analyzing algorithm,

the at least one predetermined parameter comprises plural predetermined parameters, and

the analysis is arranged so as to afford each of the plural predetermined parameters a particular weighting or relative value.

4. The monitoring system of claim 3, wherein said communication traffic controller serves to identify at least one section of traffic relative to another so as to identify a source of the predetermined parameter.

5. The monitoring system of claim 3, wherein said communication traffic controller serves to influence further monitoring actions within the apparatus.

6. The monitoring system of claim 3, wherein the analyzed contents of the at least one monitored signal comprise the interaction between at least two signals representing an at least two-way conversation.

7. The monitoring system of claim 3, wherein the recorder operates in real time to provide a real-time indication of the occurrence.

8. The monitoring system of claim 3, wherein said digital voice recorder comprises an analog/digital convertor (18) for converting analog voice into a digital signal.

9. The monitoring system of claim 3, wherein the monitoring point is a telephone exchange switch and said connections for being operatively attached to the telephony exchange switch are attached via high impedance taps (20) to telephone signal lines (24, 26) attached to said telephony exchange switch.

10. The monitoring system of claim 3, wherein the digital processor is arranged to operate in real time.

11. The monitoring system of claim 3, further comprising a replay station (32) connected to said digital processor and arranged such that the voice communication content of the at least one monitored signal can be recorded and monitored by said digital processor for identifying the at least one parameter at some later time.

12. The monitoring system of claim 3, wherein said recorder records the occurrence of the plural predetermined parameters in each of the two directions of traffic separately.

13. The monitoring system of claim 3, wherein said traffic stream identifier comprises a means for receiving an identifier tagged onto the traffic so as to identify its source.

US006338081B1

(12) **United States Patent**
Furusawa et al.

(10) **Patent No.: US 6,338,081 B1**
(45) **Date of Patent: *Jan. 8, 2002**

(54) **MESSAGE HANDLING METHOD, MESSAGE HANDLING APPARATUS, AND MEMORY MEDIA FOR STORING A MESSAGE HANDLING APPARATUS CONTROLLING PROGRAM**

(75) Inventors: **Osamu Furusawa**, Sagamihara; **Akifumi Nakda**, Kawasaki; **Toshihiro Suzuki**, Yokohama; **Hajime Tsuchitani**, Kamakura, all of (JP)

(73) Assignee: **International Business Machines Corporation**, Armonk, NY (US)

(*) Notice: This patent issued on a continued prosecution application filed under 37 CFR 1.53(d), and is subject to the twenty year patent term provisions of 35 U.S.C. 154(a)(2).

Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/092,130**

(22) Filed: **Jun. 5, 1998**

(30) **Foreign Application Priority Data**

Jun 12, 1997 (JP) 9-154688

(51) **Int. Cl.**[7] **G06F 15/16**; G06F 15/173; G06F 9/44

(52) **U.S. Cl.** **709/202**; 709/206; 709/223, 709/317

(58) **Field of Search** 709/202, 206, 709/223, 303, 317; 707/201

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,603,031	A	2/1997	White et al.	395/683
5,611,050	A *	3/1997	Theimer et al.	709/202
5,724,575	A *	3/1998	Hoover et al.	707/10
5,768,505	A *	6/1998	Gilchrist et al.	709/202
5,790,789	A *	8/1998	Suarez	709/202
5,815,665	A *	9/1998	Teper et al	709/229
5,822,585	A *	10/1998	Noble et al.	709/202
5,826,020	A *	10/1998	Randell	709/202
5,845,267	A *	12/1998	Ronen	
5,855,008	A *	12/1998	Goldhaber et al.	709/202
5,862,490	A *	1/1999	Sasuta et al.	455/525
5,884,324	A *	3/1999	Cheng et al.	707/201
5,887,171	A *	3/1999	Tada et al.	709/303
6,012,083	A *	3/1999	Savitzky et al.	709/202
5,937,161	A *	8/1999	Mulligan et al.	709/206
5,961,594	A *	10/1999	Bouvier et al.	709/223
5,961,595	A *	10/1999	Kawagoe et al.	709/223
6,006,260	A *	12/1999	Barrick, Jr. et al.	709/202
6,047,310	A *	4/2000	Kamakura et al.	709/201
6,055,512	A *	4/2000	Dean et al.	
6,065,039	A *	5/2000	Paciorek	709/202
6,119,229	A *	9/2000	Martinez et al.	713/200

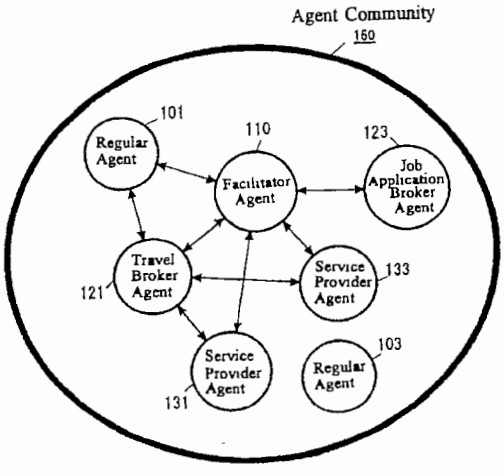
* cited by examiner

Primary Examiner—Robert B. Harrell
Assistant Examiner—Almari Romero
(74) *Attorney, Agent, or Firm*—Jerry W. Herndon; Marcia L. Doubet, Esq.

(57) **ABSTRACT**

The broker agent asks a facilitator agent to locate a service provider agent which is involved in its own job. The broker agent, upon receiving from a regular agent a message packet indicating a request for mediation of a job, analyzes the message packet to judge the outline of the requested job and applies certain conversion to the request message for sending it to the service provider agent which is relevant to the request. The broker agent receives a reply from the service provider agent and sends it to the regular agent after applying certain conversion.

20 Claims, 8 Drawing Sheets



US006477546B1

(12) **United States Patent**
Velamuri et al.

(10) **Patent No.: US 6,477,546 B1**
(45) **Date of Patent: Nov. 5, 2002**

(54) **SYSTEM AND METHOD FOR PROVIDING A TRANSACTION LOG**

(75) Inventors: **Syama S. Velamuri**, Dunwoody; **Julia Torbert**, Stone Mountain; **Prasad Nimmagadda**, Norcross, all of GA (US)

(73) Assignee: **BellSouth Intellectual Property Corporation**, Wilmington, DE (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/369,550**

(22) Filed: **Aug. 6, 1999**

Related U.S. Application Data

(62) Division of application No. 08/846,576, filed on Apr. 30, 1997.

(51) **Int. Cl.**[7] **G06F 12/00**

(52) **U.S. Cl.** **707/202**; 707/101; 707/206; 711/159

(58) **Field of Search** 707/200–204, 707/3–5, 101, 206; 711/159

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,170,480 A	*	12/1992	Mohan et al.	707/201
5,204,958 A	*	4/1993	Cheng	707/102
5,280,611 A	*	1/1994	Mohan et al.	707/8

(List continued on next page.)

FOREIGN PATENT DOCUMENTS

EP	0 350 918 A	1/1990
EP	0 750 434 A	12/1996
EP	0 751 691 A	1/1997

OTHER PUBLICATIONS

Aho et al., "Data Structures and Algorithms", Reading:Addison–Wesley, 1983, pp. 84–134 and 367–368. QA76.9.D35A38 1982.*

Kolovson, C. and Stonebraker, M. "Indexing Techniques for Historical Databases", Proceedings of the 5th International Conference on Data Engineering, Feb. 6–10, 1989, pp. 127–137.*

Ahn, I. and Snodgrass, R. "Performance Evaluation of a Temporal Database Management System", Proceedings of the 1986 ACM SIGMOD International conference on Management of Data, Jun. 1986, pp. 96–107.*

Aho, A.V. et al. Data Structures and Algorithms, Reading:Addison–Wesley, 1983, pp. 37–102. QA76.9.D35A38 1982.*

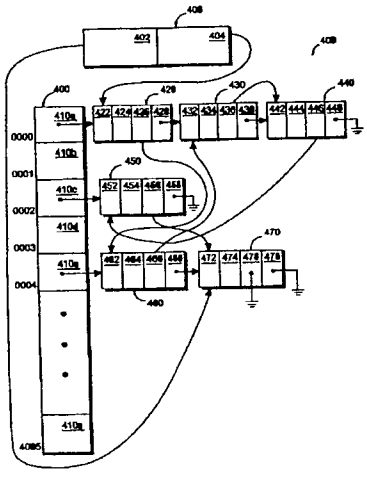
Notification Of Transmittal Of The International Search Report Or The Declaration.

Primary Examiner—Jean R. Homere
Assistant Examiner—Luke S Wassum
(74) *Attorney, Agent, or Firm*—Kilpatrick Stockton LLP

(57) **ABSTRACT**

Recording transactions using a chronological list superimposed on an indexed list. A transaction log of transaction entries is maintained as a chronological list superimposed on an indexed list. Preferably, each transaction entry includes a transaction descriptor field, a time stamp field, a chronological list pointer field and an indexed list pointer field. A first chronological list pointer points to the oldest transaction entry in the transaction log and a last chronological list pointer points to the latest transaction entry in the transaction log. The chronological list pointer field of a transaction entry points to the next oldest transaction entry. The indexed list includes a number of indexed list entry pointers. Each indexed list entry pointer corresponds to an index and points to a transaction entry with the same index. The indexed list pointer field of a transaction entry points to another transaction entry with the same index. Adding a transaction entry to the transaction log or deleting a transaction entry from the transaction log includes updating the chronological list pointers and the indexed list pointers.

20 Claims, 8 Drawing Sheets



U.S. PATENT DOCUMENTS

5,283,894 A		2/1994	Deran	707/1
5,430,719 A		7/1995	Weisser, Jr.	370/389
5,440,730 A	*	8/1995	Elmasri et al.	707/203
5,551,027 A	*	8/1996	Choy et al.	707/201
5,740,432 A	*	4/1998	Mastors	707/202
5,745,750 A	*	4/1998	Porcaro	707/102
5,832,508 A	*	11/1998	Sherman et al.	707/200
5,832,515 A	*	11/1998	Ledain et al.	707/202
5,832,518 A	*	11/1998	Mastors	707/202
5,878,410 A	*	3/1999	Zbikowski et al.	707/2
5,956,489 A	*	9/1999	San Andres et al.	709/221
5,966,708 A	*	10/1999	Clark et al.	707/101
5,996,054 A	*	11/1999	Ledain et al.	711/203
6,014,674 A	*	1/2000	McCargar	707/202
6,021,408 A	*	2/2000	Ledain et al.	707/8
6,073,109 A	*	6/2000	Flores et al.	705/8
6,092,087 A	*	7/2000	Mastors	707/202
6,148,308 A	*	11/2000	Neubauer et al.	707/203
6,219,662 B1	*	4/2001	Fuh et al.	707/3
6,230,166 B1	*	5/2001	Velamuri et al.	707/206
6,286,011 B1	*	9/2001	Velamuri et al.	707/104.1

* cited by examiner

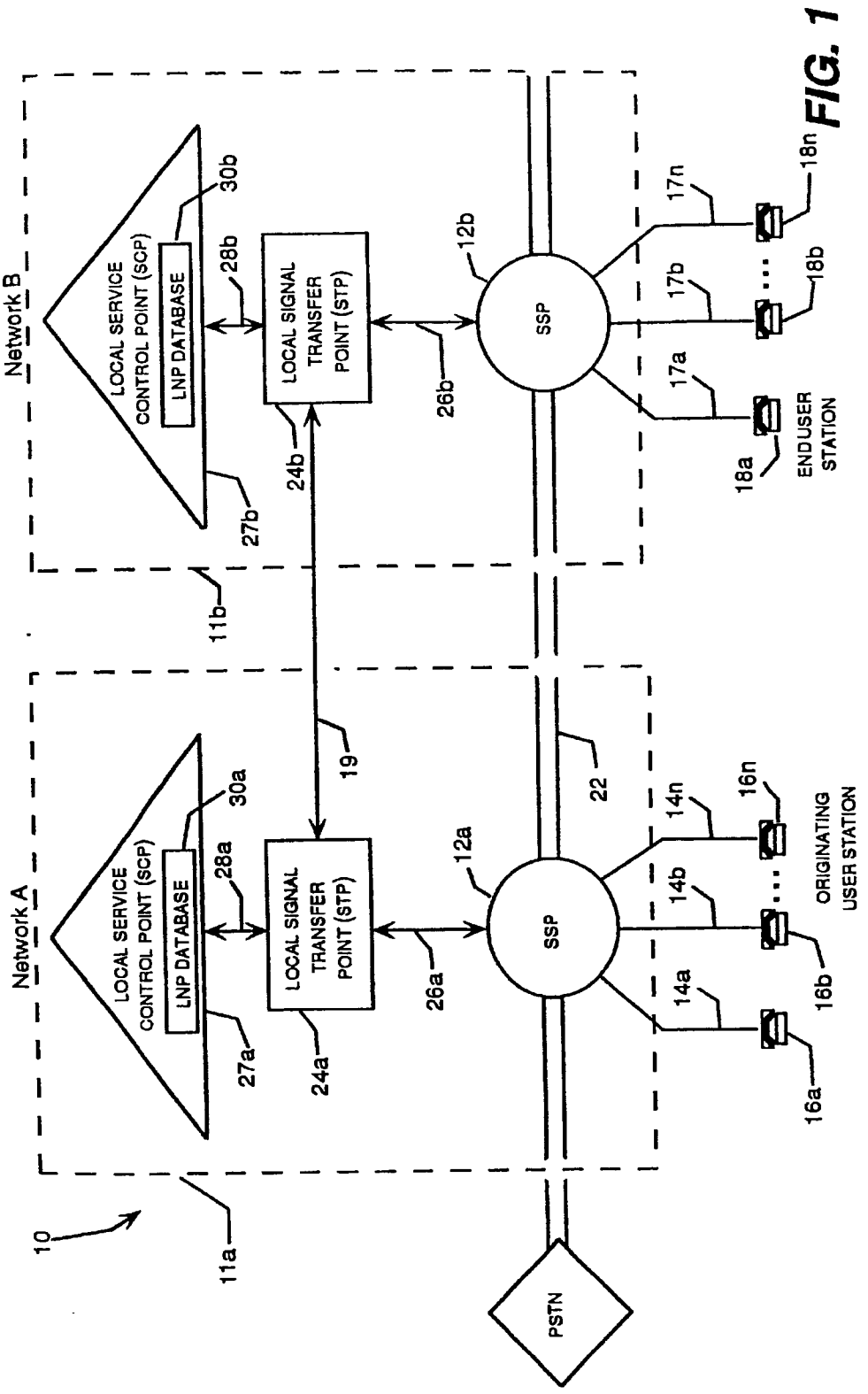

Network B
LOCAL SERVICE CONTROL POINT (SCP)
LNP DATABASE
30b
28b
LOCAL SIGNAL TRANSFER POINT (STP)
12b
SSP
17n
18n
17b
18b
17a
18a
ENDUSER STATION
24b
26b
27b
11b
19
Network A
LOCAL SERVICE CONTROL POINT (SCP)
LNP DATABASE
30a
28a
LOCAL SIGNAL TRANSFER POINT (STP)
12a
SSP
22
14n
16n
14b
16b
14a
16a
ORIGINATING USER STATION
26a
27a
24a
10
11a
PSTN

FIG. 1

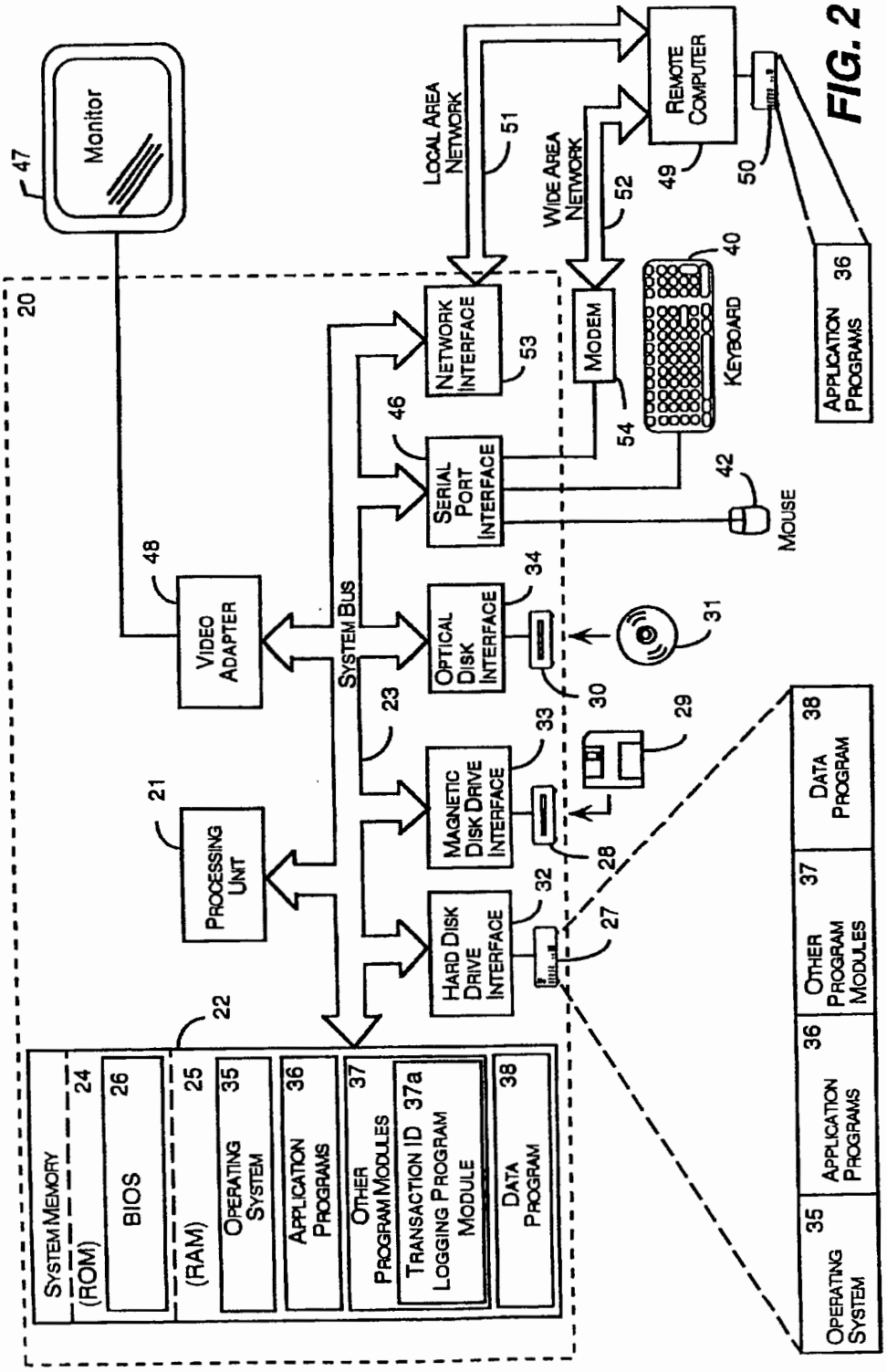

Monitor
47
20
Local Area Network
51
Wide Area Network
52
Remote Computer
49
50
Application Programs
36
Keyboard
40
Modem
54
Network Interface
53
46
Serial Port Interface
Mouse
42
48
Video Adapter
System Bus
Optical Disk Interface
34
31
30
23
Magnetic Disk Drive Interface
33
29
21
Processing Unit
28
Hard Disk Drive Interface
32
27
22
System Memory
24
(ROM)
BIOS
26
25
(RAM)
Operating System
35
Application Programs
36
Other Program Modules
37
Transaction ID
37a
Logging Program Module
Data Program
38
Data Program
38
Other Program Modules
37
Application Programs
36
Operating System
35

FIG. 2

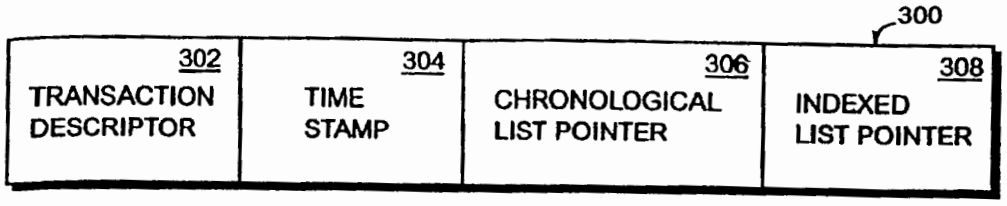

300
302
TRANSACTION DESCRIPTOR
304
TIME STAMP
306
CHRONOLOGICAL LIST POINTER
308
INDEXED LIST POINTER

FIG. 3

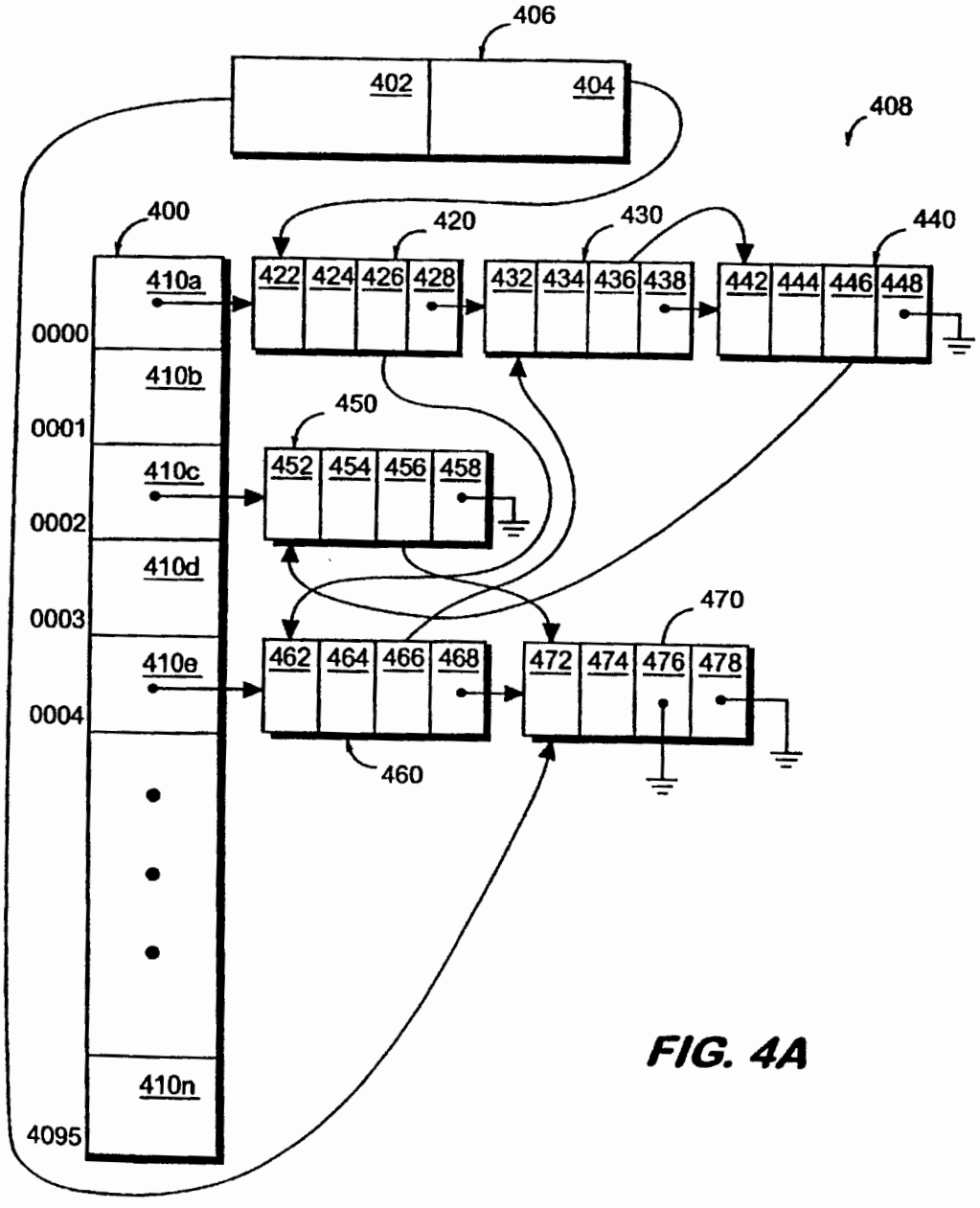

406
402
404
408
400
420
430
440
410a
422 424 426 428
432 434 436 438
442 444 446 448
0000
410b
0001
450
410c
452 454 456 458
0002
410d
0003
470
410e
462 464 466 468
472 474 476 478
0004
460
410n
4095

FIG. 4A

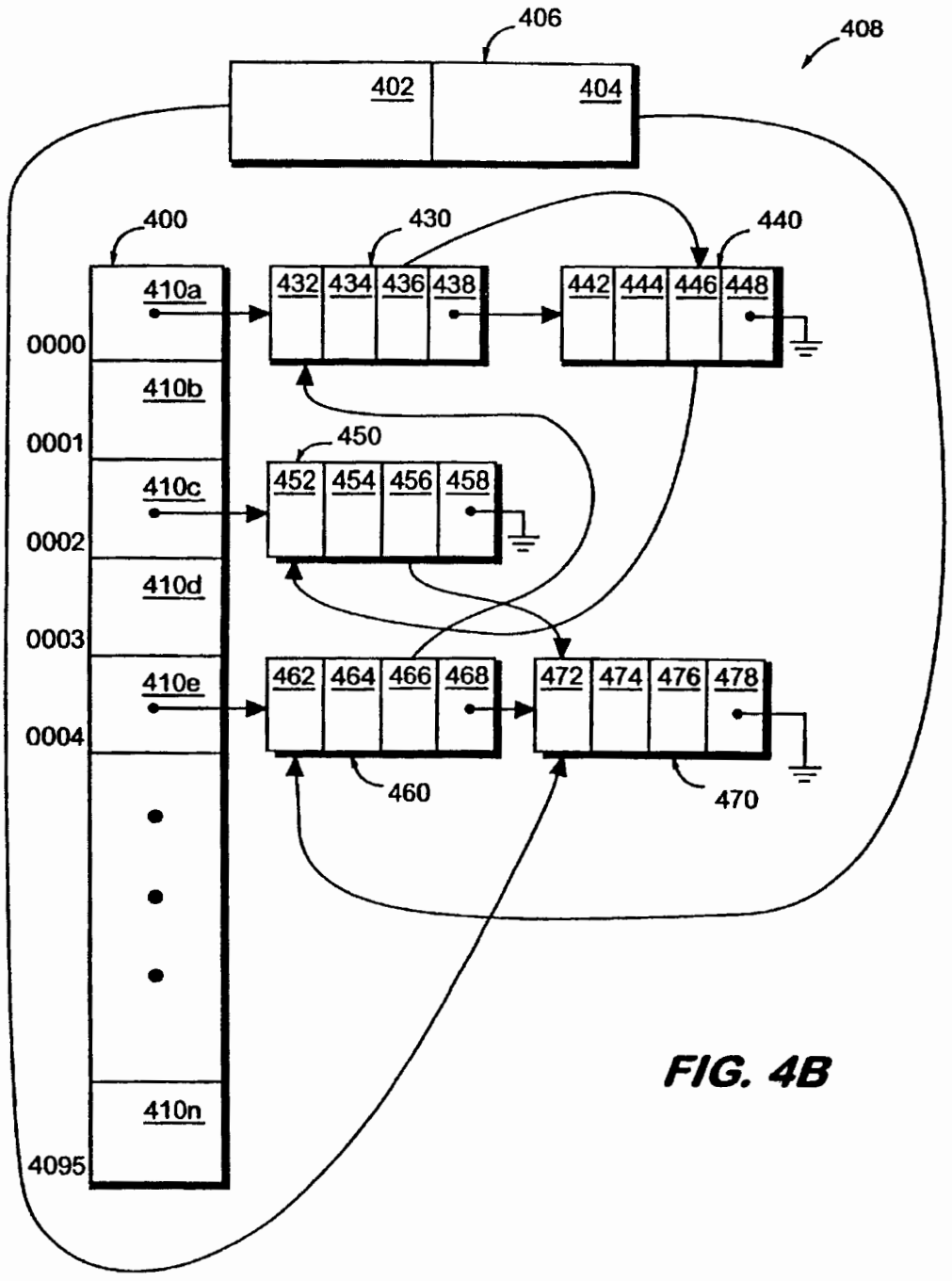

406
408
402
404
400
430
440
410a
0000
410b
0001
450
410c
0002
410d
0003
410e
0004
410n
4095
432 434 436 438
442 444 446 448
452 454 456 458
462 464 466 468
472 474 476 478
460
470

FIG. 4B

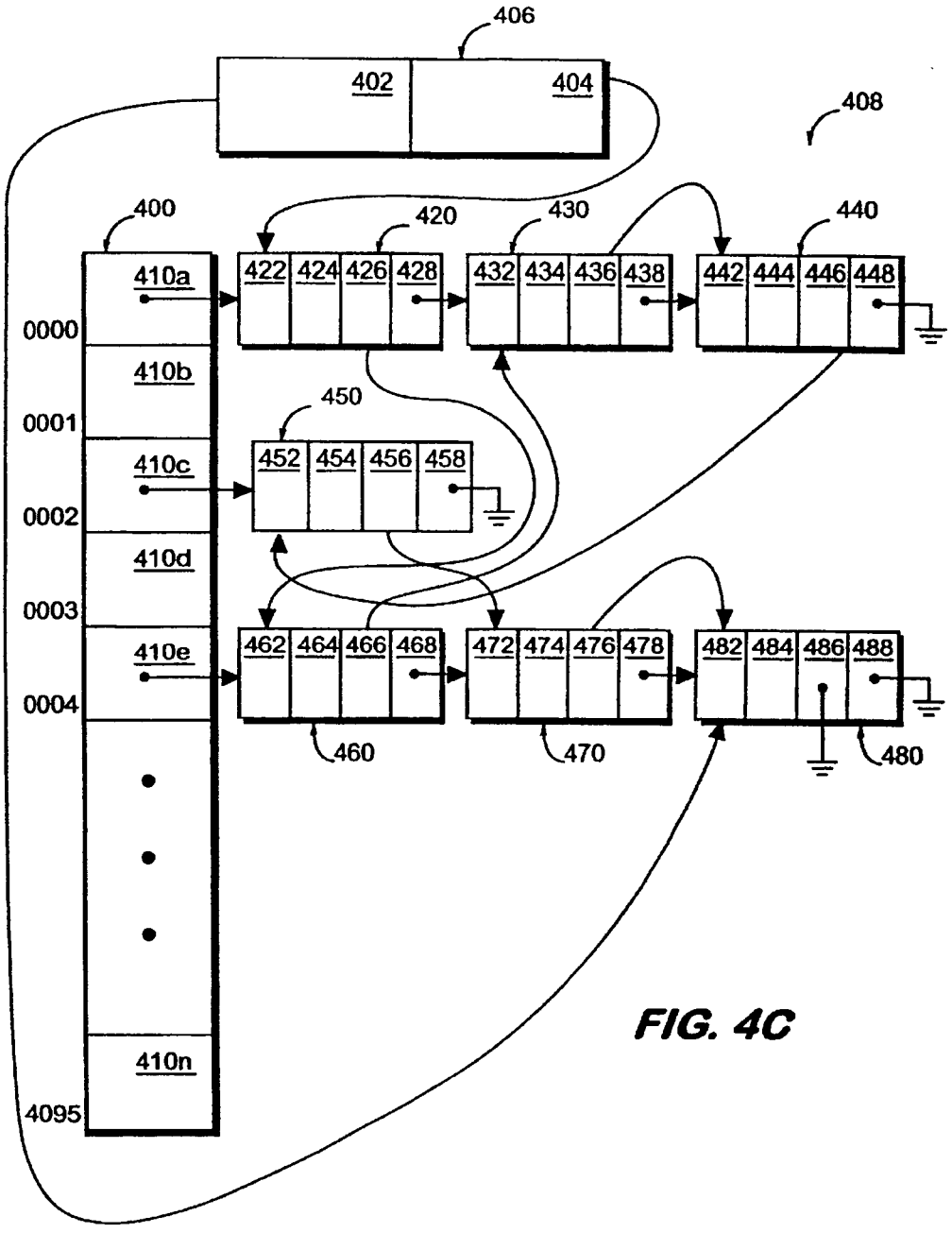

406
402
404
408
400
420
430
440
410a
422 424 426 428
432 434 436 438
442 444 446 448
0000
410b
450
0001
410c
452 454 456 458
0002
410d
0003
410e
462 464 466 468
472 474 476 478
482 484 486 488
0004
460
470
480
410n
4095

FIG. 4C

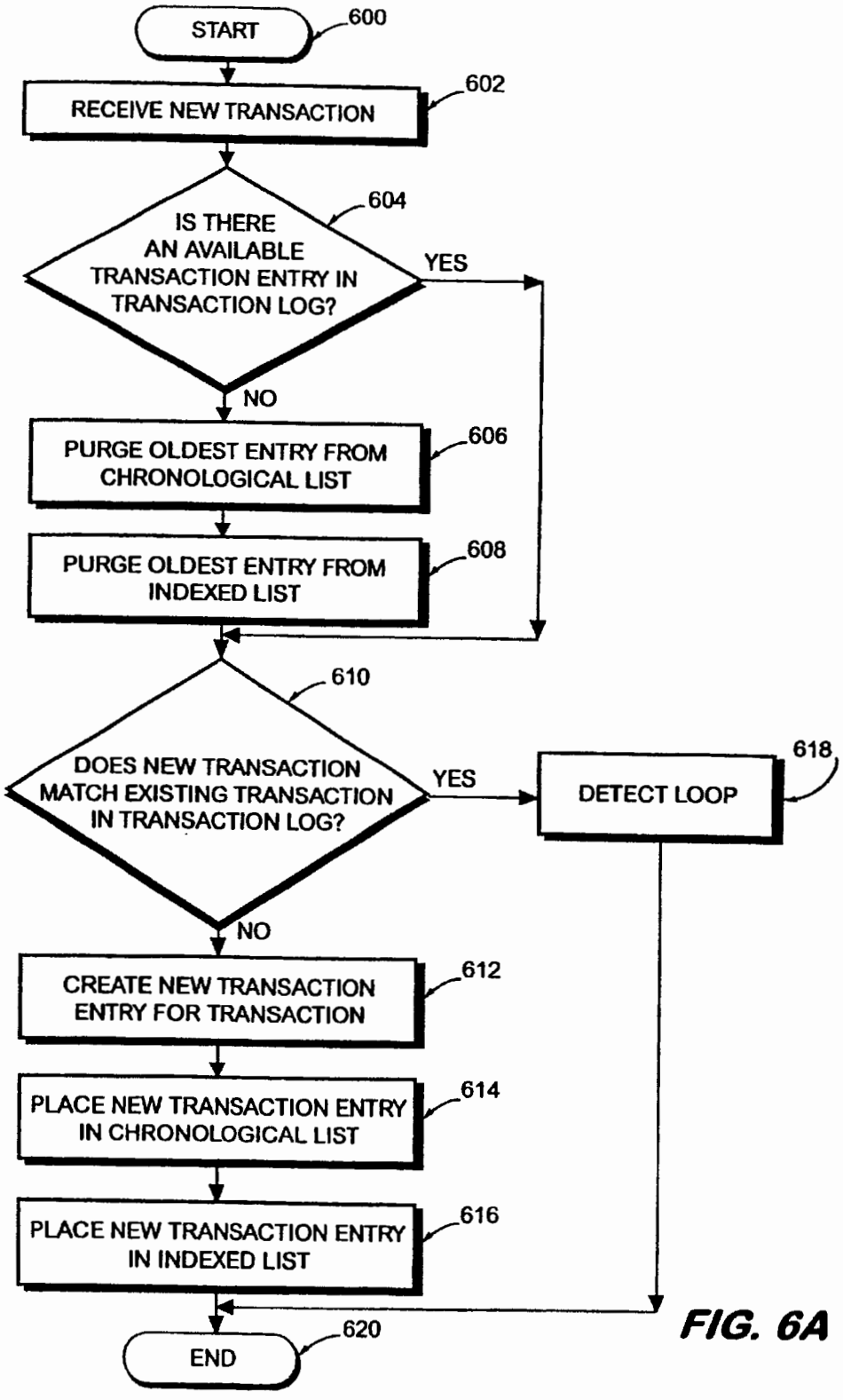

START
600
RECEIVE NEW TRANSACTION
602
IS THERE AN AVAILABLE TRANSACTION ENTRY IN TRANSACTION LOG?
604
YES
NO
PURGE OLDEST ENTRY FROM CHRONOLOGICAL LIST
606
PURGE OLDEST ENTRY FROM INDEXED LIST
608
DOES NEW TRANSACTION MATCH EXISTING TRANSACTION IN TRANSACTION LOG?
610
YES
DETECT LOOP
618
NO
CREATE NEW TRANSACTION ENTRY FOR TRANSACTION
612
PLACE NEW TRANSACTION ENTRY IN CHRONOLOGICAL LIST
614
PLACE NEW TRANSACTION ENTRY IN INDEXED LIST
616
END
620

FIG. 6A

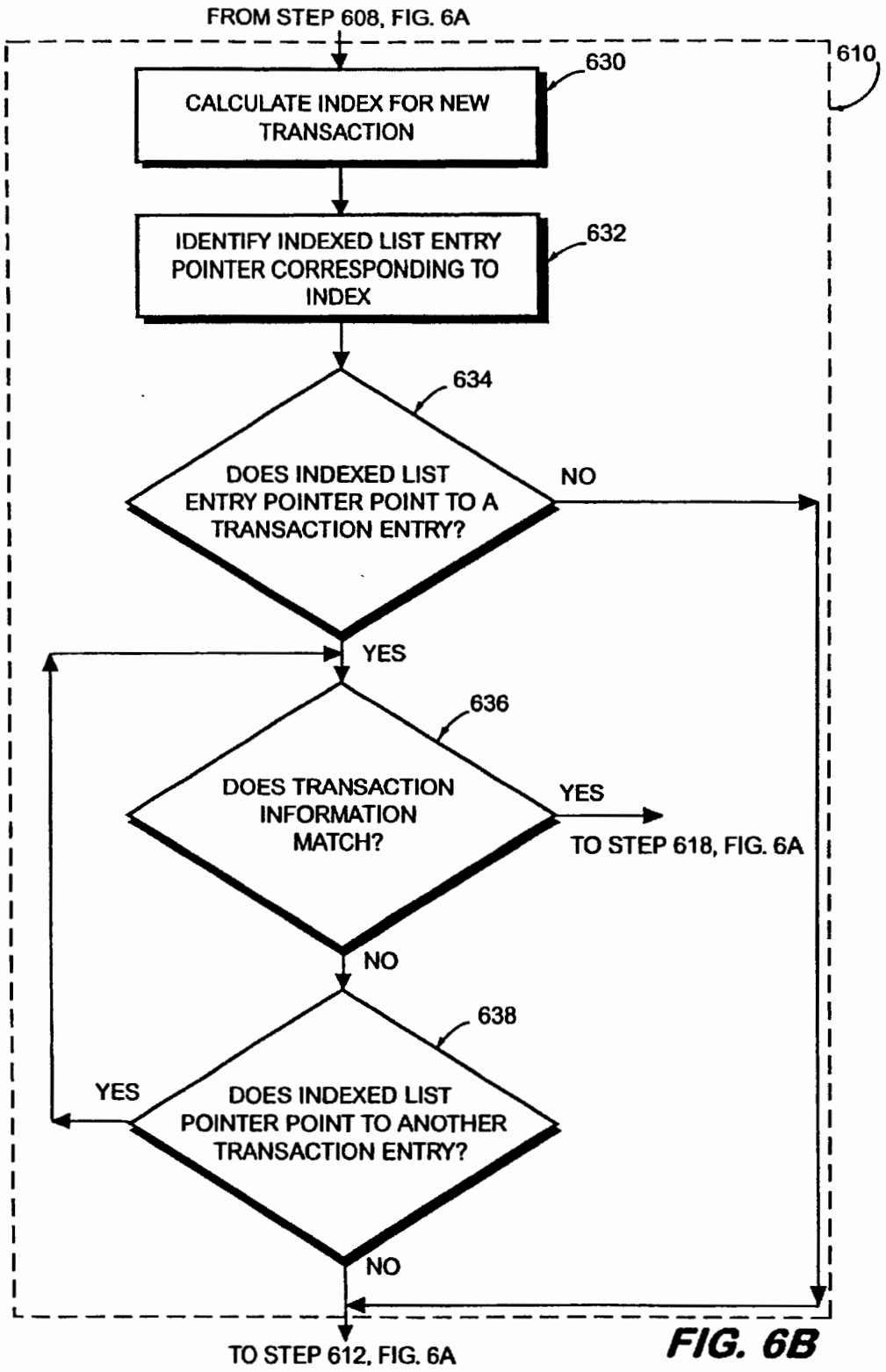

FROM STEP 608, FIG. 6A
610
630
CALCULATE INDEX FOR NEW TRANSACTION
632
IDENTIFY INDEXED LIST ENTRY POINTER CORRESPONDING TO INDEX
634
DOES INDEXED LIST ENTRY POINTER POINT TO A TRANSACTION ENTRY?
NO
YES
636
DOES TRANSACTION INFORMATION MATCH?
YES
TO STEP 618, FIG. 6A
NO
638
DOES INDEXED LIST POINTER POINT TO ANOTHER TRANSACTION ENTRY?
YES
NO
TO STEP 612, FIG. 6A

FIG. 6B

an overload condition on the network, causing legitimate calls to fail. Several solutions have been proposed to solve the looping problem. One proposed solution involves marking a message with a "dirty" bit. A dirty bit is set by the originating network before the message is sent. The originating network checks the dirty bit for each message it receives from another network. If the dirty bit is set, then the originating network detects a loop. A disadvantage of the dirty bit proposal is that the dirty bit must be preserved by all local service providers. However, there is no provision for a dirty bit in the existing message routing protocol, so there is no guarantee that the dirty bit will be preserved.

Another proposed solution is "gateway screening". This solution requires that a network screen messages received from other networks to detect a looping message. A message received from another network is screened to determine whether the message originated in the network receiving the message. If the message originated in the network receiving the message, then the message is dropped. A disadvantage of this solution is that it will only detect loops involving an originating network. If a loop occurs between two non-originating networks, it is not detected.

A third proposed solution is transaction ID logging. Transaction ID logging maintains a log of messages recently sent to other networks and compares a message received from another network to the message log. If the received message matches a message in the log, a loop is detected. The log is maintained so that it only contains messages sent within a predetermined period of time. An advantage of transaction ID logging is that it is a self-sufficient solution. Transaction ID logging may be implemented by one local service provider regardless of whether other local service providers implement it. Another advantage is that it works with existing message protocols.

Although transaction ID logging is theoretically appealing, it has not been previously implemented because of concerns that it would adversely impact network performance. The Illinois Commerce Commission ("ICC"), a group formed to study message looping in LNP enabled telecommunications networks and to provide recommended solutions, considered but did not pursue transaction ID logging. The IIC Subcommittee concluded that transaction ID logging was too processor intensive, and therefore, did not pursue transaction ID logging.

Accordingly, there is a need in the art for an implementation of transaction ID logging in which the time to compare a new transaction to the existing transaction entries in the transaction log is minimized. There is also a need in the art for an implementation of transaction ID logging in which the time to maintain a transaction log is minimized.

SUMMARY OF THE INVENTION

The present invention satisfies the above-described needs by using a chronological list superimposed on an indexed list to implement transaction ID logging. The chronological list expedites maintaining the log and the indexed list expedites searching the log. In an LNP enabled telecommunications network, the present invention may be used to detect messages looping between networks. Generally described, the present invention provides a system and method for recording transactions, such as non-call associated messages, in a transaction log using a chronological list superimposed on an indexed list. A transaction log includes a number of transaction entries. Each transaction entry corresponds to a previous transaction. When a new transaction is received, the new transaction is compared to the existing transactions in the transaction log. If a match is detected between the new transaction and an existing transaction, a loop is detected. Once a transaction loop is detected, appropriate action may be taken to break the loop. To ensure that a valid subsequent transaction is not detected as a looping transaction, stale transaction entries are deleted from the transaction log. Typically, a transaction entry is stale if it has been in the transaction log longer than a predetermined maintenance period.

A transaction entry typically includes a transaction descriptor field, a time stamp field, a chronological list pointer field and an indexed list pointer field. The transaction descriptor field contains a transaction descriptor which identifies the transaction and other transaction information. The time stamp field contains a time stamp indicating when the transaction was initiated. The chronological list pointer field may contain a chronological list pointer pointing to the next oldest transaction entry. The indexed list pointer field may contain an indexed list pointer pointing to another transaction entry with the same index.

Each transaction entry in the transaction log is placed in both the chronological list and the indexed list. The chronological list orders the transaction entries from the oldest transaction entry to the latest transaction entry. Chronological list pointers are associated with the transaction entries in the chronological list. For example, a first chronological list pointer points to the oldest transaction entry and a last chronological list pointer points to the latest transaction entry. The order of the remaining transaction entries in the chronological list is maintained using chronological list pointers. A chronological list pointer links a transaction entry to the next oldest transaction entry. The chronological list minimizes the time needed to identify and delete a stale transaction entry.

The indexed list is a list of indexed list entry pointers. Each indexed list entry pointer corresponds to an index and points to a transaction entry with the same index. The index for a transaction entry may be determined by hashing the transaction descriptor for the transaction entry. If there is more than one transaction entry with the same index, then the transaction entries are linked together via the indexed list pointer fields of the transaction entries. For example, if there are two transaction entries with the same index, then the indexed list entry pointer points to the first transaction entry and the indexed list pointer field of the first transaction entry points to the next transaction entry. The indexed list reduces the number of transaction entries which must be searched to determine whether there is a match between a new transaction and an existing transaction. Reducing the number of transaction entries minimizes the time needed to compare a new transaction with an existing transaction.

To maintain the transaction log, the transaction entries are checked to determine whether any of the transaction entries have been stored in the transaction log for longer than the predetermined maintenance period. The first chronological list pointer is used to identify the oldest transaction entry in the transaction log. The time stamp field of the oldest transaction entry is checked to determine whether the transaction entry is stale. If the oldest transaction entry is stale, then the transaction entry is deleted from the chronological list and the indexed list. To delete the oldest transaction entry from the chronological list, the first chronological pointer is updated to point to the next oldest transaction entry. To delete the oldest transaction entry from the indexed list, the indexed list entry pointer corresponding to the oldest transaction entry is updated. If the indexed list entry pointer field of the oldest transaction entry contains an end of list

indicator, then there are no other transaction entries with the same index and the indexed list entry pointer is updated to contain an end of list indicator. If the indexed list entry pointer field of the oldest transaction entry contains a pointer to another transaction entry, then the indexed list entry pointer is updated to point to that transaction entry.

To compare a new transaction with the existing transactions in the transaction log, the index for the new transaction is calculated. The index identifies an indexed list entry pointer corresponding to the index. If there is no transaction entry which corresponds to the index, then the indexed list entry pointer contains an end of list indicator. Otherwise, the indexed list entry pointer contains a pointer to a transaction entry corresponding to the index. If there is more than one transaction entry corresponding to the index, then the indexed list pointer field of the transaction entry corresponding to the index contains a pointer to another transaction entry. Once the transaction entries corresponding to the index are identified, the transaction information for each transaction entry is compared to the transaction information for the new transaction. If there is a match, then a loop is detected. If there is no match, a new transaction entry is created for the new transaction and the new transaction entry is added to the transaction log.

To add a new transaction entry to the transaction log, the last chronological list pointer is updated to point to the new transaction entry. The chronological list pointer field of the transaction entry previously pointed to by the last chronological list pointer is updated to point to the new transaction entry. If the indexed list entry pointer corresponding to the index for the new transaction contains an end of list indicator, the indexed list entry pointer is updated to point to the new transaction entry. Otherwise, the indexed list pointer field of the last transaction entry corresponding to the index is updated to point to the new transaction identifier.

Using a chronological list superimposed on an indexed list, solves the problems of quickly maintaining and searching the transaction log. The chronological list minimizes the time required to identify and delete the oldest transaction entries. There is no need to search the entire transaction log to locate the oldest transaction entry because the first chronological list pointer points to the oldest transaction entry. Similarly, only those transaction entries with the same index as the new transaction are searched to determine if a new transaction matches an existing transaction in the transaction log. The search time is minimized because only those transaction entries with the same index are searched.

These and other aspects, features and advantages of the present invention may be more clearly understood and appreciated from a review of the following detailed description of the disclosed embodiments and by reference to the appended drawings and claims.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a block diagram of two switched telephone networks illustrating the operating environment for an exemplary embodiment of the present invention.

FIG. **2** is a block diagram of a computer illustrating the operating environment for an exemplary embodiment of the present invention.

FIG. **3** is an illustration of a transaction entry created by an exemplary embodiment of the present invention.

FIG. **4A** is an illustration of a transaction log created by an exemplary embodiment of the present invention.

FIG. **4B** is an illustration of the transaction log of FIG. **4A** after the deletion of a transaction entry by an exemplary embodiment of the present invention.

FIG. **4C** is an illustration of a transaction log of FIG. **4A** after the addition of a transaction entry by an exemplary embodiment of the present invention.

FIG. **5** is a logical flow diagram illustrating the steps for maintaining a transaction log by deleting stale transaction entries in accordance with an exemplary embodiment of the present invention.

FIG. **6A** is a logical flow diagram illustrating the steps for adding a transaction entry to a transaction log in accordance with an exemplary embodiment of the present invention.

FIG. **6B** is a logical flow diagram illustrating the steps for comparing a transaction to the existing transactions in a transaction log in accordance with an exemplary embodiment of the present invention.

DETAILED DESCRIPTION

The present invention is directed toward a system and method for recording transactions in a transaction log using a chronological list superimposed on an indexed list. In one embodiment, the present invention may be used to implement transaction ID logging in an LNP enabled telecommunications network. Briefly described, a log of messages recently sent to other networks is maintained in a chronological list and in an indexed list. Each time a message is received from another network, the message is compared to the existing messages in the transaction log. If the message matches an existing message in the transaction log, then a loop is detected. The chronological list is used to detect and delete stale transaction entries. The indexed list is used to compare a new message to the existing messages in the transaction log. The use of a chronological list superimposed on an indexed list implements transaction ID logging without adversely impacting network performance.

Exemplary Telecommunications Operating Environment

FIG. **1** is a functional block diagram that illustrates a portion of a public switched telecommunications network ("PSTN") **10** configured for LNP. Two Advanced Intelligent Networks ("AIN's") **11***a* and **11***b* represent the LNP-enabled portion of the PSTN **10**. Although FIG. **1** illustrates two networks in the LNP-enabled portion of the PSTN **10**, additional networks for additional local service providers may exist. Although FIG. **1** shows both networks as AIN's, a network is not necessarily implemented as an AIN. An AIN is well-known to those skilled in the art and is described in the commonly-assigned patent to Weisser, Jr., U.S. Pat. No. 5,430,719, which is incorporated herein by reference.

The AIN's **11***a* and **11***b* may include a plurality of central office switches (not shown). Some of the central office switches are equipped with service switching points ("SSP's"). Representative SSP's **12***a* and **12***b* are shown in FIG. **1**. An SSP (specifically, a Class **5** central office switch) is the AIN component of a typical electronic central office switch used by a local exchange carrier. The terms "SSP" and "switch" are used interchangeably herein to refer to a telecommunications switch for connecting voice-channel circuits, including voice-channel lines. In FIG. **1**, the voice-channel lines for SSP **12***a* and SSP **12***b* are respectively **14***a–n* and **17***a–n*.

The switches of AIN's **11***a* and **11***b* are interconnected by a network of high capacity voice-channel circuits known as trunks **22**. Each switch of an AIN is operable for receiving a communication, such as a telephone call, originating on a line serviced by the switch, and for routing the telephone call

be used to record transactions or other data in a log so that the log may be quickly searched and maintained. FIG. 2 and the following discussion provide a brief, general description of a suitable computing environment for the invention. Those skilled in the art will appreciate that the invention may be practiced with other computer system configurations, including hand-held devices, multiprocessor systems, microprocessor-based or programmable consumer electronics, minicomputers, mainframe computers, and the like. The invention may also be practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network. In a distributed computing environment, program modules may be located in both local and remote memory storage devices.

With reference to FIG. 2, an exemplary computer system for implementing an embodiment of the invention includes a conventional personal computer **220**, including a processing unit **221**, a system memory **222**, and a system bus **223** that couples the system memory to the processing unit **221**. The personal computer **220** further includes a hard disk drive **227**, a magnetic disk drive **228**, e.g., to read from or write to a removable disk **229**, and an optical disk drive **230**, e.g., for reading a CD-ROM disk **231** or to read from or write to other optical media. The hard disk drive **227**, magnetic disk drive **228**, and optical disk drive **230** are connected to the system bus **223** by a hard disk drive interface **232**, a magnetic disk drive interface **233**, and an optical drive interface **234**, respectively. The drives and their associated computer-readable media provide nonvolatile storage for the personal computer **220**. Although the description of computer-readable media above refers to a hard disk, a removable magnetic disk and a CD-ROM disk, it should be appreciated by those skilled in the art that other types of media which are readable by a computer, such as magnetic cassettes, flash memory cards, digital video disks, Bernoulli cartridges, and the like, may also be used in the exemplary operating environment.

A number of program modules may be stored in the drives and RAM **225**, including an operating system **235**, one or more application programs **236**, other program modules **237**, such as a transaction ID logging program module **237*a***, and program data **238**. A user may enter commands and information into the personal computer **220** through a keyboard **240** and pointing device, such as a mouse **242**. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit **221** through a serial port interface **246** that is coupled to the system bus, but may be connected by other interfaces, such as a game port or a universal serial bus (USB). A monitor **247** or other type of display device is also connected to the system bus **223** via an interface, such as a video adapter **248**. In addition to the monitor, personal computers typically include other peripheral output devices (not shown), such as speakers or printers.

The personal computer **220** may operate in a networked environment using logical connections to one or more remote computers, such as a remote computer **249**. The remote computer **249** may be a server, a router, a peer device or other common network node, and typically includes many or all of the elements described relative to the personal computer **220**, although only a memory storage device **250** has been illustrated in FIG. **21**. The logical connections depicted in Figure **21** include a local area network (LAN) **251** and a wide area network (WAN) **252**. Such networking environments are commonplace in offices, enterprise-wide computer networks, intranets and the Internet.

When used in a LAN networking environment, the personal computer **220** is connected to the LAN **251** through a network interface **253**. When used in a WAN networking environment, the personal computer **220** typically includes a modem **254** or other means for establishing communications over the WAN **252**, such as the Internet. The modem **254**, which may be internal or external, is connected to the system bus **223** via the serial port interface **246**. In a networked environment, :program modules depicted relative to the personal computer **220**, or portions thereof, may be stored in the remote memory storage device. It will be appreciated that the network connections shown are exemplary and other means of establishing a communications link between the computers may be used.

LNP Embodiment of the Present Invention

The present invention is an implementation of transaction ID logging using a chronological list superimposed on an indexed list. One embodiment of the present invention may be used for detecting message looping in an LNP enabled telecommunications network. In the LNP embodiment, a transaction log is comprised of a number of transaction entries, each transaction entry corresponds to a message, such as a TCAP message, sent to another network. An exemplary transaction entry **300** is shown in FIG. 3. Preferably, the transaction entry includes a transaction descriptor field **302**, a time stamp field **304**, a chronological list pointer field **306** and an indexed list pointer field **308**. The transaction descriptor field **302** includes a transaction descriptor which identifies the transaction associated with the message. The transaction descriptor field **302** also includes other information associated with the transaction. In the LNP embodiment, the transaction descriptor may include the TCAP transaction ID, the TCAP message type, the calling party point code from the SCCP header and the TCAP message length. The time stamp field **304** contains a time stamp which indicates the time the message was sent. The chronological list pointer field **306** contains a chronological list pointer to the next entry in the chronological list. The indexed list pointer field **308** contains an indexed list pointer to the next entry with the same index in the indexed list.

The transaction entries are placed in chronological order in the chronological list using chronological list pointers. Preferably, two chronological list pointers, a first chronological list pointer and a last chronological list pointer, keep track of the beginning of the chronological list and the end of the chronological list respectively. The first chronological list pointer identifies the oldest transaction entry in the log and the last chronological list identifies the most recent transaction entry. The order of the intervening transaction entries is maintained by the chronological list pointer fields of the transaction entries.

The chronological list **406**, illustrated in FIG. **4A**, comprises a plurality of chronological pointers **402**, **404**, **426**, **436**, **446**, **456**, **466**, and **476**. The first chronological list pointer **404** points to the oldest transaction entry **4201** in the chronological list and the last chronological list pointer **402** points to the latest transaction entry **470** in the chronological list. The chronological list pointer field of a transaction entry points to the next oldest transaction entry. For example, the oldest transaction entry **420** includes a chronological list pointer field **426** which contains a chronological list pointer to the second oldest transaction entry **460**. The second oldest transaction entry **460** includes a chronological list pointer field **466** which contains a chronological list pointer to the third oldest transaction entry **430**. The chronological list

pointer field for the latest transaction entry **470** contains an end of list indicator in the chronological list pointer field **476**. The end of list indicator is illustrated in the figures by "]", the null pointer.

The chronological list helps solve the problem of efficiently maintaining the transaction log. The transaction log is maintained by deleting stale transaction entries. Preferably, a stale transaction entry is a transaction entry which has been stored in the transaction log for longer than a predetermined maintenance period. The oldest transaction entries in the transaction log may be quickly and easily identified because the oldest transaction entries are at the front of the chronological list. The first chronological list pointer identifies the oldest transaction entry. The chronological list pointer of the oldest transaction entry identifies the second oldest transaction entry. In an LNP enabled telecommunications network, maintaining the transaction log by purging stale transaction entries insures that a subsequent valid message is not detected as a looping message.

In addition to being placed in a chronological list, the transaction entries are also placed in an indexed list. The indexed list comprises a list of indexed list entry pointers. Each indexed list entry pointer corresponds to an index and points to a transaction entry which corresponds to the same index. A transaction entry corresponds to an index if the transaction descriptor for the transaction entry corresponds to the index. In one implementation of the LNP embodiment, the indexes range from 0000 to 4095. However, the range of the indexes may vary from implementation to implementation. As discussed in more detail below, the lower index is preferably zero and the upper index is preferably defined as (2^n-1), where n is an integer.

Preferably, a transaction entry corresponds to an index if the "hashed" transaction descriptor for the transaction entry matches that index. Hashing converts a transaction descriptor into a pseudo random index. The index is not truly random because for any given transaction descriptor, hashing always results in the same pseudo random index. However, the pseudo random indexes produced by hashing are sufficiently random so that the transaction entries are evenly distributed among the indexes. If the implementation uses indexes ranging from 0000 to 4095, then preferably, the transaction descriptor is hashed by calculating the modulo 4096 of the transaction descriptor. The modulo operation is performed by dividing the transaction descriptor by 4096 and returning the remainder of the division operation. The remainder of the division operation is used as an index into the indexed list. The modulo number 4096 is preferred, in part, because the modulo of the transaction descriptor for a transaction entry may be determined by performing a logical AND operation with the transaction descriptor and 4095. As will be apparent to those skilled in the art, if the indexes range from zero to 2^n-1, then the modulo 2^n may be determined by performing a logical AND operation with the transaction descriptor and 2^n-1.

FIG. **4A** also illustrates the indexed list **400**. The indexed list **400** comprises a list of indexed list entry pointers **410***a*, **410***b*, **410***c* . . . **410***n*. Each indexed list entry pointer corresponds to an index. For example, indexed list entry pointer **410***a* corresponds to 0000 and indexed list entry pointer **410***b* corresponds to 0001.

Hashing transaction descriptors may result in multiple transaction descriptors having the same index. For example, modulo 4096 of transaction descriptor 1000 and modulo 4096 of transaction descriptor 9192 are both 1000. If there are multiple transaction descriptors with the same index, then all the transaction entries with the same index are linked together via the indexed list pointer fields (e.g. **428** and **438**) of the transaction entries. As shown in FIG. **4A**, transaction entries **420**, **430**, and **440** and indexed list entry pointer **410***a* all correspond to index 0000. Indexed list entry pointer **410***a* points to transaction entry **420**. Transaction entry **420** is linked to transaction entry **430** via its indexed list pointer field **428**. Similarly, transaction entry **430** is linked to transaction entry **440** via its indexed list pointer field **438**. Since there are no other transaction entries with index 0000, transaction entry **440** contains an end of list indicator in its indexed list pointer field **448**.

The indexed list helps solve the problem of quickly searching the transaction log to determine whether a new transaction matches an existing transaction in the transaction log. When a new transaction is received, the transaction descriptor for the new transaction is hashed to determine its index. The index is used to identify an indexed list entry pointer which, in turn, is used to identify transaction entries which correspond to the index. To determine whether there is a match, the transaction information in the transaction descriptor field of the new transaction is compared to the transaction information in the transaction descriptor fields of the transaction entries which correspond to the index. Only the transaction entries which correspond to the index are checked. If the transaction information in the transaction descriptor field of the new transaction matches the transaction information in the transaction descriptor field of an existing transaction, the indexes for the two transaction descriptors will match because hashing always produces the same index for the same input.

A chronological list superimposed on an indexed list, solves the problems of efficiently maintaining and searching the transaction log. The chronological list minimizes the time required Lo identify the oldest transaction entry. There is no need to search the transaction log to locate the oldest transaction entry because the first chronological list pointer points to the oldest transaction entry. If the oldest transaction entry is stale or if there are no available transaction entries, then the oldest transaction entry may be deleted by modifying the first chronological list pointer and by modifying the indexed list entry pointer associated with the index for the oldest transaction entry. The indexed list minimizes the time required to search the list for transactions which match a new transaction. Only those transaction entries with the same index as the new transaction need be compared.

The steps for maintaining the transaction log by deleting stale transaction entries may be illustrated by reference to FIGS. **4A**, **4B** and **5**. FIG. **4A** illustrates an exemplary transaction log **408** using a chronological list **406** superimposed on an indexed list **400**. FIG. **4B** illustrates the transaction log of FIG. **4A** after a stale transaction entry is purged from the transaction log. FIG. **5** is a logical flow diagram illustrating the steps for deleting a stale transaction entry from the transaction log. FIG. **4A** illustrates the transaction log **408** prior to the deletion of any stale transaction entries. The transaction log **408** comprises six transaction entries **420**, **430**, **440**, **450**, **460** and **470**. The first chronological list pointer **404** points to transaction entry **420** and the last chronological list pointer **402** points to transaction entry **470**. The chronological order of the transaction entries is **420**, **460**, **430**, **440**, **450** and **470**. The indexed list contains indexes from 0000 to 4095. Transaction entries **420**, **430**, and **440** correspond to index 0000, transaction entry **450** corresponds to index 0002, and transaction entries **460** and **470** correspond to index 0004.

FIG. **5** illustrates the steps for deleting stale transaction entries from the transaction log. Preferably, a transaction is

stale if it has been stored in the transaction log for longer than a predetermined maintenance period. Transaction log maintenance may be initiated from an idle state as shown in step **500**. Alternatively, transaction log maintenance may be initiated whenever a new transaction is received. In either case, a timer may be used to keep track of the time elapsed since the last maintenance operation was performed and to indicate when the predetermined maintenance period has expired. The timer is checked in step **502** to determine if the predetermined maintenance period has expired. If the timer indicates that the predetermined maintenance period has expired, then the method proceeds to step **504**. In step **504**, the transaction entry pointed to by the first chronological list pointer is selected as the selected transaction entry. In FIG. **4A**, the first chronological list pointer **404** points to transaction entry **420**, so transaction entry **420** is selected as the selected transaction entry. Once a transaction entry is selected as the selected transaction entry, the time stamp field **424** of the selected transaction entry **420** is checked in step **506** to determine whether the selected transaction entry is stale. If the time stamp contained in the time stamp field **424** indicates that the selected transaction entry **420** has been stored in the transaction log **408** for longer than the predetermined maintenance period, then the selected transaction entry **420** is deleted from the chronological list **406** in step **508** and from the indexed list **400** in step **510**.

To delete the selected transaction entry **420** from the chronological list **406**, the first chronological list pointer **404** is modified to point to the next oldest transaction entry **460**. Transaction entry **460** is identified as the next oldest transaction entry by the chronological list pointer in the chronological list pointer field **426** of the selected transaction entry **420**. The other chronological list pointers remain the same.

The selected transaction entry **420** is also deleted from the indexed list. To delete the selected transaction entry from the indexed list, the index for the selected transaction entry **420** is calculated by hashing all or a predetermined portion of the transaction descriptor. In one implementation of the LNP embodiment, the index is calculated by computing the modulo 4096 of a predetermined portion of the transaction descriptor for the selected transaction entry. In FIG. **4A**, the index for the selected transaction entry **420** is 0000. The index is used to identify an indexed list entry pointer **410*a*** corresponding to the index. If the indexed list pointer field **428** of the selected transaction entry **420** contains an indexed list pointer to a next transaction entry **430**, then the indexed list entry pointer **410*a*** is modified to point to the next transaction entry **430**. If the indexed list pointer field **428** of the selected transaction entry **420** contains an end of list indicator, then the indexed list entry pointer **410*a*** is modified to include an end of list indicator. The other indexed list entry pointers and indexed list pointers remain the same.

The oldest transaction entry for a given index is always pointed to by the indexed list entry pointer because a new transaction entry is always added to the end of the indexed list. Thus, deleting a stale transaction entry from the indexed list, only requires that the indexed list entry pointer be modified. The details of adding a transaction entry to the indexed list are described below.

After the selected transaction entry is deleted, the transaction log appears as shown in FIG. **4B**. The transaction log now comprises five transaction entries **430**, **440**, **450**, **460** and **470**. The first chronological list pointer points to transaction entry **460** and the last chronological list pointer points to transaction entry **470**. The chronological order of the transaction entries is **460**, **430**, **440**, **450** and **470**. Transaction entries **430** and **440** correspond to index 0000, transaction entry **450** corresponds to index 0002, and transaction entries **460** and **470** correspond to index 0004.

Preferably, if the selected transaction entry is deleted, then the next oldest transaction entry is checked to determine whether it is also stale. FIG. **5** shows that step **504** of selecting a transaction entry as the selected transaction entry, step **506** of making a determination as to whether the selected transaction entry is stale, step **508** of deleting the selected transaction entry from the chronological list and step **510** of deleting the selected transaction entry from the indexed list are repeated until the determination in step **506** is that the selected transaction entry is not stale. By using the first chronological list pointer to select the selected transaction entry, once a determination is made that the selected transaction entry is not stale, no other transaction entries are checked because the remaining transaction entries have been stored in the transaction log for lesser periods of time.

If the determination in step **506** is that the selected transaction entry is not stale or if the determination in step **502** is that the predetermined maintenance period has not expired, then the method returns to the idle state of step **500**. Alternatively, if transaction log maintenance was initiated by the receipt of a new transaction, the method proceeds with the steps for handling a new transaction.

The steps for handling a new transaction may be illustrated by reference to FIGS. **4A**, **4C**, **6A** and **6B**. FIG. **4A** illustrates an exemplary transaction log **408** using a chronological list **406** superimposed on an indexed list **400**. FIG. **4C** illustrates the transaction log of FIG. **4A** after a new transaction entry is added to the transaction log. FIG. **6A** is a logical flow diagram illustrating the steps for adding a new transaction entry to the transaction log. FIG. **6B** is a logical flow diagram illustrating the steps for making a determination as to whether a new transaction matches an existing transaction in the transaction log.

FIG. **4A** illustrates the transaction log **408** before the new transaction is received. The steps for adding a new transaction to the transaction log begin at the START task of step **600** of FIG. **6A**. In step **602**, a new transaction is received. In step **604**, a determination is made as to whether there is an available transaction entry in the transaction log for a new transaction entry. If there is an available transaction entry for the new transaction, then the method proceeds to step **610** where a determination is made as to whether the new transaction matches an existing transaction in the transaction log.

The steps for making a determination as to whether the new transaction matches an existing transaction in the transaction log are illustrated in FIG. **6B**. In FIG. **6B**, the index for the new transaction is calculated in step **630** by hashing the transaction descriptor. In the LNP embodiment illustrated by FIG. **4A**, the index is calculated by taking the modulo 4096 of the transaction descriptor for the new transaction. For example, if the transaction descriptor for the new transaction is 0004, the index for the new transaction is hashed by calculating modulo 4096 of 0004 which is 0004. Once the index for the new transaction is calculated, the indexed list entry pointer corresponding to the index is identified in step **632**. If the index is 0004, then, as shown in FIG. **4A**, the indexed list entry pointer **410*e*** is identified. In step **634**, a determination is made as to whether the indexed list entry pointer contains a pointer to a transaction entry. If the indexed list entry pointer does not contain a pointer to a transaction entry, then the determination is that the transaction does not match an existing transaction in the transaction log and the method continues to step **612**. If the

indexed list entry pointer contains a pointer to a transaction entry, then the transaction information for the existing transaction entry is compared to the transaction information for the new transaction in step **636**. In FIG. **4**A, the indexed list entry pointer **410***e* points to transaction entry **460** so the transaction information for existing transaction entry **460** is compared to the transaction information for the new transaction.

If the transaction information for the existing transaction entry matches the transaction information for the new transaction, then a loop is detected and the method proceeds to step **618** of FIG. **6**A. In response to detecting a loop, the network takes some action which may include closing the transaction which originated the message or resending the message. Alternatively, if the transaction information for the existing transaction entry does not match the transaction information for the new transaction, then the method proceeds to step **638**. In this example, the transaction information for the existing transaction entry **460** does not match the transaction information for the new transaction so the method proceeds to step **638**. In step **638**, a determination is made as to whether the indexed list pointer field of the transaction entry points to another transaction entry. If the indexed list pointer field of the transaction entry points to another transaction entry, then the method returns to step **636**. In FIG. **4**A, the indexed list pointer field **468** of transaction entry **460** points to transaction entry **470**, so step **636** is repeated with transaction entry **470**. In this example, the determination in step **636** is that the transaction information for transaction **470** does not match the transaction information for the new transaction so the method proceeds to step **638**. If the determination in step **638** is that the indexed list pointer field of the transaction entry does not point to another transaction entry, then the method proceeds to step **612** of FIG. **6**A. The determination in step **638** for transaction entry **470** is that the indexed list pointer field **478** of the transaction entry **470** does not point to another transaction entry so the method proceeds to step **612** of FIG. **6**A.

The transaction is added to the transaction log in steps **612**–**616**. In step **612**, a new transaction entry for the transaction is created. The new transaction entry contains a transaction descriptor field, a time stamp field, a chronological list pointer field and an indexed list pointer field. In the LNP embodiment, the transaction descriptor field preferably contains a portion of the SS7 header and the time stamp field contains a time indicating when the message associated with the new transaction occurred. The new transaction entry **480** includes a transaction descriptor field **482**, a time stamp field **484**, a chronological list pointer field **486** and an indexed list pointer field **488** and is shown in FIG. **4**C. The new transaction entry is placed in the chronological list in step **614**.

To place the new transaction entry in the chronological list, the last chronological list pointer is updated to point to the new transaction entry. The chronological list pointer field for the transaction entry previously pointed to by the last chronological list pointer is also updated to point to the new transaction entry. In FIG. **4**C, the last chronological list pointer **402** is updated to point to the new transaction entry **480** and the last chronological list pointer field **476** for the transaction entry **470** previously pointed to by the last chronological list pointer is updated to point to the new transaction entry **480**. The chronological list pointer field **486** for the new transaction entry **480** contains an end of list indicator.

To place the new transaction entry in the indexed list, the new transaction entry is placed at the end of the list of transaction entries with the same index. By placing the new transaction entry at the end of the list of transaction entries with the same index, the oldest transaction entry for a given index is always pointed to by the indexed list entry pointer. The transaction entries with the same index were previously identified in step **610** where a determination was made as to whether the new transaction matches an existing transaction in the transaction log. In the example of FIG. **4**C, transaction entries **460** and **470** correspond to the same index as the transaction. The new transaction entry is added after transaction entry **470** by modifying the indexed list pointer field **478** of transaction entry **470** to point to the new transaction entry **480**. The indexed list pointer field of the new transaction entry contains an end of list indicator. Alternatively, if the indexed list entry pointer for the index corresponding to the new transaction entry contains an end of list indicator, then in step **616**, the indexed list entry pointer is updated to point to the new transaction entry.

The transaction log after the new transaction entry is added is shown in FIG. **4**C. The transaction log now comprises seven transaction entries **420**, **430**, **440**, **450**, **460**, **470** and **480**. The first chronological list pointer points to transaction entry **420** and the last chronological list pointer points to transaction entry **480**. The chronological order of the transaction entries is **420**, **460**, **430**, **440**, **450**, **470** and **480**. Transaction entries **420**, **430**, and **440** correspond to index 0000, transaction entry **450** corresponds to index 0002, and transaction entries **460**, **470** and **480** correspond to index 0004.

If a transaction entry is not available in the transaction log when a transaction is received, then the oldest transaction entry is purged from the transaction log. The steps of making a transaction entry available for a newly received transaction are shown in FIG. **6**A. The transaction is received in step **602**. In step **604**, a determination is made as to whether there is an available transaction entry in the transaction log. If a transaction entry is not available, then the oldest transaction entry is purged from the chronological list in step **606** and is purged from the indexed list in step **608**. Purging the oldest transaction entry from the chronological list and the indexed list follows the steps described above in connection with maintaining the transaction log.

The present invention is directed toward a system and method for logging transactions in a transaction log using a chronological list superimposed on an indexed list. The chronological list is used to identify and delete stale transaction entries. The time needed to identify and delete stale transaction entries is minimized because the oldest transaction entry is at the beginning of the chronological list and is at the beginning of the list pointed to by the indexed list entry pointer corresponding to the index for the oldest transaction entry.

The indexed list is used to compare a new transaction to the existing transactions in the transaction log to determine whether the new transaction matches an existing transaction. The new transaction is only compared to transaction entries with the same index as the new transaction. The time to determine whether the new transaction matches an existing transaction is minimized by limiting the number of transaction entries compared to only those transaction entries with the same index as the new transaction.

In one embodiment, the present invention may be used to implement transaction ID logging to detect non-call associated message looping in an LNP enabled telecommunications network. The use of a chronological list superimposed on an indexed list implements transaction ID logging with-

out adversely impacting network performance. Other embodiments may be used to log other types of transactions or data.

The present invention has been described in relation to particular embodiments which are intended in all respects to be illustrative rather than restrictive. Alternative embodiments will become apparent to those skilled in the art to which the present invention pertains without departing from its spirit and scope. Accordingly, the scope of the present invention is described by the appended claims and is supported by the foregoing description.

What is claimed is:

1. A method for purging stale transaction entries from a transaction log having a plurality of transaction entries organized as a chronological list superimposed on an indexed list, wherein the chronological list is ordered from an oldest transaction entry to a latest transaction entry and the indexed list comprises a plurality of indexed list entry pointers corresponding to a plurality of indexes, comprising the steps of:

checking whether a predetermined maintenance period has expired;

if the predetermined maintenance period has expired, then selecting a transaction entry as a selected transaction entry;

checking whether the selected transaction entry is stale; and

if the selected transaction entry is stale, then

(a) purging the selected transaction entry from the chronological list, and

(b) purging the selected transaction entry from the indexed list.

2. The method of claim **1**, wherein a first chronological list pointer points to the oldest transaction entry in the chronological list, and wherein the step of selecting one of the plurality of transaction entries as a selected transaction entry comprises:

selecting the oldest transaction entry as the selected transaction entry.

3. The method of claim **1** wherein the selected transaction entry corresponds to a selected transaction and comprises:

a transaction descriptor field comprising a transaction descriptor to identify the selected transaction;

a time stamp field comprising a time identifier to indicate a time when the selected transaction occurred;

a chronological list pointer field comprising a chronological list pointer to identify a subsequent transaction entry corresponding to a transaction occurring after the selected transaction; and

an indexed list pointer field comprising an indexed list pointer to identify a next transaction entry corresponding to a transaction with the same index as the selected transaction entry.

4. The method of claim **3**, wherein the step of checking whether the selected transaction entry is stale comprises comparing the time stamp field of the selected transaction entry to the predetermined maintenance period.

5. The method of claim **3** wherein the step of purging the selected transaction entry from the chronological list comprises:

updating the first chronological list pointer to point to the subsequent transaction entry.

6. The method of claim **3**, wherein the indexed list comprises a selected indexed list entry pointer corresponding to a selected index and pointing to the selected transaction entry, and wherein the step of purging the selected entry from the indexed list comprises:

updating the selected indexed list entry pointer to point to the next transaction entry.

7. The method of claim **3**, further comprising the steps of:

if the selected transaction entry is stale, then selecting the subsequent transaction entry as a second selected transaction entry;

checking whether the second selected transaction entry is stale; and

if the second selected transaction entry is stale, then

(a) purging the second selected transaction entry from the chronological list, and

(b) purging the second selected transaction entry from the indexed list.

8. A computer-readable medium having computer executable instructions for maintaining a transaction log, the transaction log comprising a plurality of transaction entries organized as a chronological list superimposed on an indexed list, wherein the transaction entries are arranged in chronological order in the chronological list beginning with an oldest transaction entry and ending with a latest transaction entry and the transaction entries are arranged in the indexed list according to indexes corresponding to the transaction entries, comprising the steps of:

maintaining a first chronological list pointer to identify the oldest transaction entry in the transaction log;

maintaining a last chronological list pointer to identify the latest transaction entry in the transaction log;

maintaining a plurality of indexed list entry pointers to identify transaction entries corresponding to the indexes;

maintaining a plurality of indexed list pointers to identify additional transaction entries corresponding to the same index; and

maintaining a plurality of chronological list pointers to identify transaction entries subsequent to the oldest transaction entry.

9. The computer-readable medium of claim **8**, wherein the oldest transaction entry comprises a chronological list pointer pointing to a second oldest transaction entry, and wherein the step of maintaining a first chronological list pointer comprises:

if the oldest transaction entry is purged from the chronological list, then updating the first chronological list pointer to point to the second oldest transaction entry.

10. The computer-readable medium of claim **8**, wherein the step of maintaining a last chronological list pointer comprises:

if a new transaction entry is added to the transaction log, then updating the last chronological list pointer to point to the new transaction entry.

11. The computer-readable medium of claim **8**, wherein the latest transaction entry comprises a chronological list pointer, and wherein the step of maintaining a plurality of chronological list pointers comprises:

if a new transaction entry is added to the transaction log, then updating the chronological list pointer for the latest transaction entry to point to the new transaction entry.

12. The computer-readable medium of claim **8**, wherein a first indexed list entry pointer corresponds to a first index, a first transaction entry has an index equal to the first index, the first indexed list entry pointer points to the first transaction entry, and the first transaction entry comprises an

indexed list pointer pointing to a next transaction entry, and wherein the step of maintaining a plurality of indexed list entry pointers comprises:

if the first transaction entry is purged, then updating the first indexed list entry pointer to point to the next transaction entry.

13. The computer-readable medium of claim 8, wherein a second indexed list entry pointer corresponds to a second index and the second indexed list entry pointer contains an end of list indicator, and wherein the step of maintaining a plurality of indexed list entry pointers comprises:

if a new transaction entry having an index equal to the second index is added to the transaction log, then updating the second indexed list entry pointer to point to the new transaction entry.

14. The computer-readable medium of claim 8, wherein a third indexed list entry pointer corresponds to a third index and points to a third transaction entry with an index equal to the third index, and the third transaction entry comprises an indexed list pointer, and w herein the step of maintaining a plurality of indexed list entry pointers comprises:

if a new transaction entry having an index equal to the third index is added to the transaction log, then updating the indexed list pointer field of the third transaction entry to point to the new transaction entry.

15. A method for creating a transaction log, comprising the steps of:

providing a chronological list superimposed on an indexed list, the chronological list having a first chronological list pointer pointing to an oldest transaction entry and a last chronological list pointer pointing to a latest transaction entry, and the indexed list having a plurality of indexed list entry pointers, each indexed list entry pointer corresponding to an index and pointing to a transaction entry that corresponds to the index;

entering a new transaction entry into the chronological list by:

updating the last chronological list pointer to point to the new transaction entry; and

entering the new transaction entry into the indexed list by:

determining an index for the new transaction entry; and

updating the indexed list entry pointer that corresponds to the index to point to the new transaction entry, so that the transaction entry is entered into both the chronological list and the indexed list.

16. The method of claim 15 wherein the new transaction entry comprises:

a transaction descriptor field comprising a transaction descriptor for identifying the transaction;

a time stamp field comprising a time identifier for indicating a time when the transaction occurred;

a chronological list pointer field comprising a chronological list pointer for identifying a subsequent transaction entry corresponding to a transaction received after the new transaction; and

an indexed list pointer field containing an indexed list pointer for identifying a next transaction entry corresponding to the same index as the new transaction entry.

17. A transaction log comprising:

an indexed list having a plurality of indexed list entry pointers, each indexed list entry pointer corresponding to an index;

a chronological list superimposed on the indexed list, the chronological list having a first chronological list pointer that points to an oldest transaction entry in the transaction log and a last chronological list pointer that points to a latest transaction entry in the transaction log; and

a plurality of transaction entries, each transaction entry corresponding to an index in the indexed list and having a chronological list pointer field including a chronological list pointer for identifying a subsequent transaction entry and an indexed list pointer field containing an indexed list pointer for identifying a next transaction entry corresponding to the same index.

18. The transaction log of claim 17, wherein each transaction entry further comprises:

a transaction descriptor field comprising a transaction descriptor for identifying a transaction corresponding to the transaction entry.

19. The transaction log of claim 18, wherein the index for a transaction entry is calculated by hashing the transaction descriptor.

20. The transaction log of claim 17, wherein a selected indexed list entry pointer corresponding to a selected index points to a transaction entry corresponding to the selected index.

* * * * *

US006651099B1

(12) **United States Patent**
Dietz et al.

(10) **Patent No.: US 6,651,099 B1**
(45) **Date of Patent: Nov. 18, 2003**

(54) **METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK**

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

(73) Assignee: **Hi/fn, Inc.**, Los Gatos, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 589 days.

(21) Appl. No.: **09/608,237**

(22) Filed: **Jun. 30, 2000**

Related U.S. Application Data

(60) Provisional application No. 60/141,903, filed on Jun. 30, 1999.

(51) **Int. Cl.**[7] **G06F 13/00**
(52) **U.S. Cl.** **709/224**; 370/389
(58) **Field of Search** 709/200, 201, 709/220, 223, 224, 231, 232, 236, 238, 239, 240, 246; 370/389, 392, 395.32

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,736,320	A	4/1988	Bristol	364/300
4,891,639	A	1/1990	Nakamura	340/825.5
5,101,402	A	3/1992	Chui et al.	370/17
5,247,517	A	9/1993	Ross et al.	370/85.5
5,247,693	A	9/1993	Bristol	395/800
5,249,292	A	9/1993	Chiappa	395/650
5,315,580	A	5/1994	Phaal	370/13
5,339,268	A	8/1994	Machida	365/49
5,351,243	A	9/1994	Kalkunte et al.	370/92
5,365,514	A	11/1994	Hershey et al.	370/17
5,375,070	A	12/1994	Hershey et al.	364/550
5,394,394	A	2/1995	Crowther et al.	370/60

(List continued on next page.)

OTHER PUBLICATIONS

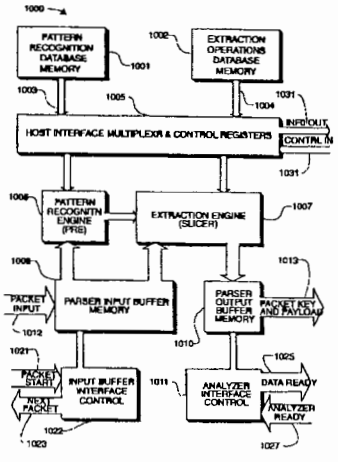
"Technical Note: the Narus System," Downloaded Apr. 29, 1999 from www.narus.com, Narus Corporation, Redwood City California.

Primary Examiner—Moustafa M. Meky
(74) *Attorney, Agent, or Firm*—Dov Rosenfeld; Inventek

(57) **ABSTRACT**

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

10 Claims, 18 Drawing Sheets



U.S. PATENT DOCUMENTS

5,414,650	A	5/1995	Hekhuis	364/715.02
5,414,704	A	5/1995	Spinney	370/60
5,430,709	A	7/1995	Galloway	370/13
5,432,776	A	7/1995	Harper	370/17
5,493,689	A	2/1996	Waclawsky et al.	395/821
5,500,855	A	3/1996	Hershey et al.	370/17
5,511,213	A	4/1996	Correa	395/800
5,511,215	A	4/1996	Terasaka et al.	395/800
5,568,471	A	10/1996	Hershey et al.	370/17
5,574,875	A	11/1996	Stansfield et al.	395/403
5,586,266	A	12/1996	Hershey et al.	395/200.11
5,606,668	A	2/1997	Shwed	395/200.11
5,608,662	A	3/1997	Large et al.	364/724.01
5,634,009	A	5/1997	Iddon et al.	395/200.11
5,651,002	A	7/1997	Van Seters et al.	370/392
5,684,954	A	11/1997	Kaiserswerth et al.	395/200.2
5,703,877	A	12/1997	Nuber et al.	370/395
5,732,213	A	3/1998	Gessel et al.	395/200.11
5,740,355	A	4/1998	Watanabe et al.	395/183.21
5,761,424	A	6/1998	Adams et al.	395/200.47
5,764,638	A	6/1998	Ketchum	370/401
5,781,735	A	7/1998	Southard	395/200.54
5,784,298	A	7/1998	Hershey et al.	364/557
5,787,253	A	7/1998	McCreery et al.	395/200.61
5,802,054	A	9/1998	Bellenger	370/351
5,805,808	A	9/1998	Hansani et al.	395/200.2
5,812,529	A	9/1998	Czarnik et al.	370/245
5,819,028	A	10/1998	Manghirmalani et al.	395/185.1
5,825,774	A	10/1998	Ready et al.	370/401
5,835,726	A	11/1998	Shwed et al.	395/200.59
5,838,919	A	11/1998	Schwaller et al.	395/200.54
5,841,895	A	11/1998	Huffman	382/155
5,850,386	A	12/1998	Anderson et al.	370/241
5,850,388	A	12/1998	Anderson et al.	370/252
5,862,335	A	1/1999	Welch, Jr. et al.	395/200.54
5,878,420	A	3/1999	de la Salle	707/10
5,893,155	A	4/1999	Cheriton	711/144
5,903,754	A	5/1999	Pearson	395/680
5,917,821	A	6/1999	Gobuyan et al.	370/392
6,014,380	A	1/2000	Hendel et al.	370/392
6,118,760	A *	9/2000	Zaumen et al.	370/229
6,243,667	B1 *	6/2001	Kerr et al.	703/27
6,452,915	B1 *	9/2002	Jorgensen	370/338
6,453,360	B1 *	9/2002	Muller et al.	709/250
6,466,985	B1 *	10/2002	Goyal et al.	709/238
6,483,804	B1 *	11/2002	Muller et al.	370/230
6,570,875	B1 *	5/2003	Hegde	370/389

* cited by examiner

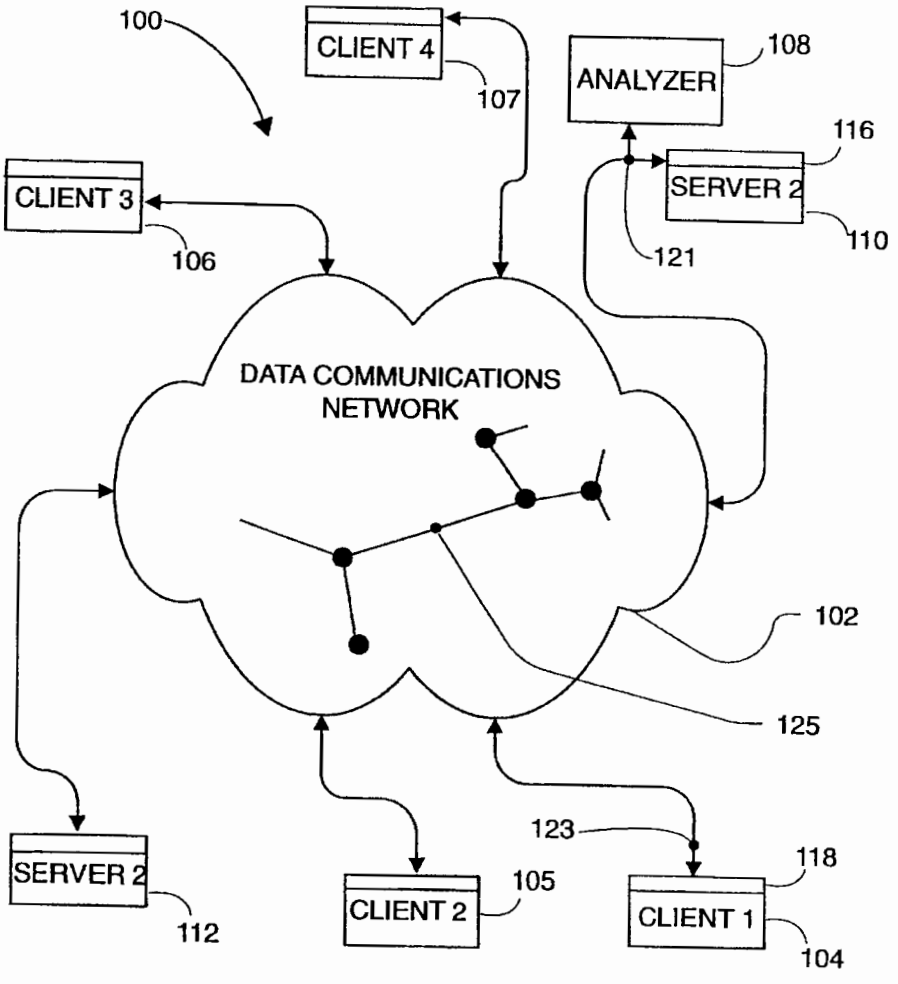

100
CLIENT 4
107
ANALYZER
108
116
SERVER 2
110
121
CLIENT 3
106
DATA COMMUNICATIONS NETWORK
102
125
123
118
SERVER 2
112
CLIENT 2
105
CLIENT 1
104

FIG. 1

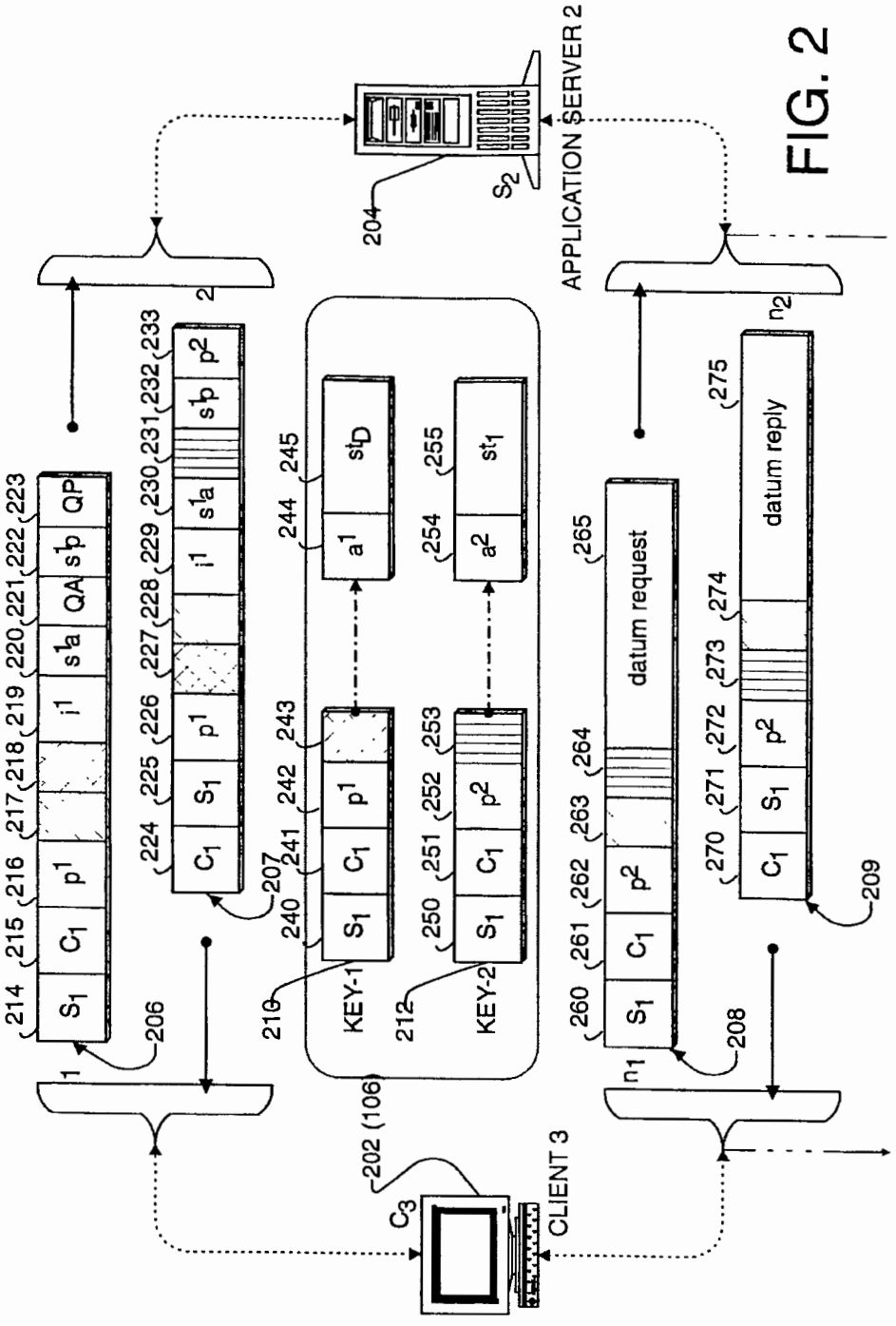

APPLICATION SERVER 2
S_2
204
CLIENT 3
C_3
202 (106)
KEY-1
KEY-2
datum request
datum reply
1
2
n_1
n_2
206
207
208
209
210
212

FIG. 2

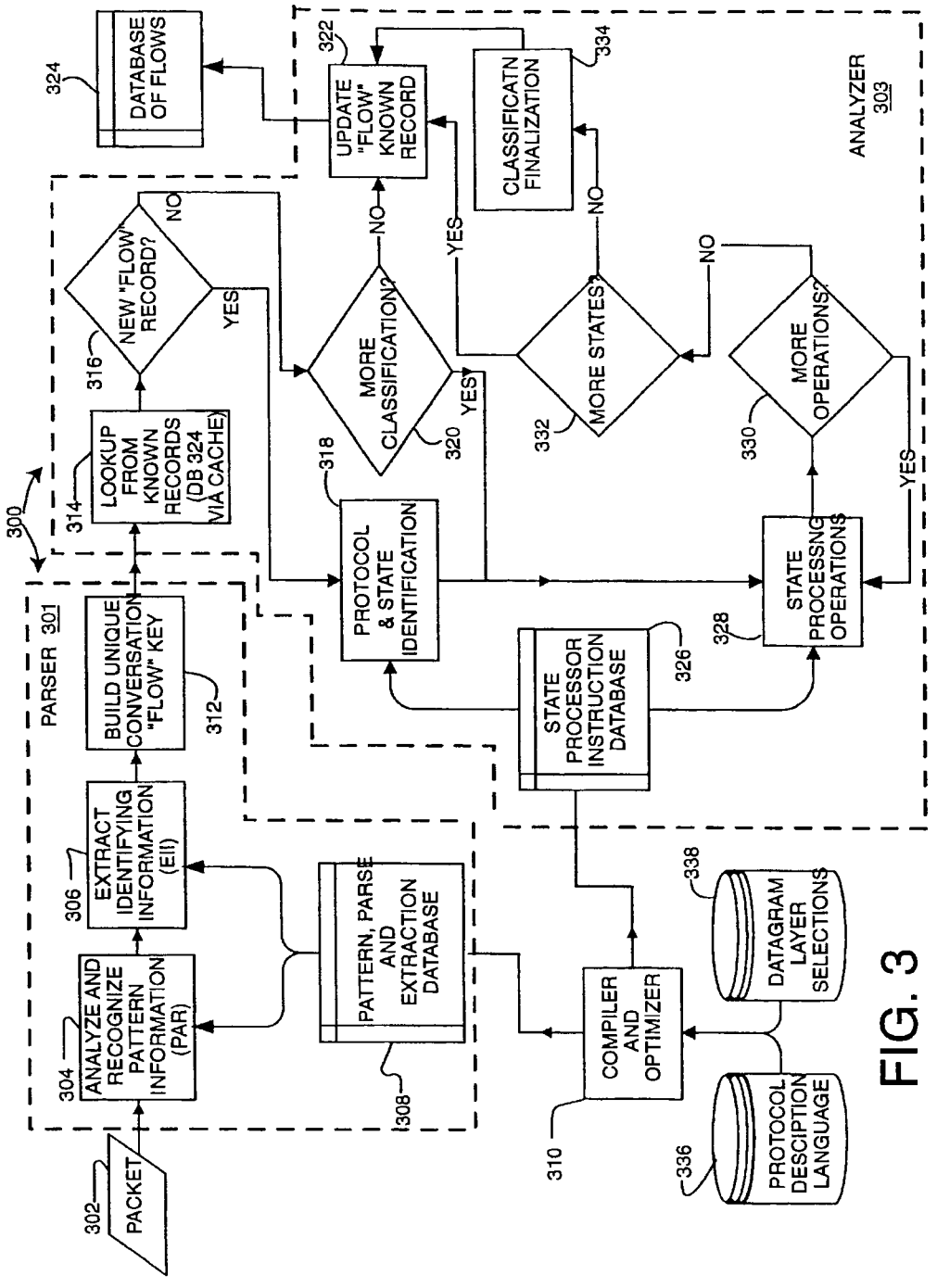

PACKET
302
PARSER 301
ANALYZE AND RECOGNIZE PATTERN INFORMATION (PAR)
304
EXTRACT IDENTIFYING INFORMATION (EII)
306
BUILD UNIQUE CONVERSATION "FLOW" KEY
312
PATTERN, PARSE AND EXTRACTION DATABASE
308
COMPILER AND OPTIMIZER
310
PROTOCOL DESCIPTION LANGUAGE
336
DATAGRAM LAYER SELECTIONS
338
STATE PROCESSOR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

FIG. 3

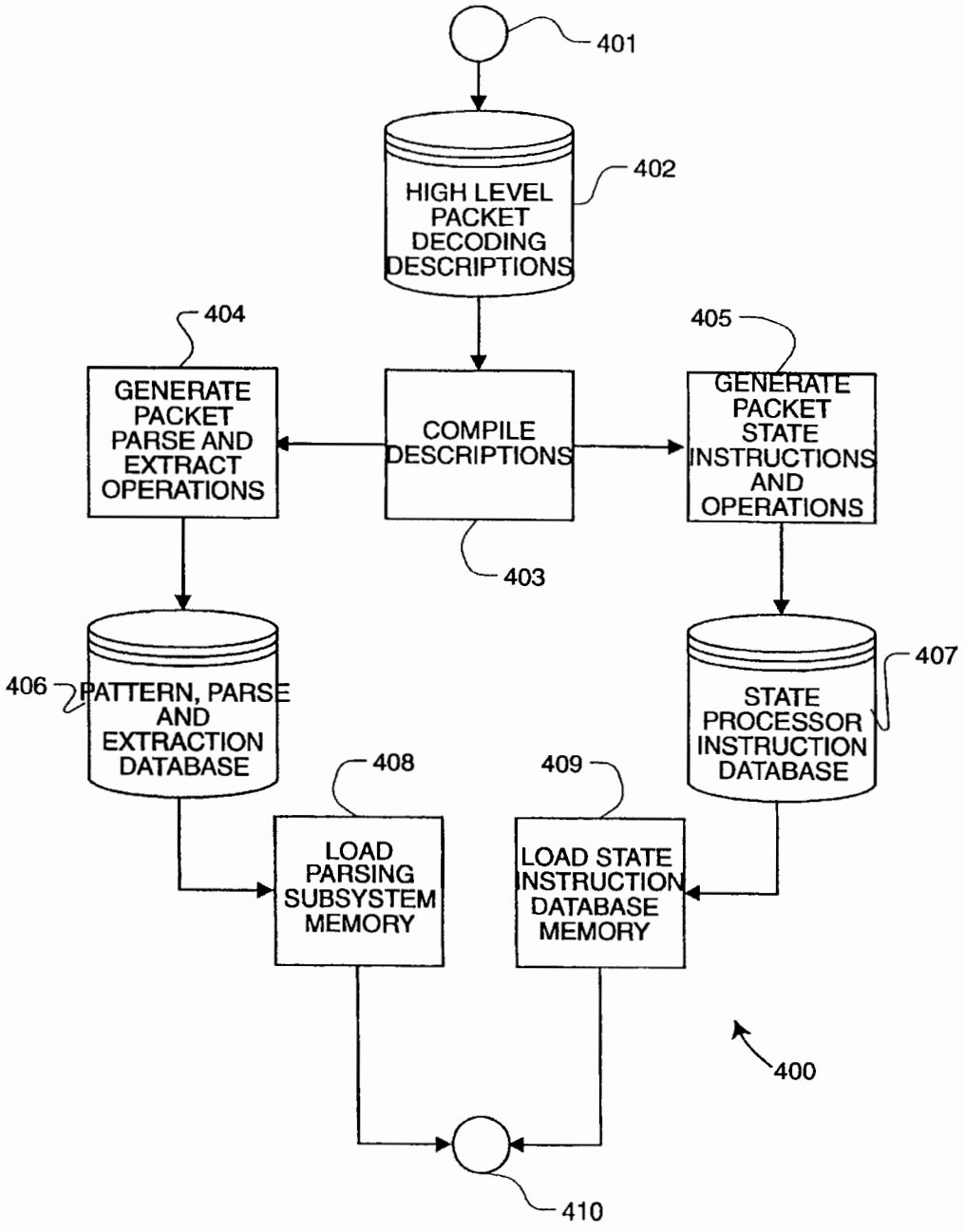

401
HIGH LEVEL PACKET DECODING DESCRIPTIONS
402
404
GENERATE PACKET PARSE AND EXTRACT OPERATIONS
COMPILE DESCRIPTIONS
403
405
GENERATE PACKET STATE INSTRUCTIONS AND OPERATIONS
406
PATTERN, PARSE AND EXTRACTION DATABASE
407
STATE PROCESSOR INSTRUCTION DATABASE
408
LOAD PARSING SUBSYSTEM MEMORY
409
LOAD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

FIG. 4

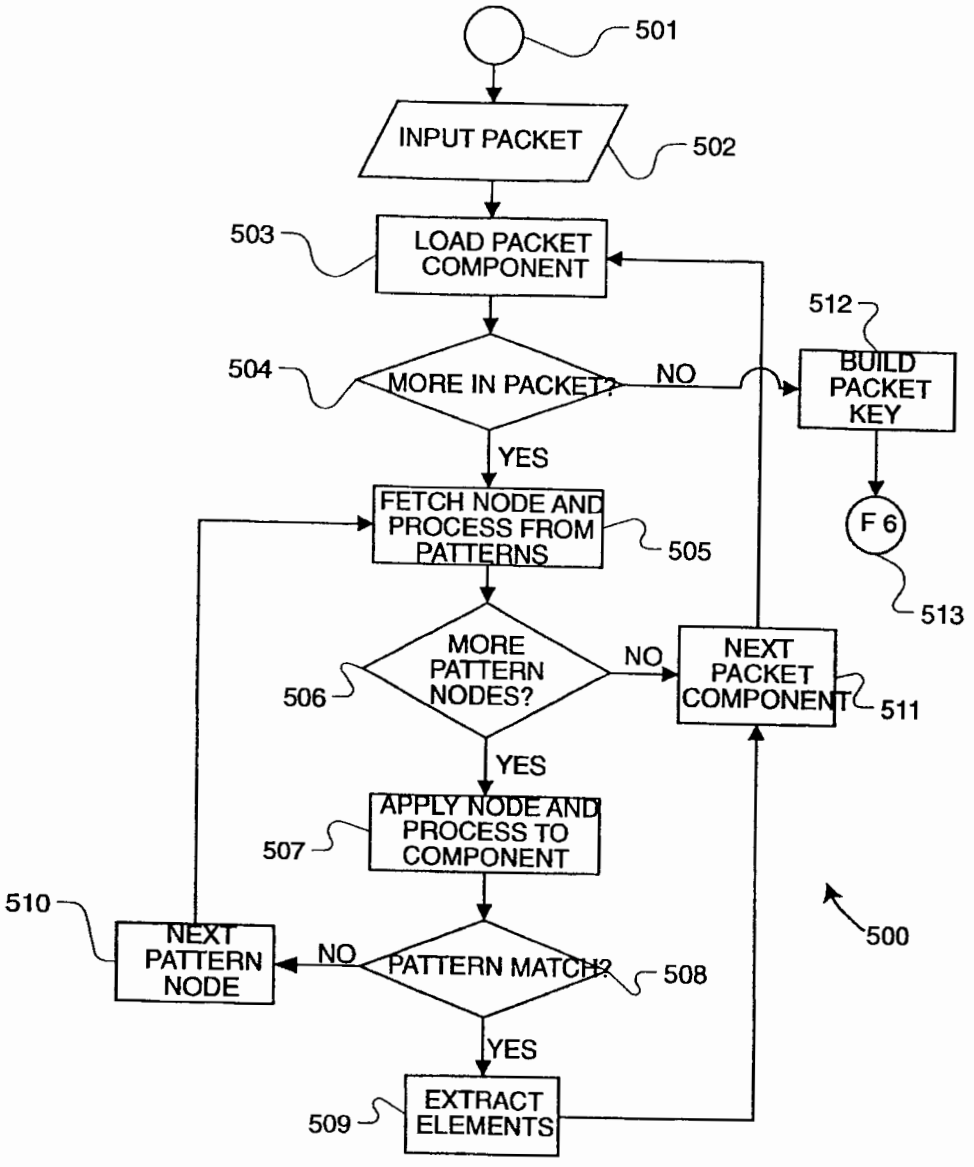

501
INPUT PACKET
502
503
LOAD PACKET COMPONENT
512
504
MORE IN PACKET?
NO
BUILD PACKET KEY
YES
FETCH NODE AND PROCESS FROM PATTERNS
505
F 6
513
MORE PATTERN NODES?
506
NO
NEXT PACKET COMPONENT
511
YES
APPLY NODE AND PROCESS TO COMPONENT
507
510
NEXT PATTERN NODE
NO
PATTERN MATCH?
508
500
YES
EXTRACT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

FIG. 5

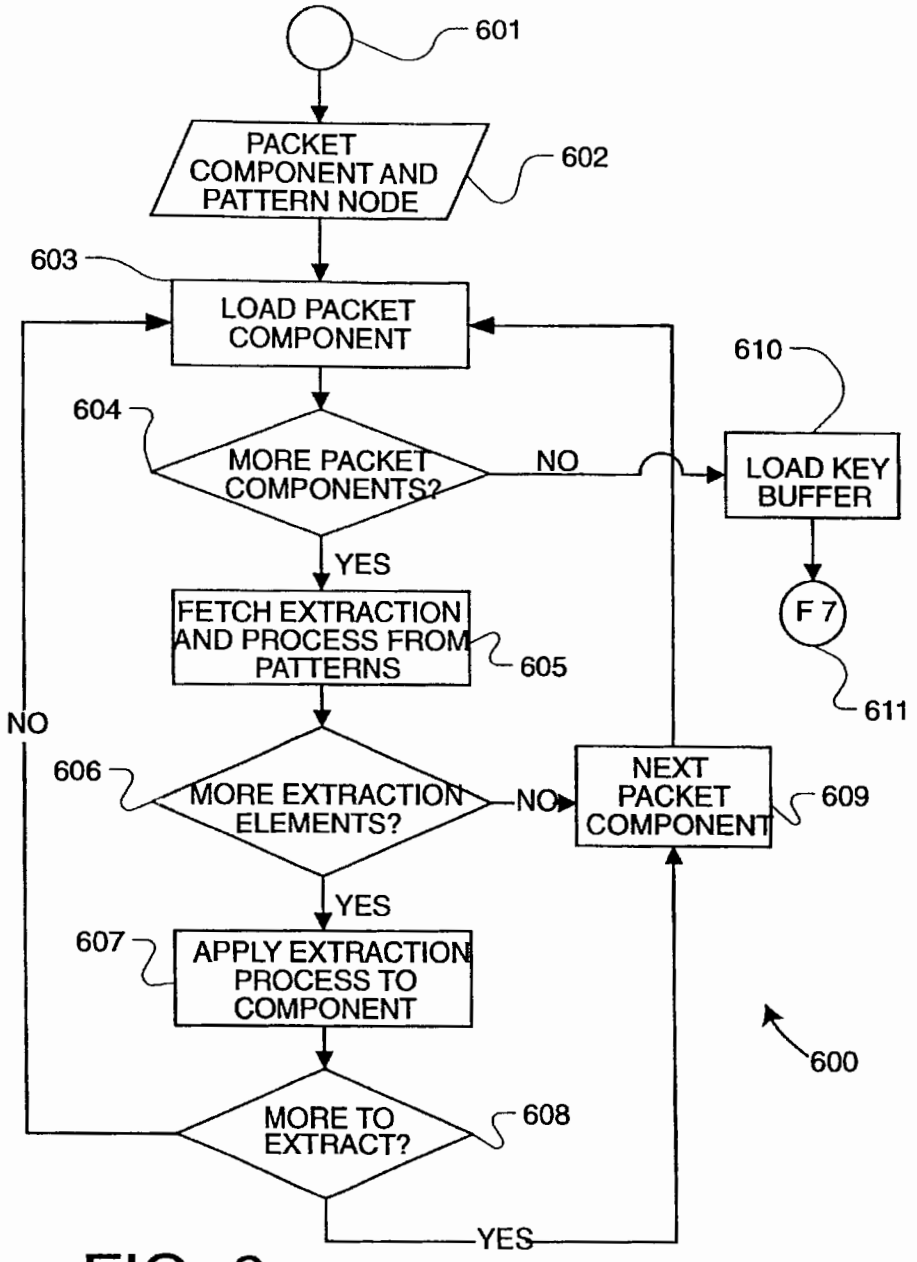

601
PACKET COMPONENT AND PATTERN NODE
602
603
LOAD PACKET COMPONENT
610
604
MORE PACKET COMPONENTS?
NO
LOAD KEY BUFFER
YES
F 7
611
FETCH EXTRACTION AND PROCESS FROM PATTERNS
605
NO
606
MORE EXTRACTION ELEMENTS?
NO
NEXT PACKET COMPONENT
609
YES
607
APPLY EXTRACTION PROCESS TO COMPONENT
600
MORE TO EXTRACT?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

FIG. 6

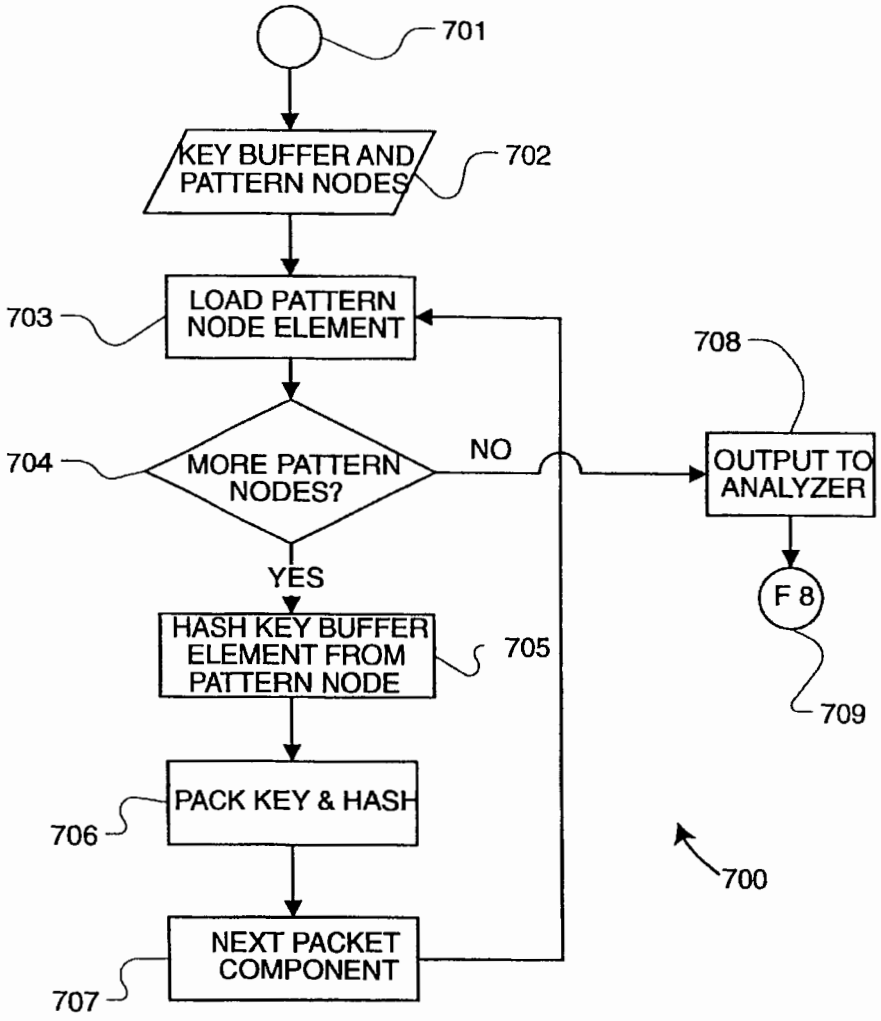

701
KEY BUFFER AND PATTERN NODES
702
LOAD PATTERN NODE ELEMENT
703
MORE PATTERN NODES?
704
NO
YES
OUTPUT TO ANALYZER
708
F 8
709
HASH KEY BUFFER ELEMENT FROM PATTERN NODE
705
PACK KEY & HASH
706
NEXT PACKET COMPONENT
707
700

FIG. 7

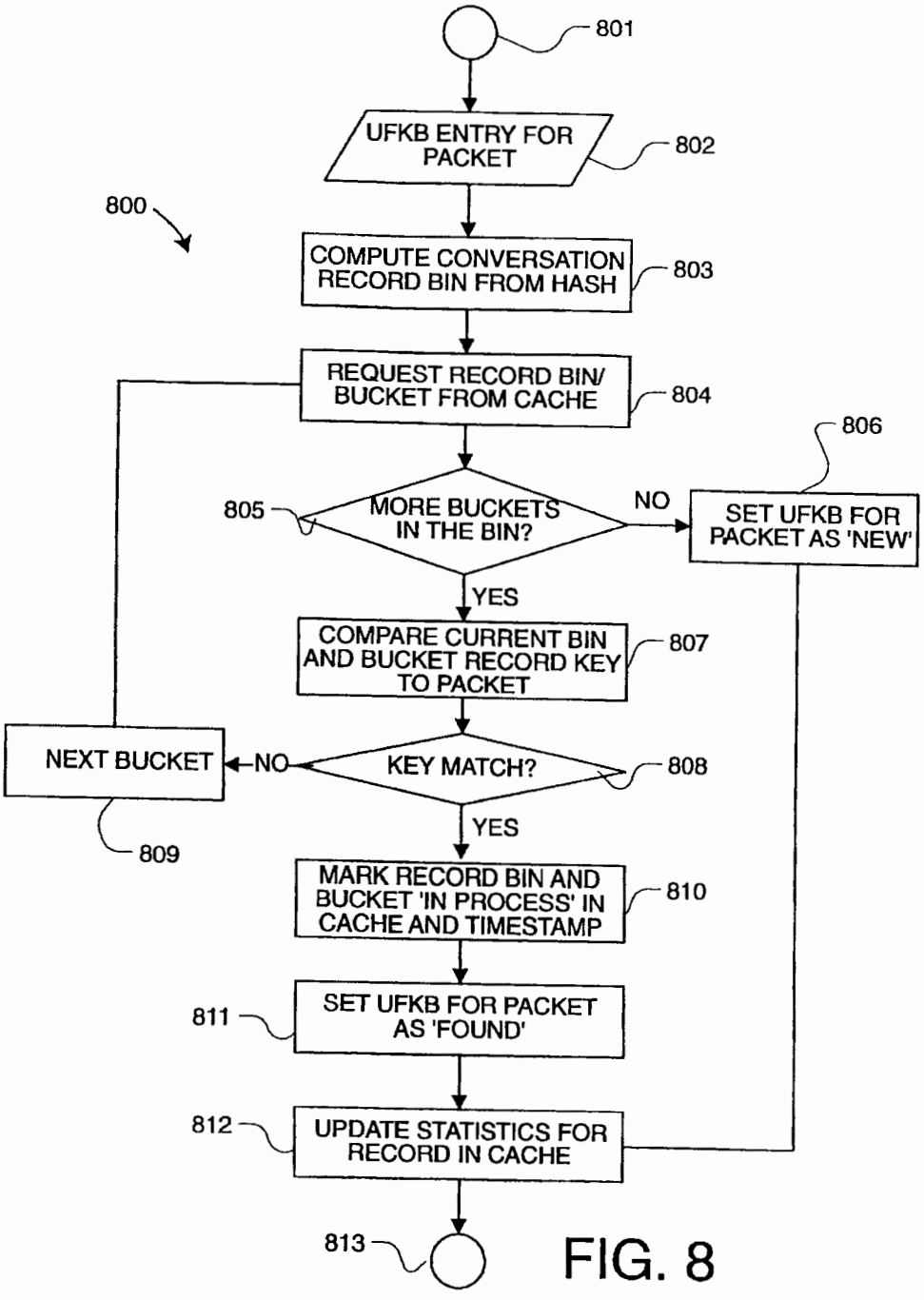

800
801
UFKB ENTRY FOR PACKET
802
COMPUTE CONVERSATION RECORD BIN FROM HASH
803
REQUEST RECORD BIN/ BUCKET FROM CACHE
804
806
MORE BUCKETS IN THE BIN?
805
NO
SET UFKB FOR PACKET AS 'NEW'
YES
COMPARE CURRENT BIN AND BUCKET RECORD KEY TO PACKET
807
NEXT BUCKET
NO
KEY MATCH?
808
YES
809
MARK RECORD BIN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

FIG. 8

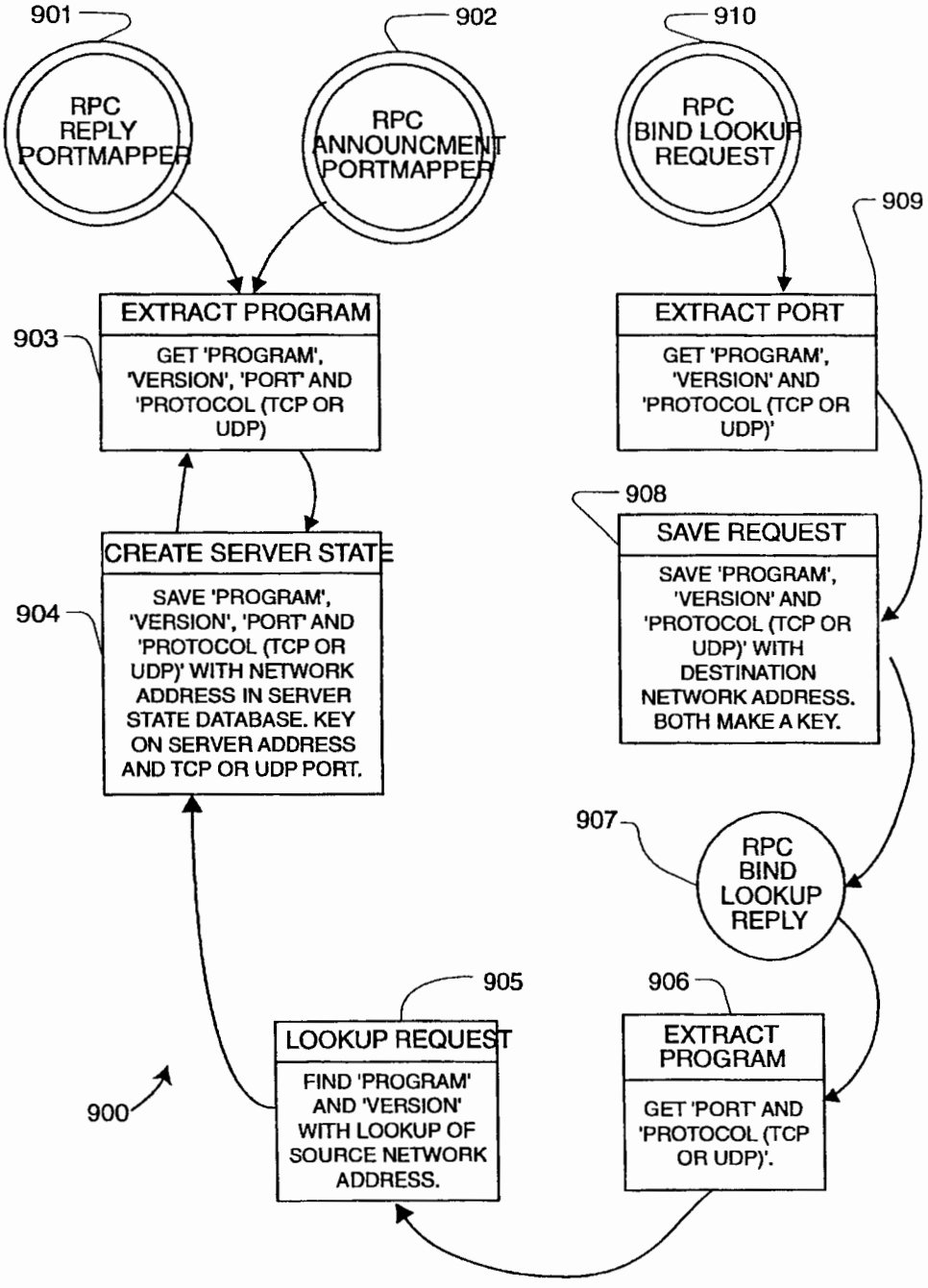

901
RPC REPLY PORTMAPPER
902
RPC ANNOUNCMENT PORTMAPPER
910
RPC BIND LOOKUP REQUEST
909
EXTRACT PROGRAM
GET 'PROGRAM', 'VERSION', 'PORT' AND 'PROTOCOL (TCP OR UDP)
903
EXTRACT PORT
GET 'PROGRAM', 'VERSION' AND 'PROTOCOL (TCP OR UDP)'
908
SAVE REQUEST
SAVE 'PROGRAM', 'VERSION' AND 'PROTOCOL (TCP OR UDP)'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

FIG. 9

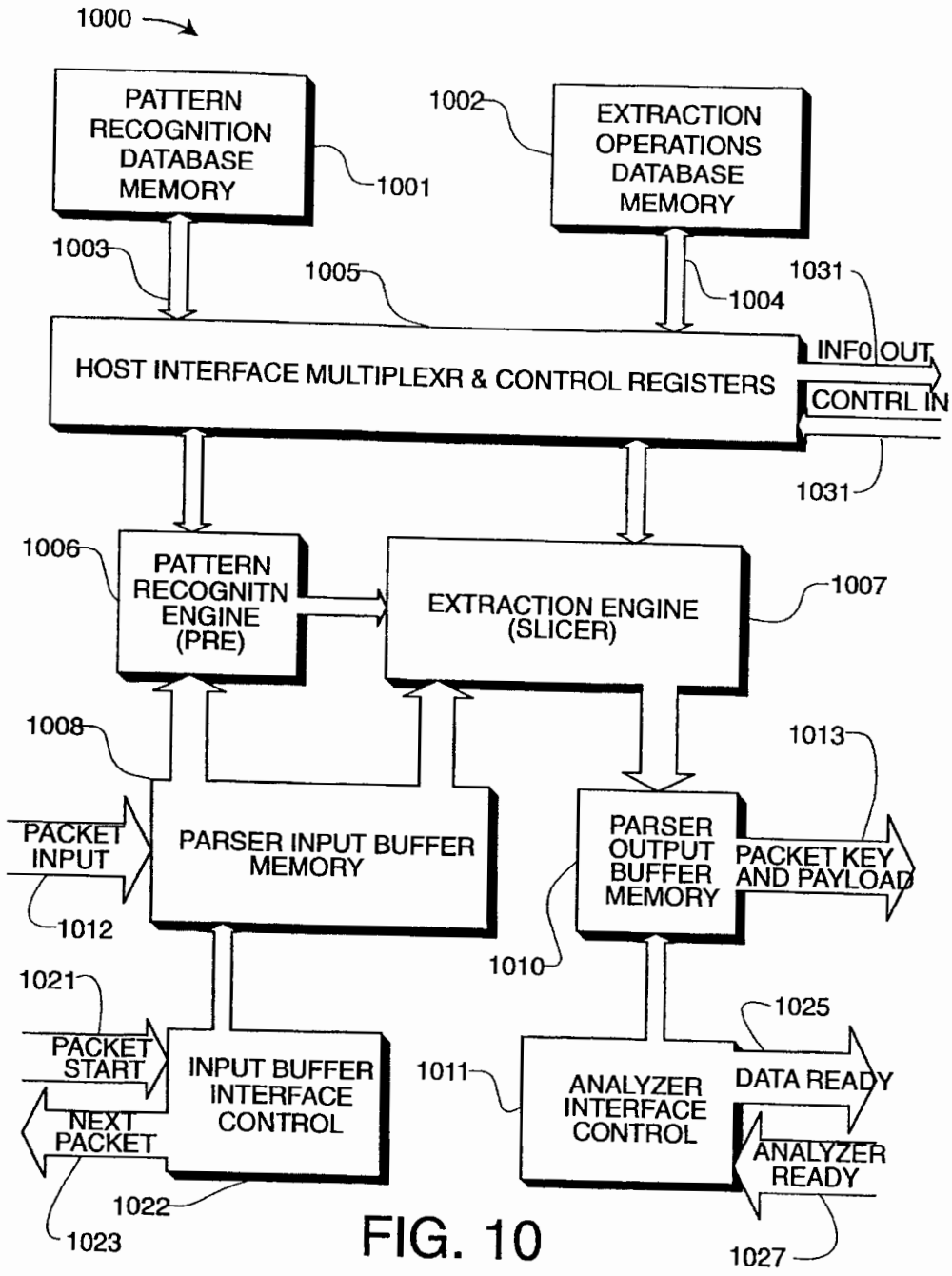

1000
PATTERN RECOGNITION DATABASE MEMORY
1001
1002
EXTRACTION OPERATIONS DATABASE MEMORY
1003
1005
1004
1031
HOST INTERFACE MULTIPLEXR & CONTROL REGISTERS
INFO OUT
CONTRL IN
1031
1006
PATTERN RECOGNITN ENGINE (PRE)
EXTRACTION ENGINE (SLICER)
1007
1008
1013
PACKET INPUT
PARSER INPUT BUFFER MEMORY
PARSER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

FIG. 10

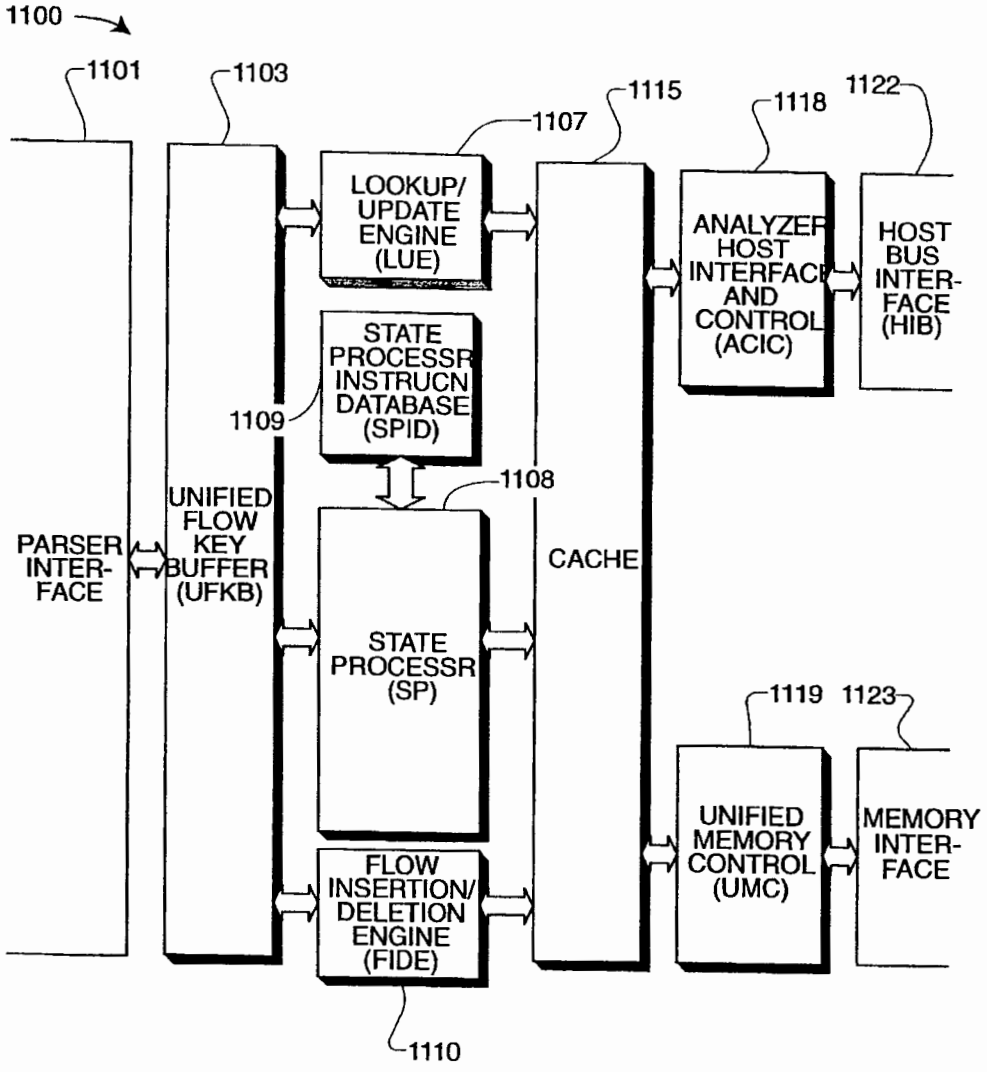

1100
1101
1103
1107
1115
1118
1122
LOOKUP/ UPDATE ENGINE (LUE)
ANALYZER HOST INTERFACE AND CONTROL (ACIC)
HOST BUS INTER-FACE (HIB)
STATE PROCESSR INSTRUCN DATABASE (SPID)
1109
1108
PARSER INTER-FACE
UNIFIED FLOW KEY BUFFER (UFKB)
STATE PROCESSR (SP)
CACHE
1119
1123
UNIFIED MEMORY CONTROL (UMC)
MEMORY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

FIG. 11

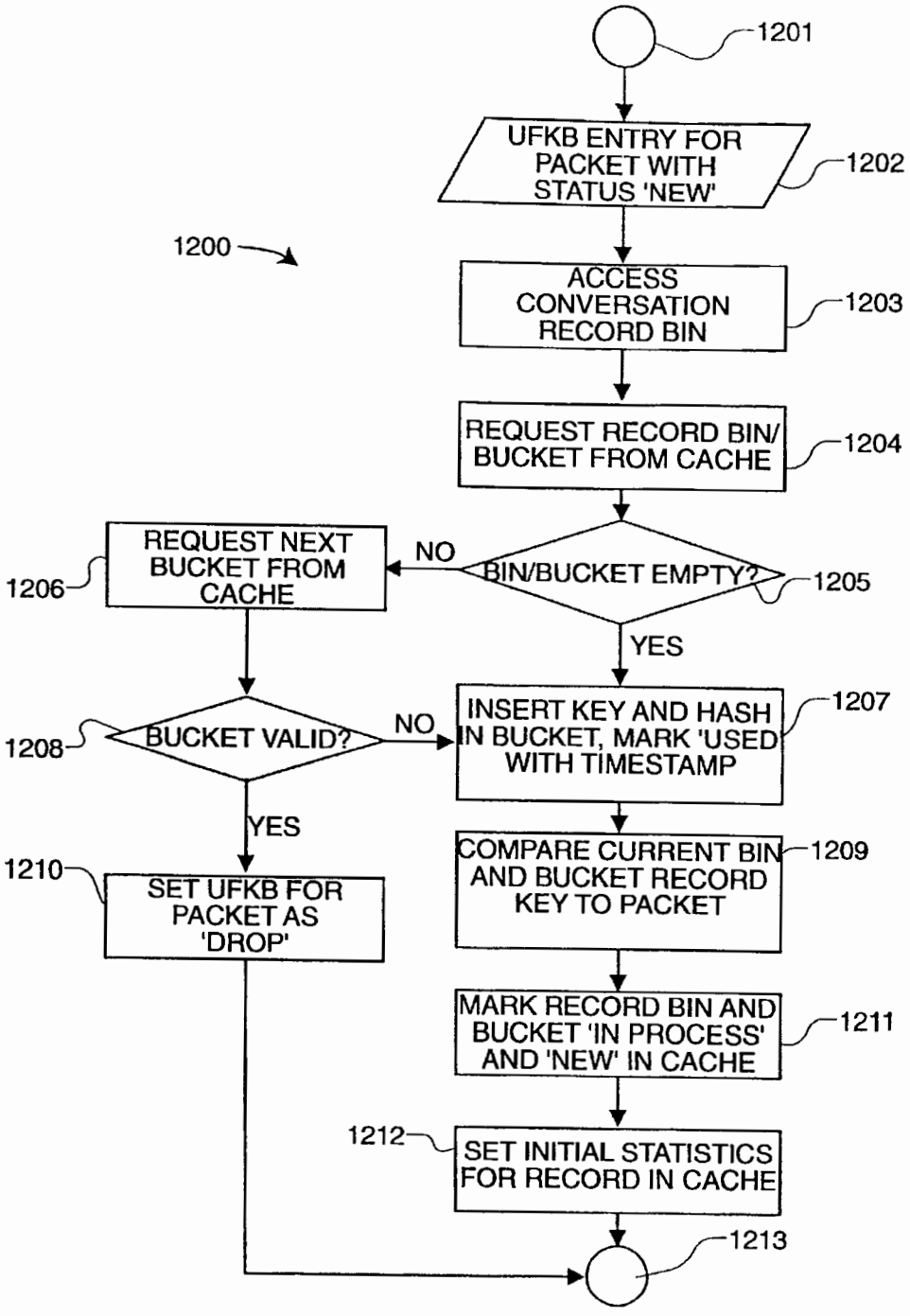

1201
UFKB ENTRY FOR PACKET WITH STATUS 'NEW'
1202
1200
ACCESS CONVERSATION RECORD BIN
1203
REQUEST RECORD BIN/ BUCKET FROM CACHE
1204
REQUEST NEXT BUCKET FROM CACHE
1206
NO
BIN/BUCKET EMPTY?
1205
YES
BUCKET VALID?
1208
NO
INSERT KEY AND HASH IN BUCKET, MARK 'USED' WITH TIMESTAMP
1207
YES
1210
SET UFKB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

FIG. 12

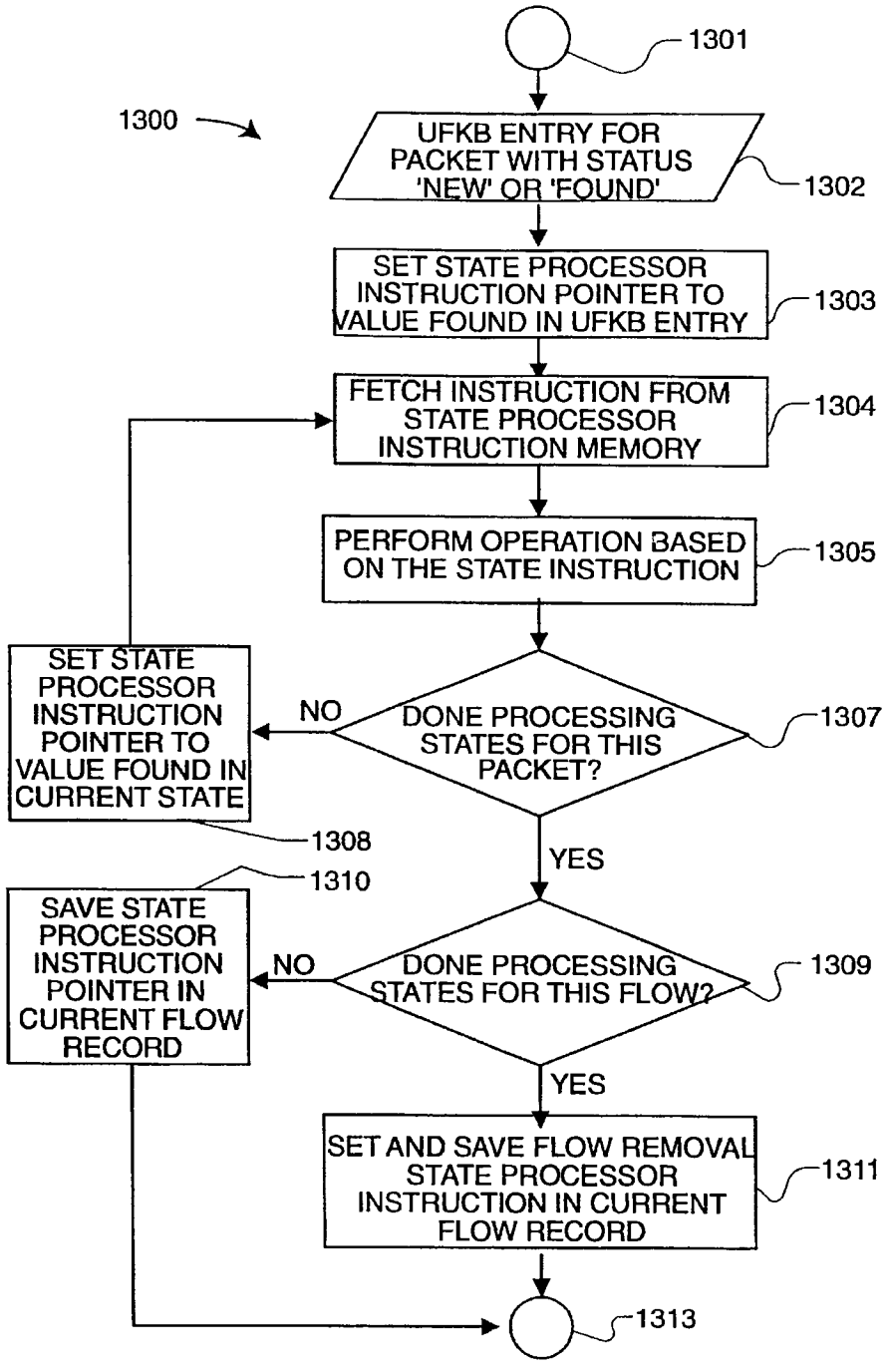

1301
1300
UFKB ENTRY FOR PACKET WITH STATUS 'NEW' OR 'FOUND'
1302
SET STATE PROCESSOR INSTRUCTION POINTER TO VALUE FOUND IN UFKB ENTRY
1303
FETCH INSTRUCTION FROM STATE PROCESSOR INSTRUCTION MEMORY
1304
PERFORM OPERATION BASED ON THE STATE INSTRUCTION
1305
DONE PROCESSING STATES FOR THIS PACKET?
1307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

FIG. 13

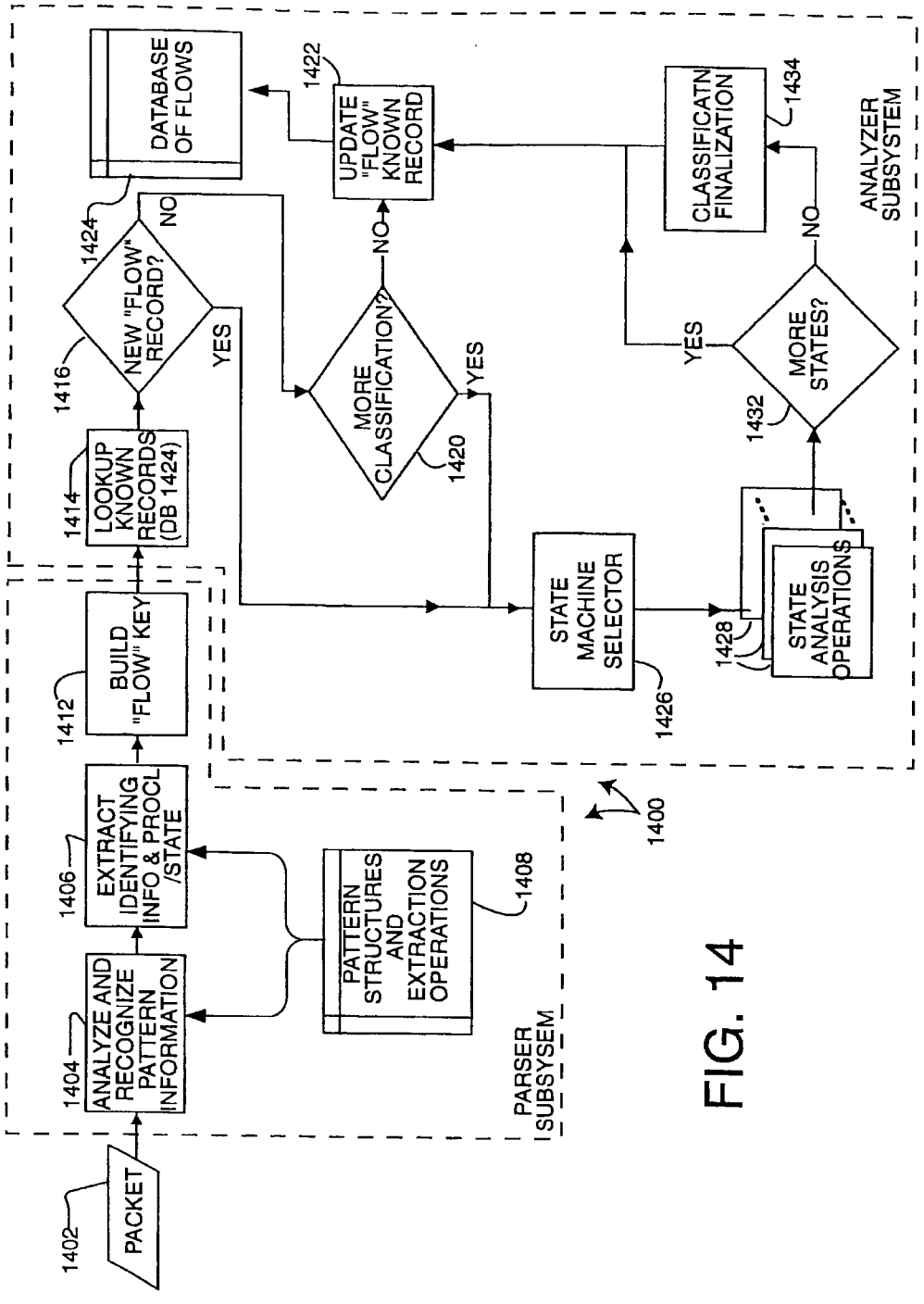

PACKET
1402
PARSER SUBSYSEM
ANALYZE AND RECOGNIZE PATTERN INFORMATION
1404
EXTRACT IDENTIFYING INFO & PROCL /STATE
1406
PATTERN STRUCTURES AND EXTRACTION OPERATIONS
1408
1400
BUILD "FLOW" KEY
1412
LOOKUP KNOWN RECORDS (DB 1424)
1414
NEW "FLOW" RECORD?
1416
DATABASE OF FLOWS
1424
MORE CLASSIFICATION?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

FIG. 14

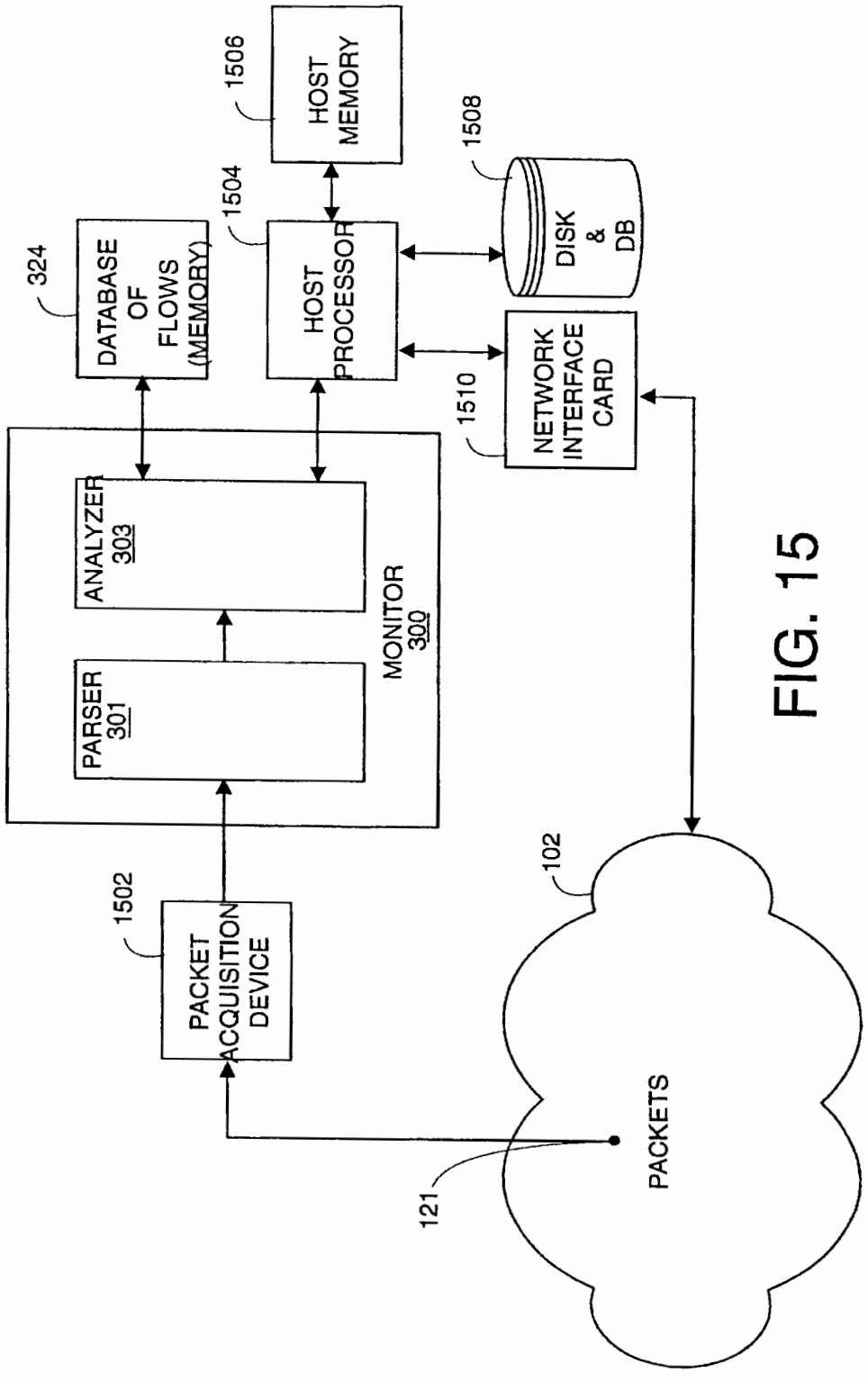

HOST MEMORY
1506
1508
DISK & DB
1504
HOST PROCESSOR
324
DATABASE OF FLOWS (MEMORY)
NETWORK INTERFACE CARD
1510
ANALYZER 303
MONITOR 300
PARSER 301
102
1502
PACKET ACQUISITION DEVICE
PACKETS
121

FIG. 15

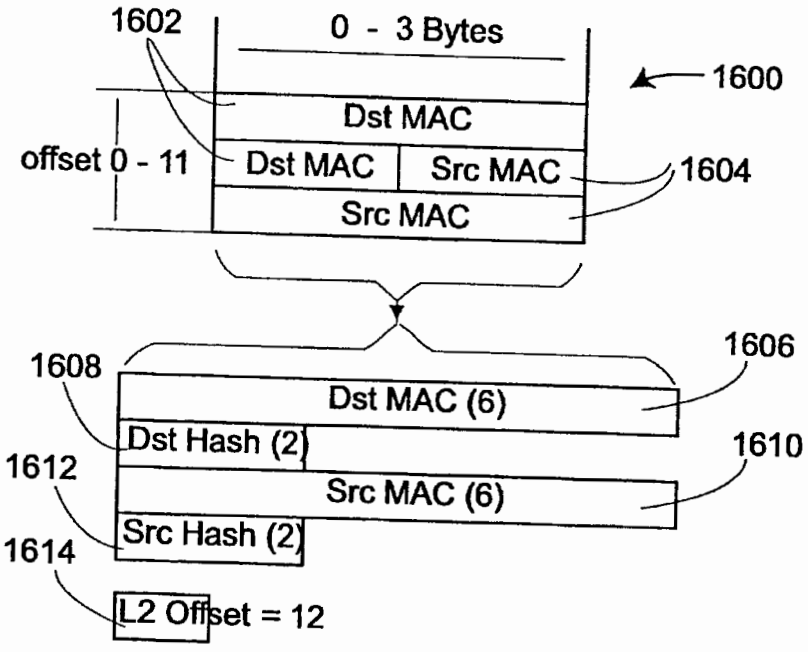

1602
0 - 3 Bytes
1600
Dst MAC
offset 0 - 11
Dst MAC
Src MAC
1604
Src MAC
1606
1608
Dst MAC (6)
Dst Hash (2)
1610
1612
Src MAC (6)
Src Hash (2)
1614
L2 Offset = 12

FIG. 16

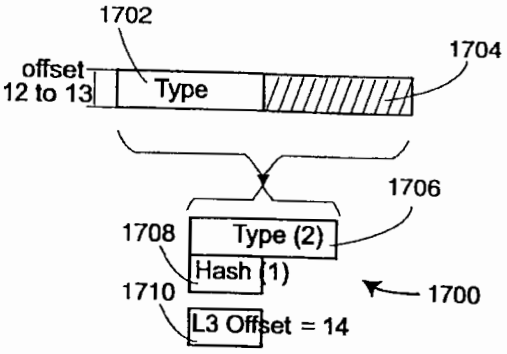


FIG. 17A

1712

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

Apple = 0x080007

* L3 Decoding
# L5 Decoding

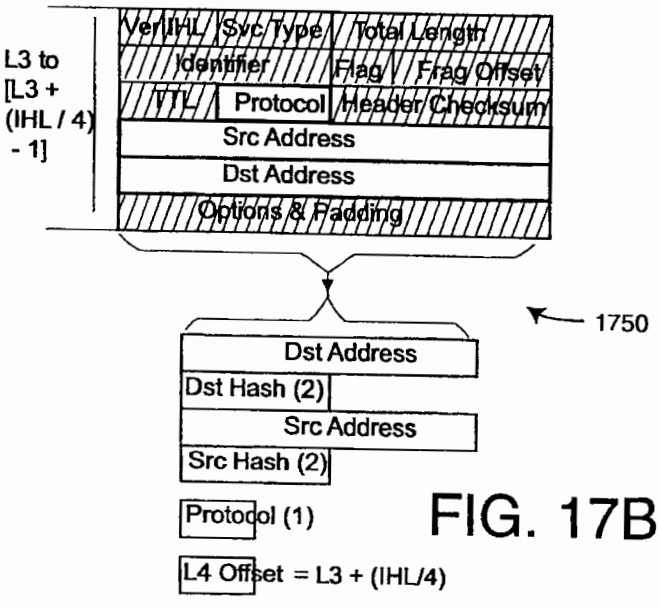


FIG. 17B

1752

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

* L4 Decoding
# L3 Re-Decoding

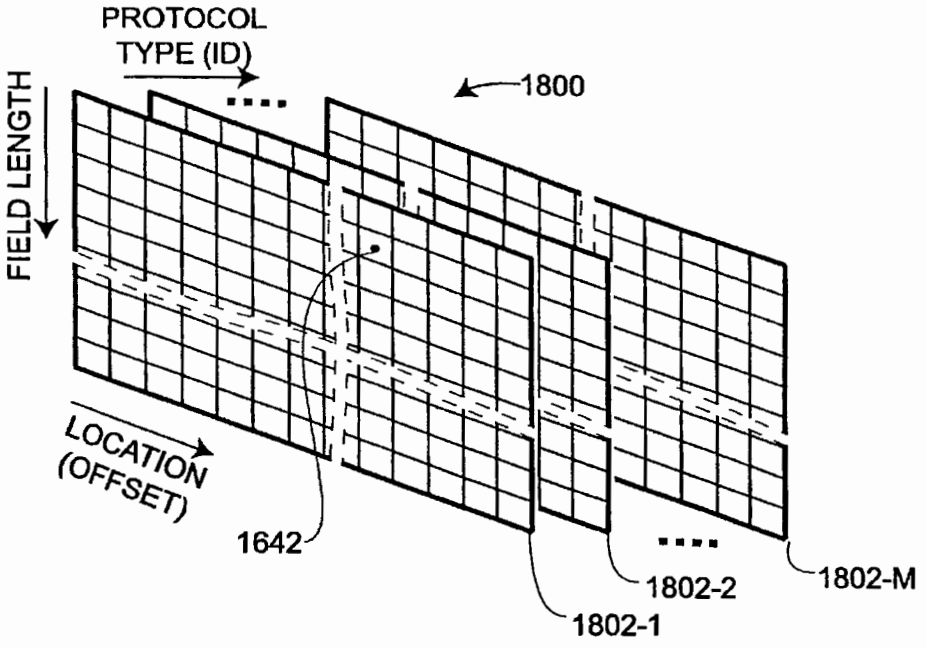


FIG. 18A

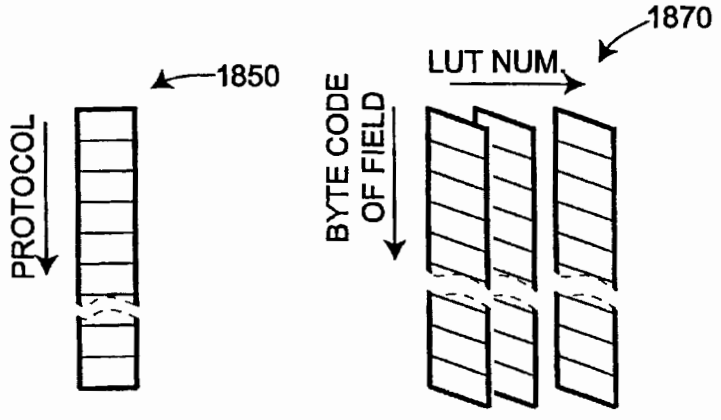


FIG. 18B

METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK

CROSS-REFERENCE TO RELATED APPLICATION

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

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

U.S. patent application Ser. No. 09/609,179 for PROCESSING PROTOCOL SPECIFIC INFORMATION IN PACKETS SPECIFIED BY A PROTOCOL DESCRIPTION LANGUAGE, to inventors Koppenhaver, et al., filed Jun. 30, 2000, still pending, and incorporated herein by reference. U.S. patent application Ser. No. 09/608,126 for RE-USING INFORMATION FROM DATA TRANSACTIONS FOR MAINTAINING STATISTICS IN NETWORK MONITORING, to inventors Dietz, et al., filed Jun. 30, 2000, still pending, and incorporated herein by reference. U.S. patent application Ser. No. 09/608,266 for ASSOCIATIVE CACHE STRUCTURE FOR LOOKUPS AND UPDATES OF FLOW RECORDS IN A NETWORK MONITOR, to inventors Sarkissian, et al., filed Jun. 30, 2000, still penting, and incorporated herein by reference. U.S. patent application Ser. No. 09/608,267 for STATE PROCESSOR FOR PATTERN MATCHING IN A NETWORK MONITOR DEVICE, to inventors Sarkissian, et al., filed Jun. 30, 2000, still pending, and incorporated herein by reference.

FIELD OF INVENTION

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

BACKGROUND TO THE PRESENT INVENTION

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

One prior art monitoring method uses log files. In this method, selected network activities may be analyzed retrospectively by reviewing log files, which are maintained by

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

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

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

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

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

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

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

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

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

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

Considering the previous SAP example again, because one features of the invention is to correctly identify the second exchange as being associated with a print service on that server, such exchange would even be recognized if the clients were not the same. What distinguishes this invention from prior art network monitors is that it has the ability to recognize disjointed flows as belonging to the same conversational flow.

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

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

Other prior art systems also are known. Phael describes a network activity monitor that processes only randomly selected packets in U.S. Pat. No. 5,315,580, titled "NETWORK MONITORING DEVICE AND SYSTEM." Nakamura teaches a network monitoring system in U.S. Pat. No. 4,891,639, titled "MONITORING SYSTEM OF NETWORK." Ross, et al., teach a method and apparatus for analyzing and monitoring network activity in U.S. Pat. No. 5,247,517, titled "METHOD AND APPARATUS FOR ANALYSIS NETWORKS," McCreery, et al., describe an Internet activity monitor that decodes packet data at the Internet protocol level layer in U.S. Pat. No. 5,787,253, titled "APPARATUS AND METHOD OF ANALYZING INTERNET ACTIVITY." The McCreery method decodes IP-packets. It goes through the decoding operations for each packet, and therefore uses the processing overhead for both recognized and unrecognized flows. In a monitor implementation of the present invention, a signature is built for every flow such that future packets of the flow are easily recognized. When a new packet in the flow arrives, the recognition process can commence from where it last left off, and a new signature built to recognize new packets of the flow.

SUMMARY

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

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

Other aspects of embodiments of the invention are:

- Properly analyzing each of the packets exchanged between a client and a server and maintaining information relevant to the current state of each of these conversational flows. p1 Providing a flexible process-

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

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

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

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

BRIEF DESCRIPTION OF THE DRAWINGS

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

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

Operation in a Network

FIG. **1** represents a system embodiment of the present invention that is referred to herein by the general reference numeral **100**. The system **100** has a computer network **102** that communicates packets (e.g., IP datagrams) between various computers, for example between the clients **104–107** and servers **110** and **112**. The network is shown schematically as a cloud with several network nodes and links shown in the interior of the cloud. A monitor **108** examines the packets passing in either direction past its connection point **121** and, according to one aspect of the invention, can elucidate what application programs are associated with

each packet. The monitor **108** is shown examining packets (i.e., datagrams) between the network interface **116** of the server **110** and the network. The monitor can also be placed at other points in the network, such as connection point **123** between the network **102** and the interface **118** of the client **104**, or some other location, as indicated schematically by connection point **125** somewhere in network **102**. Not shown is a network packet acquisition device at the location **123** on the network for converting the physical information on the network into packets for input into monitor **108**. Such packet acquisition devices are common.

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

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

ISO MODEL

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

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

In some communication arts, the term "frame" generally refers to encapsulated data at OSI layer 2, including a destination address, control bits for flow control, the data or payload, and CRC (cyclic redundancy check) data for error checking. The term "packet" generally refers to encapsulated data at OSI layer 3. In the TCP/IP world, the term "datagram" is also used. In this specification, the term "packet" is intended to encompass packets, datagrams, frames, and cells. In general, a packet format or frame format refers to how data is encapsulated with various fields and headers for transmission across a network. For example, a data packet typically includes an address destination field, a length field, an error correcting code (ECC) field, or cyclic redundancy check (CRC) field, as well as headers and footers to identify the beginning and end of the packet. The terms "packet format" and "frame format," also referred to as "cell format,"**0** are generally synonymous.

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

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

In real-world uses of the monitor **108**, the number of packets on the network **102** passing by the monitor **108**'s connection point can exceed a million per second. Consequently, the monitor has very little time available to analyze and type each packet and identify and maintain the state of the flows passing through the connection point. The monitor **108** therefore masks out all the unimportant parts of each packet that will not contribute to its classification. However, the parts to mask-out will change with each packet depending on which flow it belongs to and depending on the state of the flow.

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

In other cases, that first packet only starts the process of analyzing the conversational flow, and more packets are necessary to identify the associated application program. In such a case, a subsequent packet of a second type—but that potentially belongs to the same conversational flow—is recognized by using the signature. At such a second level, then, only a few of those application programs will have

conversational flows that can produce such a second packet type. At this level in the process of classification, all application programs that are not in the set of those that lead to such a sequence of packet types may be excluded in the process of classifying the conversational flow that includes these two packets. Based on the known patterns for the protocol and for the possible applications, a signature is produced that allows recognition of any future packets that may follow in the conversational flow.

It may be that the application is now recognized, or recognition may need to proceed to a third level of analysis using the second level signature. For each packet, therefore, the monitor parses the packet and generates a signature to determine if this signature identified a previously encountered flow, or shall be used to recognize future packets belonging to the same conversational flow. In real time, the packet is further analyzed in the context of the sequence of previously encountered packets (the state), and of the possible future sequences such a past sequence may generate in conversational flows associated with different applications. A new signature for recognizing future packets may also be generated. This process of analysis continues until the applications are identified. The last generated signature may then be used to efficiently recognize future packets associated with the same conversational flow. Such an arrangement makes it possible for the monitor **108** to cope with millions of packets per second that must be inspected.

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

The Network Monitor

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

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

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

The purpose of compiler and optimizer **310** is to provide protocol specific information to parser subsystem **301** and to analyzer subsystem **303**. The initialization occurs prior to operation of the monitor, and only needs to re-occur when new protocols are to be added.

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

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

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

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

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

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

In the preferred embodiment, the packet **302** from the acquisition device is input into a packet buffer. The pattern recognition process **304** is carried out by a pattern analysis

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

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

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

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

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

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

The determination (**316**) of whether a record with the same signature already exists is carried out by a lookup engine (LUE) that obtains new UFKB records and uses the hash in the UFKB record to lookup if there is a matching known flow. In the particular embodiment, the database of known flows **324** is in an external memory. A cache is associated with the database **324**. A lookup by the LUE for a known record is carried out by accessing the cache using the hash, and if the entry is not already present in the cache, the entry is looked up (again using the hash) in the external memory.

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

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

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

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

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

The state processor **328** analyzes both new and existing flows in order to analyze all levels of the protocol stack, ultimately classifying the flows by application (level **7** in the ISO model). It does this by proceeding from state-to-state based on predefined state transition rules and state opera-

tions as specified in state processor instruction database **326**. A state transition rule is a rule typically containing a test followed by the next-state to proceed to if the test result is true. An operation is an operation to be performed while the state processor is in a particular state—for example, in order to evaluate a quantity needed to apply the state transition rule. The state processor goes through each rule and each state process until the test is true, or there are no more tests to perform.

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

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

Thus, state processor **328** applies the first state operation to the packet for this particular flow-entry. A process **330** decides if more operations need to be performed for this state. If so, the analyzer continues looping between block **330** and **328** applying additional state operations to this particular packet until all those operations are completed—that is, there are no more operations for this packet in this state. A process **332** decides if there are further states to be analyzed for this type of flow according to the state of the flow and the protocol, in order to fully characterize the flow. If not, the conversational flow has now been fully characterized and a process **334** finalizes the classification of the conversational flow for the flow.

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

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

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

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

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

Thus, the embodiment of present invention shown in FIG. **3** automatically maintains flow-entries, which in one aspect includes storing states. The monitor of FIG. **3** also generates characteristic parts of packets—the signatures—that can be used to recognize flows. The flow-entries may be identified and accessed by their signatures. Once a packet is identified to be from a known flow, the state of the flow is known and this knowledge enables state transition analysis to be performed in real time for each different protocol and application. In a complex analysis, state transitions are traversed as more and more packets are examined. Future packets that are part of the same conversational flow have their state analysis continued from a previously achieved state. When enough packets related to an application of interest have been processed, a final recognition state is ultimately reached, i.e., a set of states has been traversed by state analysis to completely characterize the conversational flow. The signature for that final state enables each new incoming packet of the same conversational flow to be individually recognized in real time.

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

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

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

Detailed Operation

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

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

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

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

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

An alternate embodiment of the data structure used in database **308** is illustrated in FIG. **18**B. Thus, like the 3-D structure of FIG. **18**A, the data structure permits rapid searches to be performed by the pattern recognition process **304** by indexing locations in a memory rather than performing address link computations. In this alternate embodiment, the PRD **308** includes two parts, a single protocol table **1850** (PT) which has an entry for each protocol known for the monitor, and a series of Look Up Tables **1870** (LUT's) that are used to identify known protocols and their children. The protocol table includes the parameters needed by the pattern analysis and recognition process **304** (implemented by PRE **1006**) to evaluate the header information in the packet that is associated with that protocol, and parameters needed by extraction process **306** (implemented by slicer **1007**) to process the packet header. When there are children, the PT describes which bytes in the header to evaluate to determine the child protocol. In particular, each PT entry contains the header length, an offset to the child, a slicer command, and some flags.

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

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

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

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

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

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

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

FIG. **5** shows a flowchart of how actual parser subsystem **301** functions. Starting at **501**, the packet **302** is input to the packet buffer in step **502**. Step **503** loads the next (initially the first) packet component from the packet **302**. The packet components are extracted from each packet **302** one element at a time. A check is made (**504**) to determine if the load-packet-component operation **503** succeeded, indicating that there was more in the packet to process. If not, indicating all components have been loaded, the parser subsystem **301** builds the packet signature (**512**)—the next stage (FIG. **6**).

If a component is successfully loaded in **503**, the node and processes are fetched (**505**) from the pattern, parse and extraction database **308** to provide a set of patterns and

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

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

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

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

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

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

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

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

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

FIG. **8** is a flow diagram describing the operation of the lookup/update engine (LUE) that implements lookup operation **314**. The process starts at **801** from FIG. **7** with the parser record that includes a signature, the hash and at least parts of the payload. In **802** those elements are shown in the form of a UFKB-entry in the buffer. The LUE, the lookup engine **314** computes a "record bin number" from the hash for a flow-entry. A bin herein may have one or more "buckets" each containing a flow-entry. The preferred embodiment has four buckets per bin.

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

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

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

To provide for such analyses, the lookup/update engine updates one or more counters that are part of the flow-entry (in the cache) in step **812**. The process exits at **813**. In our embodiment, the counters include the total packets of the flow, the time, and a differential time from the last timestamp to the present timestamp.

It may be that the bucket of the bin did not lead to a signature match (test **808**). In such a case, the analyzer in

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

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

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

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

The Hardware System

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

FIG. 10 is a description of the parsing subsystem (301, shown here as subsystem 1000) as implemented in hardware. Memory 1001 is the pattern recognition database memory, in which the patterns that are going to be analyzed are stored. Memory 1002 is the extraction-operation database memory, in which the extraction instructions are stored. Both 1001 and 1002 correspond to internal data structure 308 of FIG. 3. Typically, the system is initialized from a microprocessor (not shown) at which time these memories are loaded through a host interface multiplexor and control register 1005 via the internal buses 1003 and 1004. Note that the contents of 1001 and 1002 are preferably obtained by compiling process 310 of FIG. 3.

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

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

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

The PRE 1006 includes of a comparison engine. The comparison engine has a first stage that checks the protocol type field to determine if it is an 802.3 packet and the field should be treated as a length. If it is not a length, the protocol is checked in a second stage. The first stage is the only protocol level that is not programmable. The second stage has two full sixteen bit content addressable memories (CAMs) defined for future protocol additions.

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

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

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

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

Once all the selected parts of the packet used to form the signature are extracted, the hash is loaded into parser output buffer memory 1010. Any additional payload from the packet that is required for further analysis is also included. The parser output memory 1010 is interfaced with the analyzer subsystem by analyzer interface control 1011. Once all the information of a packet is in the parser output buffer memory 1010, a data ready signal 1025 is asserted by

analyzer interface control. The data from the parser subsystem **1000** is moved to the analyzer subsystem via **1013** when an analyzer ready signal **1027** is asserted.

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

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

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

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

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

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

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

The cache subsystem **1115** is an associative cache that includes a set of content addressable memory cells (CAMs) each including an address portion and a pointer portion pointing to the cache memory (e.g., RAM) containing the cached flow-entries. The CAMs are arranged as a stack ordered from a top CAM to a bottom CAM. The bottom CAM's pointer points to the least recently used (LRU) cache memory entry. Whenever there is a cache miss, the contents of cache memory pointed to by the bottom CAM are replaced by the flow-entry from the flow-entry database **324**. This now becomes the most recently used entry, so the contents of the bottom CAM are moved to the top CAM and all CAM contents are shifted down. Thus, the cache is an associative cache with a true LRU replacement policy.

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

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

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

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

processing and the status for the record is set to indicate the LUE has processed the record.

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

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

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

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

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

FIG. **13** describes the operation of the state processor **1108**. The state processor is entered at **1301** with a unified flow key buffer entry to be processed. The UFKB-entry is new or corresponding to a found flow-entry. This UFKB-entry is retrieved from unified flow key buffer **1103** in **1301**. In **1303**, the protocol identifier for the UFKB-entry is used to set the state processor's instruction counter. The state processor **1108** starts the process by using the last protocol recognized by the parser subsystem **301** as an offset into a jump table. The jump table takes us to the instructions to use for that protocol. Most instructions test something in the unified flow key buffer or the flow-entry if it exists. The state processor **1108** may have to test bits, do comparisons, add or subtract to perform the test.

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

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

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

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

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

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

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

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

The process of flow insertion is now described with the aid of FIG. **12**. Flows are grouped into bins of buckets by the hash value. The engine processes a UFKB-entry that may be new or that the state processor otherwise has indicated needs to be created. FIG. **12** shows the case of a new entry being created. A conversation record bin (preferably containing 4 buckets for four records) is obtained in **1203**. This is a bin that matches the hash of the UFKB, so this bin may already have been sought for the UFKB-entry by the LUE. In **1204** the FIDE **1110** requests that the record bin/bucket be maintained in the cache system **1115**. If in **1205** the cache system **1115** indicates that the bin/bucket is empty, step **1207** inserts the flow signature (with the hash) into the bucket and the bucket is marked "used" in the cache engine of cache **1115** using a timestamp that is maintained throughout the process. In **1209**, the FIDE **1110** compares the bin and bucket record flow signature to the packet to verify that all the elements are in place to complete the record. In **1211** the system marks the record bin and bucket as "in process" and as "new" in the

cache system (and hence in the external memory). In **1212**, the initial statistical measures for the flow-record are set in the cache system. This in the preferred embodiment clears the set of counters used to maintain statistics, and may perform other procedures for statistical operations requires by the analyzer for the first packet seen for a particular flow.

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

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

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

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

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

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

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

In accordance with one hardware implementation, the elements shown in FIGS. **10** and **11** are implemented in a set of six field programmable logic arrays (FPGA's). The boundaries of these FPGA's are as follows. The parsing subsystem of FIG. **10** is implemented as two FPGAS; one FPGA, and includes blocks **1006**, **1008** and **1012**, parts of **1005**, and memory **1001**. The second FPGA includes **1002**, **1007**, **1013**, **1011** parts of **1005**. Referring to FIG. **11**, the unified look-up buffer **1103** is implemented as a single FPGA. State processor **1108** and part of state processor instruction database memory **1109** is another FPGA. Portions of the state processor instruction database memory **1109** are maintained in external SRAM's. The lookup/update engine **1107** and the flow insertion/deletion engine **1110** are in another FPGA. The sixth FPGA includes the cache system **1115**, the unified memory control **1119**, and the analyzer host interface and control **1118**.

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

Operation of the Invention

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

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

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

Examples of Packet Elucidation

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

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

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

"Session tracking" also is known as one of the primary processes for tracking applications in client/server packet exchanges. The process of tracking sessions requires an initial connection to a predefined socket or port number. This method of communication is used in a variety of transport layer protocols. It is most commonly seen in the TCP and UDP transport protocols of the IP protocol.

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

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

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

Server Announcement Example

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

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

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

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

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

1. A client (CLIENT **3**, **106** in FIG. **1**) sends a TCP packet to SERVER **2** (**110** in FIG. **1**) on port **111**, with an RPC Bind

Lookup Request (rpcBindLookup). TCP or UDP port **111** is always associated Sun RPC. This request specifies the program (as a program identifier), version, and might specify the protocol (UDP or TCP).

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

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

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

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

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

Process **909**: Extract the 'program,' 'version,' and 'protocol' (UDP or TCP).

Extract the TCP or UDP port (process **909**) which is **111** indicating Sun RPC.

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

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

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

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

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

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

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

In packet **206**, a fourth field **217** and a fifth field **218** are used to communicate port numbers that are used. The conversation direction determines where the port number field is. The diagonal pattern in field **217** is used to identify a source-port pattern, and the hash pattern in field **218** is used to identify the destination-port pattern. The order indicates the client-server message direction. A sixth field denoted "i^{1}" **219** is an element that is being requested by the client from the server. A seventh field denoted "s_1a" **220** is the service requested by the client from server **110**. The

following eighth field "QA" **221** (for question mark) indicates that the client **106** wants to know what to use to access application "s_1a". A tenth field "QP" **223** is used to indicate that the client wants the server to indicate what protocol to use for the particular application.

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

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

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

The flow signature and flow states built up as a result of this exchange are now described. When the packet monitor **300** sees the request packet **206** from the client, a first flow signature **210** is built in the parser subsystem **301** according to the pattern and extraction operations database **308**. This signature **210** includes a destination and a source address **240** and **241**. One aspect of the invention is that the flow keys are built consistently in a particular order no matter what the direction of conversation. Several mechanisms may be used to achieve this. In the particular embodiment, the numerically lower address is always placed before the numerically higher address. Such least to highest order is used to get the best spread of signatures and hashes for the lookup operations. In this case, therefore, since we assume "S_1"<"C_1", the order is address "S_1" followed by client address "C_1". The next field used to build the signature is a protocol field **242** extracted from packet **206**'s field **216**, and thus is the protocol "p^1". The next field used for the signature is field **243**, which contains the destination source port number shown as a crosshatched pattern from the field **218** of the packet **206**. This pattern will be recognized in the payload of packets to derive how this packet or sequence of packets exists as a flow. In practice, these may be TCP port numbers, or a combination of TCP port numbers. In the case of the Sun RPC example, the crosshatch represents a set of port numbers of UDS for p^1 that will be used to recognize this flow (e.g., port **111**). Port **111** indicates this is Sun RPC. Some applications, such as the Sun RPC Bind Lookups, are directly determinable ("known") at the parser level. So in this case, the signature KEY-1 points to a known application denoted "a^1" (Sun RPC Bind Lookup), and a next—state that the state processor should proceed to for more complex recognition jobs, denoted as state "st_D" is placed in the field **245** of the flow-entry.

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

The two flow signatures **210** and **212** always order the destination and source address fields with server "S_1" followed by client "C_1". Such values are automatically filled in when the addresses are first created in a particular flow signature. Preferably, large collections of flow signatures are kept in a lookup table in a least-to-highest order for the best spread of flow signatures and hashes.

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

Some network-server application recognition jobs are so simple that only a single state transition has to occur to be able to pinpoint the application that produced the packet.

Others require a sequence of state transitions to occur in order to match a known and predefined climb from state-to-state.

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

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

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

Although the present invention has been described in terms of the presently preferred embodiments, it is to be understood that the disclosure is not to be interpreted as limiting. Various alterations and modifications will no doubt become apparent to those or ordinary skill in the art after having read the above disclosure. Accordingly, it is intended that the claims be interpreted as covering all alterations and modifications as fall within the true spirit and scope of the present invention.

What is claimed is:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

* * * * *

US006650640B1

(12) **United States Patent**
Muller et al.

(10) **Patent No.: US 6,650,640 B1**
(45) **Date of Patent: Nov. 18, 2003**

(54) **METHOD AND APPARATUS FOR MANAGING A NETWORK FLOW IN A HIGH PERFORMANCE NETWORK INTERFACE**

(75) Inventors: **Shimon Muller**, Sunnyvale, CA (US); **Denton E. Gentry, Jr.**, Fremont, CA (US)

(73) Assignee: **Sun Microsystems, Inc.**, Santa Clara, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/259,932**

(22) Filed: **Mar. 1, 1999**

(51) **Int. Cl.**[7] **G06F 13/00**
(52) **U.S. Cl.** **370/392**; 370/473
(58) **Field of Search** 370/225, 230, 370/231, 235, 236, 241, 389, 392, 401, 427, 428, 473

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,858,232	A *	8/1989	Diaz et al.	370/465
5,414,704	A	5/1995	Spinney	370/60
5,583,940	A	12/1996	Vidrascu et al.	380/49

(List continued on next page.)

FOREIGN PATENT DOCUMENTS

EP	0 447 725	9/1991	G06F/15/16
EP	0 573 739	12/1993	H04L/12/56
EP	0 853 411	7/1998	H04L/29/06
EP	0 865 180	9/1998	H04L/12/56
JP	09247172	* 9/1997	H04L/12/28
WO	WO 95/14269	5/1995	G06F/7/08
WO	WO 97/28505	8/1997	G06F/13/14
WO	WO 99/00737	1/1999	G06F/13/00
WO	WO 99/00945	1/1999	H04L/12/46
WO	WO 99/00948	1/1999	H04L/12/56
WO	WO 99/00949	1/1999	H04L/12/56

OTHER PUBLICATIONS

Newman, Peter, et al., "IP Switching and Gigabit Routers" IEEE Communications Magazine, vol. 335, No. 1, Jan. 1997, pp. 64–69.

Le Faucheur, Francois, "IETF Multiprotocol Label Switching (MPLS) Architecture" IEEE International Conference, Jun. 22, 1998 pp. 6–15.

Hallsall,F., "Data Communications, Computer Networks and Open Systems", Electronic Systems Engineering Series, 1996, pp. 451–452.

Cole, R., et al., "IP Over ATM: A Framework Document" IETF Online, Apr. 1996, pp. 1–31.

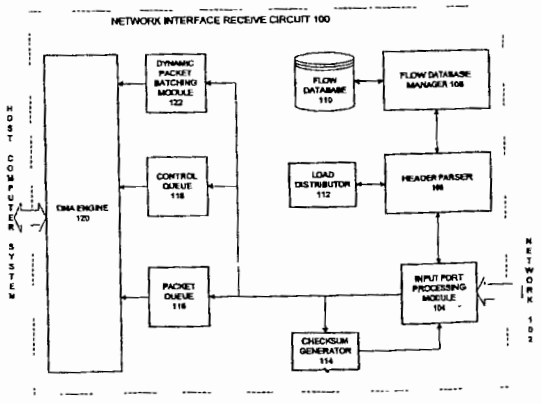
(List continued on next page.)

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

(57) **ABSTRACT**

A system and method are provided for managing information concerning a network flow comprising packets sent from a source entity to a destination entity served by a network interface. A network flow is established for each datagram sent from the source entity to the destination entity. A flow key, identifying the source and destination entities, is stored in a data structure along with information concerning validity of the flow, sequence of data in the flow datagram and how recently the flow was active. Once a flow is established, it is updated each time a packet containing data from the flow's datagram is received. When such a packet is received, an operation code is generated for identifying whether the packet is suitable for a particular network interface function. An operation code may, for example, indicate that a packet contains data to be re-assembled with other data from the same flow. Another operation code may indicate that a packet is not suitable for data re-assembly. Another operation code may specify that the packet is simply a control packet, has no data, or that the packet was received out of order.

59 Claims, 49 Drawing Sheets



U.S. PATENT DOCUMENTS

5,684,954	A		11/1997	Kaiserswerth et al. ... 295/200.2
5,742,765	A	*	4/1998	Wong et al. 709/230
5,748,905	A		5/1998	Hauser et al. 395/200.79
5,758,089	A		5/1998	Gentry et al. 395/200.64
5,778,180	A		7/1998	Gentry et al. 395/200.42
5,778,414	A		7/1998	Winter et al. 711/5
5,781,549	A	*	7/1998	Dai 370/398
5,787,255	A		7/1998	Parlan et al. 395/200.63
5,793,954	A		8/1998	Baker et al. 395/200.8
5,818,842	A	*	10/1998	Burwell et al. 370/250
5,848,067	A	*	12/1998	Osawa et al. 370/394
5,870,394	A		2/1999	Oprea 370/392
5,949,786	A	*	9/1999	Bellenger 370/401
6,157,955	A	*	12/2000	Narad et al. 709/228

OTHER PUBLICATIONS

Pending U.S. patent application Ser. No. 09/259,445, entitled "Method and Apparatus for Distributing Network Processing on a Multiprocessor Computer," by Shimon Muller et al., filed Mar. 1, 1999.

Pending U.S. patent application Ser. No. 09/260,367, entitled "Method and Apparatus for Suppressing Interrupts in a High-Speed Network Environment," by Denton Gentry, filed Mar. 1, 1999.

Pending U.S. patent application Ser. No. 09/259,736, entitled "Method and Apparatus for Modulating Interrupts in a Network Interface," by Denton Gentry et al., filed Mar. 1, 1999.

Pending U.S. patent application Ser. No. 09/259,765, entitled "A High Performance Network Interface," by Shimon Muller et al., filed Mar. 1, 1999.

Pending U.S. patent application Ser. No. 09/260,618, entitled "Method and Apparatus for Classifying Network Traffic in a High Performance Network Interface," by Shimon Muller et al., filed Mar. 1, 1999.

Pending U.S. patent application Ser. No. 09/260,324, entitled "Method and Apparatus for Dynamic Packet Batching with a High Performance Network Interface," by Shimon Muller et al., filed Mar. 1, 1999.

Pending U.S. patent application Ser. No. 09/258,952, entitled "Method and Apparatus for Early Random Discard of Packets," by Shimon Muller et al., filed Mar. 1, 1999.

Pending U.S. patent application Ser. No. 09/260,333, entitled "Method and Apparatus for Data Re-Assembly with a High Performance Network Interface," by Shimon Muller et al., filed Mar. 1, 1999.

Pending U.S. patent application Ser. No. 09/258,955, entitled "Dynamic Parsing in a High Performance Network Interface," by Denton Gentry, filed Mar. 1, 1999.

Pending U.S. patent application Ser. No. 09/259,936, entitled "Method and Apparatus for Indicating an Interrupt in a Network Interface," by Denton Gentry et al., filed Mar. 1, 1999.

Toong Shoon Chan, et al., "Parallel Architecture Support for High-Speed Protocol Processing," Feb. 1, 1997, *Microprocessors And Microsystems*, vol. 20, No. 6, pp. 325-339.

Sally Floyd & Van Jacobson, *Random Early Detection Gateways for Congestion Avoidance*, Aug., 1993, IEEE/ACM Transactions on Networking.

U.S. patent application Ser. No. 08/893,862, entitled "Mechanism for Reducing Interrupt Overhead in Device Drivers," filed Jul. 11, 1997, inventor Denton Gentry.

* cited by examiner

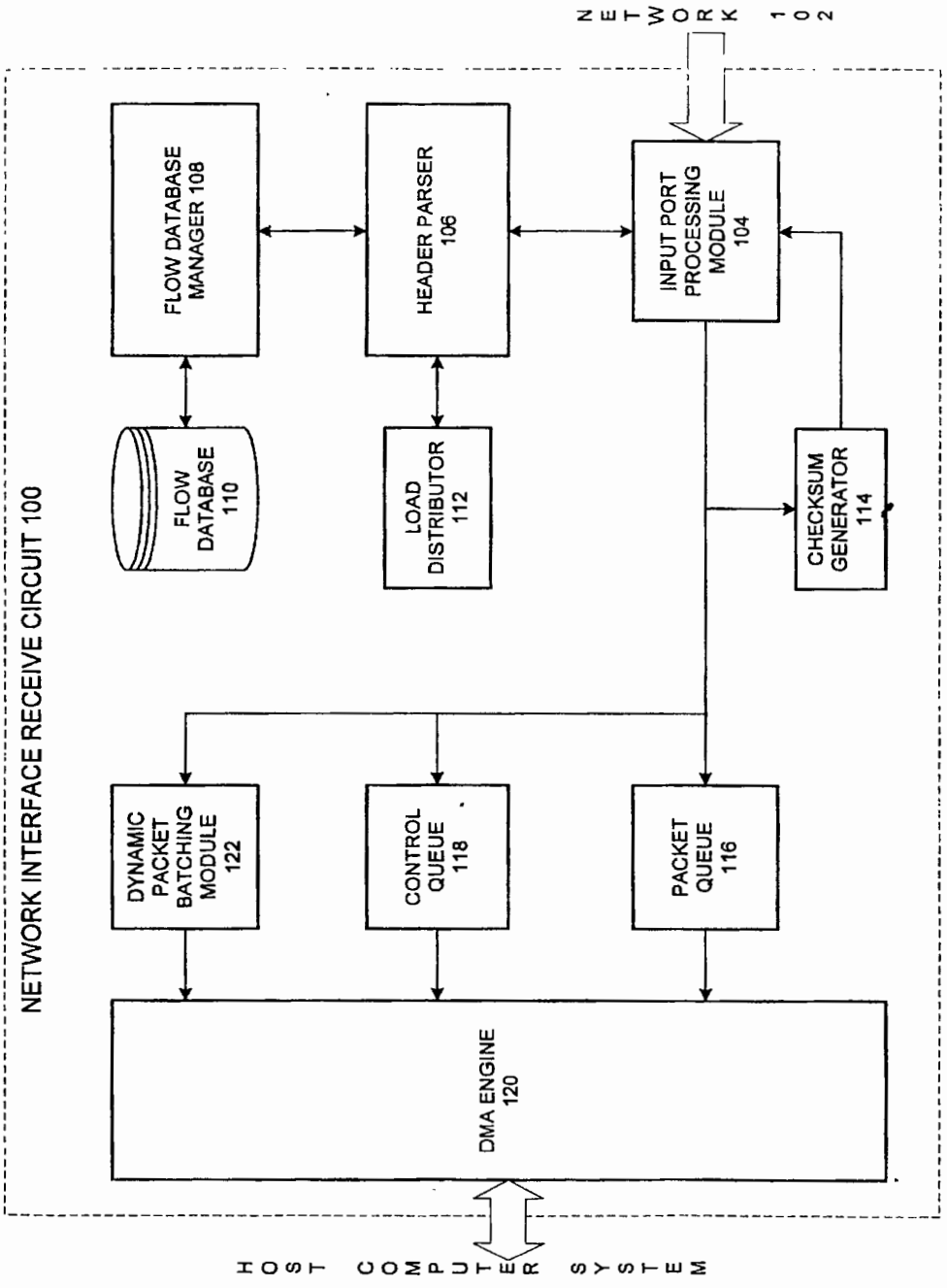

NETWORK 102
NETWORK INTERFACE RECEIVE CIRCUIT 100
FLOW DATABASE MANAGER 108
HEADER PARSER 106
INPUT PORT PROCESSING MODULE 104
FLOW DATABASE 110
LOAD DISTRIBUTOR 112
CHECKSUM GENERATOR 114
DYNAMIC PACKET BATCHING MODULE 122
CONTROL QUEUE 118
PACKET QUEUE 116
DMA ENGINE 120
HOST COMPUTER SYSTEM

FIG. 1A

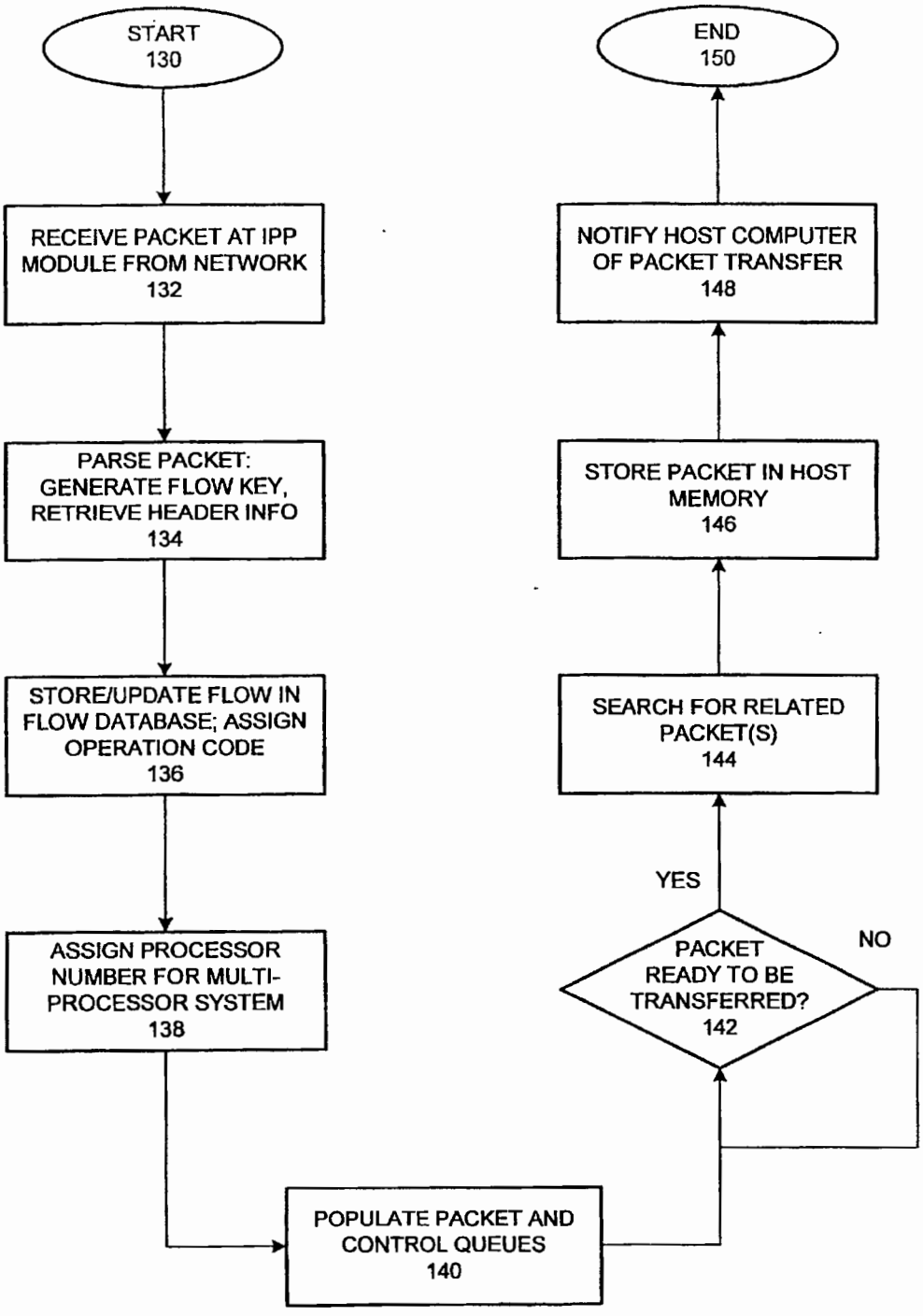
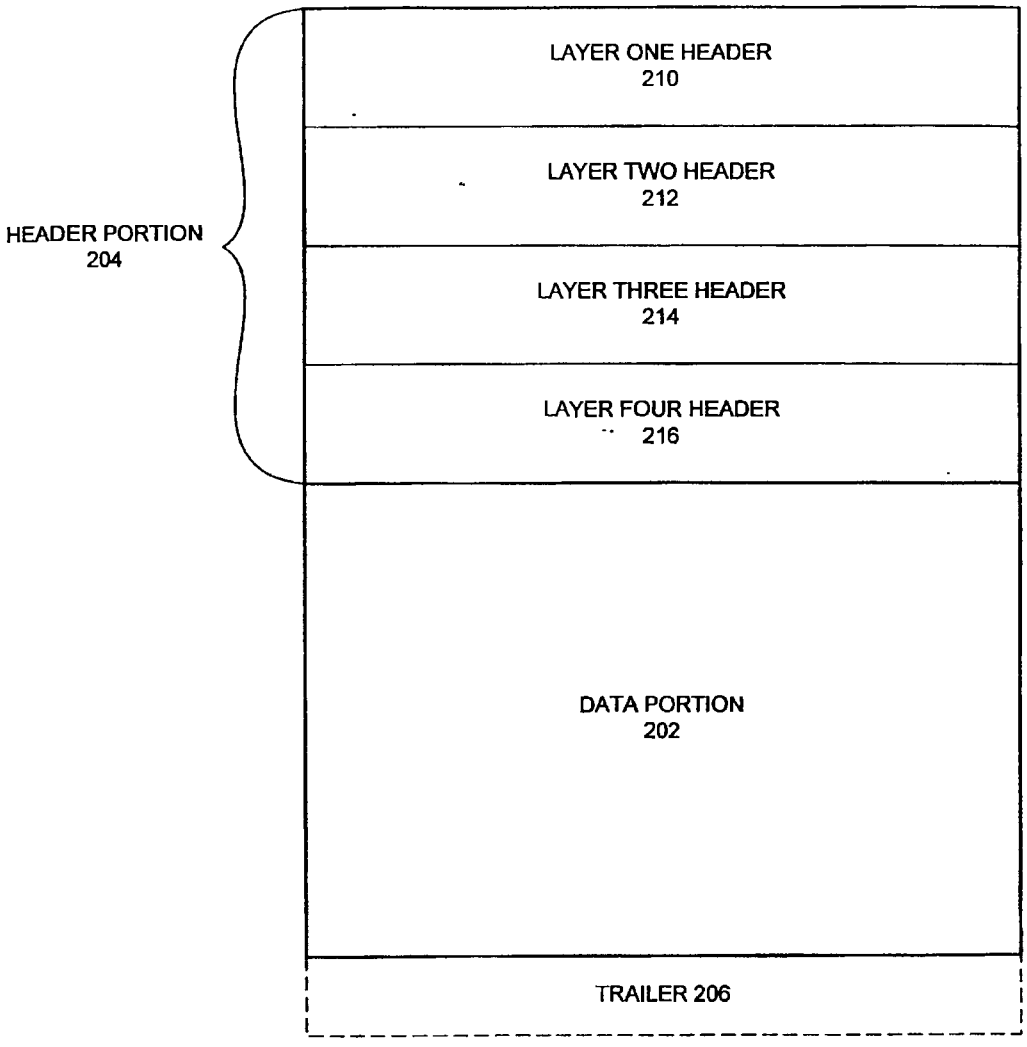

START
130
RECEIVE PACKET AT IPP MODULE FROM NETWORK
132
PARSE PACKET: GENERATE FLOW KEY, RETRIEVE HEADER INFO
134
STORE/UPDATE FLOW IN FLOW DATABASE; ASSIGN OPERATION CODE
136
ASSIGN PROCESSOR NUMBER FOR MULTI-PROCESSOR SYSTEM
138
POPULATE PACKET AND CONTROL QUEUES
140
PACKET READY TO BE TRANSFERRED?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

FIG. 1B



PACKET 200

FIG. 2

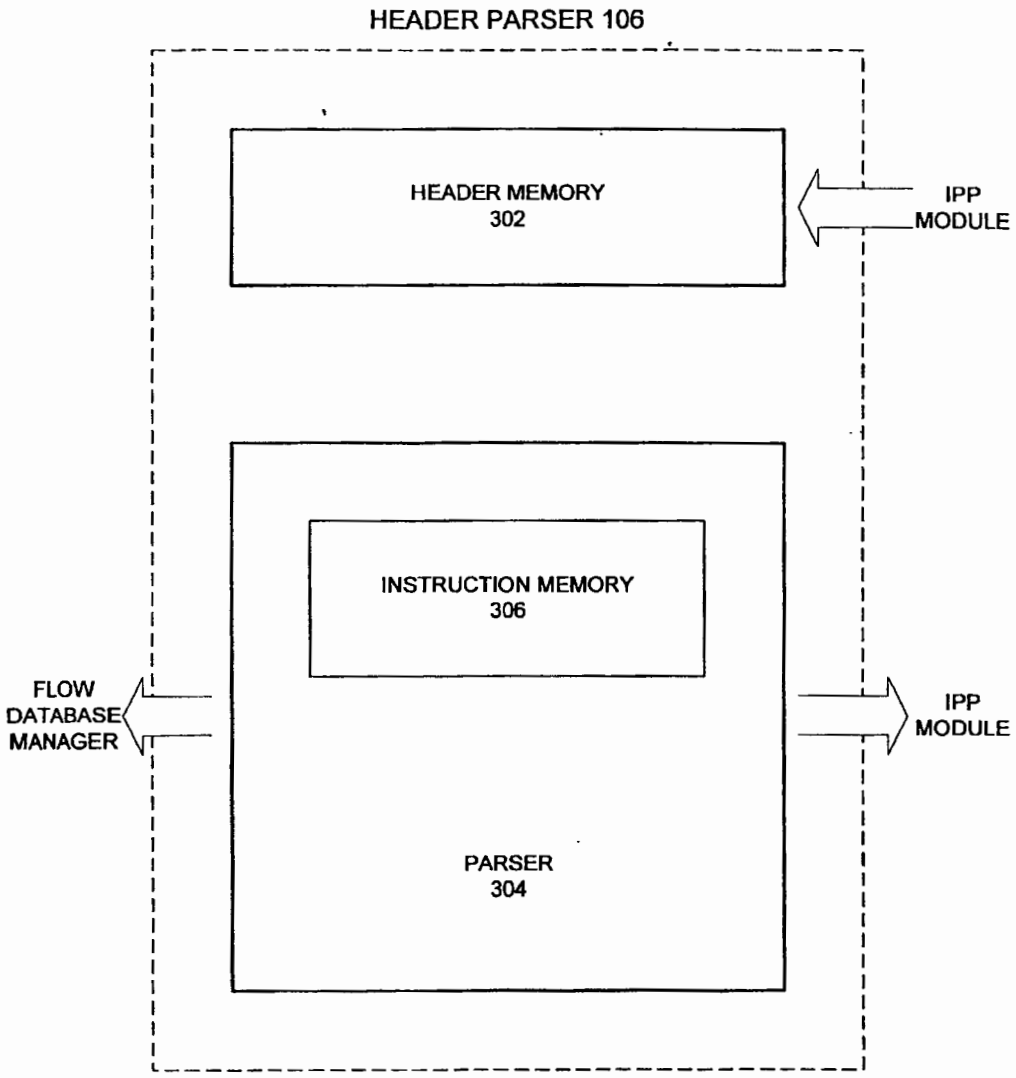

HEADER PARSER 106
HEADER MEMORY
302
IPP
MODULE
INSTRUCTION MEMORY
306
FLOW
DATABASE
MANAGER
IPP
MODULE
PARSER
304

FIG. 3

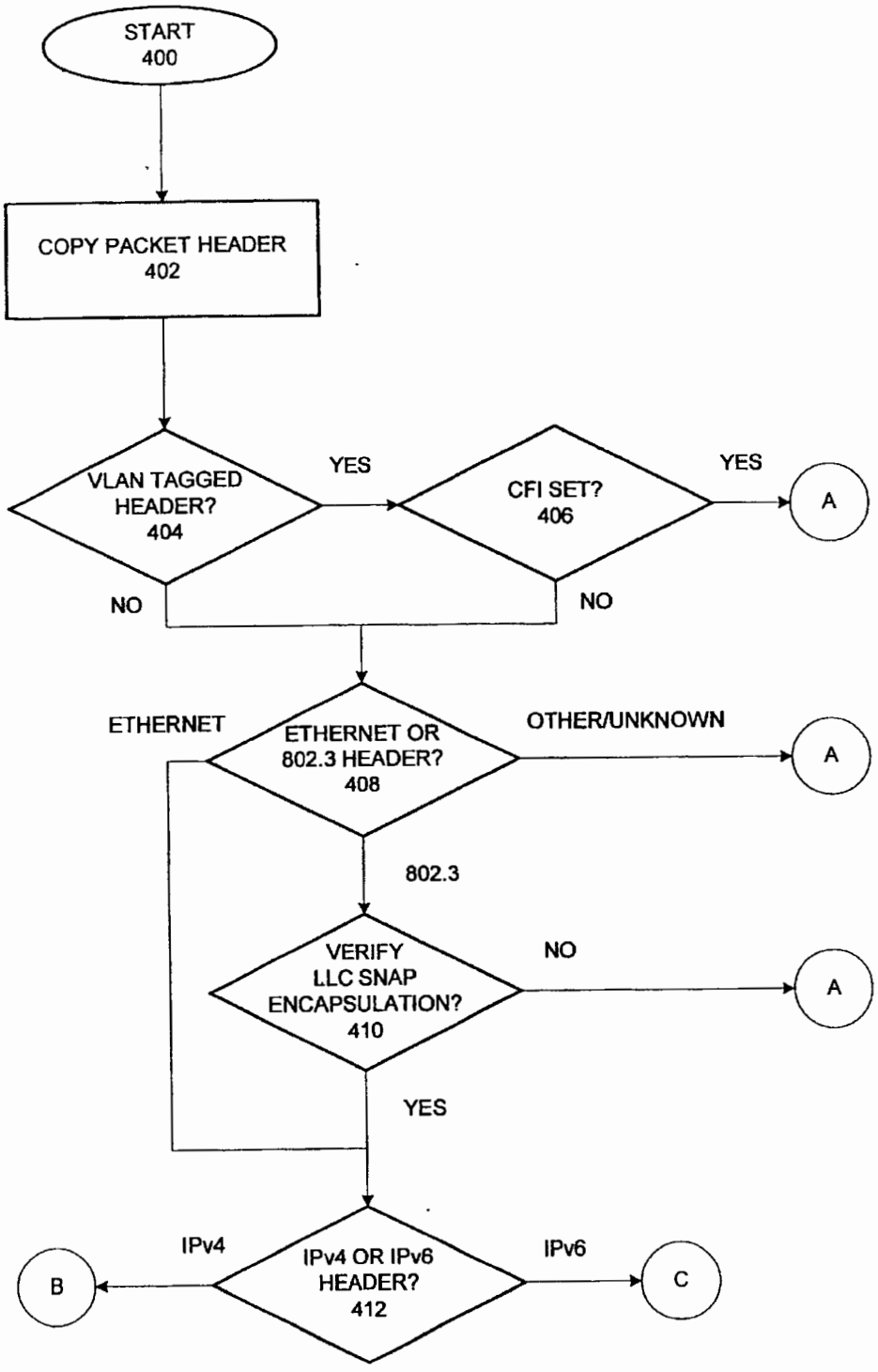

START
400
COPY PACKET HEADER
402
VLAN TAGGED HEADER?
404
YES
CFI SET?
406
YES
A
NO
NO
ETHERNET
ETHERNET OR 802.3 HEADER?
408
OTHER/UNKNOWN
A
802.3
VERIFY LLC SNAP ENCAPSULATION?
410
NO
A
YES
IPv4
IPv4 OR IPv6 HEADER?
412
IPv6
B
C

FIG. 4A

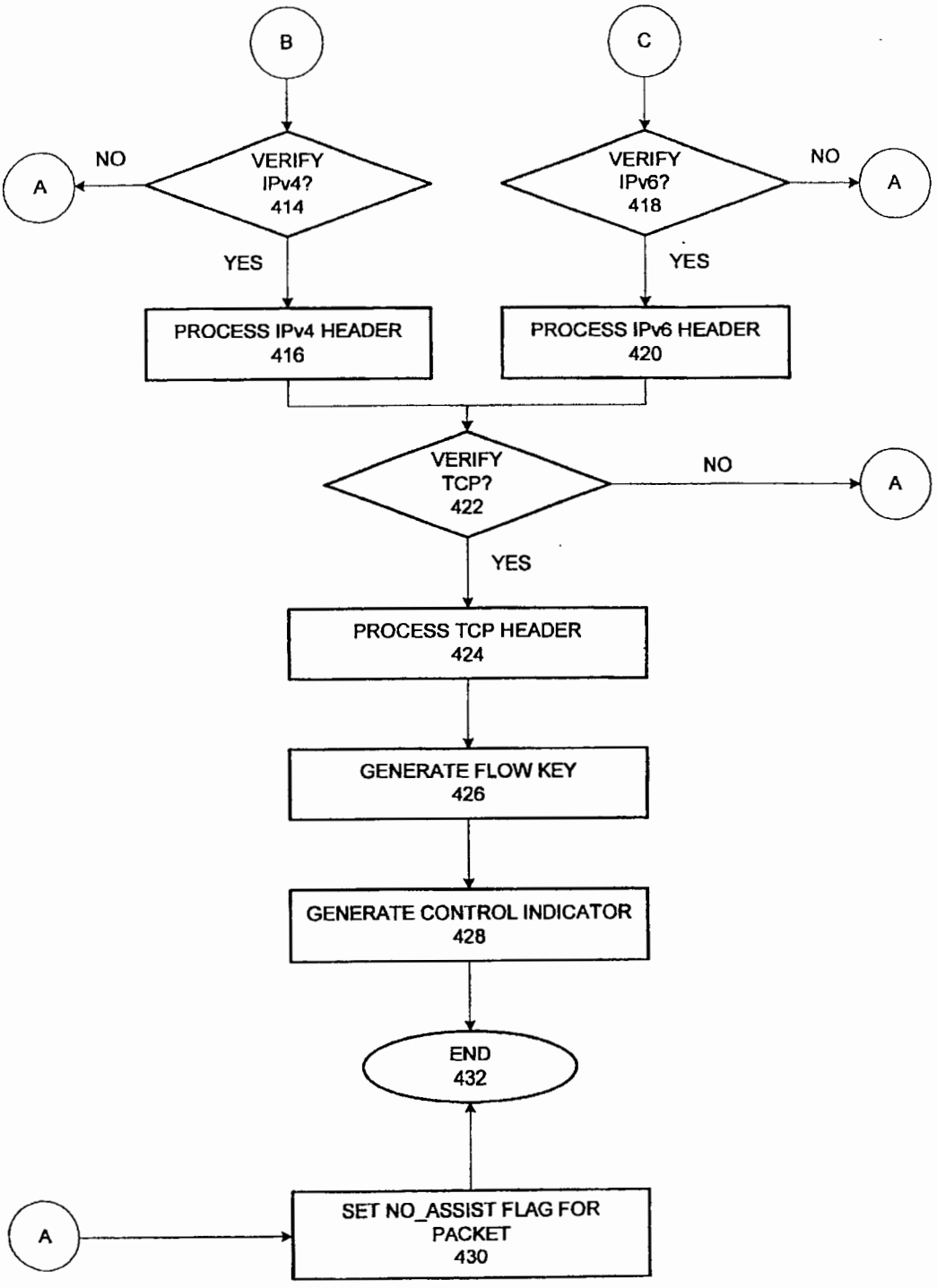

B
C
VERIFY IPv4? 414
NO
A
YES
VERIFY IPv6? 418
NO
A
YES
PROCESS IPv4 HEADER 416
PROCESS IPv6 HEADER 420
VERIFY TCP? 422
NO
A
YES
PROCESS TCP HEADER 424
GENERATE FLOW KEY 426
GENERATE CONTROL INDICATOR 428
END 432
A
SET NO_ASSIST FLAG FOR PACKET 430

FIG. 4B

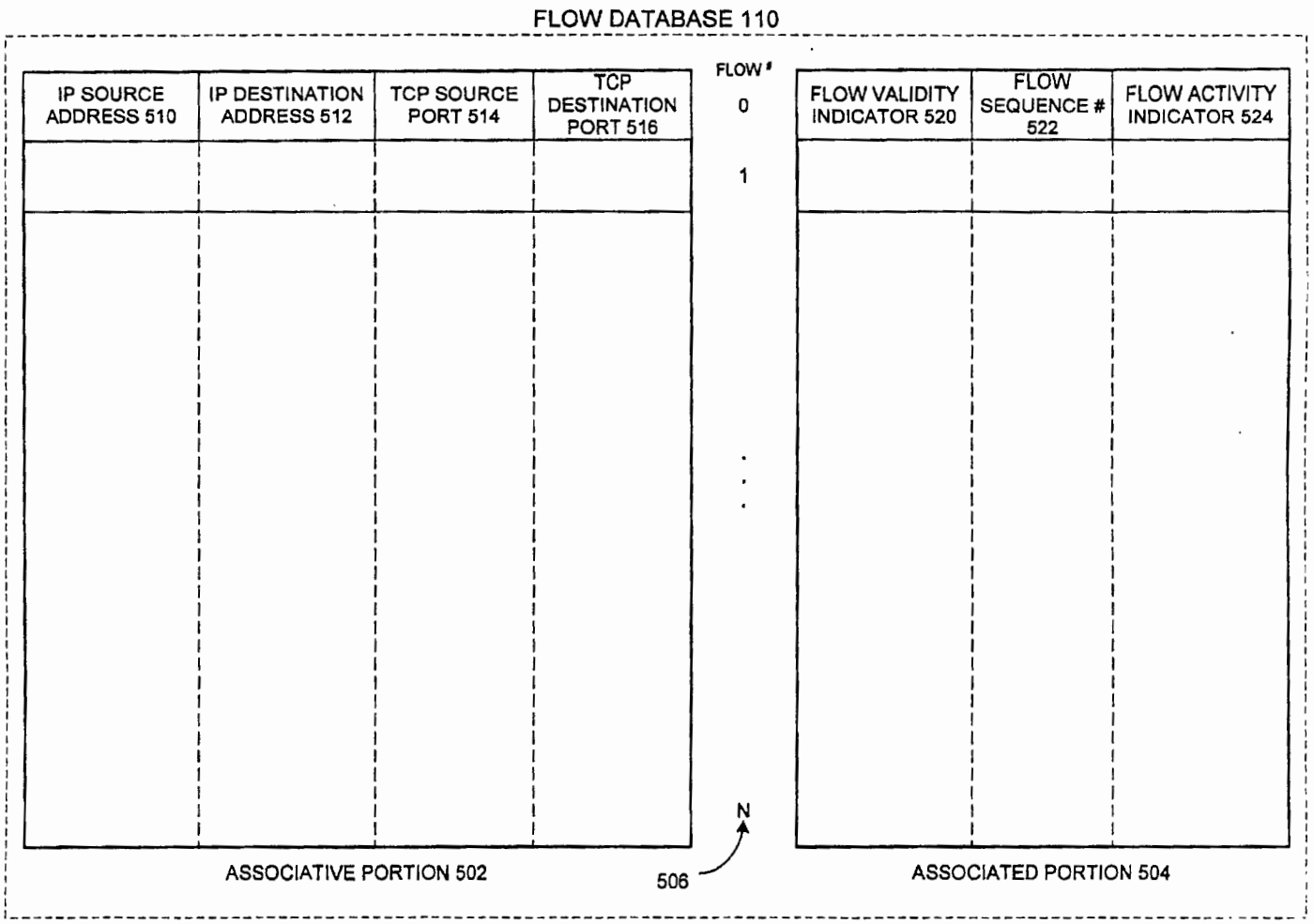

FLOW DATABASE 110
IP SOURCE ADDRESS 510
IP DESTINATION ADDRESS 512
TCP SOURCE PORT 514
TCP DESTINATION PORT 516
FLOW #
0
1
N
FLOW VALIDITY INDICATOR 520
FLOW SEQUENCE # 522
FLOW ACTIVITY INDICATOR 524
ASSOCIATIVE PORTION 502
506
ASSOCIATED PORTION 504

FIG. 5

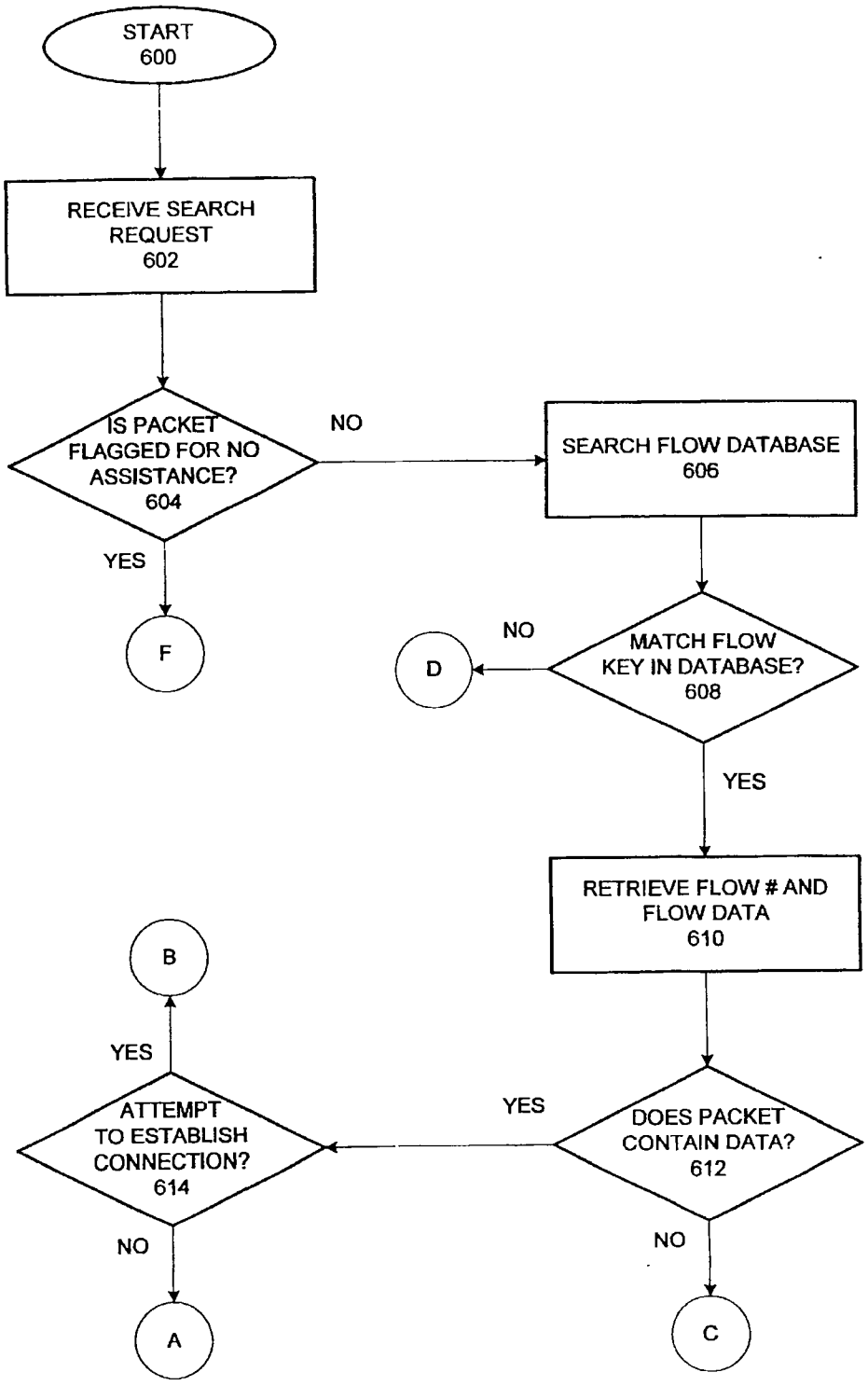

START
600
RECEIVE SEARCH REQUEST
602
IS PACKET FLAGGED FOR NO ASSISTANCE?
604
NO
SEARCH FLOW DATABASE
606
YES
F
NO
D
MATCH FLOW KEY IN DATABASE?
608
YES
RETRIEVE FLOW # AND FLOW DATA
610
B
YES
ATTEMPT TO ESTABLISH CONNECTION?
614
YES
DOES PACKET CONTAIN DATA?
612
NO
NO
A
C

FIG. 6A

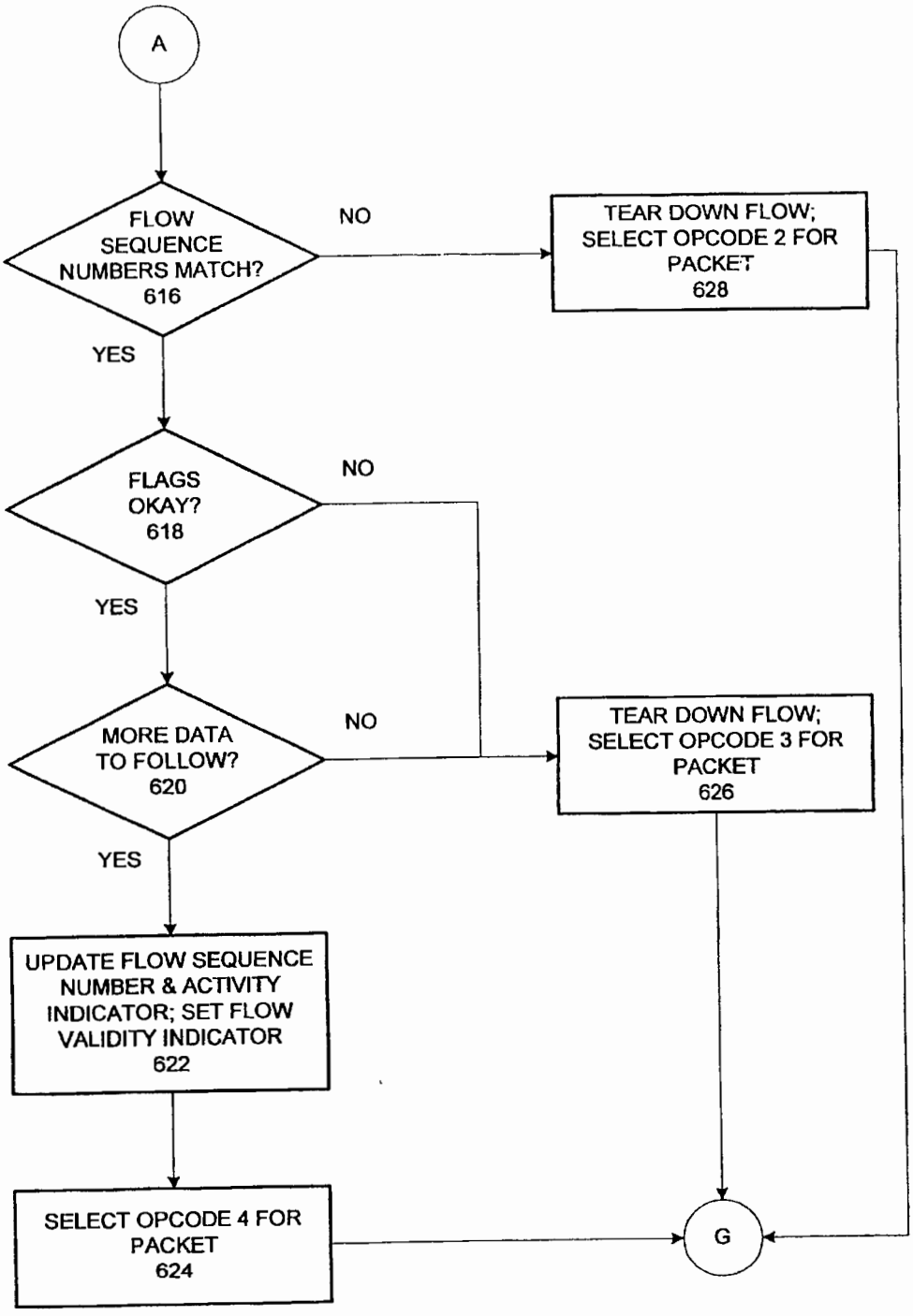

A
FLOW SEQUENCE NUMBERS MATCH? 616
NO
TEAR DOWN FLOW; SELECT OPCODE 2 FOR PACKET 628
YES
FLAGS OKAY? 618
NO
YES
MORE DATA TO FOLLOW? 620
NO
TEAR DOWN FLOW; SELECT OPCODE 3 FOR PACKET 626
YES
UPDATE FLOW SEQUENCE NUMBER & ACTIVITY INDICATOR; SET FLOW VALIDITY INDICATOR 622
SELECT OPCODE 4 FOR PACKET 624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

FIG. 6B

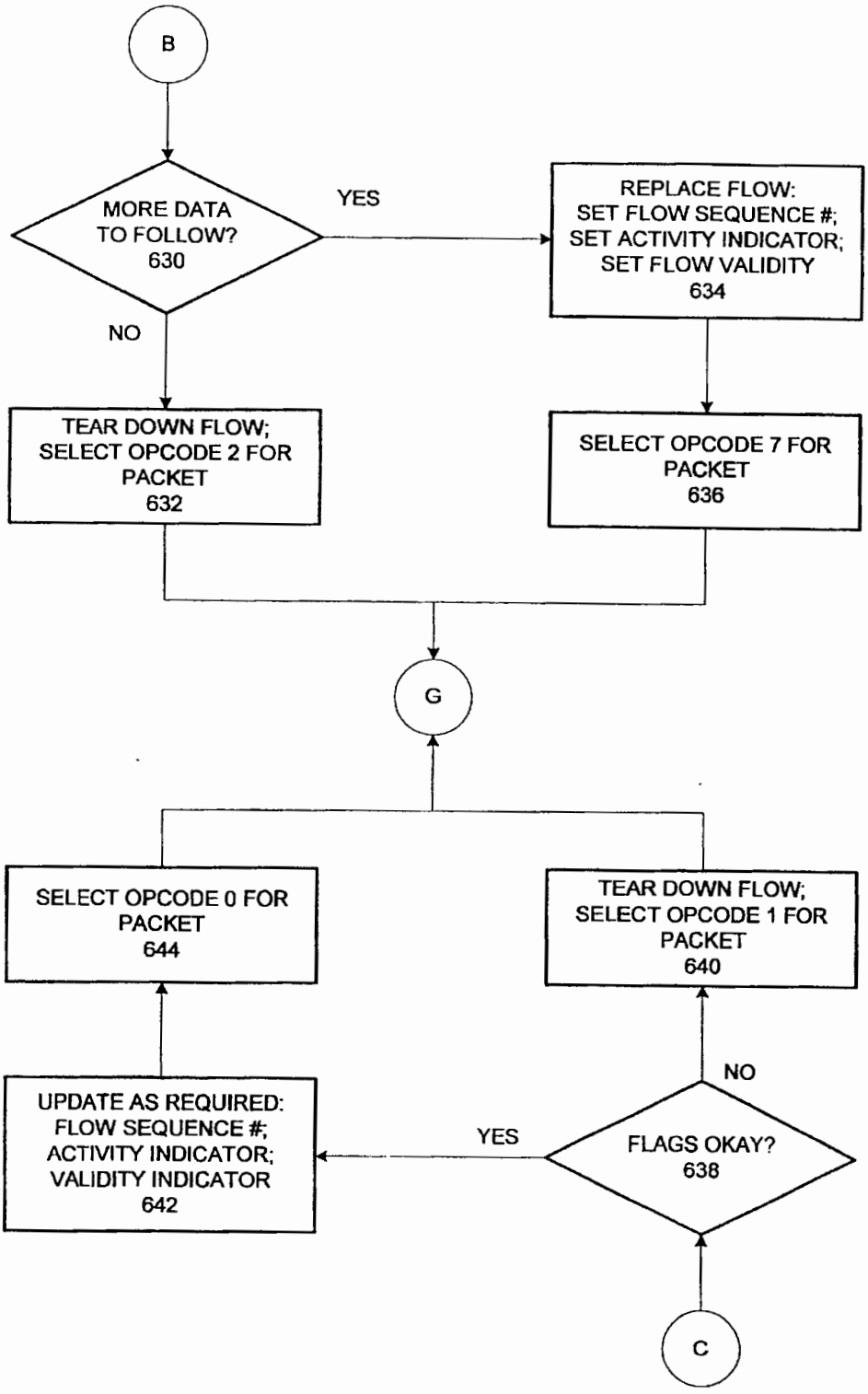

B
MORE DATA TO FOLLOW? 630
YES
NO
REPLACE FLOW: SET FLOW SEQUENCE #; SET ACTIVITY INDICATOR; SET FLOW VALIDITY 634
TEAR DOWN FLOW; SELECT OPCODE 2 FOR PACKET 632
SELECT OPCODE 7 FOR PACKET 636
G
SELECT OPCODE 0 FOR PACKET 644
TEAR DOWN FLOW; SELECT OPCODE 1 FOR PACKET 640
UPDATE AS REQUIRED: FLOW SEQUENCE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

FIG. 6C

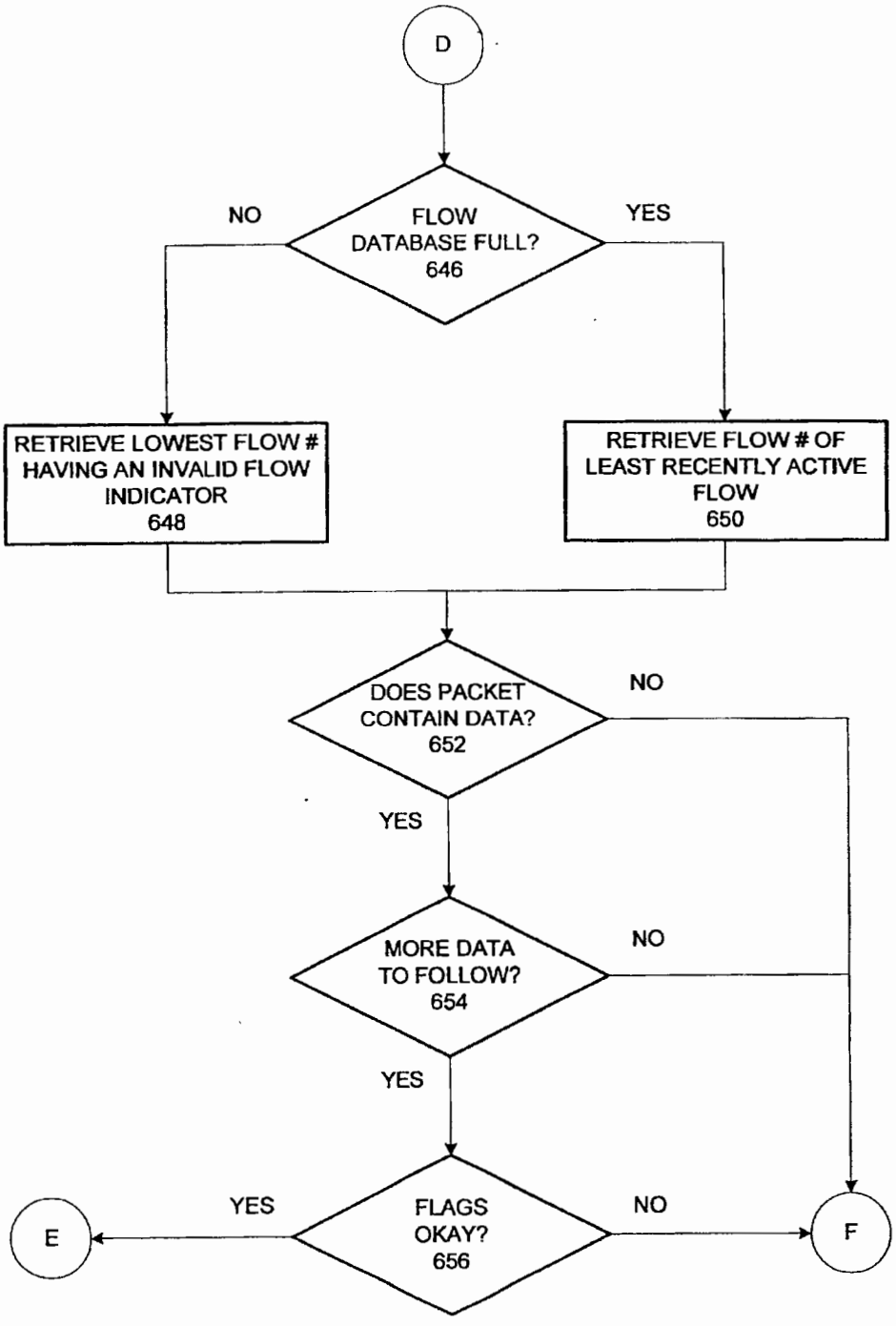

D
FLOW DATABASE FULL? 646
NO
YES
RETRIEVE LOWEST FLOW # HAVING AN INVALID FLOW INDICATOR 648
RETRIEVE FLOW # OF LEAST RECENTLY ACTIVE FLOW 650
DOES PACKET CONTAIN DATA? 652
NO
YES
MORE DATA TO FOLLOW? 654
NO
YES
FLAGS OKAY? 656
YES
NO
E
F

FIG. 6D

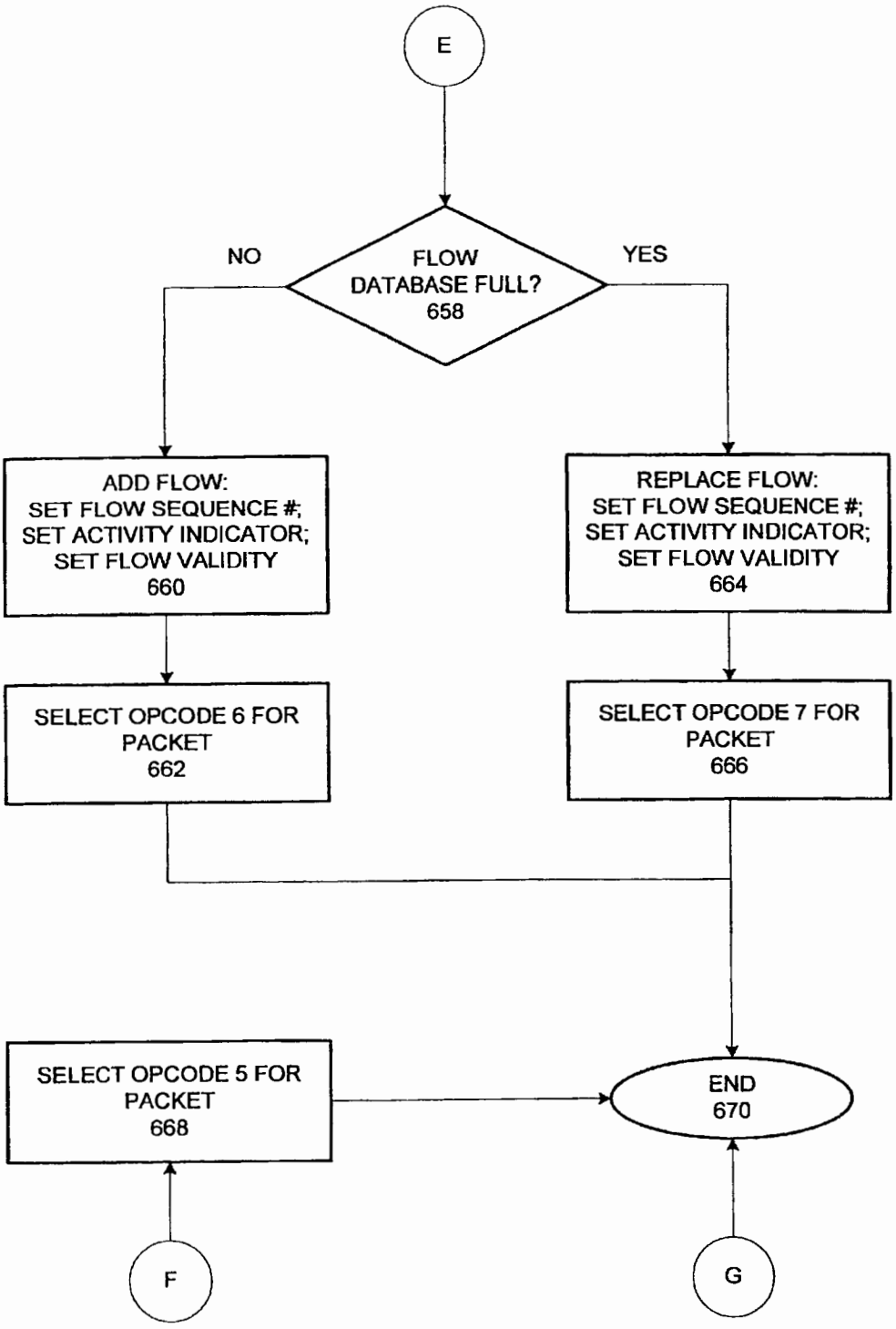

E
FLOW DATABASE FULL? 658
NO
YES
ADD FLOW: SET FLOW SEQUENCE #; SET ACTIVITY INDICATOR; SET FLOW VALIDITY 660
REPLACE FLOW: SET FLOW SEQUENCE #; SET ACTIVITY INDICATOR; SET FLOW VALIDITY 664
SELECT OPCODE 6 FOR PACKET 662
SELECT OPCODE 7 FOR PACKET 666
SELECT OPCODE 5 FOR PACKET 668
END 670
F
G

FIG. 6E

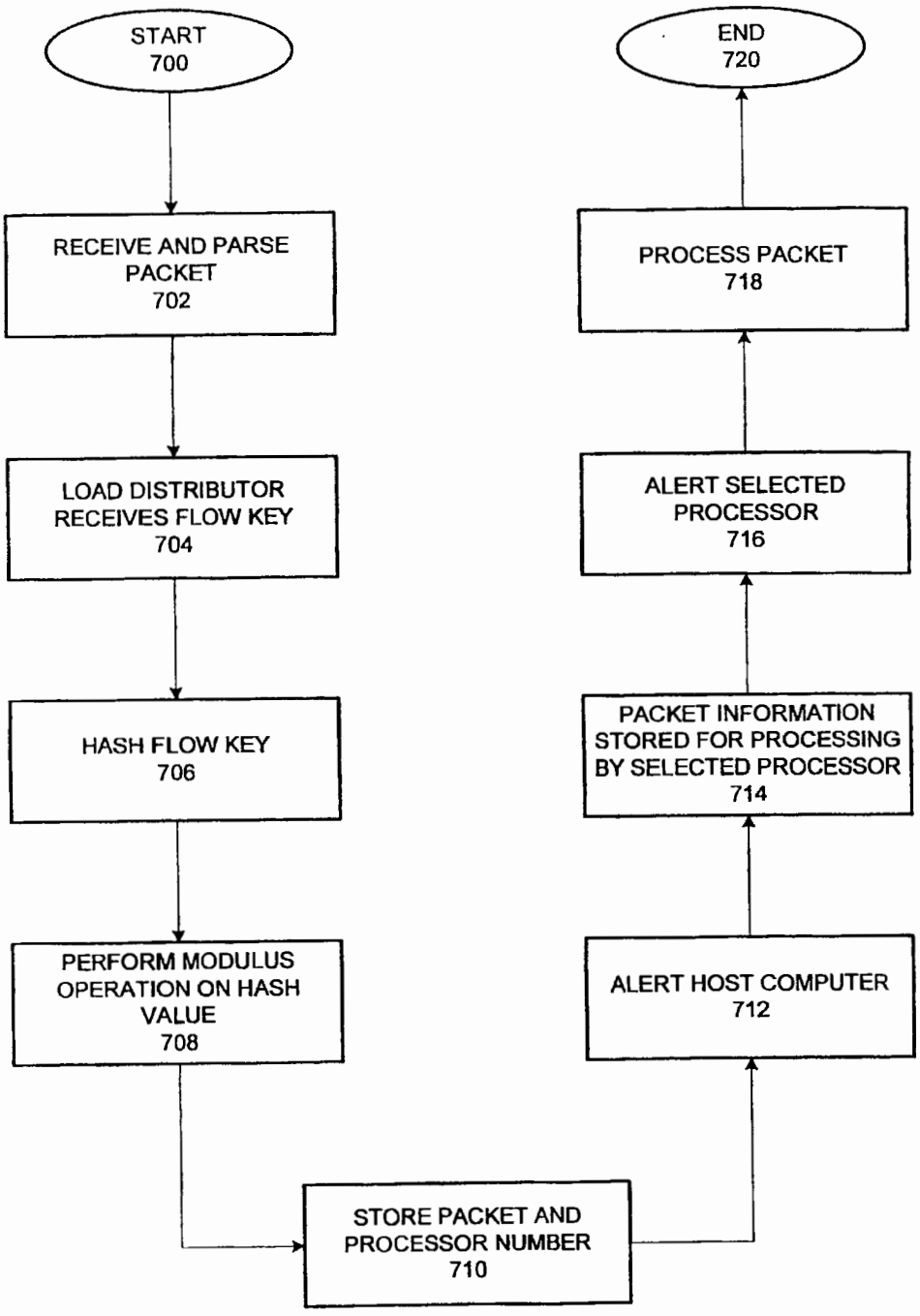

START
700
RECEIVE AND PARSE PACKET
702
LOAD DISTRIBUTOR RECEIVES FLOW KEY
704
HASH FLOW KEY
706
PERFORM MODULUS OPERATION ON HASH VALUE
708
STORE PACKET AND PROCESSOR NUMBER
710
ALERT HOST COMPUTER
712
PACKET INFORMATION STORED FOR PROCESSING BY SELECTED PROCESSOR
714
ALERT SELECTED PROCESSOR
716
PROCESS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

FIG. 7

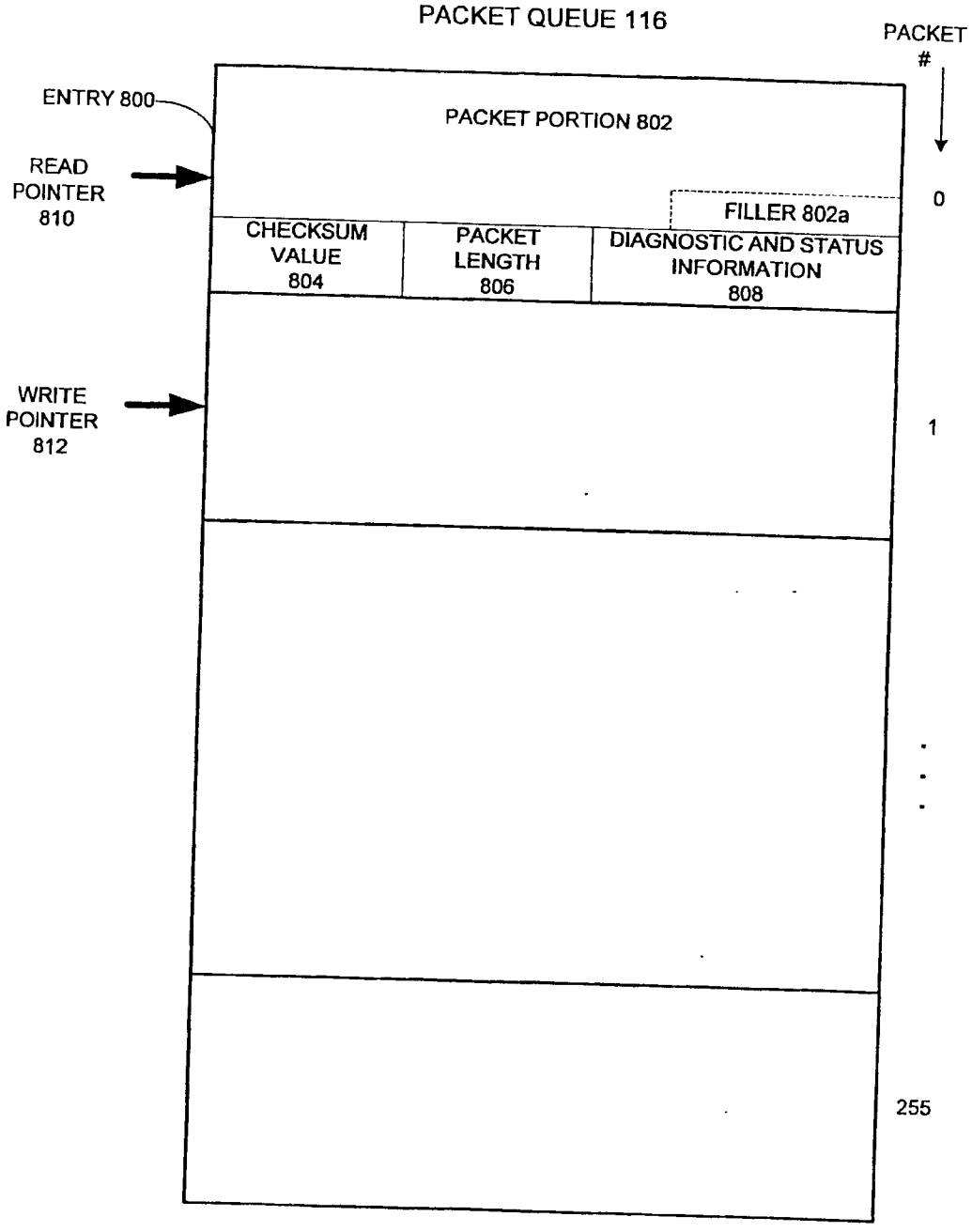

PACKET QUEUE 116
PACKET #
ENTRY 800
PACKET PORTION 802
READ POINTER 810
FILLER 802a
0
CHECKSUM VALUE 804
PACKET LENGTH 806
DIAGNOSTIC AND STATUS INFORMATION 808
WRITE POINTER 812
1
255

FIG. 8

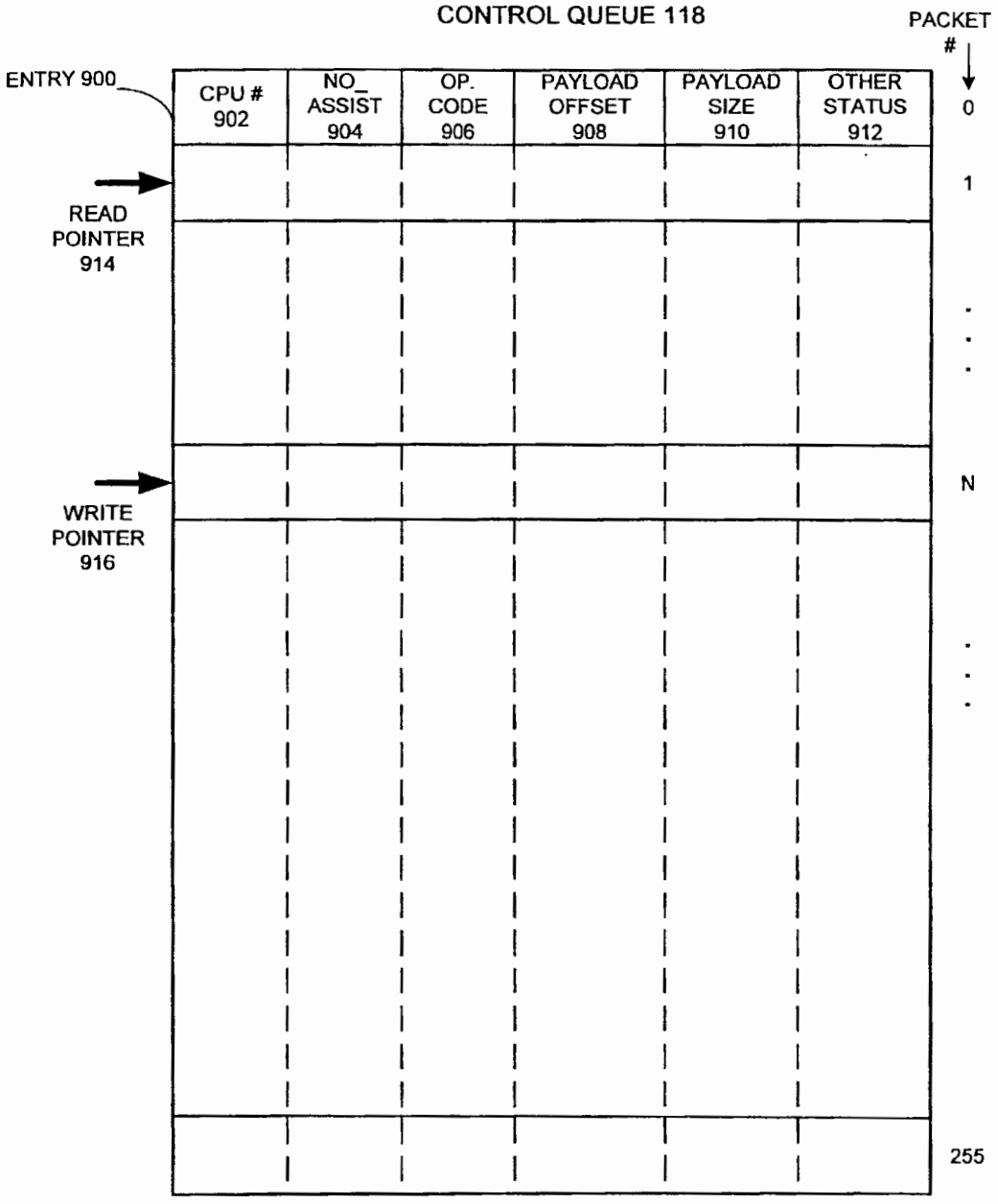

CONTROL QUEUE 118
PACKET #
ENTRY 900
CPU # 902
NO_ ASSIST 904
OP. CODE 906
PAYLOAD OFFSET 908
PAYLOAD SIZE 910
OTHER STATUS 912
0
1
READ POINTER 914
N
WRITE POINTER 916
255

FIG. 9

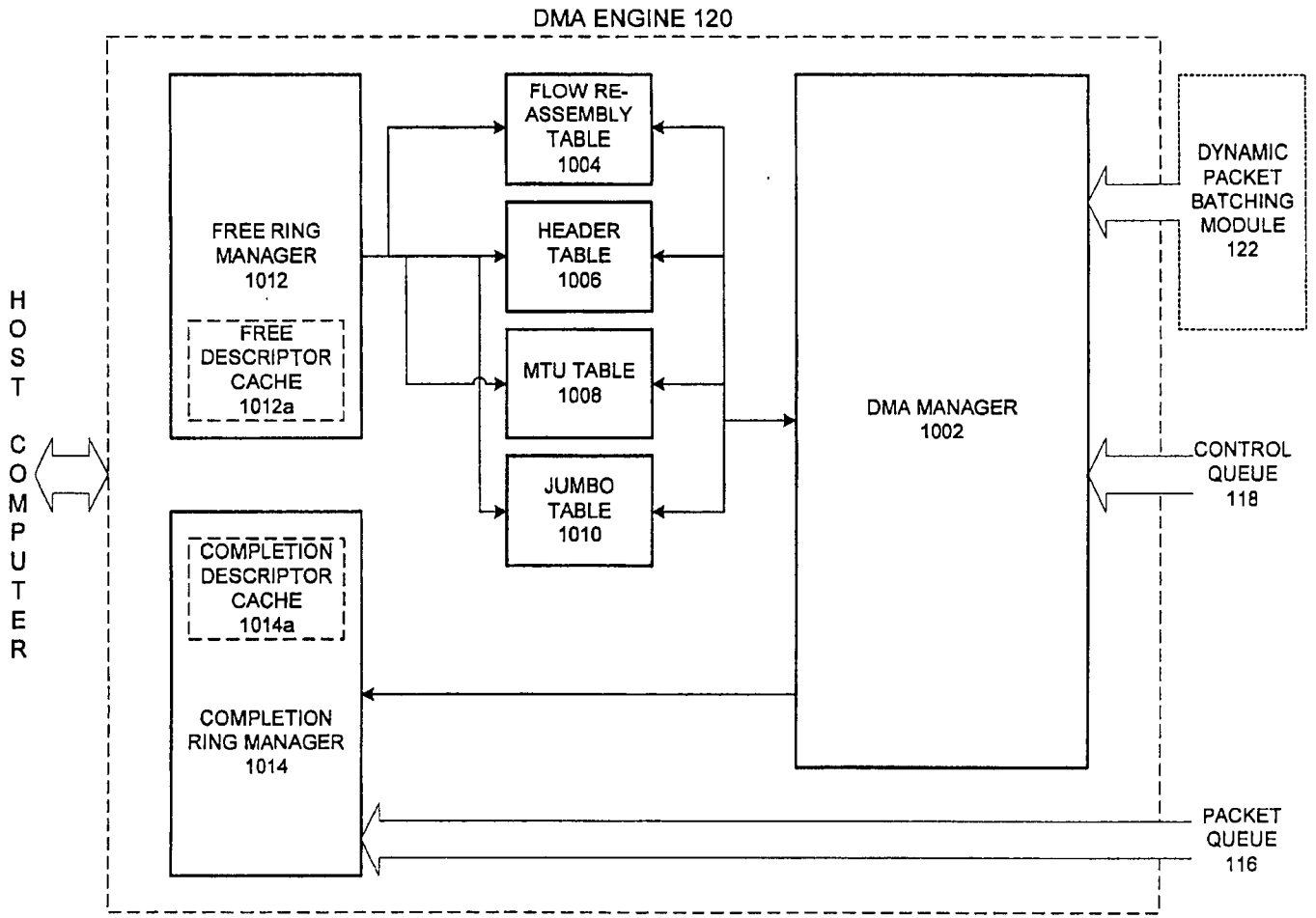

DMA ENGINE 120
HOST COMPUTER
FREE RING MANAGER 1012
FREE DESCRIPTOR CACHE 1012a
FLOW RE-ASSEMBLY TABLE 1004
HEADER TABLE 1006
MTU TABLE 1008
JUMBO TABLE 1010
COMPLETION DESCRIPTOR CACHE 1014a
COMPLETION RING MANAGER 1014
DMA MANAGER 1002
DYNAMIC PACKET BATCHING MODULE 122
CONTROL QUEUE 118
PACKET QUEUE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

FIG. 10

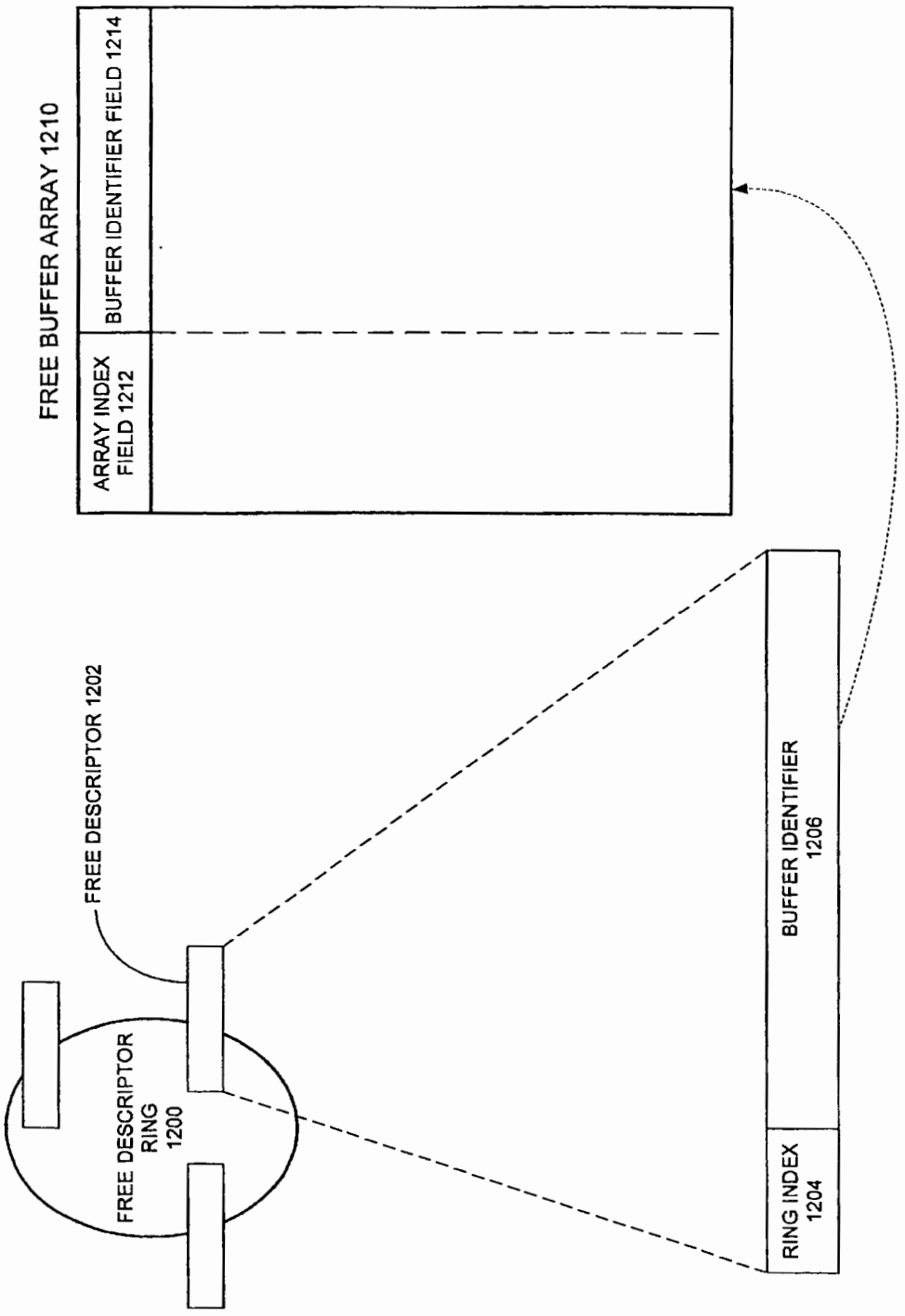

FREE BUFFER ARRAY 1210
ARRAY INDEX FIELD 1212
BUFFER IDENTIFIER FIELD 1214
FREE DESCRIPTOR 1202
FREE DESCRIPTOR RING 1200
RING INDEX 1204
BUFFER IDENTIFIER 1206

FIG. 12A

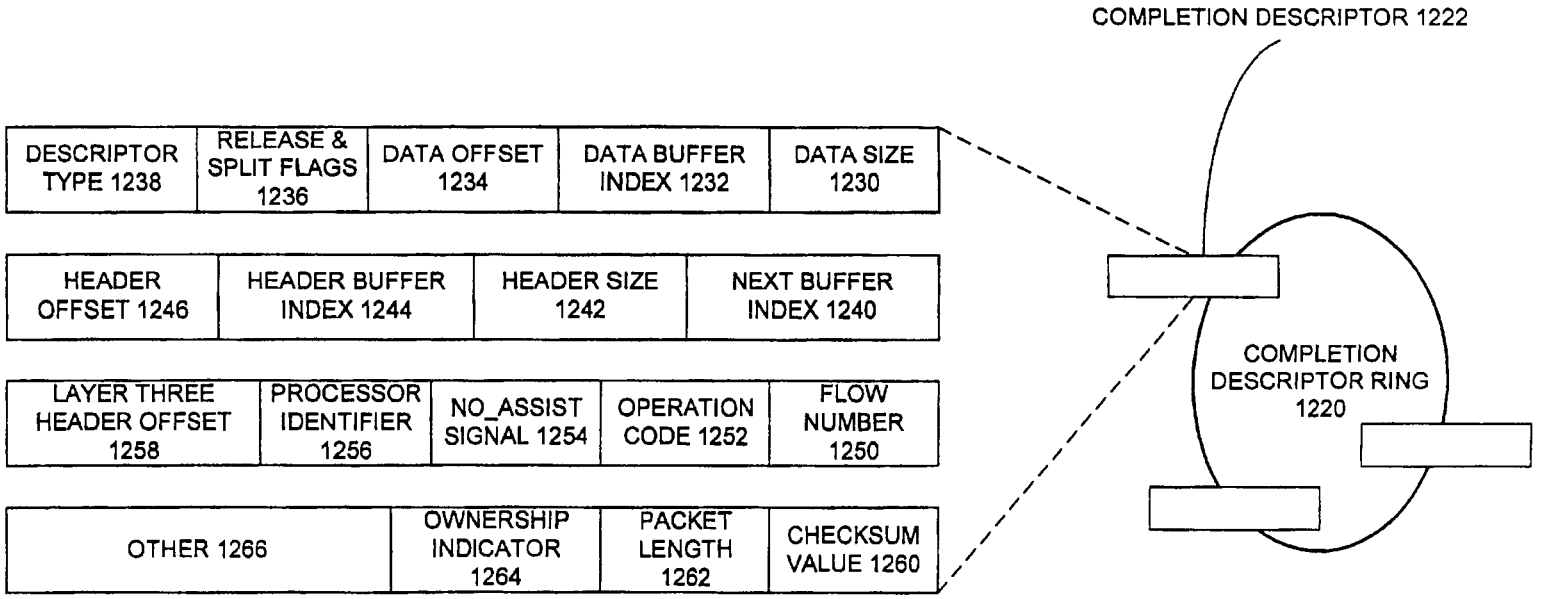

COMPLETION DESCRIPTOR 1222
DESCRIPTOR TYPE 1238
RELEASE & SPLIT FLAGS 1236
DATA OFFSET 1234
DATA BUFFER INDEX 1232
DATA SIZE 1230
HEADER OFFSET 1246
HEADER BUFFER INDEX 1244
HEADER SIZE 1242
NEXT BUFFER INDEX 1240
LAYER THREE HEADER OFFSET 1258
PROCESSOR IDENTIFIER 1256
NO_ASSIST SIGNAL 1254
OPERATION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

FIG. 12B

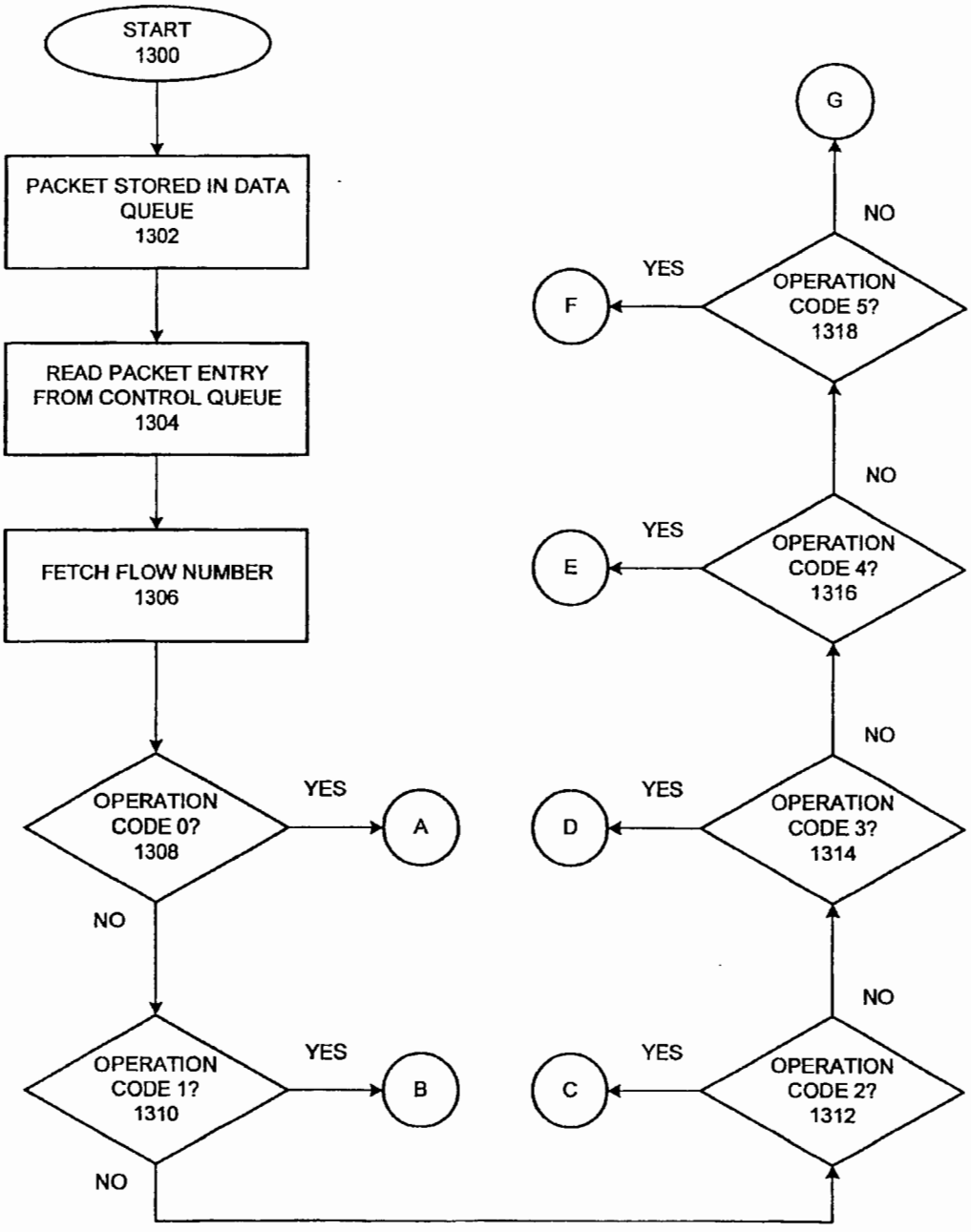

START
1300
PACKET STORED IN DATA QUEUE
1302
READ PACKET ENTRY FROM CONTROL QUEUE
1304
FETCH FLOW NUMBER
1306
OPERATION CODE 0?
1308
YES
A
NO
OPERATION CODE 1?
1310
YES
B
NO
OPERATION CODE 2?
1312
YES
C
NO
OPERATION CODE 3?
1314
YES
D
NO
OPERATION CODE 4?
1316
YES
E
NO
OPERATION CODE 5?
1318
YES
F
NO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

FIG. 13

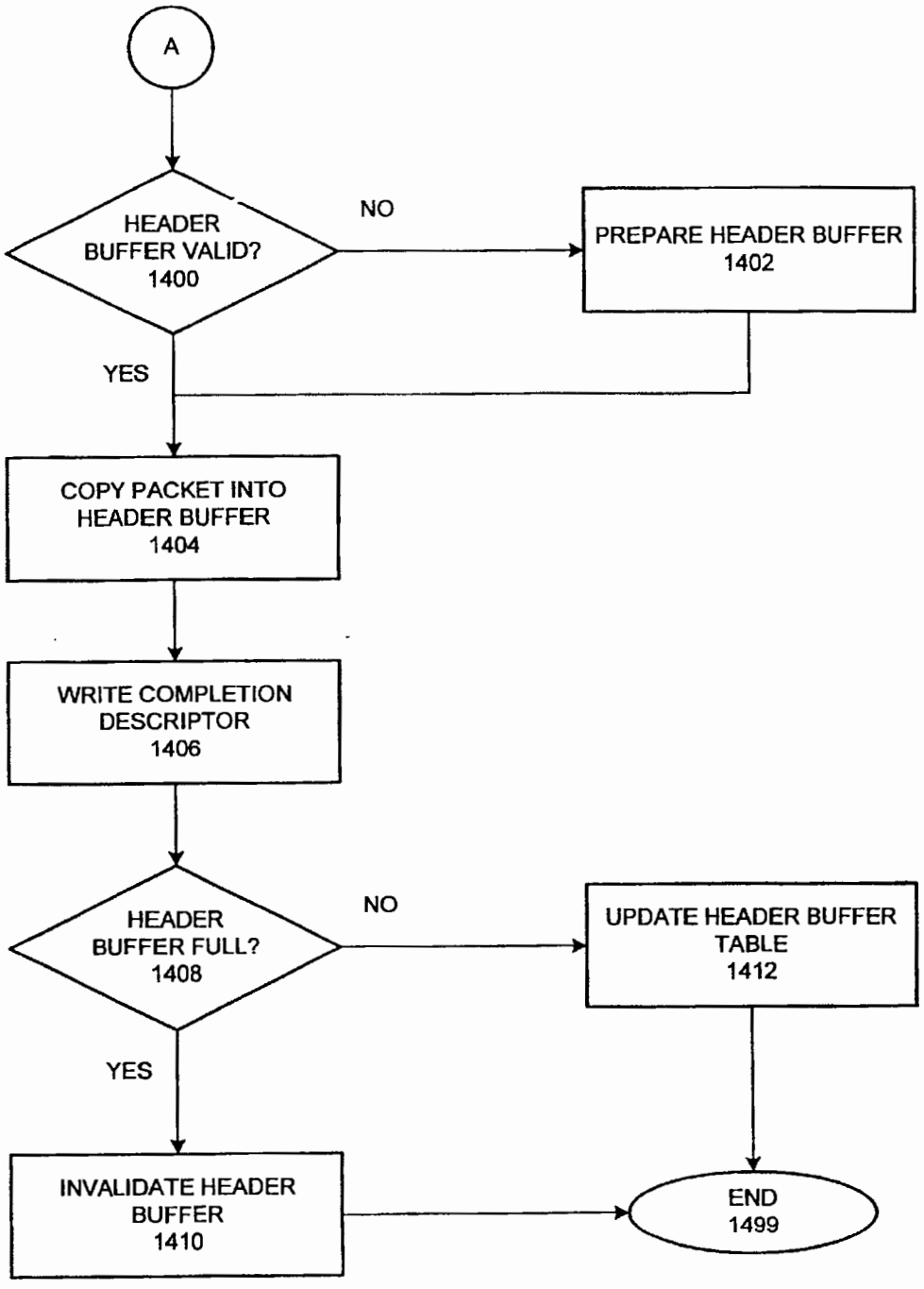

A
HEADER BUFFER VALID? 1400
NO
PREPARE HEADER BUFFER 1402
YES
COPY PACKET INTO HEADER BUFFER 1404
WRITE COMPLETION DESCRIPTOR 1406
HEADER BUFFER FULL? 1408
NO
UPDATE HEADER BUFFER TABLE 1412
YES
INVALIDATE HEADER BUFFER 1410
END 1499

FIG. 14

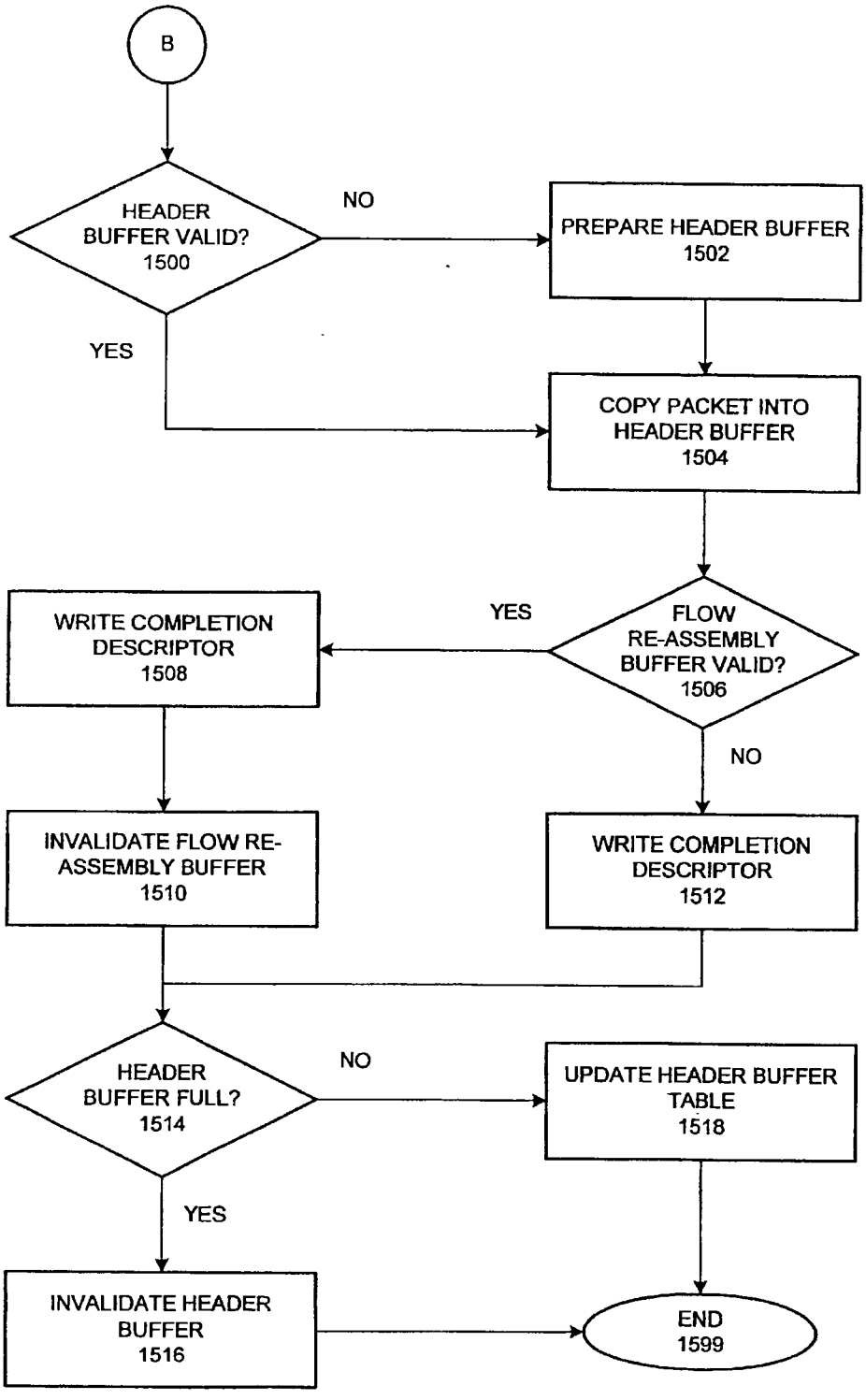

B
HEADER BUFFER VALID? 1500
NO
PREPARE HEADER BUFFER 1502
YES
COPY PACKET INTO HEADER BUFFER 1504
FLOW RE-ASSEMBLY BUFFER VALID? 1506
YES
WRITE COMPLETION DESCRIPTOR 1508
NO
INVALIDATE FLOW RE-ASSEMBLY BUFFER 1510
WRITE COMPLETION DESCRIPTOR 1512
HEADER BUFFER FULL? 1514
NO
UPDATE HEADER BUFFER TABLE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

FIG. 15

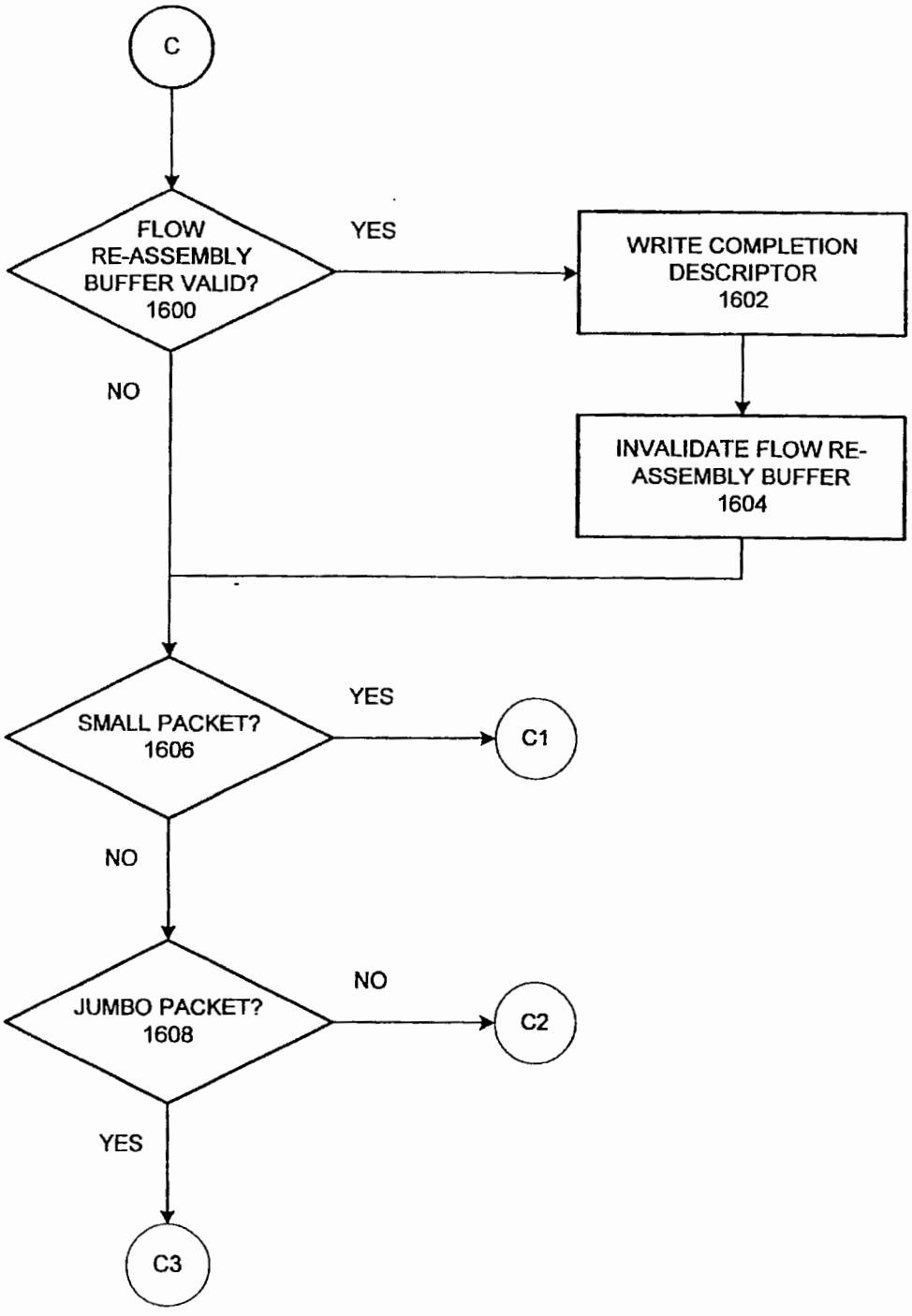

C
FLOW RE-ASSEMBLY BUFFER VALID? 1600
YES
WRITE COMPLETION DESCRIPTOR 1602
NO
INVALIDATE FLOW RE-ASSEMBLY BUFFER 1604
SMALL PACKET? 1606
YES
C1
NO
JUMBO PACKET? 1608
NO
C2
YES
C3

FIG. 16A

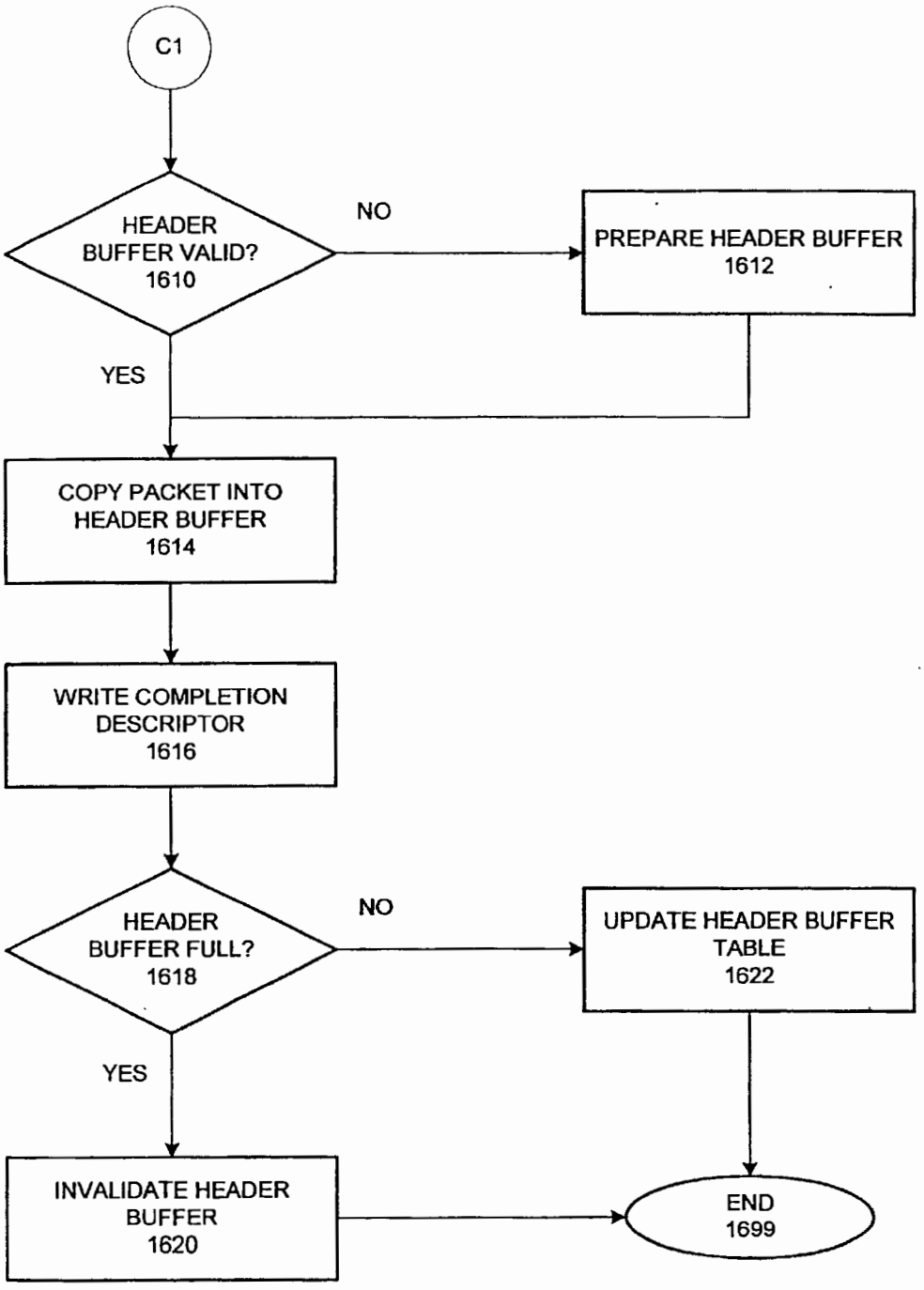

C1
HEADER BUFFER VALID? 1610
NO
PREPARE HEADER BUFFER 1612
YES
COPY PACKET INTO HEADER BUFFER 1614
WRITE COMPLETION DESCRIPTOR 1616
HEADER BUFFER FULL? 1618
NO
UPDATE HEADER BUFFER TABLE 1622
YES
INVALIDATE HEADER BUFFER 1620
END 1699

FIG. 16B

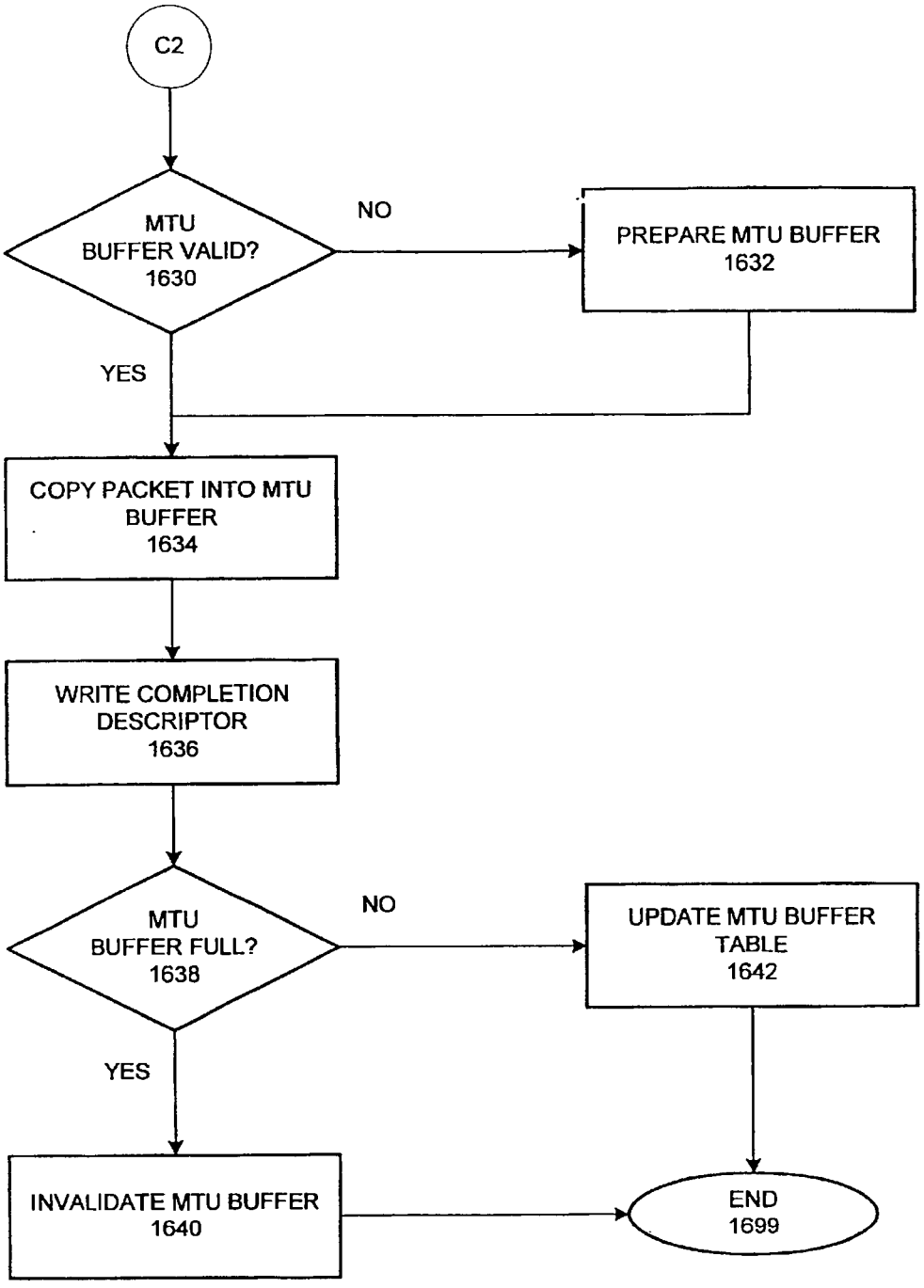

C2
MTU BUFFER VALID? 1630
NO
PREPARE MTU BUFFER 1632
YES
COPY PACKET INTO MTU BUFFER 1634
WRITE COMPLETION DESCRIPTOR 1636
MTU BUFFER FULL? 1638
NO
UPDATE MTU BUFFER TABLE 1642
YES
INVALIDATE MTU BUFFER 1640
END 1699

FIG. 16C

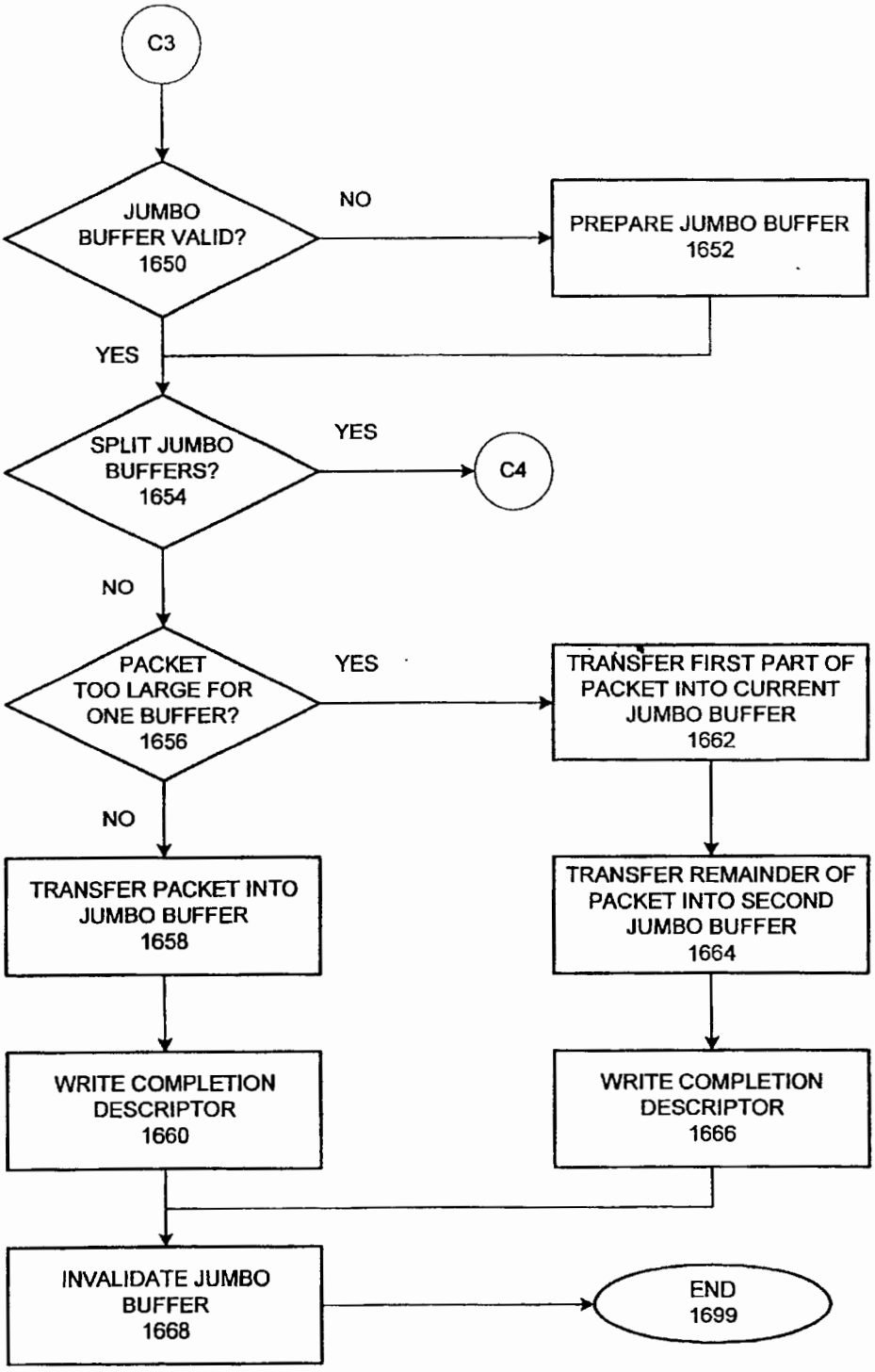

C3
JUMBO BUFFER VALID? 1650
NO
PREPARE JUMBO BUFFER 1652
YES
SPLIT JUMBO BUFFERS? 1654
YES
C4
NO
PACKET TOO LARGE FOR ONE BUFFER? 1656
YES
TRANSFER FIRST PART OF PACKET INTO CURRENT JUMBO BUFFER 1662
NO
TRANSFER PACKET INTO JUMBO BUFFER 1658
TRANSFER REMAINDER OF PACKET INTO SECOND JUMBO BUFFER 1664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

FIG. 16D

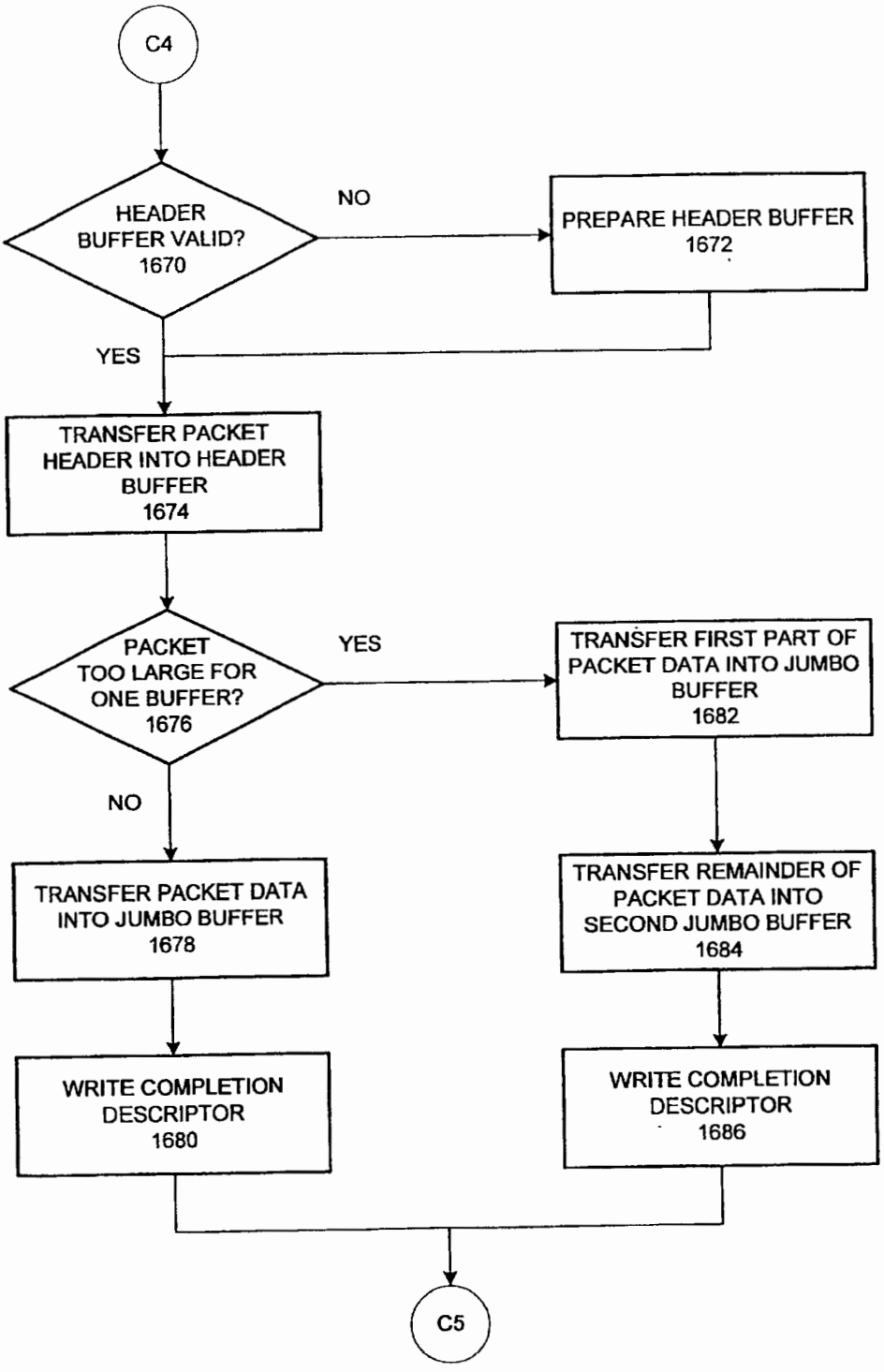

C4
HEADER BUFFER VALID? 1670
NO
PREPARE HEADER BUFFER 1672
YES
TRANSFER PACKET HEADER INTO HEADER BUFFER 1674
PACKET TOO LARGE FOR ONE BUFFER? 1676
YES
TRANSFER FIRST PART OF PACKET DATA INTO JUMBO BUFFER 1682
NO
TRANSFER PACKET DATA INTO JUMBO BUFFER 1678
TRANSFER REMAINDER OF PACKET DATA INTO SECOND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

FIG. 16E

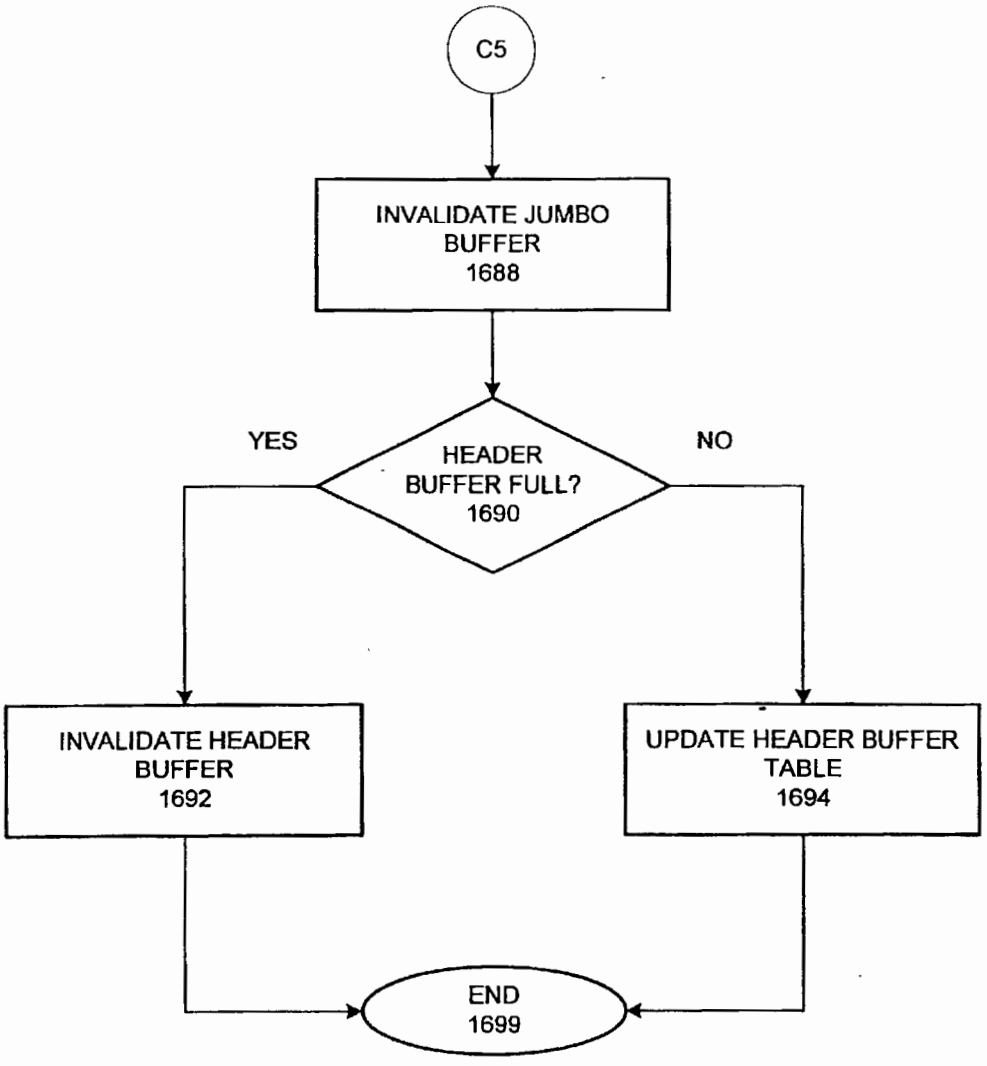

C5
INVALIDATE JUMBO BUFFER 1688
YES
HEADER BUFFER FULL? 1690
NO
INVALIDATE HEADER BUFFER 1692
UPDATE HEADER BUFFER TABLE 1694
END 1699

FIG. 16F

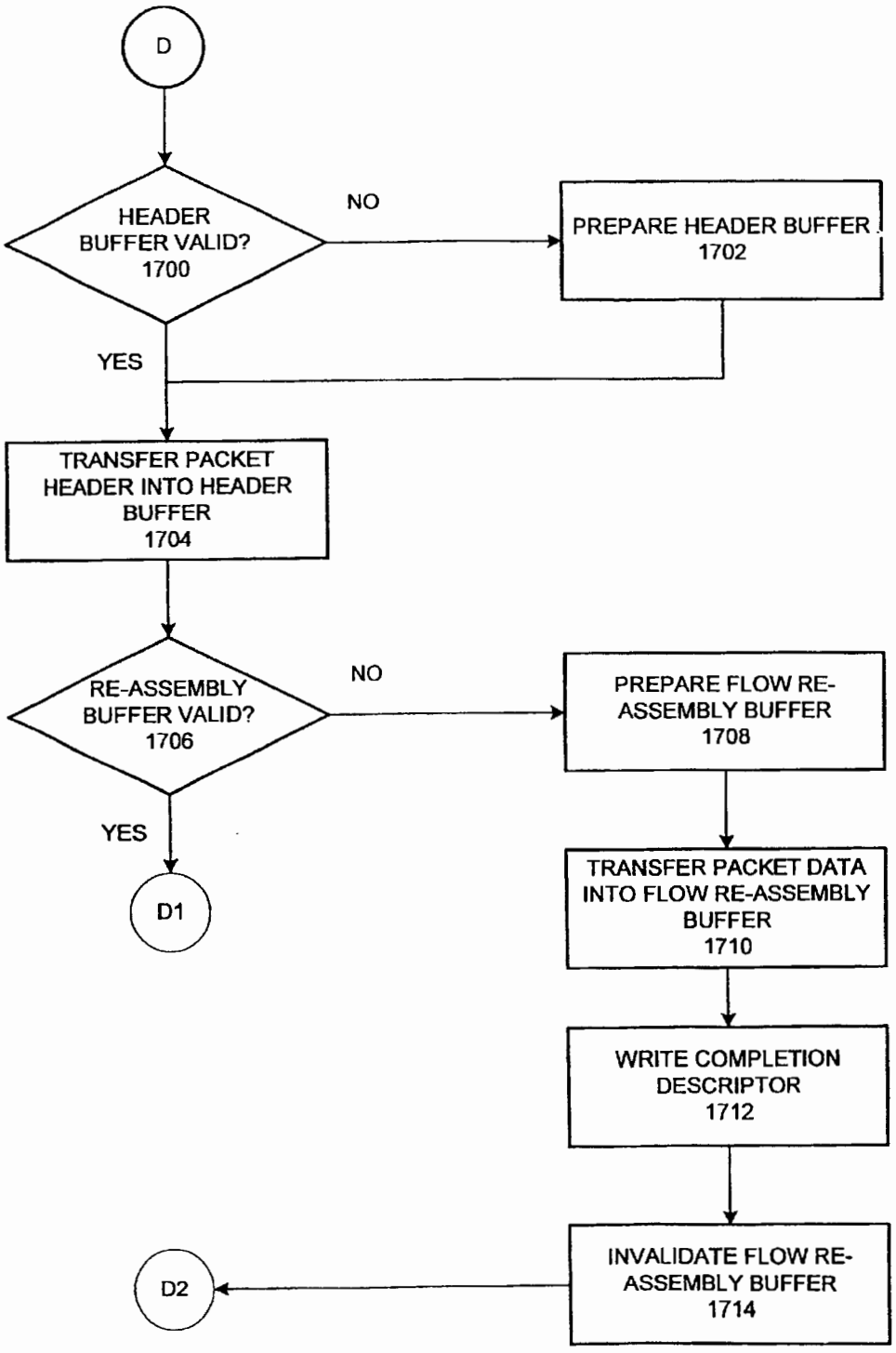

D
HEADER BUFFER VALID? 1700
NO
PREPARE HEADER BUFFER 1702
YES
TRANSFER PACKET HEADER INTO HEADER BUFFER 1704
RE-ASSEMBLY BUFFER VALID? 1706
NO
PREPARE FLOW RE-ASSEMBLY BUFFER 1708
YES
D1
TRANSFER PACKET DATA INTO FLOW RE-ASSEMBLY BUFFER 1710
WRITE COMPLETION DESCRIPTOR 1712
INVALIDATE FLOW RE-ASSEMBLY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

FIG. 17A

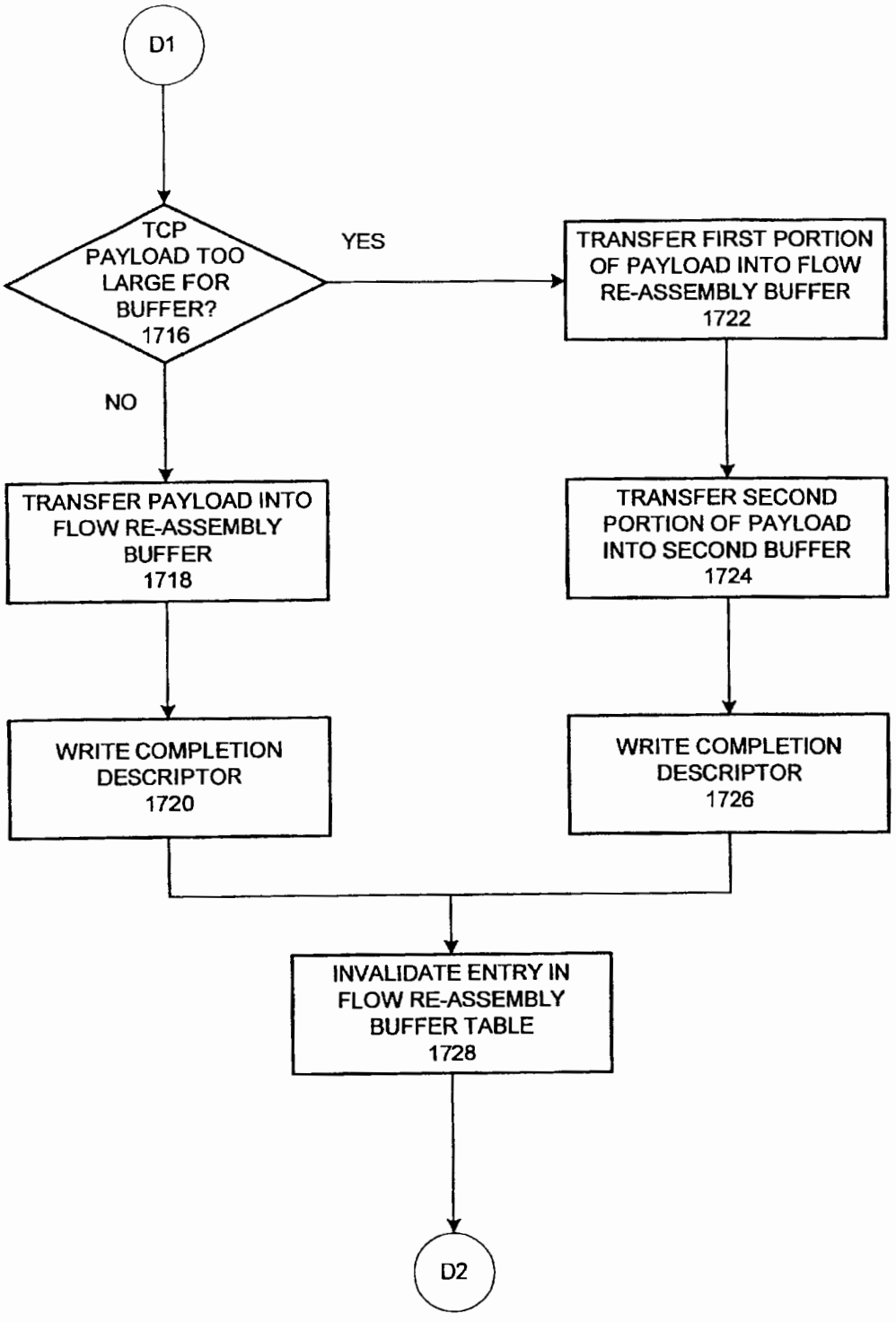

D1
TCP PAYLOAD TOO LARGE FOR BUFFER? 1716
YES
NO
TRANSFER FIRST PORTION OF PAYLOAD INTO FLOW RE-ASSEMBLY BUFFER 1722
TRANSFER PAYLOAD INTO FLOW RE-ASSEMBLY BUFFER 1718
TRANSFER SECOND PORTION OF PAYLOAD INTO SECOND BUFFER 1724
WRITE COMPLETION DESCRIPTOR 1720
WRITE COMPLETION DESCRIPTOR 1726
INVALIDATE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

FIG. 17B

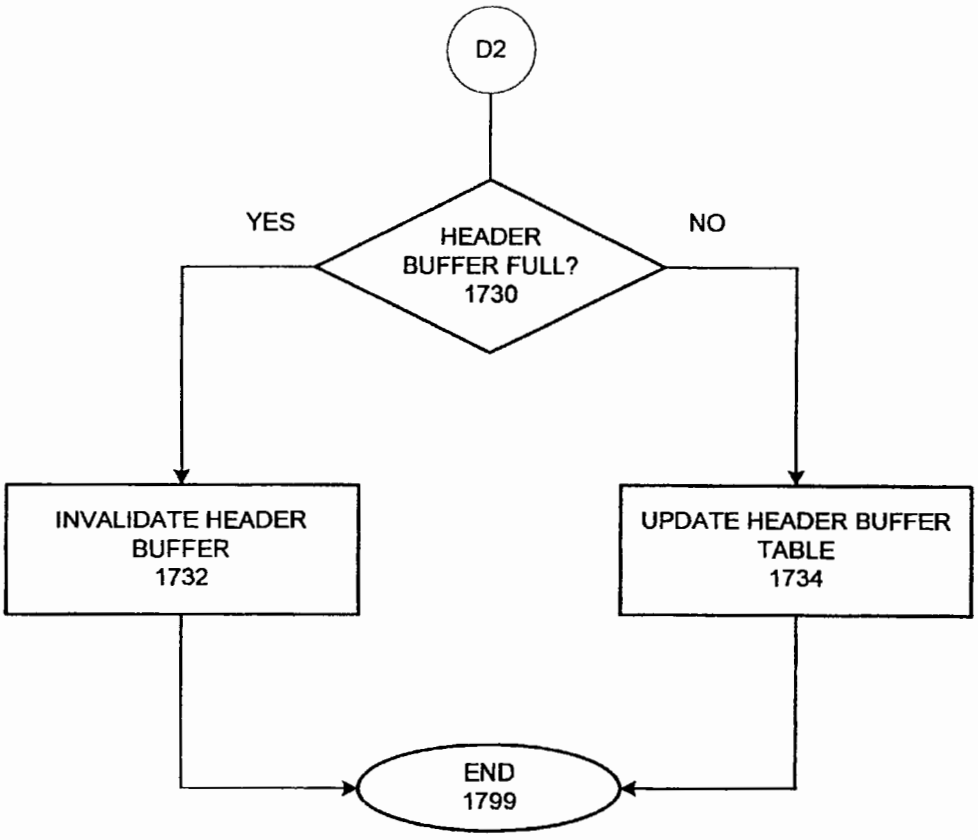

D2
HEADER BUFFER FULL? 1730
YES
NO
INVALIDATE HEADER BUFFER 1732
UPDATE HEADER BUFFER TABLE 1734
END 1799

FIG. 17C

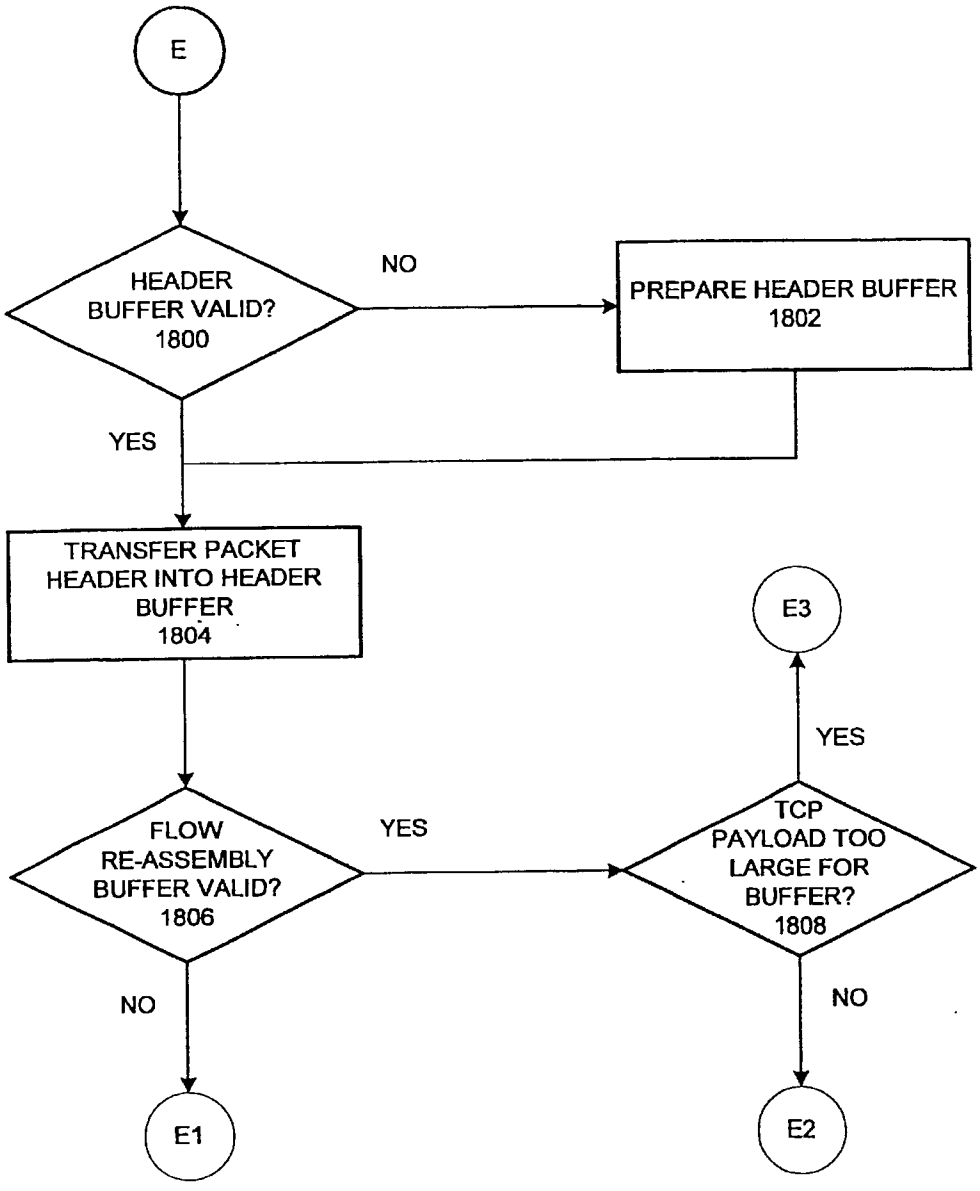

E
HEADER BUFFER VALID? 1800
NO
PREPARE HEADER BUFFER 1802
YES
TRANSFER PACKET HEADER INTO HEADER BUFFER 1804
FLOW RE-ASSEMBLY BUFFER VALID? 1806
YES
TCP PAYLOAD TOO LARGE FOR BUFFER? 1808
YES
E3
NO
NO
E1
E2

FIG. 18A

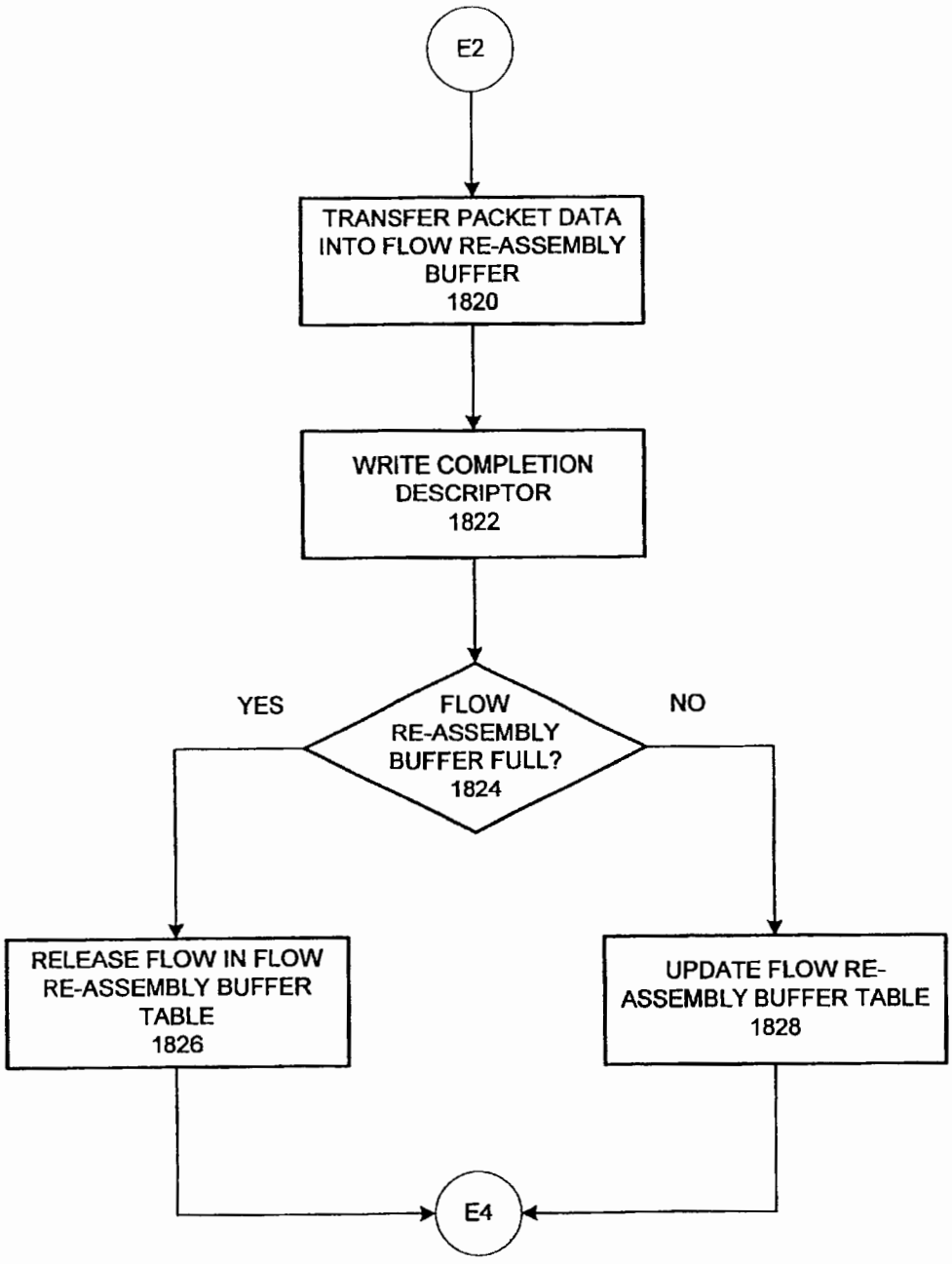

E2
TRANSFER PACKET DATA INTO FLOW RE-ASSEMBLY BUFFER 1820
WRITE COMPLETION DESCRIPTOR 1822
FLOW RE-ASSEMBLY BUFFER FULL? 1824
YES
NO
RELEASE FLOW IN FLOW RE-ASSEMBLY BUFFER TABLE 1826
UPDATE FLOW RE-ASSEMBLY BUFFER TABLE 1828
E4

FIG. 18C

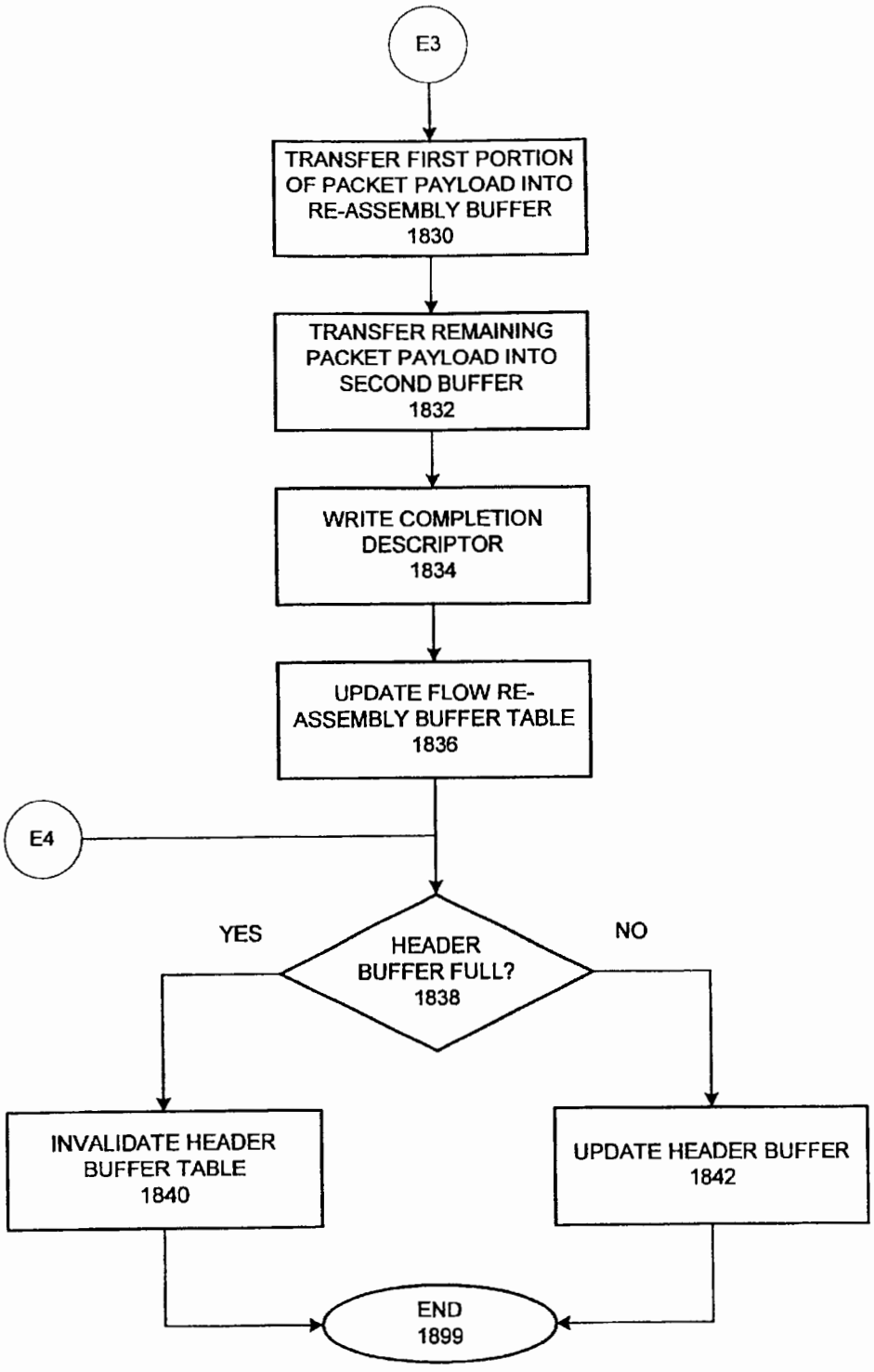

E3
TRANSFER FIRST PORTION OF PACKET PAYLOAD INTO RE-ASSEMBLY BUFFER 1830
TRANSFER REMAINING PACKET PAYLOAD INTO SECOND BUFFER 1832
WRITE COMPLETION DESCRIPTOR 1834
UPDATE FLOW RE-ASSEMBLY BUFFER TABLE 1836
E4
HEADER BUFFER FULL? 1838
YES
NO
INVALIDATE HEADER BUFFER TABLE 1840
UPDATE HEADER BUFFER 1842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

FIG. 18D

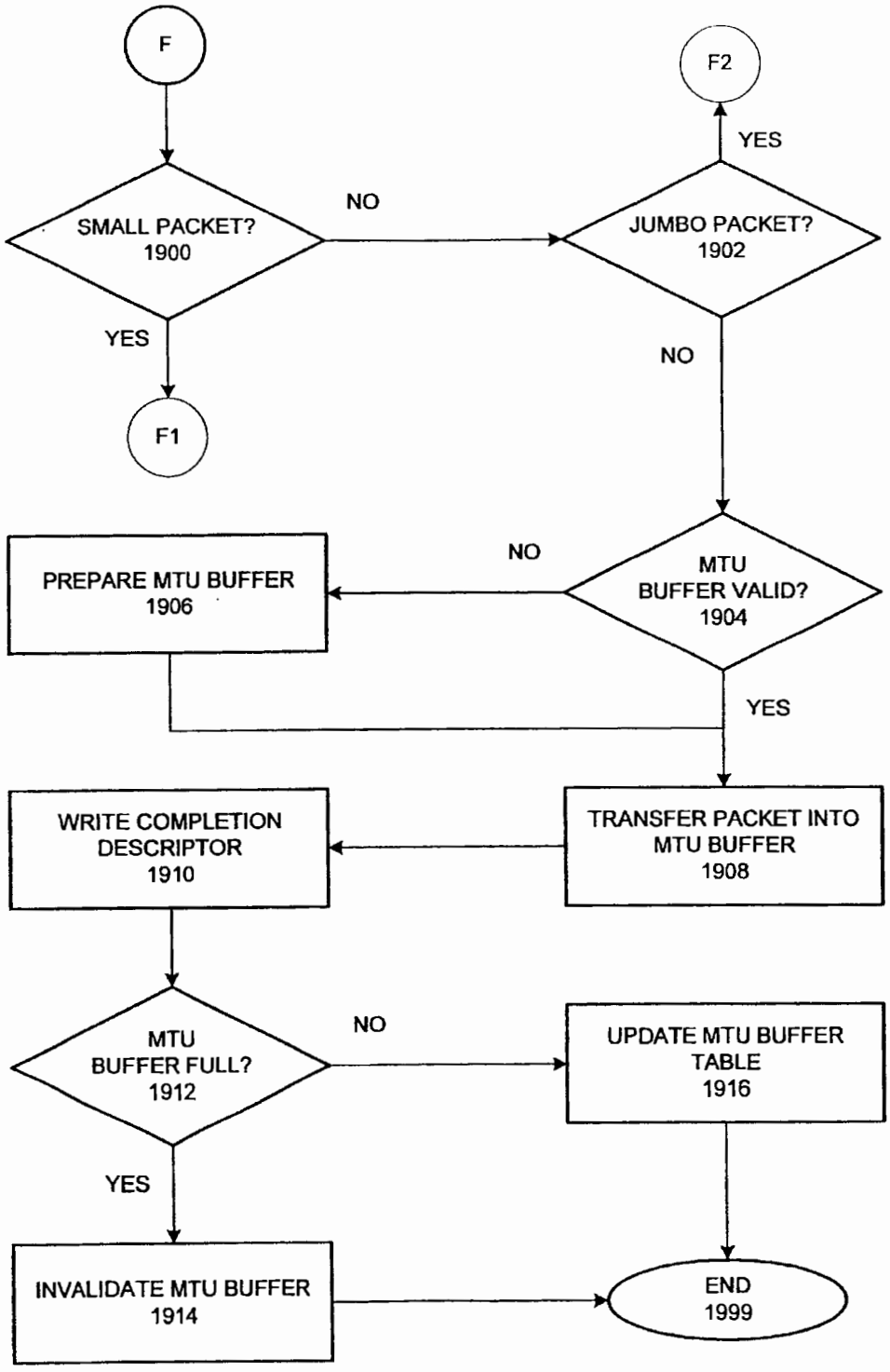

F
F2
YES
SMALL PACKET?
1900
NO
JUMBO PACKET?
1902
YES
NO
F1
PREPARE MTU BUFFER
1906
NO
MTU
BUFFER VALID?
1904
YES
WRITE COMPLETION
DESCRIPTOR
1910
TRANSFER PACKET INTO
MTU BUFFER
1908
MTU
BUFFER FULL?
1912
NO
UPDATE MTU BUFFER
TABLE
1916
YES
INVALIDATE MTU BUFFER
1914
END
1999

FIG. 19A

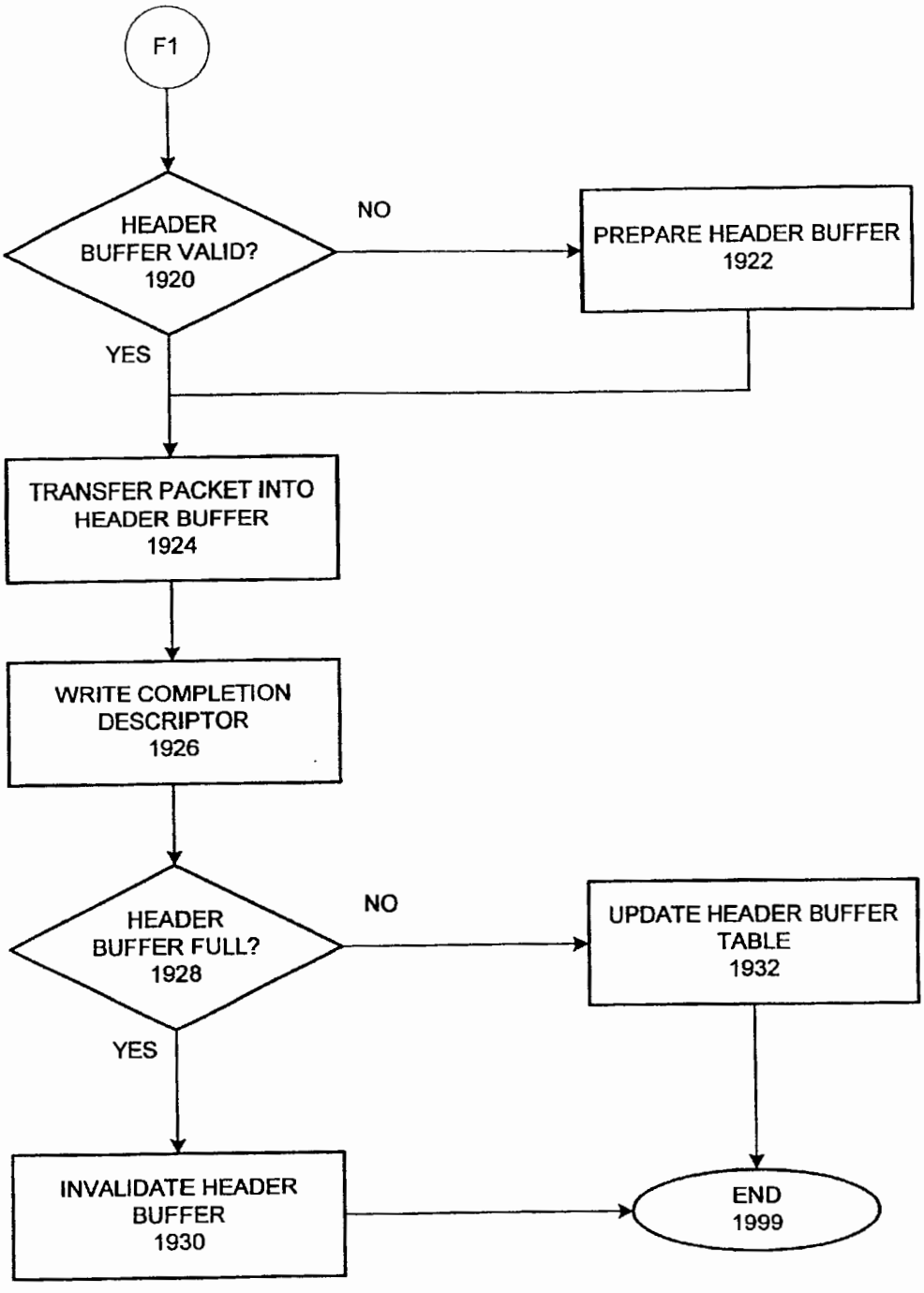

F1
HEADER BUFFER VALID? 1920
NO
PREPARE HEADER BUFFER 1922
YES
TRANSFER PACKET INTO HEADER BUFFER 1924
WRITE COMPLETION DESCRIPTOR 1926
HEADER BUFFER FULL? 1928
NO
UPDATE HEADER BUFFER TABLE 1932
YES
INVALIDATE HEADER BUFFER 1930
END 1999

FIG. 19B

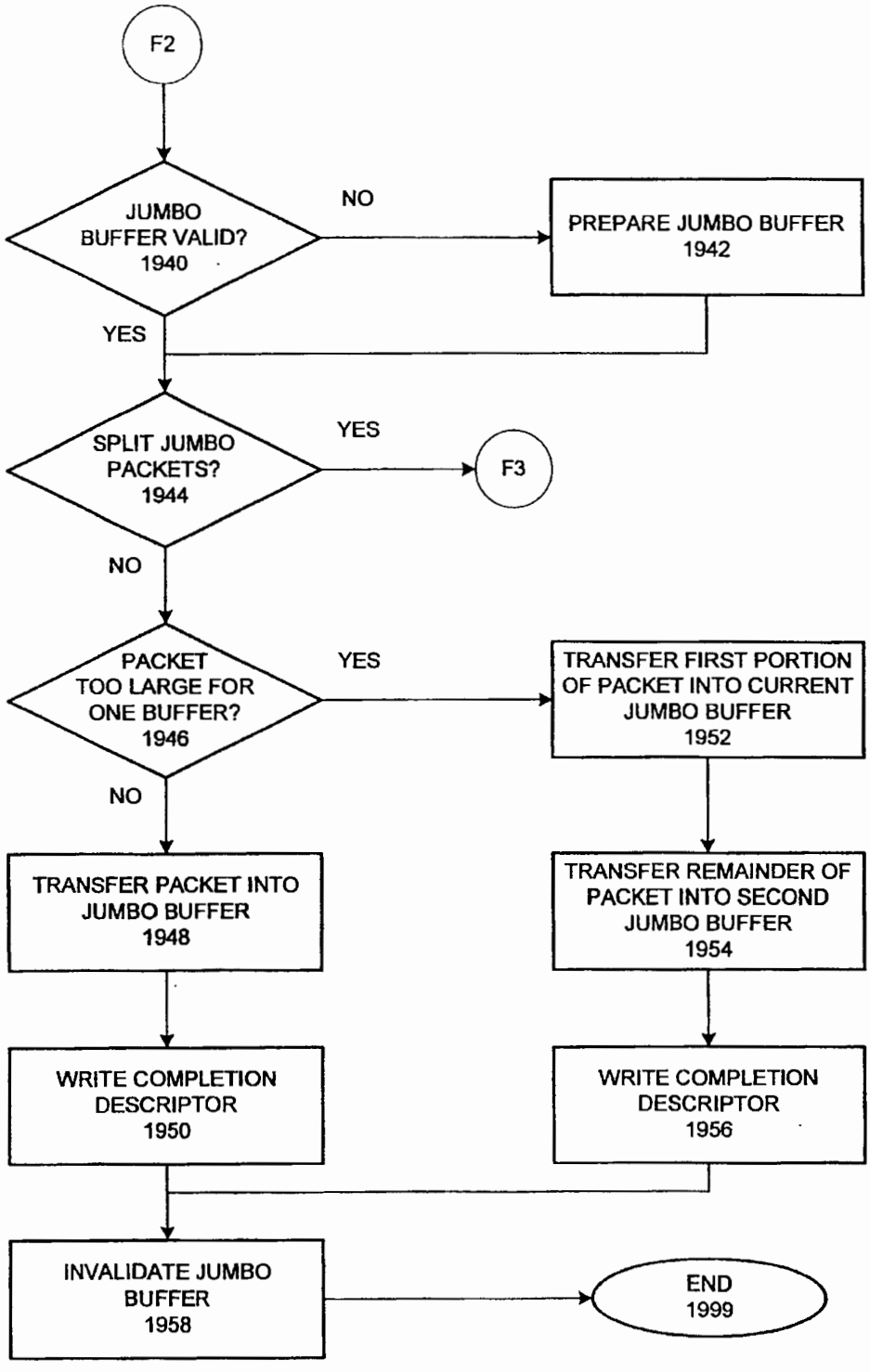

F2
JUMBO BUFFER VALID? 1940
NO
PREPARE JUMBO BUFFER 1942
YES
SPLIT JUMBO PACKETS? 1944
YES
F3
NO
PACKET TOO LARGE FOR ONE BUFFER? 1946
YES
TRANSFER FIRST PORTION OF PACKET INTO CURRENT JUMBO BUFFER 1952
NO
TRANSFER PACKET INTO JUMBO BUFFER 1948
TRANSFER REMAINDER OF PACKET INTO SECOND JUMBO BUFFER 1954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

FIG. 19C

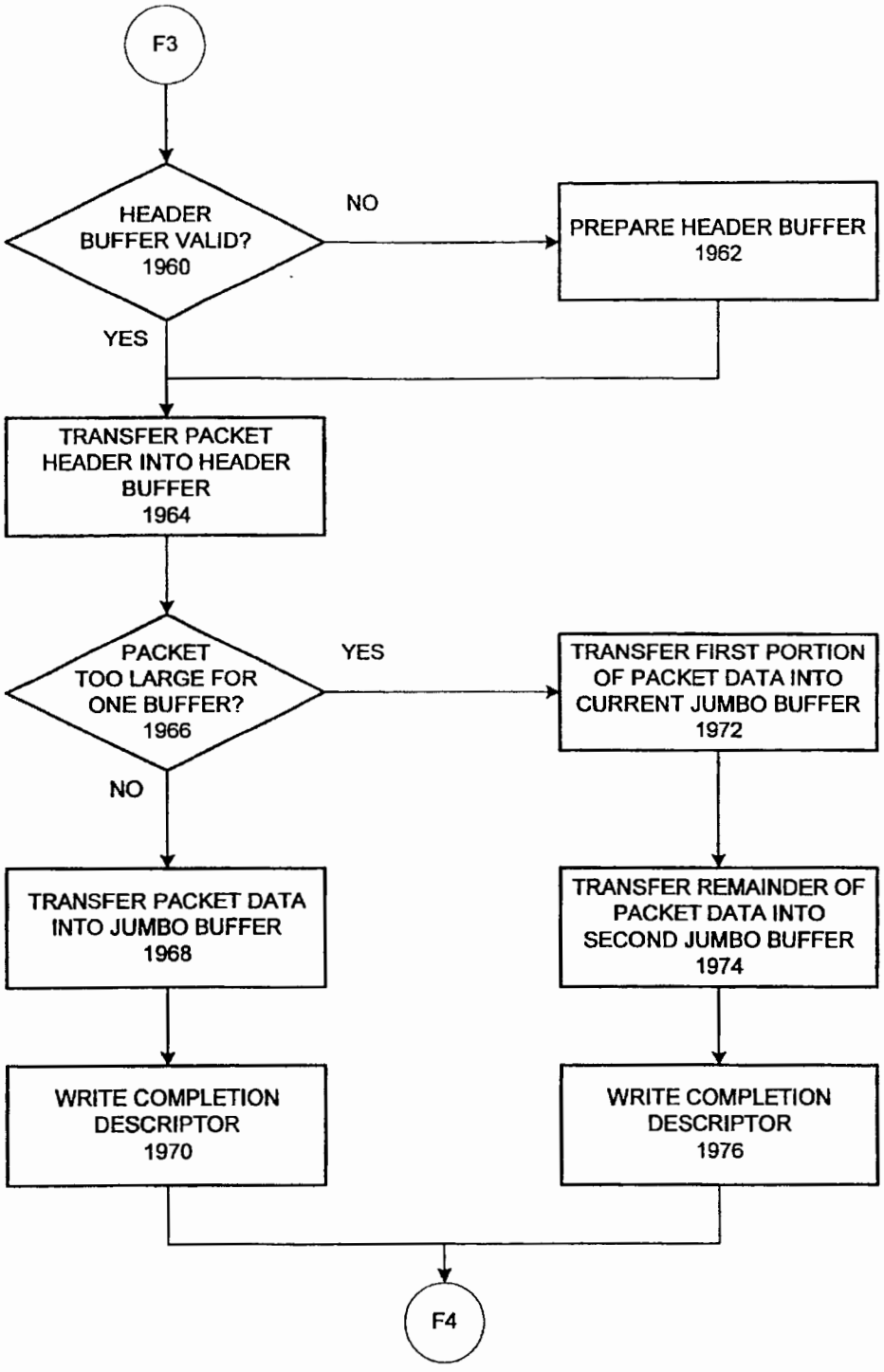

F3
HEADER BUFFER VALID? 1960
NO
PREPARE HEADER BUFFER 1962
YES
TRANSFER PACKET HEADER INTO HEADER BUFFER 1964
PACKET TOO LARGE FOR ONE BUFFER? 1966
YES
TRANSFER FIRST PORTION OF PACKET DATA INTO CURRENT JUMBO BUFFER 1972
NO
TRANSFER PACKET DATA INTO JUMBO BUFFER 1968
TRANSFER REMAINDER OF PACKET DATA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

FIG. 19D

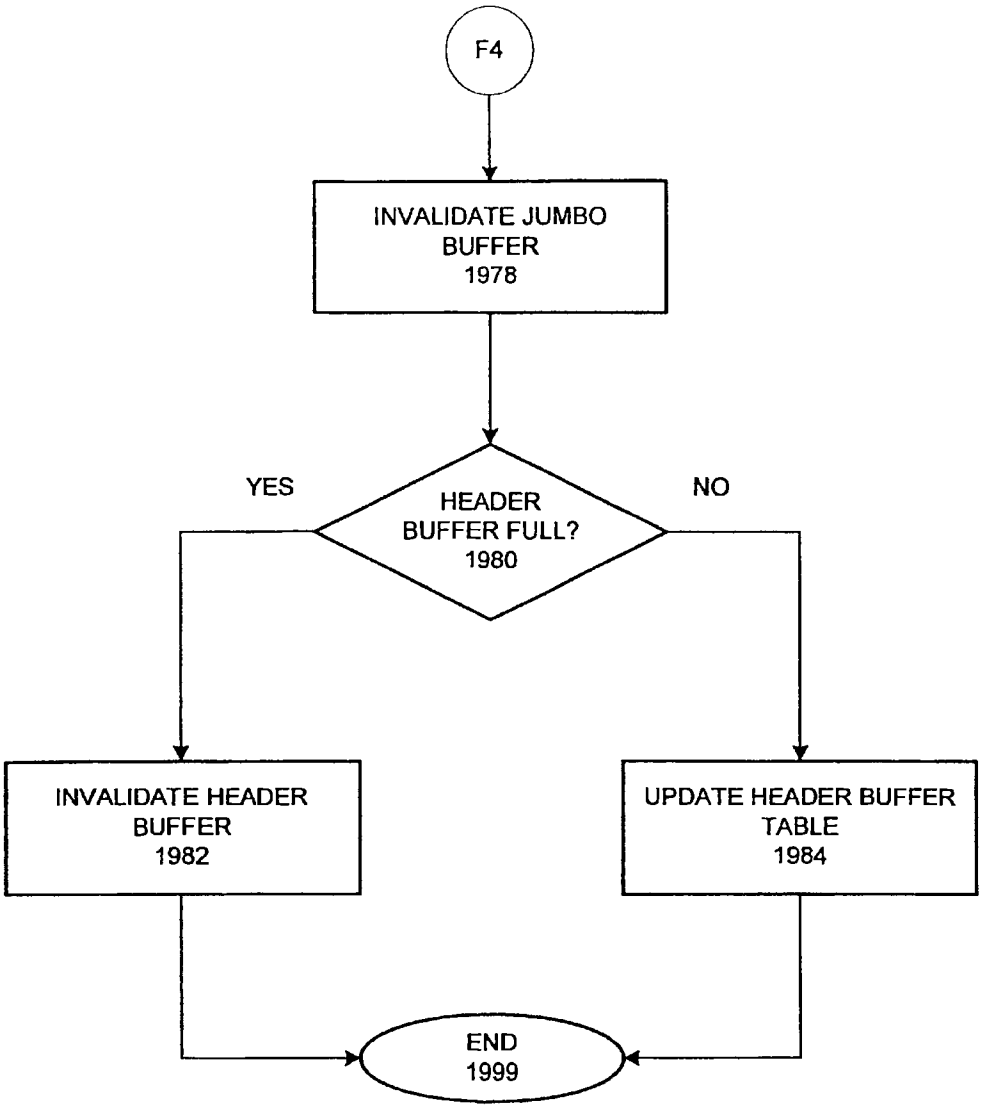

F4
INVALIDATE JUMBO BUFFER 1978
HEADER BUFFER FULL? 1980
YES
NO
INVALIDATE HEADER BUFFER 1982
UPDATE HEADER BUFFER TABLE 1984
END 1999

FIG. 19E

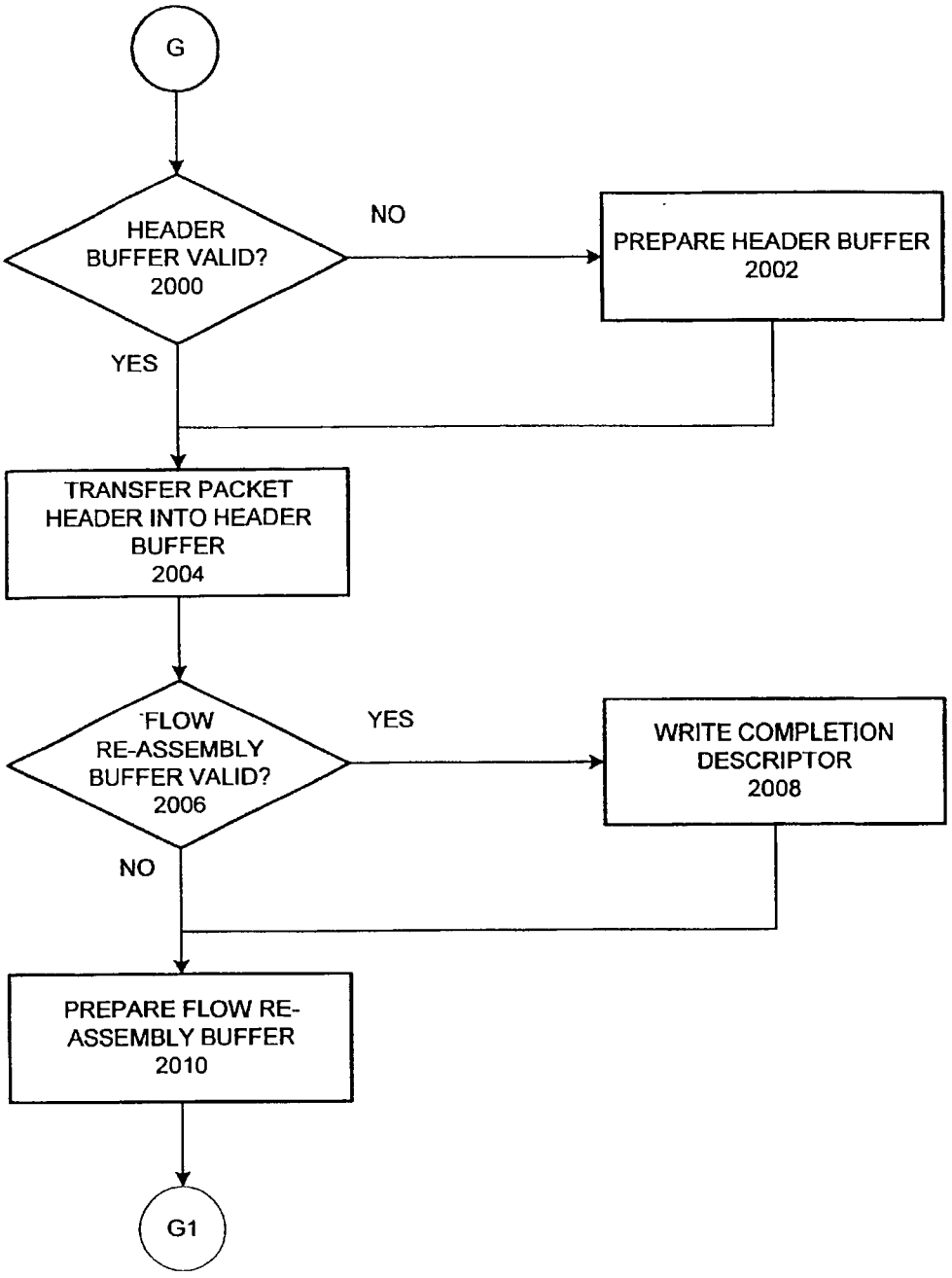

G
HEADER BUFFER VALID? 2000
NO
PREPARE HEADER BUFFER 2002
YES
TRANSFER PACKET HEADER INTO HEADER BUFFER 2004
FLOW RE-ASSEMBLY BUFFER VALID? 2006
YES
WRITE COMPLETION DESCRIPTOR 2008
NO
PREPARE FLOW RE-ASSEMBLY BUFFER 2010
G1

FIG. 20A

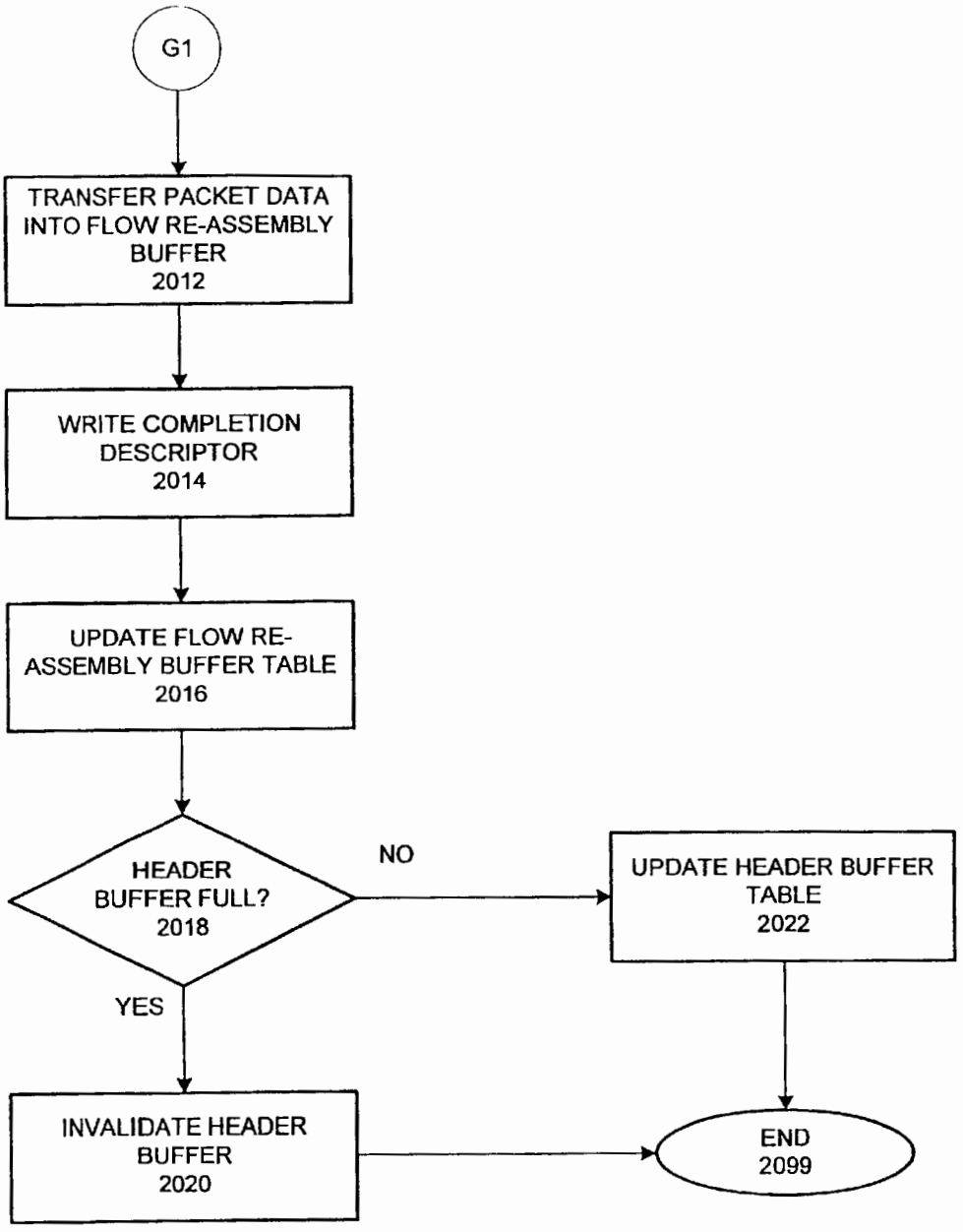

G1
TRANSFER PACKET DATA INTO FLOW RE-ASSEMBLY BUFFER 2012
WRITE COMPLETION DESCRIPTOR 2014
UPDATE FLOW RE-ASSEMBLY BUFFER TABLE 2016
HEADER BUFFER FULL? 2018
NO
UPDATE HEADER BUFFER TABLE 2022
YES
INVALIDATE HEADER BUFFER 2020
END 2099

FIG. 20B

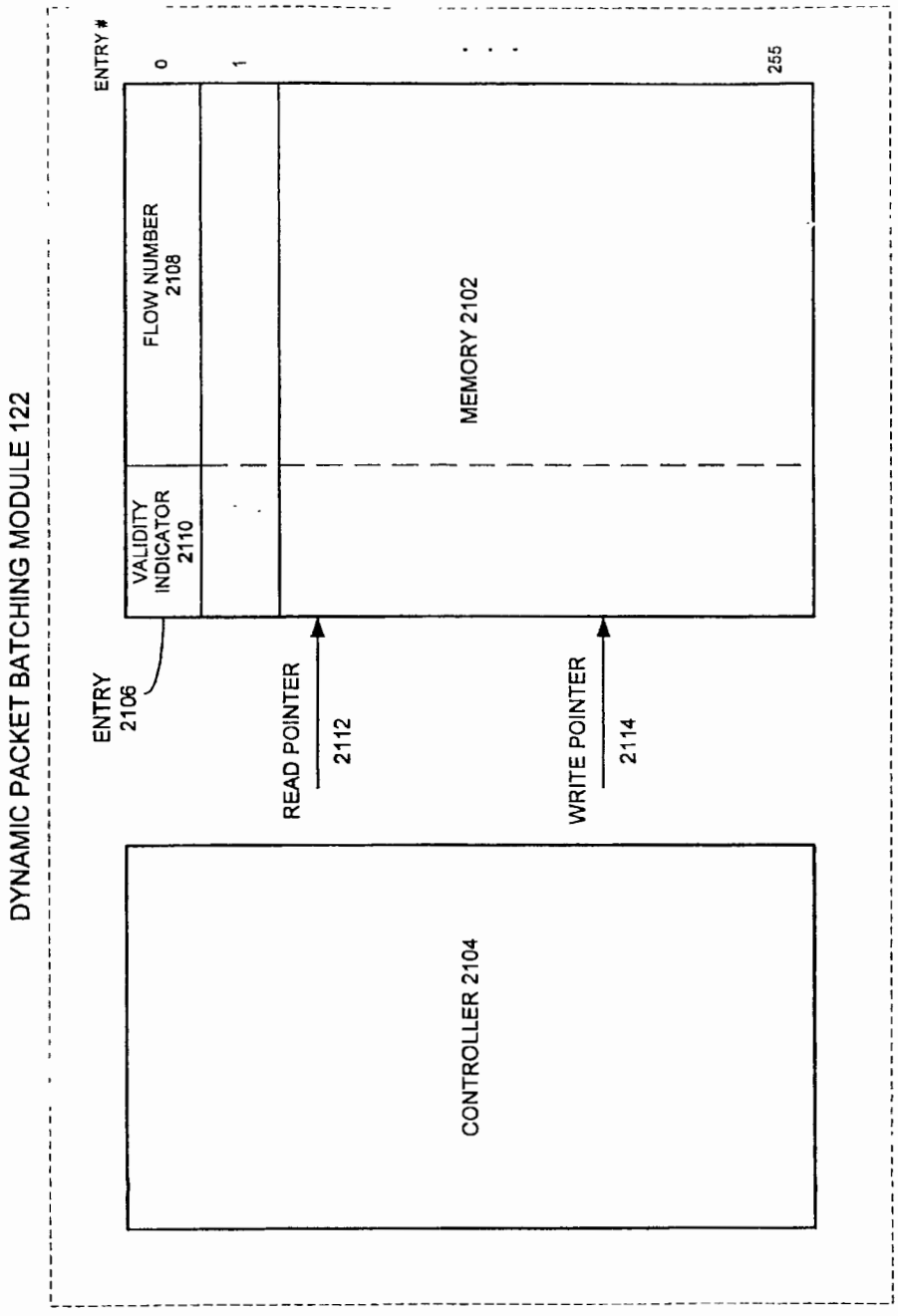

DYNAMIC PACKET BATCHING MODULE 122
ENTRY #
0
1
255
FLOW NUMBER 2108
VALIDITY INDICATOR 2110
MEMORY 2102
ENTRY 2106
READ POINTER
2112
WRITE POINTER
2114
CONTROLLER 2104

FIG. 21

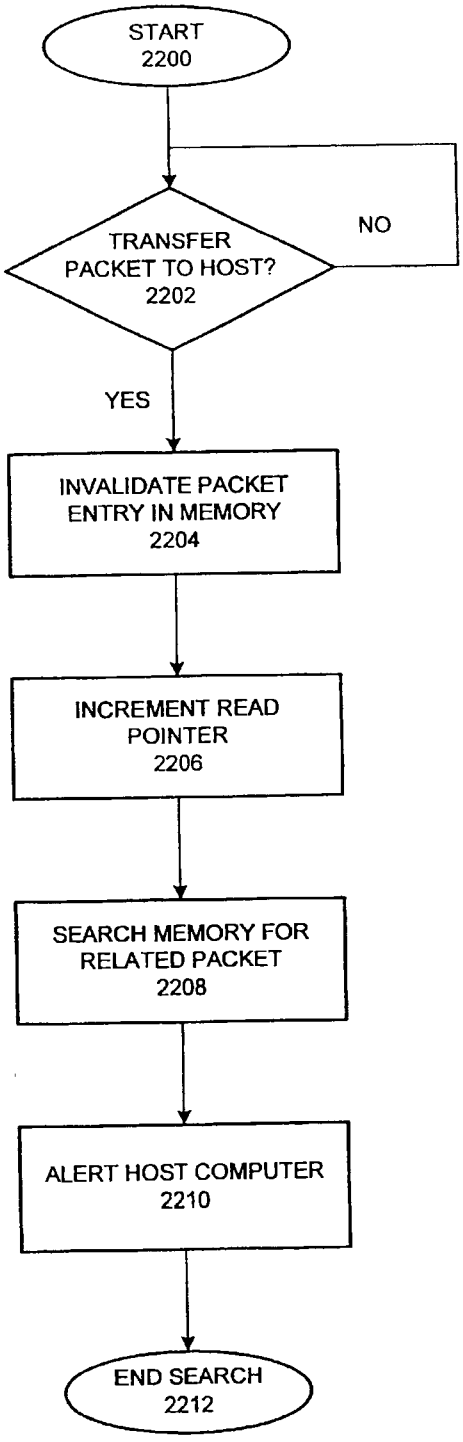

START
2200
TRANSFER PACKET TO HOST?
2202
NO
YES
INVALIDATE PACKET ENTRY IN MEMORY
2204
INCREMENT READ POINTER
2206
SEARCH MEMORY FOR RELATED PACKET
2208
ALERT HOST COMPUTER
2210
END SEARCH
2212

FIG. 22A

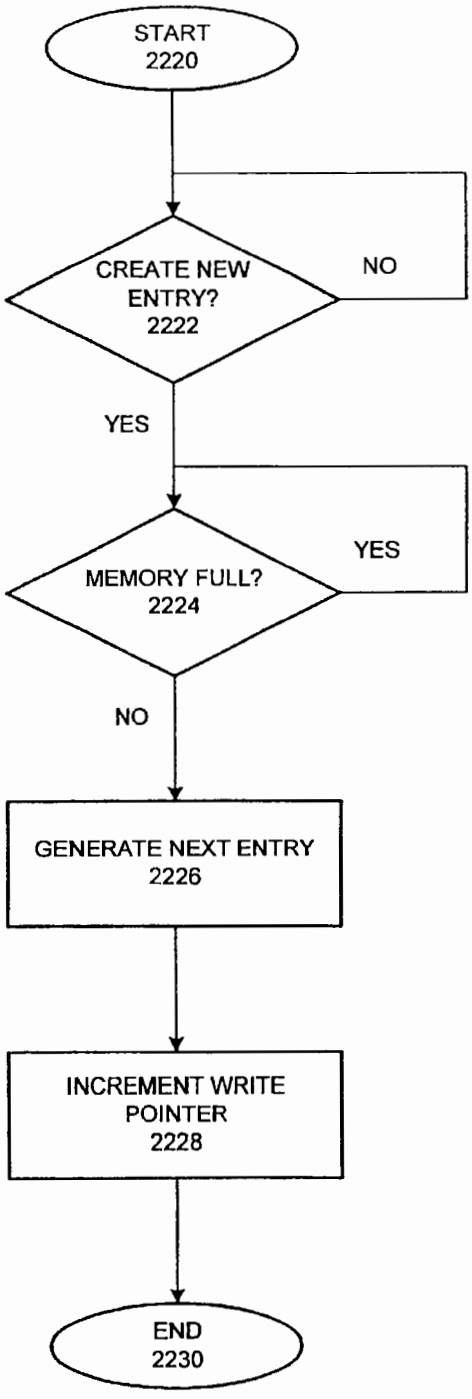

START
2220
CREATE NEW ENTRY?
2222
NO
YES
MEMORY FULL?
2224
YES
NO
GENERATE NEXT ENTRY
2226
INCREMENT WRITE POINTER
2228
END
2230

FIG. 22B

INSTR. NO. 2302	INSTR. NAME 2304	INSTRUCTION CONTENT 2306 (EXTRACTION MASK, COMPARE VALUE, OPERATOR, SUCCESS OFFSET, SUCCESS INSTRUCTION, FAILURE OFFSET, FAILURE INSTRUCTION, OUTPUT OPERATION, OPERATION ARGUMENT, OPERATION ENABLER, SHIFT, OUTPUT MASK)
0	WAIT	0xFFFF, 0x0000, NP, 6, VLAN, 0, WAIT, CLR_REG, 0x3FF, 1, 0, 0x0000
1	VLAN	0xFFFF, 0x8100, EQ, 1, CFI, 0, 802.3, IM_CTL, 0x00A, 3, 0, 0xFFFF
2	CFI	0x1000, 0x1000, EQ, 0, DONE, 1, 802.3, NONE, 0x000, 0, 0, 0x0000
3	802.3	0xFFFF, 0x0600, LT, 1, LLC_1, 0, IPV4_1, NONE, 0x000, 0, 0, 0x0000
4	LLC_1	0xFFFF, 0xAAAA, EQ, 1, LLC_2, 0, DONE, NONE, 0x000, 0, 0, 0x0000
5	LLC_2	0xFF00, 0x0300, EQ, 2, IPV4_1, 0, DONE, NONE, 0x000, 0, 0, 0x0000
6	IPV4_1	0xFFFF, 0x0800, EQ, 1, IPV4_2, 0, IPV6_1, LD_SAP, 0x100, 3, 0, 0xFFFF
7	IPV4_2	0xFF00, 0x4500, EQ, 3, IPV4_3, 0, DONE, LD_SUM, 0x00A, 1, 0, 0x0000
8	IPV4_3	0x3FFF, 0x0000, EQ, 1, IPV4_4, 0, DONE, LD_LEN, 0x03E, 1, 0, 0xFFFF
9	IPV4_4	0x00FF, 0x0006, EQ, 7, TCP_1, 0, DONE, LD_FID, 0x182, 1, 0, 0xFFFF
10	IPV6_1	0xFFFF, 0x86DD, EQ, 1, IPV6_2, 0, DONE, LD_SUM, 0x015, 1, 0x0000
11	IPV6_2	0xF000, 0x6000, EQ, 0, IPV6_3, 0, DONE, IM_R1, 0x114, 1, 0, 0xFFFF
12	IPV6_3	0x0000, 0x0000, EQ, 3, IPV6_4, 0, DONE, LD_FID, 0x484, 1, 0, 0xFFFF
13	IPV6_4	0xFF00, 0x0600, EQ, 18, TCP_1, 0, DONE, LD_LEN, 0x03F, 1, 0xFFFF
14	TCP_1	0x0000, 0x0000, EQ, 0, TCP_2, 4, TCP_2, LD_SEQ, 0x081, 3, 0, 0xFFFF
15	TCP_2	0x0000, 0x0000, EQ, 0, TCP_3, 0, TCP_3, ST_FLAG, 0x145, 3, 0, 0x002F
16	TCP_3	0x0000, 0x0000, EQ, 0, TCP_4, 0, TCP_4, LD_R1, 0x205, 3, 0xB, 0xF000
17	TCP_4	0x0000, 0x0000, EQ, 0, WAIT, 0, WAIT, LD_HDR, 0x0FF, 3, 0, 0xFFFF
18	DONE	0x0000, 0x0000, EQ, 0, WAIT, 0, WAIT, IM_CTL, 0x001, 3, 0x0000

PROGRAM 2300

FIG. 23

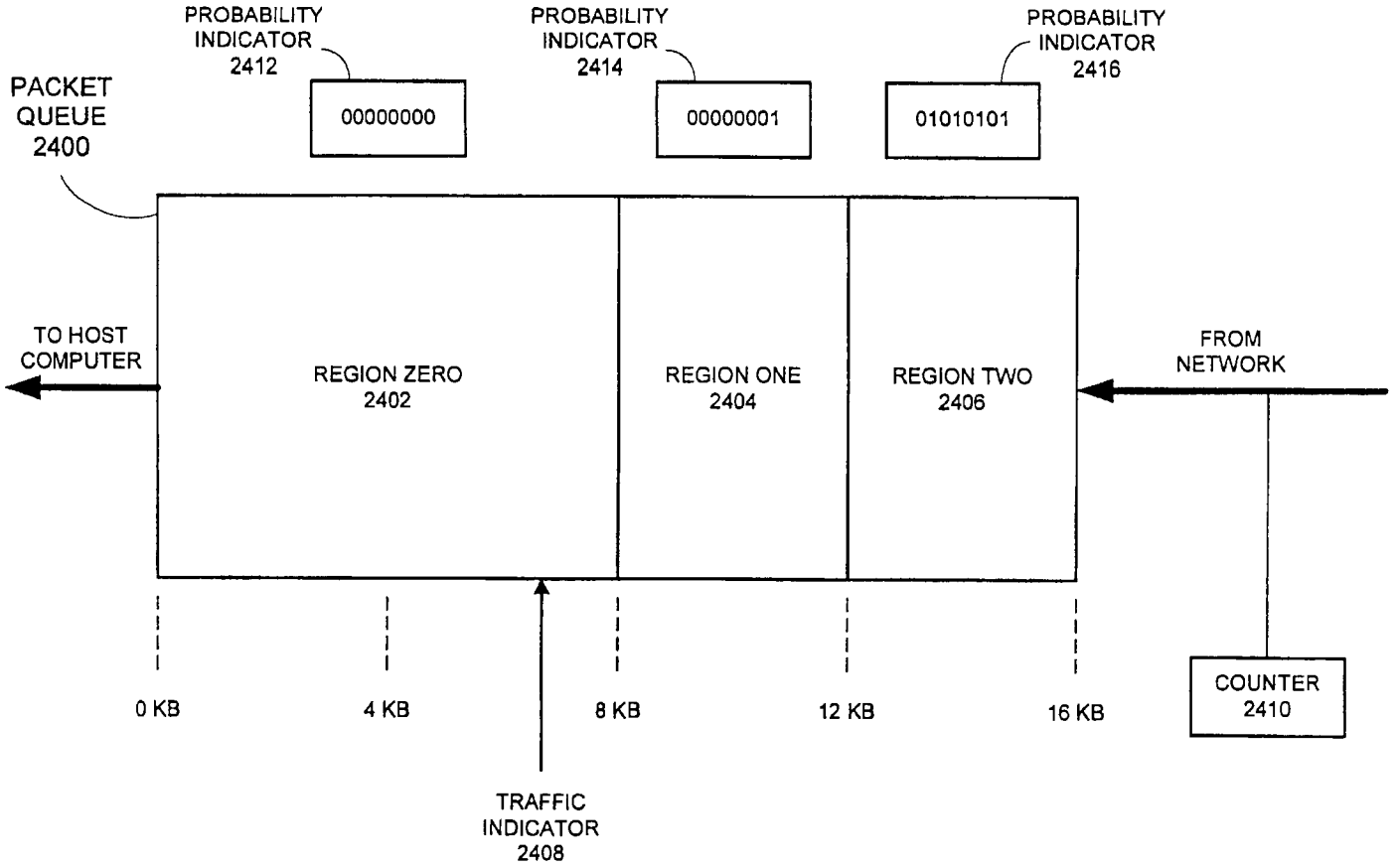

PROBABILITY INDICATOR 2412
PROBABILITY INDICATOR 2414
PROBABILITY INDICATOR 2416
PACKET QUEUE 2400
00000000
00000001
01010101
TO HOST COMPUTER
REGION ZERO 2402
REGION ONE 2404
REGION TWO 2406
FROM NETWORK
COUNTER 2410
0 KB
4 KB
8 KB
12 KB
16 KB
TRAFFIC INDICATOR 2408

FIG. 24

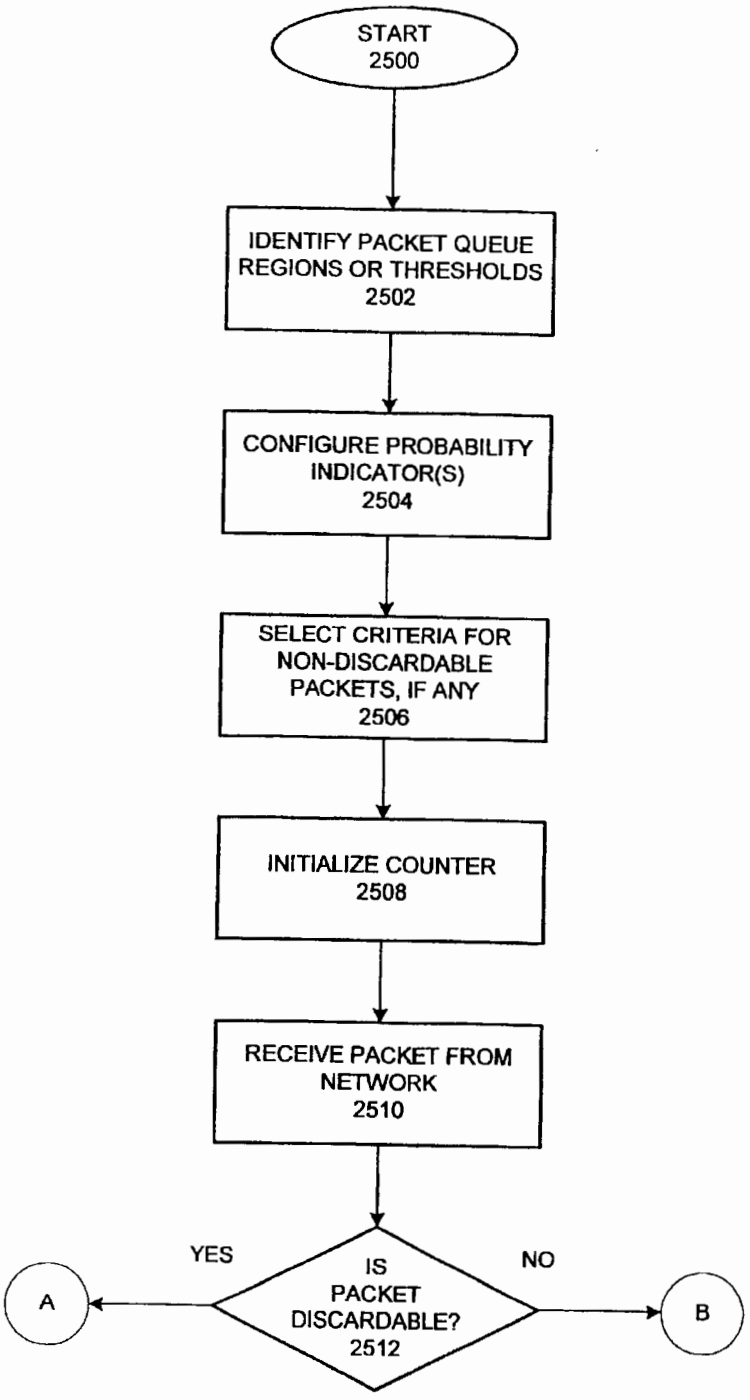

START
2500
IDENTIFY PACKET QUEUE REGIONS OR THRESHOLDS
2502
CONFIGURE PROBABILITY INDICATOR(S)
2504
SELECT CRITERIA FOR NON-DISCARDABLE PACKETS, IF ANY
2506
INITIALIZE COUNTER
2508
RECEIVE PACKET FROM NETWORK
2510
IS PACKET DISCARDABLE?
2512
YES
NO
A
B

FIG. 25A

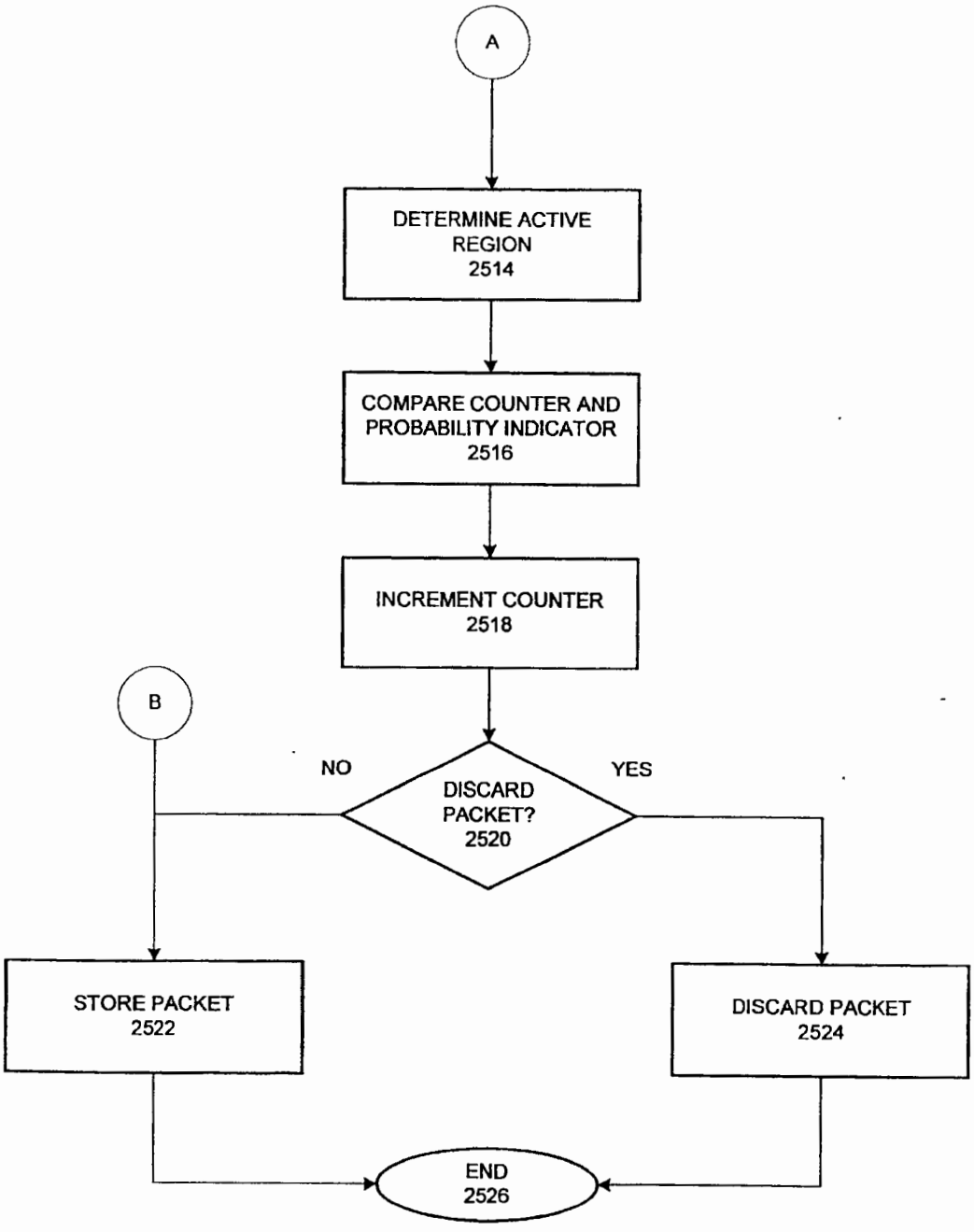

A
DETERMINE ACTIVE REGION 2514
COMPARE COUNTER AND PROBABILITY INDICATOR 2516
INCREMENT COUNTER 2518
B
NO
YES
DISCARD PACKET? 2520
STORE PACKET 2522
DISCARD PACKET 2524
END 2526

FIG. 25B

METHOD AND APPARATUS FOR MANAGING A NETWORK FLOW IN A HIGH PERFORMANCE NETWORK INTERFACE

TABLE OF CONTENTS

BACKGROUND

This invention relates to the fields of computer systems and computer networks. In particular, the present invention relates to a Network Interface Circuit (NIC) for processing communication packets exchanged between a computer network and a host computer system.

The interface between a computer and a network is often a bottleneck for communications passing between the computer and the network. While computer performance (e.g., processor speed) has increased exponentially over the years and computer network transmission speeds have undergone similar increases, inefficiencies in the way network interface circuits handle communications have become more and more evident. With each incremental increase in computer or network speed, it becomes ever more apparent that the interface between the computer and the network cannot keep pace. These inefficiencies involve several basic problems in the way communications between a network and a computer are handled.

Today's most popular forms of networks tend to be packet-based. These types of networks, including the Internet and many local area networks, transmit information in the form of packets. Each packet is separately created and transmitted by an originating endstation and is separately received and processed by a destination endstation. In addition, each packet may, in a bus topology network for example, be received and processed by numerous stations located between the originating and destination endstations.

One basic problem with packet networks is that each packet must be processed through multiple protocols or protocol levels (known collectively as a "protocol stack") on both the origination and destination endstations. When data transmitted between stations is longer than a certain minimal length, the data is divided into multiple portions, and each portion is carried by a separate packet. The amount of data that a packet can carry is generally limited by the network that conveys the packet and is often expressed as a maximum transfer unit (MTU). The original aggregation of data is sometimes known as a "datagram," and each packet carrying part of a single datagram is processed very similarly to the other packets of the datagram.

Communication packets are generally processed as follows. In the origination endstation, each separate data portion of a datagram is processed through a protocol stack. During this processing multiple protocol headers (e.g., TCP, IP, Ethernet) are added to the data portion to form a packet that can be transmitted across the network. The packet is received by a network interface circuit, which transfers the packet to the destination endstation or a host computer that serves the destination endstation. In the destination endstation, the packet is processed through the protocol stack in the opposite direction as in the origination endstation. During this processing the protocol headers are removed in the opposite order in which they were applied. The data portion is thus recovered and can be made available to a user, an application program, etc.

Several related packets (e.g., packets carrying data from one datagram) thus undergo substantially the same process in a serial manner (i.e., one packet at a time). The more data that must be transmitted, the more packets must be sent, with each one being separately handled and processed through the protocol stack in each direction. Naturally, the more packets that must be processed, the greater the demand placed upon an endstation's processor. The number of packets that must be processed is affected by factors other than just the amount of data being sent in a datagram. For example, as the amount of data that can be encapsulated in a packet increases, fewer packets need to be sent. As stated above, however, a packet may have a maximum allowable size, depending on the type of network in use (e.g., the maximum transfer unit for standard Ethernet traffic is approximately 1,500 bytes). The speed of the network also affects the number of packets that a NIC may handle in a given period of time. For example, a gigabit Ethernet network operating at peak capacity may require a NIC to receive approximately 1.48 million packets per second. Thus, the number of packets to be processed through a protocol stack may place a significant burden upon a computer's processor. The situation is exacerbated by the need to process each packet separately even though each one will be processed in a substantially similar manner.

A related problem to the disjoint processing of packets is the manner in which data is moved between "user space" (e.g., an application program's data storage) and "system space" (e.g., system memory) during data transmission and receipt. Presently, data is simply copied from one area of

memory assigned to a user or application program into another area of memory dedicated to the processor's use. Because each portion of a datagram that is transmitted in a packet may be copied separately (e.g., one byte at a time), there is a nontrivial amount of processor time required and frequent transfers can consume a large amount of the memory bus' bandwidth. Illustratively, each byte of data in a packet received from the network may be read from the system space and written to the user space in a separate copy operation, and vice versa for data transmitted over the network. Although system space generally provides a protected memory area (e.g., protected from manipulation by user programs), the copy operation does nothing of value when seen from the point of view of a network interface circuit. Instead, it risks over-burdening the host processor and retarding its ability to rapidly accept additional network traffic from the NIC. Copying each packet's data separately can therefore be very inefficient, particularly in a high-speed network environment.

In addition to the inefficient transfer of data (e.g., one packet's data at a time), the processing of headers from packets received from a network is also inefficient. Each packet carrying part of a single datagram generally has the same protocol headers (e.g., Ethernet, IP and TCP), although there may be some variation in the values within the packets' headers for a particular protocol. Each packet, however, is individually processed through the same protocol stack, thus requiring multiple repetitions of identical operations for related packets. Successively processing unrelated packets through different protocol stacks will likely be much less efficient than progressively processing a number of related packets through one protocol stack at a time.

Another basic problem concerning the interaction between present network interface circuits and host computer systems is that the combination often fails to capitalize on the increased processor resources that are available in multi-processor computer systems. In other words, present attempts to distribute the processing of network packets (e.g., through a protocol stack) among a number of protocols in an efficient manner are generally ineffective. In particular, the performance of present NICs does not come close to the expected or desired linear performance gains one may expect to realize from the availability of multiple processors. In some multi-processor systems, little improvement in the processing of network traffic is realized from the use of more than 4–6 processors, for example.

In addition, the rate at which packets are transferred from a network interface circuit to a host computer or other communication device may fail to keep pace with the rate of packet arrival at the network interface. One element or another of the host computer (e.g., a memory bus, a processor) may be over-burdened or otherwise unable to accept packets with sufficient alacrity. In this event one or more packets may be dropped or discarded. Dropping packets may cause a network entity to re-transmit some traffic and, if too many packets are dropped, a network connection may require re-initialization. Further, dropping one packet or type of packet instead of another may make a significant difference in overall network traffic. If, for example, a control packet is dropped, the corresponding network connection may be severely affected and may do little to alleviate the packet saturation of the network interface circuit because of the typically small size of a control packet. Therefore, unless the dropping of packets is performed in a manner that distributes the effect among many network connections or that makes allowance for certain types of packets, network traffic may be degraded more than necessary.

Thus, present NICs fail to provide adequate performance to interconnect today's high-end computer systems and high-speed networks. In addition, a network interface circuit that cannot make allowance for an over-burdened host computer may degrade the computer's performance.

SUMMARY

In one embodiment of the invention, a system and method are provided for managing communication flows, or connections, received at a communication device such as a network interface. In particular, communication flows are set up and torn down as network traffic is received at a network interface. Information concerning a flow is maintained for the duration of the flow to assist in determining the suitability of flow packets for certain enhanced processing operations. For example, such operations may be suitable for packets adhering to one or more pre-selected communication protocols.

In this embodiment of the invention a high performance network interface includes a flow database and a flow database manager module. A flow database in this embodiment contains an entry for each valid or active communication flow received by the network interface. Each flow may be identified by a flow key, stored in the flow's database entry, and may be indexed by a flow number.

For each valid flow, the flow database stores information indicating how recently a packet was received for the flow and sequence information concerning a datagram (e.g., a collection of data sent via multiple packets) being passed to the destination entity by the source entity. The sequence information may be used to verify correct receipt of data in the flow.

A communication flow in this embodiment comprises one or more packets sent from a source entity to a destination entity served by the network interface. A flow is thus similar, but not identical, to an end-to-end TCP (Transport Control Protocol) connection. Illustratively, a flow key comprises a combination of identifiers of the source and destination entities. In one embodiment of the invention a flow key is a combination of source and destination addresses extracted from the packet's layer three (e.g., IP or Internet Protocol) protocol header and source and destination port numbers extracted from the layer four (e.g., TCP) protocol header.

When a flow packet is received at the network interface, a flow database manager receives the packet's flow key. The flow key may be assembled by a header parser module that parses a header portion of the packet. The flow database manager may also receive control information concerning the packet, such as an indication of the size of a data portion of the packet, a flow sequence number used to identify the position of the packet data within the datagram, an indicator of the status of one or more flags in the packet's header(s), etc. Using the flow key, the flow database is searched and a database entry is added in the case of a new flow, or updated if the flow already exists.

In one embodiment of the invention, the flow database manager associates an operation code with the received packet to indicate how the packet may be further processed by the network interface and/or a host computer. The specific operation code assigned for a packet may indicate whether the packet contains data that can be re-assembled with other data passed in the flow, whether the packet is a control packet or is otherwise devoid of data, whether the packet should not be processed through a particular network interface function (e.g., due to a flag in a header of the packet), etc.

Information derived from the packet, including the flow key and control information, may be used by other portions of the network interface and/or a host computer system. Illustratively, the information may be used to re-assemble data sent from the source entity to the destination entity, to collectively process multiple packets from one flow, to distribute the processing of network traffic among multiple processors, to verify the integrity of the packet (e.g., by checksum), etc.

BRIEF DESCRIPTION OF THE FIGURES

FIG. **1A** is a block diagram depicting a network interface circuit (NIC) for receiving a packet from a network in accordance with an embodiment of the present invention.

FIG. **1B** is a flow chart demonstrating one method of operating the NIC of FIG. **1A** to transfer a packet received from a network to a host computer in accordance with an embodiment of the invention.

FIG. **2** is a diagram of a packet transmitted over a network and received at a network interface circuit in one embodiment of the invention.

FIG. **3** is a block diagram depicting a header parser of a network interface circuit for parsing a packet in accordance with an embodiment of the invention.

FIGS. **4A–4B** comprise a flow chart demonstrating one method of parsing a packet received from a network at a network interface circuit in accordance with an embodiment of the present invention.

FIG. **5** is a block diagram depicting a network interface circuit flow database in accordance with an embodiment of the invention.

FIGS. **6A–6E** comprise a flowchart illustrating one method of managing a network interface circuit flow database in accordance with an embodiment of the invention.

FIG. **7** is a flow chart demonstrating one method of distributing the processing of network packets among multiple processors on a host computer in accordance with an embodiment of the invention.

FIG. **8** is a diagram of a packet queue for a network interface circuit in accordance with an embodiment of the invention.

FIG. **9** is a diagram of a control queue for a network interface circuit in accordance with an embodiment of the invention.

FIG. **10** is a block diagram of a DMA engine for transferring a packet received from a network to a host computer in accordance with an embodiment of the invention.

FIG. **11** includes diagrams of data structures for managing the storage of network packets in host memory buffers in accordance with an embodiment of the invention.

FIGS. **12A–12B** are diagrams of a free descriptor, a completion descriptor and a free buffer array in accordance with an embodiment of the invention.

FIGS. **13–20** are flow charts demonstrating methods of transferring a packet received from a network to a buffer in a host computer memory in accordance with an embodiment of the invention.

FIG. **21** is a diagram of a dynamic packet batching module in accordance with an embodiment of the invention.

FIGS. **22A–22B** comprise a flow chart demonstrating one method of dynamically searching a memory containing information concerning packets awaiting transfer to a host computer in order to locate a packet in the same communication flow as a packet being transferred, in accordance with an embodiment of the invention.

FIG. **23** depicts one set of dynamic instructions for parsing a packet in accordance with an embodiment of the invention.

FIG. **24** depicts a system for randomly discarding a packet from a network interface in accordance with an embodiment of the invention.

FIGS. **25A–25B** comprise a flow chart demonstrating one method of discarding a packet from a network interface in accordance with an embodiment of the invention.

DETAILED DESCRIPTION

The following description is presented to enable any person skilled in the art to make and use the invention, and is provided in the context of particular applications of the invention and their requirements. Various modifications to the disclosed embodiments will be readily apparent to those skilled in the art and the general principles defined herein may be applied to other embodiments and applications without departing from the spirit and scope of the present invention. Thus, the present invention is not intended to be limited to the embodiments shown, but is to be accorded the widest scope consistent with the principles and features disclosed herein.

In particular, embodiments of the invention are described below in the form of a network interface circuit (NIC) receiving communication packets formatted in accordance with certain communication protocols compatible with the Internet. One skilled in the art will recognize, however, that the present invention is not limited to communication protocols compatible with the Internet and may be readily adapted for use with other protocols and in communication devices other than a NIC.

The program environment in which a present embodiment of the invention is executed illustratively incorporates a general-purpose computer or a special purpose device such a hand-held computer. Details of such devices (e.g., processor, memory, data storage, input/output ports and display) are well known and are omitted for the sake of clarity.

It should also be understood that the techniques of the present invention might be implemented using a variety of technologies. For example, the methods described herein may be implemented in software running on a programmable microprocessor, or implemented in hardware utilizing either a combination of microprocessors or other specially designed application specific integrated circuits, programmable logic devices, or various combinations thereof. In particular, the methods described herein may be implemented by a series of computer-executable instructions residing on a storage medium such as a carrier wave, disk drive, or other computer-readable medium.

Introduction

In one embodiment of the present invention, a network interface circuit (NIC) is configured to receive and process communication packets exchanged between a host computer system and a network such as the Internet. In particular, the NIC is configured to receive and manipulate packets formatted in accordance with a protocol stack (e.g., a combination of communication protocols) supported by a network coupled to the NIC.

A protocol stack may be described with reference to the seven-layer ISO-OSI (International Standards Organization-Open Systems Interconnection) model framework. Thus, one illustrative protocol stack includes the Transport Control Protocol (TCP) at layer four, Internet Protocol (IP) at layer three and Ethernet at layer two. For purposes of discussion,

the term "Ethernet" may be used herein to refer collectively to the standardized IEEE (Institute of Electrical and Electronics Engineers) 802.3 specification as well as version two of the non-standardized form of the protocol. Where different forms of the protocol need to be distinguished, the standard form may be identified by including the "802.3" designation.

Other embodiments of the invention are configured to work with communications adhering to other protocols, both known (e.g., AppleTalk, IPX (Internetwork Packet Exchange), etc.) and unknown at the present time. One skilled in the art will recognize that the methods provided by this invention are easily adaptable for new communication protocols.

In addition, the processing of packets described below may be performed on communication devices other than a NIC. For example, a modem, switch, router or other communication port or device (e.g., serial, parallel, USB, SCSI) may be similarly configured and operated.

In embodiments of the invention described below, a NIC receives a packet from a network on behalf of a host computer system or other communication device. The NIC analyzes the packet (e.g., by retrieving certain fields from one or more of its protocol headers) and takes action to increase the efficiency with which the packet is transferred or provided to its destination entity. Equipment and methods discussed below for increasing the efficiency of processing or transferring packets received from a network may also be used for packets moving in the reverse direction (i.e., from the NIC to the network).

One technique that may be applied to incoming network traffic involves examining or parsing one or more headers of an incoming packet (e.g., headers for the layer two, three and four protocols) in order to identify the packet's source and destination entities and possibly retrieve certain other information. Using identifiers of the communicating entities as a key, data from multiple packets may be aggregated or re-assembled. Typically, a datagram sent to one destination entity from one source entity is transmitted via multiple packets. Aggregating data from multiple related packets (e.g., packets carrying data from the same datagram) thus allows a datagram to be re-assembled and collectively transferred to a host computer. The datagram may then be provided to the destination entity in a highly efficient manner. For example, rather than providing data from one packet at a time (and one byte at a time) in separate "copy" operations, a "page-flip" operation may be performed. In a page-flip, an entire memory page of data may be provided to the destination entity, possibly in exchange for an empty or unused page.

In another technique, packets received from a network are placed in a queue to await transfer to a host computer. While awaiting transfer, multiple related packets may be identified to the host computer. After being transferred, they may be processed as a group by a host processor rather than being processed serially (e.g., one at a time).

Yet another technique involves submitting a number of related packets to a single processor of a multi-processor host computer system. By distributing packets conveyed between different pairs of source and destination entities among different processors, the processing of packets through their respective protocol stacks can be distributed while still maintaining packets in their correct order.

The techniques discussed above for increasing the efficiency with which packets are processed may involve a combination of hardware and software modules located on a network interface and/or a host computer system. In one particular embodiment, a parsing module on a host computer's NIC parses header portions of packets. Illustratively, the parsing module comprises a microsequencer operating according to a set of replaceable instructions stored as micro-code. Using information extracted from the packets, multiple packets from one source entity to one destination entity may be identified. A hardware re-assembly module on the NIC may then gather the data from the multiple packets. Another hardware module on the NIC is configured to recognize related packets awaiting transfer to the host computer so that they may be processed through an appropriate protocol stack collectively, rather than serially. The re-assembled data and the packet's headers may then be provided to the host computer so that appropriate software (e.g., a device driver for the NIC) may process the headers and deliver the data to the destination entity.

Where the host computer includes multiple processors, a load distributor (which may also be implemented in hardware on the NIC) may select a processor to process the headers of the multiple packets through a protocol stack.

In another embodiment of the invention, a system is provided for randomly discarding a packet from a NIC when the NIC is saturated or nearly saturated with packets awaiting transfer to a host computer.

One Embodiment of a High Performance Network Interface Circuit

FIG. **1A** depicts NIC **100** configured in accordance with an illustrative embodiment of the invention. A brief description of the operation and interaction of the various modules of NIC **100** in this embodiment follows. Descriptions incorporating much greater detail are provided in subsequent sections.

A communication packet may be received at NIC **100** from network **102** by a medium access control (MAC) module (not shown in FIG. **1A**). The MAC module performs low-level processing of the packet such as reading the packet from the network, performing some error checking, detecting packet fragments, detecting over-sized packets, removing the layer one preamble, etc.

Input Port Processing (IPP) module **104** then receives the packet. The IPP module stores the entire packet in packet queue **116**, as received from the MAC module or network, and a portion of the packet is copied into header parser **106**. In one embodiment of the invention IPP module **104** may act as a coordinator of sorts to prepare the packet for transfer to a host computer system. In such a role, IPP module **104** may receive information concerning a packet from various modules of NIC **100** and dispatch such information to other modules.

Header parser **106** parses a header portion of the packet to retrieve various pieces of information that will be used to identify related packets (e.g., multiple packets from one same source entity for one destination entity) and that will affect subsequent processing of the packets. In the illustrated embodiment, header parser **106** communicates with flow database manager (FDBM) **108**, which manages flow database (FDB) **110**. In particular, header parser **106** submits a query to FDBM **108** to determine whether a valid communication flow (described below) exists between the source entity that sent a packet and the destination entity. The destination entity may comprise an application program, a communication module, or some other element of a host computer system that is to receive the packet.

In the illustrated embodiment of the invention, a communication flow comprises one or more datagram packets from one source entity to one destination entity. A flow may be identified by a flow key assembled from source and desti-

nation identifiers retrieved from the packet by header parser **106**. In one embodiment of the invention a flow key comprises address and/or port information for the source and destination entities from the packet's layer three (e.g., IP) and/or layer four (e.g., TCP) protocol headers.

For purposes of the illustrated embodiment of the invention, a communication flow is similar to a TCP end-to-end connection but is generally shorter in duration. In particular, in this embodiment the duration of a flow may be limited to the time needed to receive all of the packets associated with a single datagram passed from the source entity to the destination entity.

Thus, for purposes of flow management, header parser **106** passes the packet's flow key to flow database manager **108**. The header parser may also provide the flow database manager with other information concerning the packet that was retrieved from the packet (e.g., length of the packet).

Flow database manager **108** searches FDB **110** in response to a query received from header parser **106**. Illustratively, flow database **110** stores information concerning each valid communication flow involving a destination entity served by NIC **100**. Thus, FDBM **108** updates FDB **110** as necessary, depending upon the information received from header parser **106**. In addition, in this embodiment of the invention FDBM **108** associates an operation or action code with the received packet. An operation code may be used to identify whether a packet is part of a new or existing flow, whether the packet includes data or just control information, the amount of data within the packet, whether the packet data can be re-assembled with related data (e.g., other data in a datagram sent from the source entity to the destination entity), etc. FDBM **108** may use information retrieved from the packet and provided by header parser **106** to select an appropriate operation code. The packet's operation code is then passed back to the header parser, along with an index of the packet's flow within FDB **110**.

In one embodiment of the invention the combination of header parser **106**, FDBM **108** and FDB **110**, or a subset of these modules, may be known as a traffic classifier due to their role in classifying or identifying network traffic received at NIC **100**.

In the illustrated embodiment, header parser **106** also passes the packet's flow key to load distributor **112**. In a host computer system having multiple processors, load distributor **112** may determine which processor an incoming packet is to be routed to for processing through the appropriate protocol stack. For example, load distributor **112** may ensure that related packets are routed to a single processor. By sending all packets in one communication flow or end-to-end connection to a single processor, the correct ordering of packets can be enforced. Load distributor **112** may be omitted in one alternative embodiment of the invention. In another alternative embodiment, header parser **106** may also communicate directly with other modules of NIC **100** besides the load distributor and flow database manager.

Thus, after header parser **106** parses a packet FDBM **108** alters or updates FDB **110** and load distributor **112** identifies a processor in the host computer system to process the packet. After these actions, the header parser passes various information back to IPP module **104**. Illustratively, this information may include the packet's flow key, an index of the packet's flow within flow database **110**, an identifier of a processor in the host computer system, and various other data concerning the packet (e.g., its length, a length of a packet header).

Now the packet may be stored in packet queue **116**, which holds packets for manipulation by DMA (Direct Memory Access) engine **120** and transfer to a host computer. In addition to storing the packet in a packet queue, a corresponding entry for the packet is made in control queue **118** and information concerning the packet's flow may also be passed to dynamic packet batching module **122**. Control queue **118** contains related control information for each packet in packet queue **116**.

Packet batching module **122** draws upon information concerning packets in packet queue **116** to enable the batch (i.e., collective) processing of headers from multiple related packets. In one embodiment of the invention packet batching module **122** alerts the host computer to the availability of headers from related packets so that they may be processed together.

Although the processing of a packet's protocol headers is performed by a processor on a host computer system in one embodiment of the invention, in another embodiment the protocol headers may be processed by a processor located on NIC **100**. In the former embodiment, software on the host computer (e.g., a device driver for NIC **100**) can reap the advantages of additional memory and a replaceable or upgradeable processor (e.g., the memory may be supplemented and the processor may be replaced by a faster model).

During the storage of a packet in packet queue **116**, checksum generator **114** may perform a checksum operation. The checksum may be added to the packet queue as a trailer to the packet. Illustratively, checksum generator **114** generates a checksum from a portion of the packet received from network **102**. In one embodiment of the invention, a checksum is generated from the TCP portion of a packet (e.g., the TCP header and data). If a packet is not formatted according to TCP, a checksum may be generated on another portion of the packet and the result may be adjusted in later processing as necessary. For example, if the checksum calculated by checksum generator **114** was not calculated on the correct portion of the packet, the checksum may be adjusted to capture the correct portion. This adjustment may be made by software operating on a host computer system (e.g., a device driver). Checksum generator **114** may be omitted or merged into another module of NIC **100** in an alternative embodiment of the invention.

From the information obtained by header parser **106** and the flow information managed by flow database manager **108**, the host computer system served by NIC **100** in the illustrated embodiment is able to process network traffic very efficiently. For example, data portions of related packets may be re-assembled by DMA engine **120** to form aggregations that can be more efficiently manipulated. And, by assembling the data into buffers the size of a memory page, the data can be more efficiently transferred to a destination entity through "page-flipping," in which an entire memory page filled by DMA engine **120** is provided at once. One page-flip can thus take the place of multiple copy operations. Meanwhile, the header portions of the re-assembled packets may similarly be processed as a group through their appropriate protocol stack.

As already described, in another embodiment of the invention the processing of network traffic through appropriate protocol stacks may be efficiently distributed in a multi-processor host computer system. In this embodiment, load distributor **112** assigns or distributes related packets (e.g., packets in the same communication flow) to the same processor. In particular, packets having the same source and destination addresses in their layer three protocol (e.g., IP) headers and/or the same source and destination ports in their layer four protocol (e.g., TCP) headers may be sent to a single processor.

In the NIC illustrated in FIG. 1A, the processing enhancements discussed above (e.g., re-assembling data, batch processing packet headers, distributing protocol stack processing) are possible for packets received from network **102** that are formatted according to one or more pre-selected protocol stacks. In this embodiment of the invention network **102** is the Internet and NIC **100** is therefore configured to process packets using one of several protocol stacks compatible with the Internet. Packets not configured according to the pre-selected protocols are also processed, but may not receive the benefits of the full suite of processing efficiencies provided to packets meeting the pre-selected protocols.

For example, packets not matching one of the pre-selected protocol stacks may be distributed for processing in a multi-processor system on the basis of the packets' layer two (e.g., medium access control) source and destination addresses rather than their layer three or layer four addresses. Using layer two identifiers provides less granularity to the load distribution procedure, thus possibly distributing the processing of packets less evenly than if layer three/four identifiers were used.

FIG. 1B depicts one method of using NIC **100** of FIG. 1A to receive one packet from network **102** and transfer it to a host computer. State **130** is a start state, possibly characterized by the initialization or resetting of NIC **100**.

In state **132**, a packet is received by NIC **100** from network **102**. As already described, the packet may be formatted according to a variety of communication protocols. The packet may be received and initially manipulated by a MAC module before being passed to an IPP module.

In state **134**, a portion of the packet is copied and passed to header parser **106**. Header parser **106** then parses the packet to extract values from one or more of its headers and/or its data. A flow key is generated from some of the retrieved information to identify the communication flow that includes the packet. The degree or extent to which the packet is parsed may depend upon its protocols, in that the header parser may be configured to parse headers of different protocols to different depths. In particular, header parser **106** may be optimized (e.g., its operating instructions configured) for a specific set of protocols or protocol stacks. If the packet conforms to one or more of the specified protocols it may be parsed more fully than a packet that does not adhere to any of the protocols.

In state **136**, information extracted from the packet's headers is forwarded to flow database manager **108** and/or load distributor **112**. The FDBM uses the information to set up a flow in flow database **110** if one does not already exist for this communication flow. If an entry already exists for the packet's flow, it may be updated to reflect the receipt of a new flow packet. Further, FDBM **108** generates an operation code to summarize one or more characteristics or conditions of the packet. The operation code may be used by other modules of NIC **100** to handle the packet in an appropriate manner, as described in subsequent sections. The operation code is returned to the header parser, along with an index (e.g., a flow number) of the packet's flow in the flow database.

In state **138**, load distributor **112** assigns a processor number to the packet, if the host computer includes multiple processors, and returns the processor number to the header processor. Illustratively, the processor number identifies which processor is to conduct the packet through its protocol stack on the host computer. State **138** may be omitted in an alternative embodiment of the invention, particularly if the host computer consists of only a single processor.

In state **140**, the packet is stored in packet queue **116**. As the contents of the packet are placed into the packet queue, checksum generator **114** may compute a checksum. The checksum generator may be informed by IPP module **104** as to which portion of the packet to compute the checksum on. The computed checksum is added to the packet queue as a trailer to the packet. In one embodiment of the invention, the packet is stored in the packet queue at substantially the same time that a copy of a header portion of the packet is provided to header parser **106**.

Also in state **140**, control information for the packet is stored in control queue **118** and information concerning the packet's flow (e.g., flow number, flow key) may be provided to dynamic packet batching module **122**.

In state **142**, NIC **100** determines whether the packet is ready to be transferred to host computer memory. Until it is ready to be transferred, the illustrated procedure waits.

When the packet is ready to be transferred (e.g., the packet is at the head of the packet queue or the host computer receives the packet ahead of this packet in the packet queue), in state **144** dynamic packet batching module **122** determines whether a related packet will soon be transferred. If so, then when the present packet is transferred to host memory the host computer is alerted that a related packet will soon follow. The host computer may then process the packets (e.g., through their protocol stack) as a group.

In state **146**, the packet is transferred (e.g., via a direct memory access operation) to host computer memory. And, in state **148**, the host computer is notified that the packet was transferred. The illustrated procedure then ends at state **150**.

One skilled in the art of computer systems and networking will recognize that the procedure described above is just one method of employing the modules of NIC **100** to receive a single packet from a network and transfer it to a host computer system. Other suitable methods are also contemplated within the scope of the invention.

An Illustrative Packet

FIG. **2** is a diagram of an illustrative packet received by NIC **100** from network **102**. Packet **200** comprises data portion **202** and header portion **204**, and may also contain trailer portion **206**. Depending upon the network environment traversed by packet **200**, its maximum size (e.g., its maximum transfer unit or MTU) may be limited.

In the illustrated embodiment, data portion **202** comprises data being provided to a destination or receiving entity within a computer system (e.g., user, application program, operating system) or a communication subsystem of the computer. Header portion **204** comprises one or more headers prefixed to the data portion by the source or originating entity or a computer system comprising the source entity. Each header normally corresponds to a different communication protocol.

In a typical network environment, such as the Internet, individual headers within header portion **204** are attached (e.g., prepended) as the packet is processed through different layers of a protocol stack (e.g., a set of protocols for communicating between entities) on the transmitting computer system. For example, FIG. **2** depicts protocol headers **210**, **212**, **214** and **216**, corresponding to layers one through four, respectively, of a suitable protocol stack. Each protocol header contains information to be used by the receiving computer system as the packet is received and processed through the protocol stack. Ultimately, each protocol header is removed and data portion **202** is retrieved.

As described in other sections, in one embodiment of the invention a system and method are provided for parsing

packet **200** to retrieve various bits of information. In this embodiment, packet **200** is parsed in order to identify the beginning of data portion **202** and to retrieve one or more values for fields within header portion **204**. Illustratively, however, layer one protocol header or preamble **210** corresponds to a hardware-level specification related to the coding of individual bits. Layer one protocols are generally only needed for the physical process of sending or receiving the packet across a conductor. Thus, in this embodiment of the invention layer one preamble **210** is stripped from packet **200** shortly after being received by NIC **100** and is therefore not parsed.

The extent to which header portion **204** is parsed may depend upon how many, if any, of the protocols represented in the header portion match a set of pre-selected protocols. For example, the parsing procedure may be abbreviated or aborted once it is determined that one of the packet's headers corresponds to an unsupported protocol.

In particular, in one embodiment of the invention NIC **100** is configured primarily for Internet traffic. Thus, in this embodiment packet **200** is extensively parsed only when the layer two protocol is Ethernet (either traditional Ethernet or 802.3 Ethernet, with or without tagging for Virtual Local Area Networks), the layer three protocol is IP (Internet Protocol) and the layer four protocol is TCP (Transport Control Protocol). Packets adhering to other protocols may be parsed to some (e.g., lesser) extent. NIC **100** may, however, be configured to support and parse virtually any communication protocol's header. Illustratively, the protocol headers that are parsed, and the extent to which they are parsed, are determined by the configuration of a set of instructions for operating header parser **106**.

As described above, the protocols corresponding to headers **212**, **214** and **216** depend upon the network environment in which a packet is sent. The protocols also depend upon the communicating entities. For example, a packet received by a network interface may be a control packet exchanged between the medium access controllers for the source and destination computer systems. In this case, the packet would be likely to include minimal or no data, and may not include layer three protocol header **214** or layer four protocol header **216**. Control packets are typically used for various purposes related to the management of individual connections.

Another communication flow or connection could involve two application programs. In this case, a packet may include headers **212**, **214** and **216**, as shown in FIG. **2**, and may also include additional headers related to higher layers of a protocol stack (e.g., session, presentation and application layers in the ISO-OSI model). In addition, some applications may include headers or header-like information within data portion **202**. For example, for a Network File System (NFS) application, data portion **202** may include NFS headers related to individual NFS datagrams. A datagram may be defined as a collection of data sent from one entity to another, and may comprise data transmitted in multiple packets. In other words, the amount of data constituting a datagram may be greater than the amount of data that can be included in one packet.

One skilled in the art will appreciate that the methods for parsing a packet that are described in the following section are readily adaptable for packets formatted in accordance with virtually any communication protocol.

One Embodiment of a Header Parser

FIG. **3** depicts header parser **106** of FIG. **1A** in accordance with a present embodiment of the invention. Illustratively, header parser **106** comprises header memory **302** and parser **304**, and parser **304** comprises instruction memory **306**. Although depicted as distinct modules in FIG. **3**, in an alternative embodiment of the invention header memory **302** and instruction memory **306** are contiguous.

In the illustrated embodiment, parser **304** parses a header stored in header memory **302** according to instructions stored in instruction memory **306**. The instructions are designed for the parsing of particular protocols or a particular protocol stack, as discussed above. In one embodiment of the invention, instruction memory **306** is modifiable (e.g., the memory is implemented as RAM, EPROM, EEPROM or the like), so that new or modified parsing instructions may be downloaded or otherwise installed. Instructions for parsing a packet are further discussed in the following section.

In FIG. **3**, a header portion of a packet stored in IPP module **104** (shown in FIG. **1A**) is copied into header memory **302**. Illustratively, a specific number of bytes (e.g., **114**) at the beginning of the packet are copied. In an alternative embodiment of the invention, the portion of a packet that is copied may be of a different size. The particular amount of a packet copied into header memory **302** should be enough to capture one or more protocol headers, or at least enough information (e.g., whether included in a header or data portion of the packet) to retrieve the information described below. The header portion stored in header memory **302** may not include the layer one header, which may be removed prior to or in conjunction with the packet being processed by IPP module **104**.

After a header portion of the packet is stored in header memory **302**, parser **304** parses the header portion according to the instructions stored in instruction memory **306**. In the presently described embodiment, instructions for operating parser **304** apply the formats of selected protocols to step through the contents of header memory **302** and retrieve specific information. In particular, specifications of communication protocols are well known and widely available. Thus, a protocol header may be traversed byte by byte or some other fashion by referring to the protocol specifications. In a present embodiment of the invention the parsing algorithm is dynamic, with information retrieved from one field of a header often altering the manner in which another part is parsed.

For example, it is known that the Type field of a packet adhering to the traditional, form of Ethernet (e.g., version two) begins at the thirteenth byte of the (layer two) header. By comparison, the Type field of a packet following the IEEE 802.3 version of Ethernet begins at the twenty-first byte of the header. The Type field is in yet other locations if the packet forms part of a Virtual Local Area Network (VLAN) communication (which illustratively involves tagging or encapsulating an Ethernet header). Thus, in a present embodiment of the invention, the values in certain fields are retrieved and tested in order to ensure that the information needed from a header is drawn from the correct portion of the header. Details concerning the form of a VLAN packet may be found in specifications for the IEEE 802.3p and IEEE 802.3q forms of the Ethernet protocol.

The operation of header parser **106** also depends upon other differences between protocols, such as whether the packet uses version four or version six of the Internet Protocol, etc. Specifications for versions four and six of IP may be located in IETF (Internet Engineering Task Force) RFCs (Request for Comment) **791** and **2460**, respectively.

The more protocols that are "known" by parser **304**, the more protocols a packet may be tested for, and the more complicated the parsing of a packet's header portion may become. One skilled in the art will appreciate that the protocols that may be parsed by parser **304** are limited only

by the instructions according to which it operates. Thus, by augmenting or replacing the parsing instructions stored in instruction memory **306**, virtually all known protocols may be handled by header parser **106** and virtually any information may be retrieved from a packet's headers.

If, of course, a packet header does not conform to an expected or suspected protocol, the parsing operation may be terminated. In this case, the packet may not be suitable for one more of the efficiency enhancements offered by NIC **100** (e.g., data re-assembly, packet batching, load distribution).

Illustratively, the information retrieved from a packet's headers is used by other portions of NIC **100** when processing that packet. For example, as a result of the packet parsing performed by parser **304** a flow key is generated to identify the communication flow or communication connection that comprises the packet. Illustratively, the flow key is assembled by concatenating one or more addresses corresponding to one or more of the communicating entities. In a present embodiment, a flow key is formed from a combination of the source and destination addresses drawn from the IP header and the source and destination ports taken from the TCP header. Other indicia of the communicating entities may be used, such as the Ethernet source and destination addresses (drawn from the layer two header), NFS file handles or source and destination identifiers for other application datagrams drawn from the data portion of the packet.

One skilled in the art will appreciate that the communicating entities may be identified with greater resolution by using indicia drawn from the higher layers of the protocol stack associated with a packet. Thus, a combination of IP and TCP indicia may identify the entities with greater particularity than layer two information.

Besides a flow key, parser **304** also generates a control or status indicator to summarize additional information concerning the packet. In one embodiment of the invention a control indicator includes a sequence number (e.g., TCP sequence number drawn from a TCP header) to ensure the correct ordering of packets when re-assembling their data. The control indicator may also reveal whether certain flags in the packet's headers are set or cleared, whether the packet contains any data, and, if the packet contains data, whether the data exceeds a certain size. Other data are also suitable for inclusion in the control indicator, limited only by the information that is available in the portion of the packet parsed by parser **304**.

In one embodiment of the invention, header parser **106** provides the flow key and all or a portion of the control indicator to flow database manager **108**. As discussed in a following section, FDBM **108** manages a database or other data structure containing information relevant to communication flows passing through NIC **100**.

In other embodiments of the invention, parser **304** produces additional information derived from the header of a packet for use by other modules of NIC **100**. For example, header parser **106** may report the offset, from the beginning of the packet or from some other point, of the data or payload portion of a packet received from a network. As described above, the data portion of a packet typically follows the header portion and may be followed by a trailer portion. Other data that header parser **106** may report include the location in the packet at which a checksum operation should begin, the location in the packet at which the layer three and/or layer four headers begin, diagnostic data, payload information, etc. The term "payload" is often used to refer to the data portion of a packet. In particular, in one embodiment of the invention header parser **106** provides a payload offset and payload size to control queue **118**.

In appropriate circumstances, header parser **106** may also report (e.g., to IPP module **104** and/or control queue **118**) that the packet is not formatted in accordance with the protocols that parser **304** is configured to manipulate. This report may take the form of a signal (e.g., the No_Assist signal described below), alert, flag or other indicator. The signal may be raised or issued whenever the packet is found to reflect a protocol other than the pre-selected protocols that are compatible with the processing enhancements described above (e.g., data re-assembly, batch processing of packet headers, load distribution). For example, in one embodiment of the invention parser **304** may be configured to parse and efficiently process packets using TCP at layer four, IP at layer three and Ethernet at layer two. In this embodiment, an IPX (Internetwork Packet Exchange) packet would not be considered compatible and IPX packets therefore would not be gathered for data re-assembly and batch processing.

At the conclusion of parsing in one embodiment of the invention, the various pieces of information described above are disseminated to appropriate modules of NIC **100**. After this (and as described in a following section), flow database manager **108** determines whether an active flow is associated with the flow key derived from the packet and sets an operation code to be used in subsequent processing. In addition, IPP module **104** transmits the packet to packet queue **116**. IPP module **104** may also receive some of the information extracted by header parser **106**, and pass it to another module of NIC **100**.

In the embodiment of the invention depicted in FIG. **3**, an entire header portion of a received packet to be parsed is copied and then parsed in one evolution, after which the header parser turns its attention to another packet. However, in an alternative embodiment multiple copy and/or parsing operations may be performed on a single packet. In particular, an initial header portion of the packet may be copied into and parsed by header parser **106** in a first evolution, after which another header portion may be copied into header parser **106** and parsed in a second evolution. A header portion in one evolution may partially or completely overlap the header portion of another evolution. In this manner, extensive headers may be parsed even if header memory **302** is of limited size. Similarly, it may require more than one operation to load a full set of instructions for parsing a packet into instruction memory **306**. Illustratively, a first portion of the instructions may be loaded and executed, after which other instructions are loaded.

With reference now to FIGS. **4A–4B**, a flow chart is presented to illustrate one method by which a header parser may parse a header portion of a packet received at a network interface circuit from a network. In this implementation, the header parser is configured, or optimized, for parsing packets conforming to a set of pre-selected protocols (or protocol stacks). For packets meeting these criteria, various information is retrieved from the header portion to assist in the re-assembly of the data portions of related packets (e.g., packets comprising data from a single datagram). Other enhanced features of the network interface circuit may also be enabled.

The information generated by the header parser includes, in particular, a flow key with which to identify the communication flow or communication connection that comprises the received packet. In one embodiment of the invention, data from packets having the same flow key may be identified and re-assembled to form a datagram. In addition, headers of packets having the same flow key may be processed collectively through their protocol stack (e.g., rather than serially).

In another embodiment of the invention, information retrieved by the header parser is also used to distribute the processing of network traffic received from a network. For example, multiple packets having the same flow key may be submitted to a single processor of a multi-processor host computer system.

In the method illustrated in FIGS. **4A–4B**, the set of pre-selected protocols corresponds to communication protocols frequently transmitted via the Internet. In particular, the set of protocols that may be extensively parsed in this method include the following. At layer two: Ethernet (traditional version), 802.3 Ethernet, Ethernet VLAN (Virtual Local Area Network) and 802.3 Ethernet VLAN. At layer three: IPv4 (with no options) and IPv6 (with no options). Finally, at layer four, only TCP protocol headers (with or without options) are parsed in the illustrated method. Header parsers in alternative embodiments of the invention parse packets formatted through other protocol stacks. In particular, a NIC may be configured in accordance with the most common protocol stacks in use on a given network, which may or may not include the protocols compatible with the header parser method illustrated in FIGS. **4A–4B**.

As described below, a received packet that does not correspond to the protocols parsed by a given method may be flagged and the parsing algorithm terminated for that packet. Because the protocols under which a packet has been formatted can only be determined, in the present method, by examining certain header field values, the determination that a packet does not conform to the selected set of protocols may be made at virtually any time during the procedure. Thus, the illustrated parsing method has as one goal the identification of packets not meeting the formatting criteria for re-assembly of data.

Various protocol header fields appearing in headers for the selected protocols are discussed below. Communication protocols that may be compatible with an embodiment of the present invention (e.g., protocols that may be parsed by a header parser) are well known to persons skilled in the art and are described with great particularity in a number of references. They therefore need not be visited in minute detail herein. In addition, the illustrated method of parsing a header portion of a packet for the selected protocols is merely one method of gathering the information described below. Other parsing procedures capable of doing so are equally suitable.

In a present embodiment of the invention, the illustrated procedure is implemented as a combination of hardware and software. For example, updateable micro-code instructions for performing the procedure may be executed by a microsequencer. Alternatively, such instructions may be fixed (e.g., stored in read-only memory) or may be executed by a processor or microprocessor.

In FIGS. **4A–4B**, state **400** is a start state during which a packet is received by NIC **100** (shown in FIG. **1A**) and initial processing is performed. NIC **100** is coupled to the Internet for purposes of this procedure. Initial processing may include basic error checking and the removal of the layer one preamble. After initial processing, the packet is held by IPP module **104** (also shown in FIG. **1A**). In one embodiment of the invention, state **400** comprises a logical loop in which the header parser remains in an idle or wait state until a packet is received.

In state **402**, a header portion of the packet is copied into memory (e.g., header memory **302** of FIG. **3**). In a present embodiment of the invention a predetermined number of bytes at the beginning (e.g., 114 bytes) of the packet are copied. Packet portions of different sizes are copied in alternative embodiments of the invention, the sizes of which are guided by the goal of copying enough of the packet to capture and/or identify the necessary header information. Illustratively, the full packet is retained by IPP module **104** while the following parsing operations are performed, although the packet may, alternatively, be stored in packet queue **116** prior to the completion of parsing.

Also in state **402**, a pointer to be used in parsing the packet may be initialized. Because the layer one preamble was removed, the header portion copied to memory should begin with the layer two protocol header. Illustratively, therefore, the pointer is initially set to point to the twelfth byte of the layer two protocol header and the two-byte value at the pointer position is read. As one skilled in the art will recognize, these two bytes may be part of a number of different fields, depending upon which protocol constitutes layer two of the packet's protocol stack. For example, these two bytes may comprise the Type field of a traditional Ethernet header, the Length field of an 802.3 Ethernet header or the TPID (Tag Protocol IDentifier) field of a VLAN-tagged header.

In state **404**, a first examination is made of the layer two header to determine if it comprises a VLAN-tagged layer two protocol header. Illustratively, this determination depends upon whether the two bytes at the pointer position store the hexadecimal value 8100. If so, the pointer is probably located at the TPID field of a VLAN-tagged header. If not a VLAN header, the procedure proceeds to state **408**.

If, however, the layer two header is a VLAN-tagged header, in state **406** the CFI (Canonical Format Indicator) bit is examined. If the CFI bit is set (e.g., equal to one), the illustrated procedure jumps to state **430**, after which it exits. In this embodiment of the invention the CFI bit, when set, indicates that the format of the packet is not compatible with (i.e., does not comply with) the pre-selected protocols (e.g., the layer two protocol is not Ethernet or 802.3 Ethernet). If the CFI bit is clear (e.g., equal to zero), the pointer is incremented (e.g., by four bytes) to position it at the next field that must be examined.

In state **408**, the layer two header is further tested. Although it is now known whether this is or is not a VLAN-tagged header, depending upon whether state **408** was reached through state **406** or directly from state **404**, respectively, the header may reflect either the traditional Ethernet format or the 802.3 Ethernet format. At the beginning of state **408**, the pointer is either at the twelfth or sixteenth byte of the header, either of which may correspond to a Length field or a Type field. In particular, if the two-byte value at the position identified by the pointer is less than 0600 (hexadecimal), then the packet corresponds to 802.3 Ethernet and the pointer is understood to identify a Length field. Otherwise, the packet is a traditional (e.g., version two) Ethernet packet and the pointer identifies a Type field.

If the layer two protocol is 802.3 Ethernet, the procedure continues at state **410**. If the layer two protocol is traditional Ethernet, the Type field is tested for the hexadecimal values of 0800 and 08DD. If the tested field has one of these values, then it has also been determined that the packet's layer three protocol is the Internet Protocol. In this case the illustrated procedure continues at state **412**. Lastly, if the field is a Type field having a value other than 0800 or 86DD (hexadecimal), then the packet's layer three protocol does not match the pre-selected protocols according to which the header parser was configured. Therefore, the procedure continues at state **430** and then ends.

In one embodiment of the invention the packet is examined in state **408** to determine if it is a jumbo Ethernet frame. This determination would likely be made prior to deciding whether the layer two header conforms to Ethernet or 802.3 Ethernet. Illustratively, the jumbo frame determination may be made based on the size of the packet, which may be reported by IPP module **104** or a MAC module. If the packet is a jumbo frame, the procedure may continue at state **410**; otherwise, it may resume at state **412**.

In state **410**, the procedure verifies that the layer two protocol is 802.3 Ethernet with LLC SNAP encapsulation. In particular, the pointer is advanced (e.g., by two bytes) and the six-byte value following the Length field in the layer two header is retrieved and examined. If the header is an 802.3 Ethernet header, the field is the LLC_SNAP field and should have a value of AAAA03000000 (hexadecimal). The original specification for an LLC SNAP header may be found in the specification for IEEE 802.2. If the value in the packet's LLC_SNAP field matches the expected value the pointer is incremented another six bytes, the two-byte 802.3 Ethernet Type field is read and the procedure continues at state **412**. If the values do not match, then the packet does not conform to the specified protocols and the procedure enters state **430** and then ends.

In state **412**, the pointer is advanced (e.g., another two bytes) to locate the beginning of the layer three protocol header. This pointer position may be saved for later use in quickly identifying the beginning of this header. The packet is now known to conform to an accepted layer two protocol (e.g., traditional Ethernet, Ethernet with VLAN tagging, or 802.3 Ethernet with LLC SNAP) and is now checked to ensure that the packet's layer three protocol is IP. As discussed above, in the illustrated embodiment only packets conforming to the IP protocol are extensively processed by the header parser.

Illustratively, if the value of the Type field in the layer two header (retrieved in state **402** or state **410**) is 0800 (hexadecimal), the layer three protocol is expected to be IP, version four. If the value is 86DD (hexadecimal), the layer three protocol is expected to be IP, version six. Thus, the Type field is tested in state **412** and the procedure continues at state **414** or state **418**, depending upon whether the hexadecimal value is 0800 or 86DD, respectively.

In state **414**, the layer three header's conformity with version four of IP is verified. In one embodiment of the invention the Version field of the layer three header is tested to ensure that it contains the hexadecimal value 4, corresponding to version four of IP. If in state **414** the layer three header is confirmed to be IP version four, the procedure continues at state **416**; otherwise, the procedure proceeds to state **430** and then ends at state **432**.

In state **416**, various pieces of information from the IP header are saved. This information may include the IHL (IP Header Length), Total Length, Protocol and/or Fragment Offset fields. The IP source address and the IP destination addresses may also be stored. The source and destination address values are each four bytes long in version four of IP. These addresses are used, as described above, to generate a flow key that identifies the communication flow in which this packet was sent. The Total Length field stores the size of the IP segment of this packet, which illustratively comprises the IP header, the TCP header and the packet's data portion. The TCP segment size of the packet (e.g., the size of the TCP header plus the size of the data portion of the packet) may be calculated by subtracting twenty bytes (the size of the IP version four header) from the Total Length value. After state **416**, the illustrated procedure advances to state **422**.

In state **418**, the layer three header's conformity with version six of IP is verified by testing the Version field for the hexadecimal value 6. If the Version field does not contain this value, the illustrated procedure proceeds to state **430**.

In state **420**, the values of the Payload Length (e.g., the size of the TCP segment) and Next Header field are saved, plus the IP source and destination addresses. Source and destination addresses are each sixteen bytes long in version six of IP.

In state **422** of the illustrated procedure, it is determined whether the IP header (either version four or version six) indicates that the layer four header is TCP. Illustratively, the Protocol field of a version four IP header is tested while the Next Header field of a version six header is tested. In either case, the value should be 6 (hexadecimal). The pointer is then incremented as necessary (e.g., twenty bytes for IP version four, forty bytes for IP version six) to reach the beginning of the TCP header. If it is determined in state **422** that the layer four header is not TCP, the procedure advances to state **430** and ends at end state **432**.

In one embodiment of the invention, other fields of a version four IP header may be tested in state **422** to ensure that the packet meets the criteria for enhanced processing by NIC **100**. For example, an IHL field value other than 5 (hexadecimal) indicates that IP options are set for this packet, in which case the parsing operation is aborted. A fragmentation field value other than zero indicates that the IP segment of the packet is a fragment, in which case parsing is also aborted. In either case, the procedure jumps to state **430** and then ends at end state **432**.

In state **424**, the packet's TCP header is parsed and various data are collected from it. In particular, the TCP source port and destination port values are saved. The TCP sequence number, which is used to ensure the correct re-assembly of data from multiple packets, is also saved. Further, the values of several components of the Flags field—illustratively, the URG (urgent), PSH (push), RST (reset), SYN (synch) and FIN (finish) bits—are saved. As will be seen in a later section, in one embodiment of the invention these flags signal various actions to be performed or statuses to be considered in the handling of the packet.

Other signals or statuses may be generated in state **424** to reflect information retrieved from the TCP header. For example, the point from which a checksum operation is to begin may be saved (illustratively, the beginning of the TCP header); the ending point of a checksum operation may also be saved (illustratively, the end of the data portion of the packet). An offset to the data portion of the packet may be identified by multiplying the value of the Header Length field of the TCP header by four. The size of the data portion may then be calculated by subtracting the offset to the data portion from the size of the entire TCP segment.

In state **426**, a flow key is assembled by concatenating the IP source and destination addresses and the TCP source and destination ports. As already described, the flow key may be used to identify a communication flow or communication connection, and may be used by other modules of NIC **100** to process network traffic more efficiently. Although the sizes of the source and destination addresses differ between IP versions four and six (e.g., four bytes each versus sixteen bytes each, respectively), in the presently described embodiment of the invention all flow keys are of uniform size. In particular, in this embodiment they are thirty-six bytes long, including the two-byte TCP source port and two-byte TCP destination port. Flow keys generated from IP, version four, packet headers are padded as necessary (e.g., with twenty-four clear bytes) to fill the flow key's allocated space.

In state **428**, a control or status indicator is assembled to provide various information to one or more modules of NIC **100**. In one embodiment of the invention a control indicator includes the packet's TCP sequence number, a flag or identifier (e.g., one or more bits) indicating whether the packet contains data (e.g., whether the TCP payload size is greater than zero), a flag indicating whether the data portion of the packet exceeds a pre-determined size, and a flag indicating whether certain entries in the TCP Flags field are equivalent to pre-determined values. The latter flag may, for example, be used to inform another module of NIC **100** that components of the Flags field do or do not have a particular configuration. After state **428**, the illustrated procedure ends with state **432**.

State **430** may be entered at several different points of the illustrated procedure. This state is entered, for example, when it is determined that a header portion that is being parsed by a header parser does not conform to the pre-selected protocol stacks identified above. As a result, much of the information described above is not retrieved. A practical consequence of the inability to retrieve this information is that it then cannot be provided to other modules of NIC **100** and the enhanced processing described above and in following sections may not be performed for this packet. In particular, and as discussed previously, in a present embodiment of the invention one or more enhanced operations may be performed on parsed packets to increase the efficiency with which they are processed. Illustrative operations that may be applied include the re-assembly of data from related packets (e.g., packets containing data from a single datagram), batch processing of packet headers through a protocol stack, load distribution or load sharing of protocol stack processing, efficient transfer of packet data to a destination entity, etc.

In the illustrated procedure, in state **430** a flag or signal (illustratively termed No_Assist) is set or cleared to indicate that the packet presently held by IPP module **104** (e.g., which was just processed by the header parser) does not conform to any of the pre-selected protocol stacks. This flag or signal may be relied upon by another module of NIC **100** when deciding whether to perform one of the enhanced operations.

Another flag or signal may be set or cleared in state **430** to initialize a checksum parameter indicating that a checksum operation, if performed, should start at the beginning of the packet (e.g., with no offset into the packet). Illustratively, incompatible packets cannot be parsed to determine a more appropriate point from which to begin the checksum operation. After state **430**, the procedure ends with end state **432**.

After parsing a packet, the header parser may distribute information generated from the packet to one or more modules of NIC **100**. For example, in one embodiment of the invention the flow key is provided to flow database manager **108**, load distributor **112** and one or both of control queue **118** and packet queue **116**. Illustratively, the control indicator is provided to flow database manager **108**. This and other control information, such as TCP payload size, TCP payload offset and the No_Assist signal may be returned to IPP module **104** and provided to control queue **118**. Yet additional control and/or diagnostic information, such as offsets to the layer three and/or layer four headers, may be provided to IPP module **104**, packet queue **116** and/or control queue **118**.

Checksum information (e.g., a starting point and either an ending point or other means of identifying a portion of the packet from which to compute a checksum) may be provided to checksum generator **114**.

As discussed in a following section, although a received packet is parsed on NIC **100** (e.g., by header parser **106**), the packets are still processed (e.g., through their respective protocol stacks) on the host computer system in the illustrated embodiment of the invention. However, after parsing a packet in an alternative embodiment of the invention, NIC **100** also performs one or more subsequent processing steps. For example, NIC **100** may include one or more protocol processors for processing one or more of the packet's protocol headers.

Dynamic Header Parsing Instructions in One Embodiment of the Invention

In one embodiment of the present invention, header parser **106** parses a packet received from a network according to a dynamic sequence of instructions. The instructions may be stored in the header parser's instruction memory (e.g., RAM, SRAM, DRAM, flash) that is re-programmable or that can otherwise be updated with new or additional instructions. In one embodiment of the invention software operating on a host computer (e.g., a device driver) may download a set of parsing instructions for storage in the header parser memory.

The number and format of instructions stored in a header parser's instruction memory may be tailored to one or more specific protocols or protocol stacks. An instruction set configured for one collection of protocols, or a program constructed from that instruction set, may therefore be updated or replaced by a different instruction set or program. For packets received at the network interface that are formatted in accordance with the selected protocols (e.g., "compatible" packets), as determined by analyzing or parsing the packets, various enhancements in the handling of network traffic become possible as described in the following sections. In particular, packets from one datagram that are configured according to a selected protocol may be re-assembled for efficient transfer in a host computer. In addition, header portions of such packets may be processed collectively rather than serially. And, the processing of packets from different datagrams by a multi-processor host computer may be shared or distributed among the processors. Therefore, one objective of a dynamic header parsing operation is to identify a protocol according to which a received packet has been formatted or determine whether a packet header conforms to a particular protocol.

FIG. **23**, discussed in detail shortly, presents an illustrative series of instructions for parsing the layer two, three and four headers of a packet to determine if they are Ethernet, IP and TCP, respectively. The illustrated instructions comprise one possible program or microcode for performing a parsing operation. As one skilled in the art will recognize, after a particular set of parsing instructions is loaded into a parser memory, a number of different programs may be assembled. FIG. **23** thus presents merely one of a number of programs that may be generated from the stored instructions. The instructions presented in FIG. **23** may be performed or executed by a microsequencer, a processor, a microprocessor or other similar module located within a network interface circuit.

In particular, other instruction sets and other programs may be derived for different communication protocols, and may be expanded to other layers of a protocol stack. For example, a set of instructions could be generated for parsing NFS (Network File System) packets. Illustratively, these instructions would be configured to parse layer five and six headers to determine if they are Remote Procedure Call (RPC) and External Data Representation (XDR), respectively. Other instructions could be configured to parse a

portion of the packet's data (which may be considered layer seven). An NFS header may be considered a part of a packet's layer six protocol header or part of the packet's data.

One type of instruction executed by a microsequencer may be designed to locate a particular field of a packet (e.g., at a specific offset within the packet) and compare the value stored at that offset to a value associated with that field in a particular communication protocol. For example, one instruction may require the microsequencer to examine a value in a packet header at an offset that would correspond to a Type field of an Ethernet header. By comparing the value actually stored in the packet with the value expected for the protocol, the microsequencer can determine if the packet appears to conform to the Ethernet protocol. Illustratively, the next instruction applied in the parsing program depends upon whether the previous comparison was successful. Thus, the particular instructions applied by the microsequencer, and the sequence in which applied, depend upon which protocols are represented by the packet's headers.

The microsequencer may test one or more field values within each header included in a packet. The more fields that are tested and that are found to comport with the format of a known protocol, the greater the certainty that the packet conforms to that protocol. As one skilled in the art will appreciate, one communication protocol may be quite different than another protocol, thus requiring examination of different parts of packet headers for different protocols. Illustratively, the parsing of one packet may end in the event of an error or because it was determined that the packet being parsed does or does not conform to the protocol(s) the instructions are designed for.

Each instruction in FIG. **23** may be identified by a number and/or a name. A particular instruction may perform a variety of tasks other than comparing a header field to an expected value. An instruction may, for example, call another instruction to examine another portion of a packet header, initialize, load or configure a register or other data structure, prepare for the arrival and parsing of another packet, etc. In particular, a register or other storage structure may be configured in anticipation of an operation that is performed in the network interface after the packet is parsed. For example, a program instruction in FIG. **23** may identify an output operation that may or may not be performed, depending upon the success or failure of the comparison of a value extracted from a packet with an expected value. An output operation may store a value in a register, configure a register (e.g., load an argument or operator) for a post-parsing operation, clear a register to await a new packet, etc.

A pointer may be employed to identify an offset into a packet being parsed. In one embodiment, such a pointer is initially located at the beginning of the layer two protocol header. In another embodiment, however, the pointer is situated at a specific location within a particular header (e.g., immediately following the layer two destination and/or source addresses) when parsing commences. Illustratively, the pointer is incremented through the packet as the parsing procedure executes. In one alternative embodiment, however, offsets to areas of interest in the packet may be computed from one or more known or computed locations.

In the parsing program depicted in FIG. **23**, a header is navigated (e.g., the pointer is advanced) in increments of two bytes (e.g., sixteen-bit words). In addition, where a particular field of a header is compared to a known or expected value, up to two bytes are extracted at a time from the field. Further, when a value or header field is copied for storage in a register or other data structure, the amount of data that may be copied in one operation may be expressed in multiples of two-byte units or in other units altogether (e.g., individual bytes). This unit of measurement (e.g., two bytes) may be increased or decreased in an alternative embodiment of the invention. Altering the unit of measurement may alter the precision with which a header can be parsed or a header value can be extracted.

In the embodiment of the invention illustrated in FIG. **23**, a set of instructions loaded into the header parser's instruction memory comprises a number of possible operations to be performed while testing a packet for compatibility with selected protocols. Program **2300** is generated from the instruction set. Program **2300** is thus merely one possible program, microcode or sequence of instructions that can be formed from the available instruction set.

In this embodiment, the loaded instruction set enables the following sixteen operations that may be performed on a packet that is being parsed. Specific implementations of these operations in program **2300** are discussed in additional detail below. These instructions will be understood to be illustrative in nature and do not limit the composition of instruction sets in other embodiments of the invention. In addition, any subset of these operations may be employed in a particular parsing program or microcode. Further, multiple instructions may employ the same operation and have different effects.

A CLR_REG operation allows the selective initialization of registers or other data structures used in program **2300** and, possibly, data structures used in functions performed after a packet is parsed. Initialization may comprise storing the value zero. A number of illustrative registers that may be initialized by a CLR_REG operation are identified in the remaining operations.

A LD_FID operation copies a variable amount of data from a particular offset within the packet into a register configured to store a packet's flow key or other flow identifier. This register may be termed a FLOWID register. The effect of an LD_FID operation is cumulative. In other words, each time it is invoked for one packet the generated data is appended to the flow key data stored previously.

A LD_SEQ operation copies a variable amount of data from a particular offset within the packet into a register configured to store a packet's sequence number (e.g., a TCP sequence number). This register may be assigned the label SEQNO. This operation is also cumulative—the second and subsequent invocations of this operation for the packet cause the identified data to be appended to data stored previously.

A LD_CTL operation loads a value from a specified offset in the packet into a CONTROL register. The CONTROL register may comprise a control indicator discussed in a previous section for identifying whether a packet is suitable for data re-assembly, packet batching, load distribution or other enhanced functions of NIC **100**. In particular, a control indicator may indicate whether a No_Assist flag should be raised for the packet, whether the packet includes any data, whether the amount of packet data is larger than a predetermined threshold, etc. Thus, the value loaded into a CONTROL register in a LD_CTL operation may affect the post-parsing handling of the packet.

A LD_SAP operation loads a value into the CONTROL register from a variable offset within the packet. The loaded value may comprise the packet's ethertype. In one option that may be associated with a LD_SAP operation, the offset of the packet's layer three header may also be stored in the CONTROL register or elsewhere. As one skilled in the art will recognize, a packet's layer three header may immedi-

ately follow its layer two ethertype field if the packet conforms to the Ethernet and IP protocols.

A LD_R1 operation may be used to load a value into a temporary register (e.g., named R1) from a variable offset within the packet. A temporary register may be used for a variety of tasks, such as accumulating values to determine the length of a header or other portion of the packet. A LD_R1 operation may also cause a value from another variable offset to be stored in a second temporary register (e.g., named R2). The values stored in the R1 and/or R2 registers during the parsing of a packet may or may not be cumulative.

A LD_L3 operation may load a value from the packet into a register configured to store the location of the packet's layer three header. This register may be named L3OFFSET. In one optional method of invoking this operation, it may be used to load a fixed value into the L3OFFSET register. As another option, the LD_L3 operation may add a value stored in a temporary register (e.g., R1) to the value being stored in the L3OFFSET register.

A LD_SUM operation stores the starting point within the packet from which a checksum should be calculated. The register in which this value is stored may be named a CSUMSTART register. In one alternative invocation of this operation, a fixed or predetermined value is stored in the register. As another option, the LD_SUM operation may add a value stored in a temporary register (e.g., R1) to the value being stored in the CSUMSTART register.

A LD_HDR operation loads a value into a register configured to store the location within the packet at which the header portion may be split. The value that is stored may, for example, be used during the transfer of the packet to the host computer to store a data portion of the packet in a separate location than the header portion. The loaded value may thus identify the beginning of the packet data or the beginning of a particular header. In one invocation of a LD_HDR operation, the stored value may be computed from a present position of a parsing pointer described above. In another invocation, a fixed or predetermined value may be store. As yet another alternative, a value stored in a temporary register (e.g., R1) and/or a constant may be added to the loaded value.

A LD_LEN operation stores the length of the packet's payload into a register (e.g., a PAYLOADLEN register).

An IM_FID operation appends or adds a fixed or predetermined value to the existing contents of the FLOWID register described above.

An IM_SEQ operation appends or adds a fixed or predetermined value to the contents of the SEQNO register described above.

An IM_SAP operation loads or stores a fixed or predetermined value in the CSUMSTART register described above.

An IM_R1 operation may add or load a predetermined value in one or more temporary registers (e.g., R1, R2).

An IM_CTL operation loads or stores a fixed or predetermined value in the CONTROL register described above.

A ST_FLAG operation loads a value from a specified offset in the packet into a FLAGS register. The loaded value may comprise one or more fields or flags from a packet header.

One skilled in the art will recognize that the labels assigned to the operations and registers described above and elsewhere in this section are merely illustrative in nature and in no way limit the operations and parsing instructions that may be employed in other embodiments of the invention.

Instructions in program **2300** comprise instruction number field **2302**, which contains a number of an instruction within the program, and instruction name field **2304**, which contains a name of an instruction. In an alternative embodiment of the invention instruction number and instruction name fields may be merged or one of them may be omitted.

Instruction content field **2306** includes multiple portions for executing an instruction. An "extraction mask" portion of an instruction is a two-byte mask in hexadecimal notation. An extraction mask identifies a portion of a packet header to be copied or extracted, starting from the current packet offset (e.g., the current position of the parsing pointer). Illustratively, each bit in the packet's header that corresponds to a one in the hexadecimal value is copied for comparison to a comparison or test value. For example, a value of 0xFF00 in the extraction mask portion of an instruction signifies that the entire first byte at the current packet offset is to be copied and that the contents of the second byte are irrelevant. Similarly, an extraction mask of 0x3FFF signifies that all but the two most significant bits of the first byte are to be copied. A two-byte value is constructed from the extracted contents, using whatever was copied from the packet. Illustratively, the remainder of the value is padded with zeros. One skilled in the art will appreciate that the format of an extraction mask (or an output mask, described below) may be adjusted as necessary to reflect little endian or big endian representation.

One or more instructions in a parsing program may not require any data extracted from the packet at the pointer location to be able to perform its output operation. These instructions may have an extraction mask value of 0x0000 to indicate that although a two-byte value is still retrieved from the pointer position, every bit of the value is masked off. Such an extraction mask thus yields a definite value of zero. This type of instruction may be used when, for example, an output operation needs to be performed before another substantive portion of header data is extracted with an extraction mask other than 0x0000.

A "compare value" portion of an instruction is a two-byte hexadecimal value with which the extracted packet contents are to be compared. The compare value may be a value known to be stored in a particular field of a specific protocol header. The compare value may comprise a value that the extracted portion of the header should match or have a specified relationship to in order for the packet to be considered compatible with the pre-selected protocols.

An "operator" portion of an instruction identifies an operator signifying how the extracted and compare values are to be compared. Illustratively, EQ signifies that they are tested for equality, NE signifies that they are tested for inequality, LT signifies that the extracted value must be less than the compare value for the comparison to succeed, GE signifies that the extracted value must be greater than or equal to the compare value, etc. An instruction that awaits arrival of a new packet to be parsed may employ an operation of NP. Other operators for other functions may be added and the existing operators may be assigned other monikers.

A "success offset" portion of an instruction indicates the number of two-byte units that the pointer is to advance if the comparison between the extracted and test values succeeds. A "success instruction" portion of an instruction identifies the next instruction in program **2300** to execute if the comparison is successful.

Similarly, "failure offset" and "failure instruction" portions indicate the number of two-byte units to advance the pointer and the next instruction to execute, respectively, if the comparison fails. Although offsets are expressed in units of two bytes (e.g., sixteen-bit words) in this embodiment of

the invention, in an alternative embodiment of the invention they may be smaller or larger units. Further, as mentioned above an instruction may be identified by number or name.

Not all of the instructions in a program are necessarily used for each packet that is parsed. For example, a program may include instructions to test for more than one type or version of a protocol at a particular layer. In particular, program **2300** tests for either version four or six of the IP protocol at layer three. The instructions that are actually executed for a given packet will thus depend upon the format of the packet. Once a packet has been parsed as much as possible with a given program or it has been determined that the packet does or does not conform to a selected protocol, the parsing may cease or an instruction for halting the parsing procedure may be executed. Illustratively, a next instruction portion of an instruction (e.g., "success instruction" or "failure instruction") with the value "DONE" indicates the completion of parsing of a packet. A DONE, or similar, instruction may be a dummy instruction. In other words, "DONE" may simply signify that parsing to be terminated for the present packet. Or, like instruction eighteen of program **2300**, a DONE instruction may take some action to await a new packet (e.g., by initializing a register).

The remaining portions of instruction content field **2306** are used to specify and complete an output or other data storage operation. In particular, in this embodiment an "output operation" portion of an instruction corresponds to the operations included in the loaded instruction set. Thus, for program **2300**, the output operation portion of an instruction identifies one of the sixteen operations described above. The output operations employed in program **2300** are further described below in conjunction with individual instructions.

An "operation argument" portion of an instruction comprises one or more arguments or fields to be stored, loaded or otherwise used in conjunction with the instruction's output operation. Illustratively, the operation argument portion takes the form of a multi-bit hexadecimal value. For program **2300**, operation arguments are eleven bits in size. An argument or portion of an argument may have various meanings, depending upon the output operation. For example, an operation argument may comprise one or more numerical values to be stored in a register or to be used to locate or delimit a portion of a header. Or, an argument bit may comprise a flag to signal an action or status. In particular, one argument bit may specify that a particular register is to be reset; a set of argument bits may comprise an offset into a packet header to a value to be stored in a register, etc. Illustratively, the offset specified by an operation argument is applied to the location of the parsing pointer position before the pointer is advanced as specified by the applicable success offset or failure offset. The operation arguments used in program **2300** are explained in further detail below.

An "operation enabler" portion of an instruction content field specifies whether or when an instruction's output operation is to be performed. In particular, in the illustrated embodiment of the invention an instruction's output operation may or may not be performed, depending on the result of the comparison between a value extracted from a header and the compare value. For example, an output enabler may be set to a first value (e.g., zero) if the output operation is never to be performed. It may take different values if it is to be performed only when the comparison does or does not satisfy the operator (e.g., one or two, respectively). An operation enabler may take yet another value (e.g., three) if it is always to be performed.

A "shift" portion of an instruction comprises a value indicating how an output value is to be shifted. A shift may be necessary because different protocols sometime require values to be formatted differently. In addition, a value indicating a length or location of a header or header field may require shifting in order to reflect the appropriate magnitude represented by the value. For example, because program **2300** is designed to use two-byte units, a value may need to be shifted if it is to reflect other units (e.g., bytes). A shift value in a present embodiment indicates the number of positions (e.g., bits) to right-shift an output value. In another embodiment of the invention a shift value may represent a different shift type or direction.

Finally, an "output mask" specifies how a value being stored in a register or other data structure is to be formatted. As stated above, an output operation may require an extracted, computed or assembled value to be stored. Similar to the extraction mask, the output mask is a two-byte hexadecimal value. For every position in the output mask that contains a one, in this embodiment of the invention the corresponding bit in the two-byte value identified by the output operation and/or operation argument is to be stored. For example, a value of 0xFFFF indicates that the specified two-byte value is to be stored as is. Illustratively, for every position in the output mask that contains a zero, a zero is stored. Thus, a value of 0xF000 indicates that the most significant four bits of the first byte are to be stored, but the rest of the stored value is irrelevant, and may be padded with zeros.

An output operation of "NONE" may be used to indicate that there is no output operation to be performed or stored, in which case other instruction portions pertaining to output may be ignored or may comprise specified values (e.g., all zeros). In the program depicted in FIG. **23**, however, a CLR_REG output operation, which allows the selective re-initialization of registers, may be used with an operation argument of zero to effectively perform no output. In particular, an operation argument of zero for the CLR_REG operation indicates that no registers are to be reset. In an alternative embodiment of the invention the operation enabler portion of an instruction could be set to a value (e.g., zero) indicating that the output operation is never to be performed.

The format and sequence of instructions in FIG. **23** will be understood to represent just one method of parsing a packet to determine whether it conforms to a particular communication protocol. In particular, the instructions are designed to examine one or more portions of one or more packet headers for comparison to known or expected values and to configure or load a register or other storage location as necessary. As one skilled in the art will appreciate, instructions for parsing a packet may take any of a number of forms and be performed in a variety of sequences without exceeding the scope of the invention.

With reference now to FIG. **23**, instructions in program **2300** may be described in detail. Prior to execution of the program depicted in FIG. **23**, a parsing pointer is situated at the beginning of a packet's layer two header. The position of the parsing pointer may be stored in a register for easy reference and update during the parsing procedure. In particular, the position of the parsing pointer as an offset (e.g., from the beginning of the layer two header) may be used in computing the position of a particular position within a header.

Program **2300** begins with a WAIT instruction (e.g., instruction zero) that waits for a new packet (e.g., indicated by operator NP) and, when one is received, sets a parsing pointer to the twelfth byte of the layer two header. This offset to the twelfth byte is indicated by the success offset portion

of the instruction. Until a packet is received, the WAIT instruction loops on itself. In addition, a CLR_REG operation is conducted, but the operation enabler setting indicates that it is only conducted when the comparison succeeds (e.g., when a new packet is received).

The specified CLR_REG operation operates according to the WAIT instruction's operation argument (i.e., 0x3FF). In this embodiment, each bit of the argument corresponds to a register or other data structure. The registers initialized in this operation may include the following: ADDR (e.g., to store the parsing pointer's address or location), FLOWID (e.g., to store the packet's flow key), SEQNO (e.g., to store a TCP sequence number), SAP (e.g., the packet's ethertype) and PAYLOADLEN (e.g., payload length). The following registers configured to store certain offsets may also be reset: FLOWOFF (e.g., offset within FLOWID register), SEQOFF (e.g., offset within SEQNO register), L3OFFSET (e.g., offset of the packet's layer three header), HDRSPLIT (e.g., location to split packet) and CSUMSTART (e.g., starting location for computing a checksum). Also, one or more status or control indicators (e.g., CONTROL or FLAGS register) for reporting the status of one or more flags of a packet header may be reset. In addition, one or more temporary registers (e.g., R1, R2) or other data structures may also be initialized. These registers are merely illustrative of he data structures that may be employed in one embodiment of the invention. Other data structures may be employed in other embodiments for the same or different output operations.

Temporary registers such as R1 and/or R2 may be used in program **2300** to track various headers and header fields. One skilled in the art will recognize the number of possible combinations of communication protocols and the effect of those various combinations on the structure and format of a packet's headers. More information may need to be examined or gathered from a packet conforming to one protocol or set of protocols than from a packet conforming to another protocol or set of protocols. For example, if extension headers are used with an Internet Protocol header, values from those extension headers and/or their lengths may need to be stored, which values are not needed if extension headers are not used. When calculating a particular offset, such as an offset to the beginning of a packet's data portion for example, multiple registers may need to be maintained and their values combined or added. In this example, one register or temporary register may track the size or format of an extension header, while another register tracks the base IP header.

Instruction VLAN (e.g., instruction one) examines the two-byte field at the parsing pointer position (possibly a Type, Length or TPID field) for a value indicating a VLAN-tagged header (e.g., 8100 in hexadecimal). If the header is VLAN-tagged, the pointer is incremented a couple of bytes (e.g., one two-byte unit) and execution continues with instruction CFI; otherwise, execution continues with instruction 802.3. In either event, the instruction's operation enabler indicates that an IM_CTL operation is always to be performed.

As described above, an IM_CTL operation causes a control register or other data structure to be populated with one or more flags to report the status or condition of a packet. As described in the previous section, a control indicator may indicate whether a packet is suitable for enhanced processing (e.g., whether a No_Assist signal should be generated for the packet), whether a packet includes any data and, if so, whether the size of the data portion exceeds a specified threshold. The operation argument 0x00A for instruction VLAN comprises the value to be stored in the control register, with individual bits of the argument corresponding to particular flags. Illustratively, flags associated with the conditions just described may be set to one, or true, in this IM_CTL operation.

Instruction CFI (e.g., instruction two) examines the CFI bit or flag in a layer two header. If the CFI bit is set, then the packet is not suitable for the processing enhancements described in other sections and the parsing procedure ends by calling instruction DONE (e.g., instruction eighteen). If the CFI bit is not set, then the pointer is incremented another couple of bytes and execution continues with instruction 802.3. As explained above, a null output operation (e.g., "NONE") indicates that no output operation is performed. In addition, the output enabler value (e.g., zero) further ensures that no output operation is performed.

In instruction 802.3 (e.g., instruction three), a Type or Length field (depending on the location of the pointer and format of the packet) is examined to determine if the packet's layer two format is traditional Ethernet or 802.3 Ethernet. If the value in the header field appears to indicate 802.3 Ethernet (e.g., contains a hexadecimal value less than 0600), the pointer is incremented two bytes (to what should be an LLC SNAP field) and execution continues with instruction LLC_1. Otherwise, the layer two protocol may be considered traditional Ethernet and execution continues with instruction IPV4_1. Instruction 802.3 in this embodiment of the invention does not include an output operation.

In instructions LLC_1 and LLC_2 (e.g., instructions four and five), a suspected layer two LLC SNAP field is examined to ensure that the packet conforms to the 802.3 Ethernet protocol. In instruction LLC_1, a first part of the field is tested and, if successful, the pointer is incremented two bytes and a second part is tested in instruction LLC_2. If instruction LLC_2 succeeds, the parsing pointer is advanced four bytes to reach what should be a Type field and execution continues with instruction IPV4_1. If either test fails, however, the parsing procedure exits. In the illustrated embodiment of the invention, no output operation is performed while testing the LLC SNAP field.

In instruction IPV4_1 (e.g., instruction six), the parsing pointer should be at an Ethernet Type field. This field is examined to determine if the layer three protocol appears to correspond to version four of the Internet Protocol. If this test is successful (e.g., the Type field contains a hexadecimal value of 0800), the pointer is advanced two bytes to the beginning of the layer three header and execution of program **2300** continues with instruction IPV4_2. If the test is unsuccessful, then execution continues with instruction IPV6_1. Regardless of the test results, the operation enabler value (e.g., three) indicates that the specified LD_SAP output operation is always performed.

As described previously, in a LD_SAP operation a packet's ethertype (or Service Access Point) is stored in a register. Part of the operation argument of 0x100, in particular the right-most six bits (e.g., zero) constitute an offset to a two-byte value comprising the ethertype. The offset in this example is zero because, in the present context, the parsing pointer is already at the Type field that contains the ethertype. In the presently described embodiment, the remainder of the operation argument constitutes a flag specifying that the starting position of the layer three header (e.g., an offset from the beginning of the packet) is also to be saved (e.g., in the L3OFFSET register). In particular, the beginning of the layer three header is known to be located immediately after the two-byte Type field.

Instruction IPV4_2 (e.g., instruction seven) tests a suspected layer three version field to ensure that the layer three

protocol is version four of IP. In particular, a specification for version four of IP specifies that the first four bits of the layer three header contain a value of 0x4. If the test fails, the parsing procedure ends with instruction DONE. If the test succeeds, the pointer advances six bytes and instruction IPV4_3 is called.

The specified LD_SUM operation, which is only performed if the comparison in instruction IPV4_2 succeeds, indicates that an offset to the beginning of a point from which a checksum may be calculated should be stored. In particular, in the presently described embodiment of the invention a checksum should be calculated from the beginning of the TCP header (assuming that the layer four header is TCP). The value of the operation argument (e.g., 0x00A) indicates that the checksum is located twenty bytes (e.g., ten two-byte increments) from the current pointer. Thus, a value of twenty bytes is added to the parsing pointer position and the result is stored in a register or other data structure (e.g., the CSUMSTART register).

Instruction IPV4_3 (e.g., instruction eight) is designed to determine whether the packet's IP header indicates IP fragmentation. If the value extracted from the header in accordance with the extraction mask does not equal the comparison value, then the packet indicates fragmentation. If fragmentation is detected, the packet is considered unsuitable for the processing enhancements described in other sections and the procedure exits (e.g., through instruction DONE). Otherwise, the pointer is incremented two bytes and instruction IPV4_4 is called after performing a LD_LEN operation.

In accordance with the LD_LEN operation, the length of the IP segment is saved. The illustrated operation argument (e.g., 0x03E) comprises an offset to the Total Length field where this value is located. In particular, the least-significant six bits constitute the offset. Because the pointer has already been advanced past this field, the operation argument comprises a negative value. One skilled in the art will recognize that this binary value (e.g., 111110) may be used to represent the decimal value of negative two. Thus, the present offset of the pointer, minus four bytes (e.g., two two-byte units), is saved in a register or other data structure (e.g., the PAYLOADLEN register). Any other suitable method of representing a negative offset may be used. Or, the IP segment length may be saved while the pointer is at a location preceding the Total Length field (e.g., during a previous instruction).

In instruction IPV4_4 (e.g., instruction nine), a one-byte Protocol field is examined to determine whether the layer four protocol appears to be TCP. If so, the pointer is advanced fourteen bytes and execution continues with instruction TCP_1; otherwise the procedure ends.

The specified LD_FID operation, which is only performed when the comparison in instruction IPV4_4 succeeds, involves retrieving the packet's flow key and storing it in a register or other location (e.g., the FLOWID register). One skilled in the art will appreciate that in order for the comparison in instruction IPV4_4 to be successful, the packet's layer three and four headers must conform to IP (version four) and TCP, respectively. If so, then the entire flow key (e.g., IP source and destination addresses plus TCP source and destination port numbers) is stored contiguously in the packet's header portion. In particular, the flow key comprises the last portion of the IP header and the initial portion of the TCP header and may be extracted in one operation. The operation argument (e.g., 0x182) thus comprises two values needed to locate and delimit the flow key. Illustratively, the right-most six bits of the argument (e.g., 0x02) identify an offset from the pointer position, in two-byte units, to the beginning of the flow key. The other five bits of the argument (e.g., 0x06) identify the size of the flow key, in two-byte units, to be stored.

In instruction IPV6_1 (e.g., instruction ten), which follows the failure of the comparison performed by instruction IPV4_1, the parsing pointer should be at a layer two Type field. If this test is successful (e.g., the Type field holds a hexadecimal value of 86DD), instruction IPV6_2 is executed after a LD_SUM operation is performed and the pointer is incremented two bytes to the beginning of the layer three protocol. If the test is unsuccessful, the procedure exits.

The indicated LD_SUM operation in instruction IPV6_1 is similar to the operation conducted in instruction IPV4_2 but utilizes a different argument. Again, the checksum is to be calculated from the beginning of the TCP header (assuming the layer four header is TCP). The specified operation argument (e.g., 0x015) thus comprises an offset to the beginning of the TCP header—twenty-one two-byte steps ahead. The indicated offset is added to the present pointer position and saved in a register or other data structure (e.g., the CSUMSTART register).

Instruction IPV6_2 (e.g., instruction eleven) tests a suspected layer three version field to further ensure that the layer three protocol is version six of IP. If the comparison fails, the parsing procedure ends with the invocation of instruction DONE. If it succeeds, instruction IPV6_3 is called. Operation IM_R1, which is performed only when the comparison succeeds in this embodiment, saves the length of the IP header from a Payload Length field. As one skilled in the art will appreciate, the Total Length field (e.g., IP segment size) of an IP, version four, header includes the size of the version four header. However, the Payload Length field (e.g., IP segment size) of an IP, version six, header does not include the size of the version six header. Thus, the size of the version six header, which is identified by the right-most eight bits of the output argument (e.g., 0x14, indicating twenty two-byte units) is saved. Illustratively, the remainder of the argument identifies the data structure in which to store the header length (e.g., temporary register R1). Because of the variation in size of layer three headers between protocols, in one embodiment of the invention the header size is indicated in different units to allow greater precision. In particular, in one embodiment of the invention the size of the header is specified in bytes in instruction IPV6_2, in which case the output argument could be 0x128.

Instruction IPV6_3 (e.g., instruction twelve) in this embodiment does not examine a header value. In this embodiment, the combination of an extraction mask of 0x0000 with a comparison value of 0x0000 indicates that an output operation is desired before the next examination of a portion of a header. After the LD_FID operation is performed, the parsing pointer is advanced six bytes to a Next Header field of the version six IP header. Because the extraction mask and comparison values are both 0x0000, the comparison should never fail and the failure branch of instruction should never be invoked.

As described previously, a LD_FID operation stores a flow key in an appropriate register or other data structure (e.g., the FLOWID register). Illustratively, the operation argument of 0x484 comprises two values for identifying and delimiting the flow key. In particular, the right-most six bits (e.g., 0x04) indicates that the flow key portion is located at an offset of eight bytes (e.g., four two-byte increments) from the current pointer position. The remainder of the operation

argument (e.g., 0x12) indicates that thirty-six bytes (e.g., the decimal equivalent of 0x12 two-byte units) are to be copied from the computed offset. In the illustrated embodiment of the invention the entire flow key is copied intact, including the layer three source and destination addresses and layer four source and destination ports.

In instruction IPV6_4 (e.g., instruction thirteen), a suspected Next Header field is examined to determine whether the layer four protocol of the packet's protocol stack appears to be TCP. If so, the procedure advances thirty-six bytes (e.g., eighteen two-byte units) and instruction TCP_1 is called; otherwise the procedure exits (e.g., through instruction DONE). Operation LD_LEN is performed if the value in the Next Header field is 0x06. As described above, this operation stores the IP segment size. Once again the argument (e.g., 0x03F) comprises a negative offset, in this case negative one. This offset indicates that the desired Payload Length field is located two bytes before the pointer's present position. Thus, the negative offset is added to the present pointer offset and the result saved in an appropriate register or other data structure (e.g., the PAYLOADLEN register).

In instructions TCP_1, TCP_2, TCP_3 and TCP_4 (e.g., instructions fourteen through seventeen), no header values—other than certain flags specified in the instruction's output operations—are examined, but various data from the packet's TCP header are saved. In the illustrated embodiment, the data that is saved includes a TCP sequence number, a TCP header length and one or more flags. For each instruction, the specified operation is performed and the next instruction is called. As described above, a comparison between the comparison value of 0x0000 and a null extraction value, as used in each of these instructions, will never fail. After instruction TCP_4, the parsing procedure returns to instruction WAIT to await a new packet.

For operation LD_SEQ in instruction TCP_1, the operation argument (e.g., 0x081) comprises two values to identify and extract a TCP sequence number. The right-most six bits (e.g., 0x01) indicate that the sequence number is located two bytes from the pointer's current position. The rest of the argument (e.g., 0x2) indicates the number of two-byte units that must be copied from that position in order to capture the sequence number. Illustratively, the sequence number is stored in the SEQNO register.

For operation ST_FLAG in instruction TCP_2, the operation argument (e.g., 0x145) is used to configure a register (e.g., the FLAGS register) with flags to be used in a post-parsing task. The right-most six bits (e.g., 0x05) constitute an offset, in two-byte units, to a two-byte portion of the TCP header that contains flags that may affect whether the packet is suitable for post-parsing enhancements described in other sections. For example, URG, PSH, RST, SYN and FIN flags may be located at the offset position and be used to configure the register. The output mask (e.g., 0x002F) indicates that only particular portions (e.g., bits) of the TCP header's Flags field are stored.

Operation LD_R1 of instruction TCP_3 is similar to the operation conducted in instruction IPV6_2. Here, an operation argument of 0x205 includes a value (e.g., the least-significant six bits) identifying an offset of five two-byte units from the current pointer position. That location should include a Header Length field to be stored in a data structure identified by the remainder of the argument (e.g., temporary register R1). The output mask (e.g., 0xF000) indicates that only the first four bits are saved (e.g., the Header Length field is only four bits in size).

As one skilled in the art may recognize, the value extracted from the Header Length field may need to be adjusted in order to reflect the use of two-byte units (e.g., sixteen bit words) in the illustrated embodiment. Therefore, in accordance with the shift portion of instruction TCP_3, the value extracted from the field and configured by the output mask (e.g., 0xF000) is shifted to the right eleven positions when stored in order to simplify calculations.

Operation LD_HDR of instruction TCP_4 causes the loading of an offset to the first byte of packet data following the TCP header. As described in a later section, packets that are compatible with a pre-selected protocol stack may be separated at some point into header and data portions. Saving an offset to the data portion now makes it easier to split the packet later. Illustratively, the right-most seven bits of the 0x0FF operation argument comprise a first element of the offset to the data. One skilled in the art will recognize the bit pattern (e.g., 1111111) as equating to negative one. Thus, an offset value equal to the current parsing pointer (e.g., the value in the ADDR register) minus two bytes—which locates the beginning of the TCP header—is saved. The remainder of the argument signifies that the value of a temporary data structure (e.g., temporary register R1) is to be added to this offset. In this particular context, the value saved in the previous instruction (e.g., the length of the TCP header) is added. These two values combine to form an offset to the beginning of the packet data, which is stored in an appropriate register or other data structure (e.g., the HDRSPLIT register).

Finally, and as mentioned above, instruction DONE (e.g., instruction eighteen) indicates the end of parsing of a packet when it is determined that the packet does not conform to one or more of the protocols associated with the illustrated instructions. This may be considered a "clean-up" instruction. In particular, output operation LD_CTL, with an operation argument of 0x001 indicates that a No_Assist flag is to be set (e.g., to one) in the control register described above in conjunction with instruction VLAN. The No_Assist flag, as described elsewhere, may be used to inform other modules of the network interface that the present packet, is unsuitable for one or more processing enhancements described elsewhere.

It will be recognized by one skilled in the art that the illustrated program or microcode merely provides one method of parsing a packet. Other programs, comprising the same instructions in a different sequence or different instructions altogether, with similar or dissimilar formats, may be employed to examine and store portions of headers and to configure registers and other data structures.

The efficiency gains to be realized from the application of the enhanced processing described in following sections more than offset the time required to parse a packet with the illustrated program. Further, even though a header parser parses a packet on a NIC in a current embodiment of the invention, the packet may still need to be processed through its protocol stack (e.g., to remove the protocol headers) by a processor on a host computer. Doing so avoids burdening the communication device (e.g., network interface) with such a task.

One Embodiment of a Flow Database

FIG. **5** depicts flow database (FDB) **110** according to one embodiment of the invention. Illustratively FDB **110** is implemented as a CAM (Content Addressable Memory) using a re-writeable memory component (e.g., RAM, SRAM, DRAM). In this embodiment, FDB **110** comprises associative portion **502** and associated portion **504**, and may be indexed by flow number **506**.

The scope of the invention does not limit the form or structure of flow database **110**. In alternative embodiments

of the invention virtually any form of data structure may be employed (e.g., database, table, queue, list, array), either monolithic or segmented, and may be implemented in hardware or software. The illustrated form of FDB **110** is merely one manner of maintaining useful information concerning communication flows through NIC **100**. As one skilled in the art will recognize, the structure of a CAM allows highly efficient and fast associative searching.

In the illustrated embodiment of the invention, the information stored in FDB **110** and the operation of flow database manager (FDBM) **108** (described below) permit functions such as data re-assembly, batch processing of packet headers, and other enhancements. These functions are discussed in detail in other sections but may be briefly described as follows.

One form of data re-assembly involves the re-assembly or combination of data from multiple related packets (e.g., packets from a single communication flow or a single datagram). One method for the batch processing of packet headers entails processing protocol headers from multiple related packets through a protocol stack collectively rather than one packet at a time. Another illustrative function of NIC **100** involves the distribution or sharing of such protocol stack processing (and/or other functions) among processors in a multi-processor host computer system. Yet another possible function of NIC **100** is to enable the transfer of re-assembled data to a destination entity (e.g., an application program) in an efficient aggregation (e.g., a memory page), thereby avoiding piecemeal and highly inefficient transfers of one packet's data at a time. Thus, in this embodiment of the invention, one purpose of FDB **110** and FDBM **108** is to generate information for the use of NIC **100** and/or a host computer system in enabling, disabling or performing one or more of these functions.

Associative portion **502** of FDB **110** in FIG. **5** stores the flow key of each valid flow destined for an entity served by NIC **100**. Thus, in one embodiment of the invention associative portion **502** includes IP source address **510**, IP destination address **512**, TCP source port **514** and TCP destination port **516**. As described in a previous section these fields may be extracted from a packet and provided to FDBM **108** by header parser **106**.

Although each destination entity served by NIC **100** may participate in multiple communication flows or end-to-end TCP connections, only one flow at a time will exist between a particular source entity and a particular destination entity. Therefore, each flow key in associative portion **502** that corresponds to a valid flow should be unique from all other valid flows. In alternative embodiments of the invention, associative portion **502** is composed of different fields, reflecting alternative flow key forms, which may be determined by the protocols parsed by the header parser and the information used to identify communication flows.

Associated portion **504** in the illustrated embodiment comprises flow validity indicator **520**, flow sequence number **522** and flow activity indicator **524**. These fields provide information concerning the flow identified by the flow key stored in the corresponding entry in associative portion **502**. The fields of associated portion **504** may be retrieved and/or updated by FDBM **108** as described in the following section.

Flow validity indicator **520** in this embodiment indicates whether the associated flow is valid or invalid. Illustratively, the flow validity indicator is set to indicate a valid flow when the first packet of data in a flow is received, and may be reset to reassert a flow's validity every time a portion of a flow's datagram (e.g., a packet) is correctly received.

Flow validity indicator **520** may be marked invalid after the last packet of data in a flow is received. The flow validity indicator may also be set to indicate an invalid flow whenever a flow is to be torn down (e.g., terminated or aborted) for some reason other than the receipt of a final data packet. For example, a packet may be received out of order from other packets of a datagram, a control packet indicating that a data transfer or flow is being aborted may be received, an attempt may be made to re-establish or re-synchronize a flow (in which case the original flow is terminated), etc. In one embodiment of the invention flow validity indicator **520** is a single bit, flag or value.

Flow sequence number **522** in the illustrated embodiment comprises a sequence number of the next portion of data that is expected in the associated flow. Because the datagram being sent in a flow is typically received via multiple packets, the flow sequence number provides a mechanism to ensure that the packets are received in the correct order. For example, in one embodiment of the invention NIC **100** re-assembles data from multiple packets of a datagram. To perform this re-assembly in the most efficient manner, the packets need to be received in order. Thus, flow sequence number **522** stores an identifier to identify the next packet or portion of data that should be received.

In one embodiment of the invention, flow sequence number **522** corresponds to the TCP sequence number field found in TCP protocol headers. As one skilled in the art will recognize, a packet's TCP sequence number identifies the position of the packet's data relative to other data being sent in a datagram. For packets and flows involving protocols other than TCP, an alternative method of verifying or ensuring the receipt of data in the correct order may be employed.

Flow activity indicator **524** in the illustrated embodiment reflects the recency of activity of a flow or, in other words, the age of a flow. In this embodiment of the invention flow activity indicator **524** is associated with a counter, such as a flow activity counter (not depicted in FIG. **5**). The flow activity counter is updated (e.g., incremented) each time a packet is received as part of a flow that is already stored in flow database **110**. The updated counter value is then stored in the flow activity indicator field of the packet's flow. The flow activity counter may also be incremented each time a first packet of a new flow that is being added to the database is received. In an alternative embodiment, a flow activity counter is only updated for packets containing data (e.g., it is not updated for control packets). In yet another alternative embodiment, multiple counters are used for updating flow activity indicators of different flows.

Because it can not always be determined when a communication flow has ended (e.g., the final packet may have been lost), the flow activity indicator may be used to identify flows that are obsolete or that should be torn down for some other reason. For example, if flow database **110** appears to be fully populated (e.g., flow validity indicator **520** is set for each flow number) when the first packet of a new flow is received, the flow having the lowest flow activity indicator may be replaced by the new flow.

In the illustrated embodiment of the invention, the size of fields in FDB **110** may differ from one entry to another. For example, IP source and destination addresses are four bytes large in version four of the protocol, but are sixteen bytes large in version six. In one alternative embodiment of the invention, entries for a particular field may be uniform in size, with smaller entries being padded as necessary.

In another alternative embodiment of the invention, fields within FDB **110** may be merged. In particular, a flow's flow key may be stored as a single entity or field instead of being stored as a number of separate fields as shown in FIG. **5**.

Similarly, flow validity indicator **520**, flow sequence number **522** and flow activity indicator **524** are depicted as separate entries in FIG. **5**. However, in an alternative embodiment of the invention one or more of these entries may be combined. In particular, in one alternative embodiment flow validity indicator **520** and flow activity indicator **524** comprise a single entry having a first value (e.g., zero) when the entry's associated flow is invalid. As long as the flow is valid, however, the combined entry is incremented as packets are received, and is reset to the first value upon termination of the flow.

In one embodiment of the invention FDB **110** contains a maximum of sixty-four entries, indexed by flow number **506**, thus allowing the database to track sixty-four valid flows at a time. In alternative embodiments of the invention, more or fewer entries may be permitted, depending upon the size of memory allocated for flow database **110**. In addition to flow number **506**, a flow may be identifiable by its flow key (stored in associative portion **502**).

In the illustrated embodiment of the invention, flow database **110** is empty (e.g., all fields are filled with zeros) when NIC **100** is initialized. When the first packet of a flow is received header parser **106** parses a header portion of the packet. As described in a previous section, the header parser assembles a flow key to identify the flow and extracts other information concerning the packet and/or the flow. The flow key, and other information, is passed to flow database manager **108**. FDBM **108** then searches FDB **110** for an active flow associated with the flow key. Because the database is empty, there is no match.

In this example, the flow key is therefore stored (e.g., as flow number zero) by copying the IP source address, IP destination address, TCP source port and TCP destination port into the corresponding fields. Flow validity indicator **520** is then set to indicate a valid flow, flow sequence number **522** is derived from the TCP sequence number (illustratively provided by the header parser), and flow activity indicator **524** is set to an initial value (e.g., one), which may be derived from a counter. One method of generating an appropriate flow sequence number, which may be used to verify that the next portion of data received for the flow is received in order, is to add the TCP sequence number and the size of the packet's data. Depending upon the configuration of the packet (e.g., whether the SYN bit in a Flags field of the packet's TCP header is set), however, the sum may need to be adjusted (e.g., by adding one) to correctly identify the next expected portion of data.

As described above, one method of generating an appropriate initial value for a flow activity indicator is to copy a counter value that is incremented for each packet received as part of a flow. For example, for the first packet received after NIC **100** is initialized, a flow activity counter may be incremented to the value of one. This value may then be stored in flow activity indicator **524** for the associated flow. The next packet received as part of the same (or a new) flow causes the counter to be incremented to two, which value is stored in the flow activity indicator for the associated flow. In this example, no two flows should have the same flow activity indicator except at initialization, when they may all equal zero or some other predetermined value.

Upon receipt and parsing of a later packet received at NIC **100**, the flow database is searched for a valid flow matching that packet's flow key. Illustratively, only the flow keys of active flows (e.g., those flows for which flow validity indicator **520** is set) are searched. Alternatively, all flow keys (e.g., all entries in associative portion **502**) may be searched but a match is only reported if its flow validity indicator indicates a valid flow. With a CAM such as FDB **110** in FIG. **5**, flow keys and flow validity indicators may be searched in parallel.

If a later packet contains the next portion of data for a previous flow (e.g., flow number zero), that flow is updated appropriately. In one embodiment of the invention this entails updating flow sequence number **522** and incrementing flow activity indicator **524** to reflect its recent activity. Flow validity indicator **520** may also be set to indicate the validity of the flow, although it should already indicate that the flow is valid.

As new flows are identified, they are added to FDB **110** in a similar manner to the first flow. When a flow is terminated or torn down, the associated entry in FDB **110** is invalidated. In one embodiment of the invention, flow validity indicator **520** is merely cleared (e.g., set to zero) for the terminated flow. In another embodiment, one or more fields of a terminated flow are cleared or set to an arbitrary or predetermined value. Because of the bursty nature of network packet traffic, all or most of the data from a datagram is generally received in a short amount of time. Thus, each valid flow in FDB **110** normally only needs to be maintained for a short period of time, and its entry can then be used to store a different flow.

Due to the limited amount of memory available for flow database **110** in one embodiment of the invention, the size of each field may be limited. In this embodiment, sixteen bytes are allocated for IP source address **510** and sixteen bytes are allocated for IP destination address **512**. For IP addresses shorter than sixteen bytes in length, the extra space may be padded with zeros. Further, TCP source port **514** and TCP destination port **516** are each allocated two bytes. Also in this embodiment, flow validity indicator **520** comprises one bit, flow sequence number **522** is allocated four bytes and flow activity indicator **524** is also allocated four bytes.

As one skilled in the art will recognize from the embodiments described above, a flow is similar, but not identical, to an end-to-end TCP connection. A TCP connection may exist for a relatively extended period of time, sufficient to transfer multiple datagrams from a source entity to a destination entity. A flow, however, may exist only for one datagram. Thus, during one end-to-end TCP connection, multiple flows may be set up and torn down (e.g., once for each datagram). As described above, a flow may be set up (e.g., added to FDB **110** and marked valid) when NIC **100** detects the first portion of data in a datagram and may be torn down (e.g., marked invalid in FDB **110**) when the last portion of data is received. Illustratively, each flow set up during a single end-to-end TCP connection will have the same flow key because the layer three and layer four address and port identifiers used to form the flow key will remain the same.

In the illustrated embodiment, the size of flow database **110** (e.g., the number of flow entries) determines the maximum number of flows that may be interleaved (e.g., simultaneously active) at one time while enabling the functions of data re-assembly and batch processing of protocol headers. In other words, in the embodiment depicted in FIG. **5**, NIC **100** can set up sixty-four flows and receive packets from up to sixty-four different datagrams (i.e., sixty-four flows may be active) without tearing down a flow. If a maximum number of flows through NIC **100** were known, flow database **110** could be limited to the corresponding number of entries.

The flow database may be kept small because a flow only lasts for one datagram in the presently described embodiment and, because of the bursty nature of packet traffic, a datagram's packets are generally received in a short period

of time. The short duration of a flow compensates for a limited number of entries in the flow database. In one embodiment of the invention, if FDB **110** is filled with active flows and a new flow is commenced (i.e., a first portion of data in a new datagram), the oldest (e.g., the least recently active) flow is replaced by the new one.

In an alternative embodiment of the invention, flows may be kept active for any number of datagrams (or other measure of network traffic) or for a specified length or range of time. For example, when one datagram ends its flow in FDB **110** may be kept "open" (i.e., not torn down) if the database is not full (e.g., the flow's entry is not needed for a different flow). This scheme may further enhance the efficient operation of NIC **100** if another datagram having the same flow key is received. In particular, the overhead involved in setting up another flow is avoided and more data re-assembly and packet batching (as described below) may be performed. Advantageously, a flow may be kept open in flow database **110** until the end-to-end TCP connection that encompasses the flow ends.

One Embodiment of a Flow Database Manager

FIGS. **6A–6E** depict one method of operating a flow database manager (FDBM), such as flow database manager **108** of FIG. **1A**, for managing flow database (FDB) **110**. Illustratively, FDBM **108** stores and updates flow information stored in flow database **110** and generates an operation code for a packet received by NIC **100**. FDBM **108** also tears down a flow (e.g., replaces, removes or otherwise invalidates an entry in FDB **110**) when the flow is terminated or aborted.

In one embodiment of the invention a packet's operation code reflects the packet's compatibility with pre-determined criteria for performing one or more functions of NIC **100** (e.g., data re-assembly, batch processing of packet headers, load distribution). In other words, depending upon a packet's operation code, other modules of NIC **100** may or may not perform one of these functions, as described in following sections.

In another embodiment of the invention, an operation code indicates a packet status. For example, an operation code may indicate that a packet: contains no data, is a control packet, contains more than a specified amount of data, is the first packet of a new flow, is the last packet of an existing flow, is out of order, contains a certain flag (e.g., in a protocol header) that does not have an expected value (thus possibly indicating an exceptional circumstance), etc.

The operation of flow database manager **108** depends upon packet information provided by header parser **106** and data drawn from flow database **110**. After FDBM **108** processes the packet information and/or data, control information (e.g., the packet's operation code) is stored in control queue **118** and FDB **110** may be altered (e.g., a new flow may be entered or an existing one updated or torn down).

With reference now to FIGS. **6A–6E**, state **600** is a start state in which FDBM **108** awaits information drawn from a packet received by NIC **100** from network **102**. In state **602**, header parser **106** or another module of NIC **100** notifies FDBM **108** of a new packet by providing the packet's flow key and some control information. Receipt of this data may be interpreted as a request to search FDB **110** to determine whether a flow having this flow key already exists.

In one embodiment of the invention the control information passed to FDBM **108** includes a sequence number (e.g., a TCP sequence number) drawn from a packet header. The control information may also indicate the status of certain flags in the packet's headers, whether the packet includes data and, if so, whether the amount of data exceeds a certain size. In this embodiment, FDBM **108** also receives a No_Assist signal for a packet if the header parser determines that the packet is not formatted according to one of the pre-selected protocol stacks (i.e., the packet is not "compatible"), as discussed in a previous section. Illustratively, the No_Assist signal indicates that one or more functions of NIC **100** (e.g., data re-assembly, batch processing, load-balancing) may not be provided for the packet.

In state **604**, FDBM **108** determines whether a No_Assist signal was asserted for the packet. If so, the procedure proceeds to state **668** (FIG. **6E**). Otherwise, FDBM **108** searches FDB **110** for the packet's flow key in state **606**. In one embodiment of the invention only valid flow entries in the flow database are searched. As discussed above, a flow's validity may be reflected by a validity indicator such as flow validity indicator **520** (shown in FIG. **5**). If, in state **608**, it is determined that the packet's flow key was not found in the database, or that a match was found but the associated flow is not valid, the procedure advances to state **646** (FIG. **6D**).

If a valid match is found in the flow database, in state **610** the flow number (e.g., the flow database index for the matching entry) of the matching flow is noted and flow information stored in FDB **110** is read. Illustratively, this information includes flow validity indicator **520**, flow sequence number **522** and flow activity indicator **524** (shown in FIG. **5**).

In state **612**, FDBM **108** determines from information received from header parser **106** whether the packet contains TCP payload data. If not, the illustrated procedure proceeds to state **638** (FIG. **6C**); otherwise the procedure continues to state **614**.

In state **614**, the flow database manager determines whether the packet constitutes an attempt to reset a communication connection or flow. Illustratively, this may be determined by examining the state of a SYN bit in one of the packet's protocol headers (e.g., a TCP header). In one embodiment of the invention the value of one or more control or flag bits (such as the SYN bit) are provided to the FDBM by the header parser. As one skilled in the art will recognize, one TCP entity may attempt to reset a communication flow or connection with another entity (e.g., because of a problem on one of the entity's host computers) and send a first portion of data along with the re-connection request. This is the situation the flow database manager attempts to discern in state **614**. If the packet is part of an attempt to re-connect or reset a flow or connection, the procedure continues at state **630** (FIG. **6C**).

In state **616**, flow database manager **108** compares a sequence number (e.g., a TCP sequence number) extracted from a packet header with a sequence number (e.g., flow sequence number **522** of FIG. **5**) of the next expected portion of data for this flow. As discussed in a previous section, these sequence numbers should correlate if the packet contains the flow's next portion of data. If the sequence numbers do not match, the procedure continues at state **628**.

In state **618**, FDBM **108** determines whether certain flags extracted from one or more of the packet's protocol headers match expected values. For example, in one embodiment of the invention the URG, PSH, RST and FIN flags from the packet's TCP header are expected to be clear (i.e., equal to zero). If any of these flags are set (e.g., equal to one) an exceptional condition may exist, thus making it possible that one or more of the functions (e.g., data re-assembly, batch processing, load distribution) offered by NIC **100** should not be performed for this packet. As long as the flags are clear, the procedure continues at state **620**; otherwise the procedure continues at state **626**.

In state **620**, the flow database manager determines whether more data is expected during this flow. As discussed above, a flow may be limited in duration to a single datagram. Therefore, in state **620** the FDBM determines if this packet appears to be the final portion of data for this flow's datagram. Illustratively, this determination is made on the basis of the amount of data included with the present packet. As one skilled in the art will appreciate, a datagram comprising more data than can be carried in one packet is sent via multiple packets. The typical manner of disseminating a datagram among multiple packets is to put as much data as possible into each packet. Thus, each packet except the last is usually equal or nearly equal in size to the maximum transfer unit (MTU) allowed for the network over which the packets are sent. The last packet will hold the remainder, usually causing it to be smaller than the MTU.

Therefore, one manner of identifying the final portion of data in a flow's datagram is to examine the size of each packet and compare it to a figure (e.g., MTU) that a packet is expected to exceed except when carrying the last data portion. It was described above that control information is received by FDBM **108** from header parser **106**. An indication of the size of the data carried by a packet may be included in this information. In particular, header parser **106** in one embodiment of the invention is configured to compare the size of each packet's data portion to a pre-selected value. In one embodiment of the invention this value is programmable. This value is set, in the illustrated embodiment of the invention, to the maximum amount of data a packet can carry without exceeding MTU. In one alternative embodiment, the value is set to an amount somewhat less than the maximum amount of data that can be carried.

Thus, in state **620**, flow database manager **108** determines whether the received packet appears to carry the final portion of data for the flow's datagram. If not, the procedure continues to state **626**.

In state **622**, it has been ascertained that the packet is compatible with pre-selected protocols and is suitable for one or more functions offered by NIC **100**. In particular, the packet has been formatted appropriately for one or more of the functions discussed above. FDBM **108** has determined that the received packet is part of an existing flow, is compatible with the pre-selected protocols and contains the next portion of data for the flow (but not the final portion). Further, the packet is not part of an attempt to re-set a flow/connection, and important flags have their expected values. Thus, flow database **110** can be updated as follows.

The activity indicator (e.g., flow activity indicator **524** of FIG. **5**) for this flow is modified to reflect the recent flow activity. In one embodiment of the invention flow activity indicator **524** is implemented as a counter, or is associated with a counter, that is incremented each time data is received for a flow. In another embodiment of the invention, an activity indicator or counter is updated every time a packet having a flow key matching a valid flow (e.g., whether or not the packet includes data) is received.

In the illustrated embodiment, after a flow activity indicator or counter is incremented it is examined to determine if it "rolled over" to zero (i.e., whether it was incremented past its maximum value). If so, the counter and/or the flow activity indicators for each entry in flow database **110** are set to zero and the current flow's activity indicator is once again incremented. Thus, in one embodiment of the invention the rolling over of a flow activity counter or indicator causes the re-initialization of the flow activity mechanism for flow database **110**. Thereafter, the counter is incremented and the flow activity indicators are again updated as described previously. One skilled in the art will recognize that there are many other suitable methods that may be applied in an embodiment of the present invention to indicate that one flow was active more recently than another was.

Also in state **622**, flow sequence number **522** is updated. Illustratively, the new flow sequence number is determined by adding the size of the newly received data to the existing flow sequence number. Depending upon the configuration of the packet (e.g., values in its headers), this sum may need to be adjusted. For example, this sum may indicate simply the total amount of data received thus far for the flow's datagram. Therefore, a value may need to be added (e.g., one byte) in order to indicate a sequence number of the next byte of data for the datagram. As one skilled in the art will recognize, other suitable methods of ensuring that data is received in order may be used in place of the scheme described here.

Finally, in state **622** in one embodiment of the invention, flow validity indicator **520** is set or reset to indicate the flow's validity.

Then, in state **624**, an operation code is associated with the packet. In the illustrated embodiment of the invention, operation codes comprise codes generated by flow database manager **108** and stored in control queue **118**. In this embodiment, an operation code is three bits in size, thus allowing for eight operation codes. Operation codes may have a variety of other forms and ranges in alternative embodiments. For the illustrated embodiment of the invention, TABLE 1 describes each operation code in terms of the criteria that lead to each code's selection and the ramifications of that selection. For purposes of TABLE 1, setting up a flow comprises inserting a flow into flow database **110**. Tearing down a flow comprises removing or invalidating a flow in flow database **110**. The re-assembly of data is discussed in a following section describing DMA engine **120**.

In the illustrated embodiment of the invention, operation code 4 is selected in state **624** for packets in the present context of the procedure (e.g., compatible packets carrying the next, but not last, data portion of a flow). Thus, the existing flow is not torn down and there is no need to set up a new flow. As described above, a compatible packet in this embodiment is a packet conforming to one or more of the pre-selected protocols. By changing or augmenting the pre-selected protocols, virtually any packet may be compatible in an alternative embodiment of the invention.

Returning now to FIGS. **6A–6E**, after state **624** the illustrated procedure ends at state **670**.

In state **626** (reached from state **618** or state **620**), operation code 3 is selected for the packet. Illustratively, operation code 3 indicates that the packet is compatible and matches a valid flow (e.g., the packet's flow key matches the flow key of a valid flow in FDB **110**). Operation code 3 may also signify that the packet contains data, does not constitute an attempt to re-synchronize or reset a communication flow/connection and the packet's sequence number matches the expected sequence number (from flow database **110**). But, either an important flag (e.g., one of the TCP flags URG, PSH, RST or FIN) is set (determined in state **618**) or the packet's data is less than the threshold value described above (in state **620**), thus indicating that no more data is likely to follow this packet in this flow. Therefore, the existing flow is torn down but no new flow is created. Illustratively, the flow may be torn down by clearing the flow's validity indicator (e.g., setting it to zero). After state **626**, the illustrated procedure ends at state **670**.

In state **628** (reached from state **616**), operation code 2 is selected for the packet. In the present context, operation

code 2 may indicate that the packet is compatible, matches a valid flow (e.g., the packet's flow key matches the flow key of a valid flow in FDB **110**), contains data and does not constitute an attempt to re-synchronize or reset a communication flow/connection. However, the sequence number extracted from the packet (in state **616**) does not match the expected sequence number from flow database **110**. This may occur, for example, when a packet is received out of order. Thus, the existing flow is torn down but no new flow is established. Illustratively, the flow may be torn down by clearing the flow's validity indicator (e.g., setting it to zero). After state **628**, the illustrated procedure ends at state **670**.

State **630** is entered from state **614** when it is determined that the received packet constitutes an attempt to reset a communication flow or connection (e.g., the TCP SYN bit is set). In state **630**, flow database manager **108** determines whether more data is expected to follow. As explained in conjunction with state **620**, this determination may be made on the basis of control information received by the flow database manager from the header parser. If more data is expected (e.g., the amount of data in the packet equals or exceeds a threshold value), the procedure continues at state **634**.

In state **632**, operation code 2 is selected for the packet. Operation code 2 was also selected in state **628** in a different context. In the present context, operation code 2 may indicate that the packet is compatible, matches a valid flow and contains data. Operation code 2 may also signify in this context that the packet constitutes an attempt to re-synchronize or reset a communication flow or connection, but that no more data is expected once the flow/connection is reset. Therefore, the existing flow is torn down and no new flow is established. Illustratively, the flow may be torn down by clearing the flow's validity indicator (e.g., setting it to zero). After state **632**, the illustrated procedure ends at state **670**.

In state **634**, flow database manager **108** responds to an attempt to reset or re-synchronize a communication flow/connection whereby additional data is expected. Thus, the existing flow is torn down and replaced as follows. The existing flow may be identified by the flow number retrieved in state **610** or by the packet's flow key. The flow's sequence number (e.g., flow sequence number **522** in FIG. **5**) is set to the next expected value. Illustratively, this value depends upon the sequence number (e.g., TCP sequence number) retrieved from the packet (e.g., by header parser **106**) and the amount of data included in the packet. In one embodiment of the invention these two values are added to determine a new flow sequence number. As discussed previously, this sum may need to be adjusted (e.g., by adding one). Also in state **634**, the flow activity indicator is updated (e.g., incremented). As explained in conjunction with state **622**, if the flow activity indicator rolls over, the activity indicators for all flows in the database are set to zero and the present flow is again incremented. Finally, the flow validity indicator is set to indicate that the flow is valid.

In state **636**, operation code 7 is selected for the packet. In the present context, operation code 7 indicates that the packet is compatible, matches a valid flow and contains data. Operation code 7 may further signify, in this context, that the packet constitutes an attempt to re-synchronize or reset a communication flow/connection and that additional data is expected once the flow/connection is reset. In effect, therefore, the existing flow is torn down and a new one (with the same flow key) is stored in its place. After state **636**, the illustrated procedure ends at end state **670**.

State **638** is entered after state **612** when it is determined that the received packet contains no data. This often indicates that the packet is a control packet. In state **638**, flow database manager **108** determines whether one or more flags extracted from the packet by the header parser match expected or desired values. For example, in one embodiment of the invention the TCP flags URG, PSH, RST and FIN must be clear in order for DMA engine **120** to re-assemble data from multiple related packets (e.g., packets having an identical flow key). As discussed above, the TCP SYN bit may also be examined. In the present context (e.g., a packet with no data), the SYN bit is also expected to be clear (e.g., to store a value of zero). If the flags (and SYN bit) have their expected values the procedure continues at state **642**. If, however, any of these flags are set, an exceptional condition may exist, thus making it possible that one or more functions offered by NIC **100** (e.g., data re-assembly, batch processing, load distribution) are unsuitable for this packet, in which case the procedure proceeds to state **640**.

In state **640**, operation code 1 is selected for the packet. Illustratively, operation code 1 indicates that the packet is compatible and matches a valid flow, but does not contain any data and one or more important flags or bits in the packet's header(s) are set. Thus, the existing flow is torn down and no new flow is established. Illustratively, the flow may be torn down by clearing the flow's validity indicator (e.g., setting it to zero). After state **640**, the illustrated procedure ends at end state **670**.

In state **642**, the flow's activity indicator is updated (e.g., incremented) even though the packet contains no data. As described above in conjunction with state **622**, if the activity indicator rolls over, in a present embodiment of the invention all flow activity indicators in the database are set to zero and the current flow is again incremented. The flow's validity indicator may also be reset, as well as the flow's sequence number.

In state **644**, operation code 0 is selected for the packet. Illustratively, operation code 0 indicates that the packet is compatible, matches a valid flow, and that the packet does not contain any data. The packet may, for example, be a control packet. Operation code 0 further indicates that none of the flags checked by header parser **106** and described above (e.g., URG, PSH, RST and FIN) are set. Thus, the existing flow is not torn down and no new flow is established. After state **644**, the illustrated procedure ends at end state **670**.

State **646** is entered from state **608** if the packet's flow key does not match any of the flow keys of valid flows in the flow database. In state **646**, FDBM **108** determines whether flow database **110** is full and may save some indication of whether the database is full. In one embodiment of the invention the flow database is considered full when the validity indicator (e.g., flow validity indicator **520** of FIG. **5**) is set for every flow number (e.g., for every flow in the database). If the database is full, the procedure continues at state **650**, otherwise it continues at state **648**.

In state **648**, the lowest flow number of an invalid flow (e.g., a flow for which the associated flow validity indicator is equal to zero) is determined. Illustratively, this flow number is where a new flow will be stored if the received packet warrants the creation of a new flow. After state **648**, the procedure continues at state **652**.

In state **650**, the flow number of the least recently active flow is determined. As discussed above, in the illustrated embodiment of the invention a flow's activity indicator (e.g., flow activity indicator **524** of FIG. **5**) is updated (e.g., incremented) each time data is received for a flow. Therefore, in this embodiment the least recently active flow can be identified as the flow having the least recently

updated (e.g., lowest) flow activity indicator. Illustratively, if multiple flows have flow activity indicators set to a common value (e.g., zero), one flow number may be chosen from them at random or by some other criteria. After state **650**, the procedure continues at state **652**.

In state **652**, flow database manager **108** determines whether the packet contains data. Illustratively, the control information provided to FDBM **108** by the header parser indicates whether the packet has data. If the packet does not include data (e.g., the packet is a control packet), the illustrated procedure continues at state **668**.

In state **654**, flow database manager **108** determines whether the data received with the present packet appears to contain the final portion of data for the associated datagram/flow. As described in conjunction with state **620**, this determination may be made on the basis of the amount of data included with the packet. If the amount of data is less than a threshold value (a programmable value in the illustrated embodiment), then no more data is expected and this is likely to be the only data for this flow. In this case the procedure continues at state **668**. If, however, the data meets or exceeds the threshold value, in which case more data may be expected, the procedure proceeds to state **656**.

In state **656**, the values of certain flags are examined. These flags may include, for example, the URG, PSH, RST, FIN bits of a TCP header. If any of the examined flags do not have their expected or desired values (e.g., if any of the flags are set), an exceptional condition may exist making one or more of the functions of NIC **100** (e.g., data re-assembly, batch processing, load distribution) unsuitable for this packet. In this case the procedure continues at state **668**; otherwise the procedure proceeds to state **658**.

In state **658**, the flow database manager retrieves the information stored in state **646** concerning whether flow database **110** is full. If the database is full, the procedure continues at state **664**; otherwise the procedure continues at state **660**.

In state **660**, a new flow is added to flow database **110** for the present packet. Illustratively, the new flow is stored at the flow number identified or retrieved in state **648**. The addition of a new flow may involve setting a sequence number (e.g., flow sequence number **522** from FIG. **5**). Flow sequence number **522** may be generated by adding a sequence number (e.g., TCP sequence number) retrieved from the packet and the amount of data included in the packet. As discussed above, this sum may need to be adjusted (e.g., by adding one).

Storing a new flow may also include initializing an activity indicator (e.g., flow activity indicator **524** of FIG. **5**). In one embodiment of the invention this initialization involves storing a value retrieved from a counter that is incremented each time data is received for a flow. Illustratively, if the counter or a flow activity indicator is incremented past its maximum storable value, the counter and all flow activity indicators are cleared or reset. Also in state **660**, a validity indicator (e.g., flow validity indicator **520** of FIG. **5**) is set to indicate that the flow is valid. Finally, the packet's flow key is also stored in the flow database, in the entry corresponding to the assigned flow number.

In state **662**, operation code 6 is selected for the packet. Illustratively, operation code 6 indicates that the packet is compatible, did not match any valid flows and contains the first portion of data for a new flow. Further, the packet's flags have their expected or necessary values, additional data is expected in the flow and the flow database is not full. Thus, operation code 6 indicates that there is no existing flow to tear down and that a new flow has been stored in the flow database. After state **662**, the illustrated procedure ends at state **670**.

In state **664**, an existing entry in the flow database is replaced so that a new flow, initiated by the present packet, can be stored. Therefore, the flow number of the least recently active flow, identified in state **650**, is retrieved. This flow may be replaced as follows. The sequence number of the existing flow (e.g., flow sequence number **522** of FIG. **5**) is replaced with a value derived by combining a sequence number extracted from the packet (e.g., TCP sequence number) with the size of the data portion of the packet. This sum may need to be adjusted (e.g., by adding one). Then the existing flow's activity indicator (e.g., flow activity indicator **524**) is replaced. For example, the value of a flow activity counter may be copied into the flow activity indicator, as discussed above. The flow's validity indicator (e.g., flow validity indicator **520** of FIG. **5**) is then set to indicate that the flow is valid. Finally, the flow key of the new flow is stored.

In state **666**, operation code 7 is selected for the packet. Operation code 7 was also selected in state **636**. In the present context, operation code 7 may indicate that the packet is compatible, did not match the flow key of any valid flows and contains the first portion of data for a new flow. Further, the packet's flags have compatible values and additional data is expected in the flow. Lastly, however, in this context operation code 7 indicates that the flow database is full, so an existing entry was torn down and the new one stored in its place. After state **666**, the illustrated procedure ends at end state **670**.

In state **668**, operation code 5 is selected for the packet. State **668** is entered from various states and operation code 5 thus represents a variety of possible conditions or situations. For example, operation code 5 may be selected when a No_Assist signal is detected (in state **604**) for a packet. As discussed above, the No_Assist signal may indicate that the corresponding packet is not compatible with a set of pre-selected protocols. In this embodiment of the invention, incompatible packets are ineligible for one or more of the various functions of NIC **100** (e.g., data re-assembly, batch processing, load distribution).

State **668** may also be entered, and operation code 5 selected, from state **652**, in which case the code may indicate that the received packet does not match any valid flow keys and, further, contains no data (e.g., it may be a control packet).

State **668** may also be entered from state **654**. In this context operation code 5 may indicate that the packet does not match any valid flow keys. It may further indicate that the packet contains data, but that the size of the data portion is less than the threshold discussed in conjunction with state **654**. In this context, it appears that the packet's data is complete (e.g., comprises all of the data for a datagram), meaning that there is no other data to re-assemble with this packet's data and therefore there is no reason to make a new entry in the database for this one-packet flow.

Finally, state **668** may also be entered from state **656**. In this context, operation code 5 may indicate that the packet does not match any valid flow keys, contains data, and more data is expected, but at least one flag in one or more of the packet's protocol headers does not have its expected value. For example, in one embodiment of the invention the TCP flags URG, PSH, RST and FIN are expected to be clear. If any of these flags are set an exceptional condition may exist, thus making it possible that one of the functions offered by NIC **100** is unsuitable for this packet.

As TABLE 1 reflects, there is no flow to tear down and no new flow is established when operation code 5 is selected. Following state **668**, the illustrated procedure ends at state **670**.

One skilled in the art will appreciate that the procedure illustrated in FIGS. **6A–6E** and discussed above is but one suitable procedure for maintaining and updating a flow database and for determining a packet's suitability for certain processing functions. In particular, different operation codes may be utilized or may be implemented in a different manner, a goal being to produce information for later processing of the packet through NIC **100**.

Although operation codes are assigned for all packets by a flow database manager in the illustrated procedure, in an alternative procedure an operation code assigned by the FDBM may be replaced or changed by another module of NIC **100**. This may be done to ensure a particular method of treating certain types of packets. For example, in one embodiment of the invention IPP module **104** assigns a predetermined operation code (e.g., operation code 2 of TABLE 1) to jumbo packets (e.g., packets greater in size than MTU) so that DMA engine **120** will not re-assemble them. In particular, the IPP module may independently determine that the packet is a jumbo packet (e.g., from information provided by a MAC module) and therefore assign the predetermined code. Illustratively, header parser **106** and FDBM **108** perform their normal functions for a jumbo packet and IPP module **104** receives a first operation code assigned by the FDBM. However, the IPP module replaces that code before storing the jumbo packet and information concerning the packet. In one alternative embodiment header parser **106** and/or flow database manager **108** may be configured to recognize a particular type of packet (e.g., jumbo) and assign a predetermined operation code.

The operation codes applied in the embodiment of the invention illustrated in FIGS. **6A–6E** are presented and explained in the following TABLE 1. TABLE 1 includes illustrative criteria used to select each operation code and illustrative results or effects of each code.

TABLE 1

Op. Code	Criteria for Selection	Result of Operation Code
0	Compatible control packet with clear flags; a flow was previously established for this flow key.	Do not set up a new flow; Do not tear down existing flow; Do not re-assemble data (packet contains no data).
1	Compatible control packet with at least one flag or SYN bit set; a flow was previously established.	Do not set up a new flow; Tear down existing flow; Do no re-assemble data (packet contains no data).
2	Compatible packet whose sequence number does not match sequence number in flow database, or SYN bit is set (indicating attempt to re-establish a connection) but there is no more data to come; a flow was previously established. -- Or -- Jumbo packet.	Do not set up a new flow; Tear down existing flow; Do not re-assemble packet data.
3	A compatible packet carrying a final portion of flow data, or a flag is set (but packet is in sequence, unlike operation code 2); a flow was previously established.	Do not set up a new flow; Tear down existing flow; Re-assemble data with previous packets.
4	Receipt of next compatible packet in sequence; a flow was previously established.	Do not set up a new flow; Do not tear down existing flow; Re-assemble data with other packets.
5	Packet cannot be re-assembled because: incompatible, a flag is set, packet contains no data or there is no more data to come. No flow was previously established.	Do not set up a flow; There is no flow to tear down; Do not re-assemble.

TABLE 1-continued

Op. Code	Criteria for Selection	Result of Operation Code
6	First compatible packet of a new flow; no flow was previously established.	Set up a new flow; There is no flow to tear down; Re-assemble data with packets to follow.
7	First compatible packet of a new flow, but flow database is full; no flow was previously established. -- Or -- Compatible packet, SYN bit is set and additional data will follow; a flow was previously established.	Replace existing flow; Re-assemble data with packets to follow.

One Embodiment of a Load Distributor

In one embodiment of the invention, load distributor **112** enables the processing of packets through their protocol stacks to be distributed among a number of processors. Illustratively, load distributor **112** generates an identifier (e.g., a processor number) of a processor to which a packet is to be submitted. The multiple processors may be located within a host computer system that is served by NIC **100**. In one alternative embodiment, one or more processors for manipulating packets through a protocol stack are located on NIC **100**.

Without an effective method of sharing or distributing the processing burden, one processor could become overloaded if it were required to process all or most network traffic received at NIC **100**, particularly in a high-speed network environment. The resulting delay in processing network traffic could deteriorate operations on the host computer system as well as other computer systems communicating with the host system via the network.

As one skilled in the art will appreciate, simply distributing packets among processors in a set of processors (e.g., such as in a round-robin scheme) may not be an efficient plan. Such a plan could easily result in packets being processed out of order. For example, if two packets from one communication flow or connection that are received at a network interface in the correct order were submitted to two different processors, the second packet may be processed before the first. This could occur, for example, if the processor that received the first packet could not immediately process the packet because it was busy with another task. When packets are processed out of order a recovery scheme must generally be initiated, thus introducing even more inefficiency and more delay.

Therefore, in a present embodiment of the invention packets are distributed among multiple processors based upon their flow identities. As described above, a header parser may generate a flow key from layer three (e.g., IP) and layer four (e.g., TCP) source and destination identifiers retrieved from a packet's headers. The flow key may be used to identify the communication flow to which the packet belongs. Thus, in this embodiment of the invention all packets having an identical flow key are submitted to a single processor. As long as the packets are received in order by NIC **100**, they should be provided to the host computer and processed in order by their assigned processor.

Illustratively, multiple packets sent from one source entity to one destination entity will have the same flow key even if the packets are part of separate datagrams, as long as their layer three and layer four identifiers remain the same. As discussed above, separate flows are set up and torn down for each datagram within one TCP end-to-end connection. Therefore, just as all packets within one flow are sent to one

processor, all packets within a TCP end-to-end connection will also be sent to the same processor. This helps ensure the correct ordering of packets for the entire connection, even between datagrams.

Depending upon the network environment in which NIC **100** operates (e.g., the protocols supported by network **102**), the flow key may be too large to use as an identifier of a processor. In one embodiment of the invention described above, for example, a flow key measures 288 bits. Meanwhile, the number of processors participating in the load-balancing scheme may be much smaller. For example, in the embodiment of the invention described below in conjunction with FIG. **7**, a maximum of sixty-four processors is supported. Thus, in this embodiment only a six-bit number is needed to identify the selected processor. The larger flow key may therefore be mapped or hashed into a smaller range of values.

FIG. **7** depicts one method of generating an identifier (e.g., a processor number) to specify a processor to process a packet received by NIC **100**, based on the packet's flow key. In this embodiment of the invention, network **102** is the Internet and a received packet is formatted according to a compatible protocol stack (e.g., Ethernet at layer two, IP at layer three and TCP at layer four).

State **700** is a start state. In state **702** a packet is received by NIC **100** and a header portion of the packet is parsed by header parser **106** (a method of parsing a packet is described in a previous section). In state **704**, load distributor **112** receives the packet's flow key that was generated by header parser **106**.

Because a packet's flow key is 288 bits wide in this embodiment, in state **706** a hashing function is performed to generate a value that is smaller in magnitude. The hash operation may, for example, comprise a thirty-two bit CRC (cyclic redundancy check) function such as ATM (Asynchronous Transfer Mode) Adaptation Layer 5 (AAL5). AAL5 generates thirty-two bit numbers that are fairly evenly distributed among the 2^{32} possible values. Another suitable method of hashing is the standard Ethernet CRC-32 function. Other hash functions that are capable of generating relatively small numbers from relatively large flow keys, where the numbers generated are well distributed among a range of values, are also suitable.

With the resulting hash value, in state **708** a modulus operation is performed over the number of processors available for distributing or sharing the processing. Illustratively, software executing on the host computer (e.g., a device driver for NIC **100**) programs or stores the number of processors such that it may be read or retrieved by load distributor **112** (e.g., in a register). The number of processors available for load balancing may be all or a subset of the number of processors installed on the host computer system. In the illustrated embodiment, the number of processors available in a host computer system is programmable, with a maximum value of sixty-four. The result of the modulus operation in this embodiment, therefore, is the number of the processor (e.g., from zero to sixty-three) to which the packet is to be submitted for processing. In this embodiment of the invention, load distributor **112** is implemented in hardware, thus allowing rapid execution of the hashing and modulus functions. In an alternative embodiment of the invention, virtually any number of processors may be accommodated.

In state **710**, the number of the processor that will process the packet through its protocol stack is stored in the host computer's memory. Illustratively, state **710** is performed in parallel with the storage of the packet in a host memory buffer. As described in a following section, in one embodiment of the invention a descriptor ring in the host computer's memory is constructed to hold the processor number and possibly other information concerning the packet (e.g., a pointer to the packet, its size, its TCP checksum).

A descriptor ring in this embodiment is a data structure comprising a number of entries, or "descriptors," for storing information to be used by a network interface circuit's host computer system. In the illustrated embodiment, a descriptor temporarily stores packet information after the packet has been received by NIC **100**, but before the packet is processed by the host computer system. The information stored in a descriptor may be used, for example, by the device driver for NIC **100** or for processing the packet through its protocol stack.

In state **712**, an interrupt or other alert is issued to the host computer to inform it that a new packet has been delivered from NIC **100**. In an embodiment of the invention in which NIC **100** is coupled to the host computer by a PCI (Peripheral Component Interconnect) bus, the INTA signal may be asserted across the bus. A PCI controller in the host receives the signal and the host operating system is alerted (e.g., via an interrupt).

In state **714**, software operating on the host computer (e.g., a device driver for NIC **100**) is invoked (e.g., by the host computer's operating system interrupt handler) to act upon a newly received packet. The software gathers information from one or more descriptors in the descriptor ring and places information needed to complete the processing of each new packet into a queue for the specified processor (i.e., according to the processor number stored in the packet's descriptor). Illustratively, each descriptor corresponds to a separate packet. The information stored in the processor queue for each packet may include a pointer to a buffer containing the packet, the packet's TCP checksum, offsets of one or more protocol headers, etc. In addition, each processor participating in the load distribution scheme may have an associated queue for processing network packets. In an alternative embodiment of the invention, multiple queues may be used (e.g., for multiple priority levels or for different protocol stacks).

Illustratively, one processor on the host computer system is configured to receive all alerts and/or interrupts associated with the receipt of network packets from NIC **100** and to alert the appropriate software routine or device driver. This initial processing may, alternatively, be distributed among multiple processors. In addition, in one embodiment of the invention a portion of the retrieval and manipulation of descriptor contents is performed as part of the handling of the interrupt that is generated when a new packet is stored in the descriptor ring. The processor selected to process the packet will perform the remainder of the retrieval/manipulation procedure.

In state **716**, the processor designated to process a new packet is alerted or woken. In an embodiment of the invention operating on a Solaris™ workstation, individual processes executed by the processor are configured as "threads." A thread is a process running in a normal mode (e.g., not at an interrupt level) so as to have minimal impact on other processes executing on the workstation. A normal mode process may, however, execute at a high priority. Alternatively, a thread may run at a relatively low interrupt level.

A thread responsible for processing an incoming packet may block itself when it has no packets to process, and awaken when it has work to do. A "condition variable" may be used to indicate whether the thread has a packet to process. Illustratively, the condition variable is set to a first

value when the thread is to process a packet (e.g., when a packet is received for processing by the processor) and is set to a second value when there are no more packets to process. In the illustrated embodiment of the invention, one condition variable may be associated with each processor's queue.

In an alternative embodiment, the indicated processor is alerted in state **716** by a "cross-processor call." A cross-processor call is one way of communicating among processors whereby one processor is interrupted remotely by another processor. Other methods by which one processor alerts, or dispatches a process to, another processor may be used in place of threads and cross-processor calls.

In state **718**, a thread or other process on the selected processor begins processing the packet that was stored in the processor's queue. Methods of processing a packet through its protocol stack are well known to those skilled in the art and need not be described in detail. The illustrated procedure then ends with end state **720**.

In one alternative embodiment of the invention, a high-speed network interface is configured to receive and process ATM (Asynchronous Transfer Mode) traffic. In this embodiment, a load distributor is implemented as a set of instructions (e.g., as software) rather than as a hardware module. As one skilled in the art is aware, ATM traffic is connection-oriented and may be identified by a virtual connection identifier (VCI), which corresponds to a virtual circuit established between the packet's source and destination entities. Each packet that is part of a virtual circuit includes the VCI in its header.

Advantageously, a VCI is relatively small in size (e.g., sixteen bits). In this alternative embodiment, therefore, a packet's VCI may be used in place of a flow key for the purpose of distributing or sharing the burden of processing packets through their protocol stacks. Illustratively, traffic from different VCIs is sent to different processors, but, to ensure correct ordering of packets, all packets having the same VCI are sent to the same processor. When an ATM packet is received at a network interface, the VCI is retrieved from its header and provided to the load distributor. The modulus of the VCI over the number of processors that are available for load distribution is then computed. Similar to the illustrated embodiment, the packet and its associated processor number are then provided to the host computer.

As described above, load distribution in a present embodiment of the invention is performed on the basis of a packet's layer three and/or layer four source and destination entity identifiers. In an alternative embodiment of the invention, however, load distribution may be performed on the basis of layer two addresses. In this alternative embodiment, packets having the same Ethernet source and destination addresses, for example, are sent to a single processor.

As one of skill in the art will recognize, however, this may result in a processor receiving many more packets than it would if layer three and/or layer four identifiers were used. For example, if a large amount of traffic is received through a router situated near (in a logical sense) to the host computer, the source Ethernet address for all of the traffic may be the router's address even though the traffic is from a multitude of different end users and/or computers. In contrast, if the host computer is on the same Ethernet segment as all of the end users/computers, the layer two source addresses will show greater variety and allow more effective load sharing.

Other methods of distributing the processing of packets received from a network may differ from the embodiment illustrated in FIG. **7** without exceeding the scope of the invention. In particular, one skilled in the art will appreciate that many alternative procedures for assigning a flow's packets to a processor and delivering those packets to the processor may be employed.

One Embodiment of a Packet Queue

As described above, packet queue **116** stores packets received from IPP module **104** prior to their re-assembly by DMA engine **120** and their transfer to the host computer system. FIG. **8** depicts packet queue **116** according to one embodiment of the invention.

In the illustrated embodiment, packet queue **116** is implemented as a FIFO (First-In First-Out) queue containing up to **256** entries. Each packet queue entry in this embodiment stores one packet plus various information concerning the packet. For example, entry **800** includes packet portion **802** plus a packet status portion. Because packets of various sizes are stored in packet queue **116**, packet portion **802** may include filler **802***a* to supplement the packet so that the packet portion ends at an appropriate boundary (e.g., byte, word, double word).

Filler **802***a* may comprise random data or data having a specified pattern. Filler **802***a* may be distinguished from the stored packet by the pattern of the filler data or by a tag field.

Illustratively, packet status information includes TCP checksum value **804** and packet length **806** (e.g., length of the packet stored in packet portion **802**). Storing the packet length may allow the packet to be easily identified and retrieved from packet portion **802**. Packet status information may also include diagnostic/status information **808**. Diagnostic/status information **808** may include a flag indicating that the packet is bad (e.g., incomplete, received with an error), an indicator that a checksum was or was not computed for the packet, an indicator that the checksum has a certain value, an offset to the portion of the packet on which the checksum was computed, etc. Other flags or indicators may also be included for diagnostics, filtering, or other purposes. In one embodiment of the invention, the packet's flow key (described above and used to identify the flow comprising the packet) and/or flow number (e.g., the corresponding index of the packet's flow in flow database **110**) are included in diagnostic/status information **808**. In another embodiment, a tag field to identify or delimit filler **802***a* is included in diagnostic/status information **808**.

In one alternative embodiment of the invention, any or all of the packet status information described above is stored in control queue **118** rather than packet queue **116**.

In the illustrated embodiment of the invention packet queue **116** is implemented in hardware (e.g., as random access memory). In this embodiment, checksum value **804** is sixteen bits in size and may be stored by checksum generator **114**. Packet length **806** is fourteen bits large and may be stored by header parser **106**. Finally, portions of diagnostic/status information **808** may be stored by one or more of IPP module **104**, header parser **106**, flow database manager **108**, load distributor **112** and checksum generator **114**.

Packet queue **116** in FIG. **8** is indexed with two pointers. Read pointer **810** identifies the next entry to be read from the queue, while write pointer **812** identifies the entry in which the next received packet and related information is to be stored. As explained in a subsequent section, the packet stored in packet portion **802** of an entry is extracted from packet queue **116** when its data is to be-reassembled by DMA engine **120** and/or transferred to the host computer system.

One Embodiment of a Control Queue

In one embodiment of the invention, control queue **118** stores control and status information concerning a packet received by NIC **100**. In this embodiment, control queue **118**

retains information used to enable the batch processing of protocol headers and/or the re-assembly of data from multiple related packets. Control queue **118** may also store information to be used by the host computer or a series of instructions operating on a host computer (e.g., a device driver for NIC **100**). The information stored in control queue **118** may supplement or duplicate information stored in packet queue **116**.

FIG. **9** depicts control queue **118** in one embodiment of the invention. The illustrated control queue contains one entry for each packet stored in packet queue **116** (e.g., up to 256 entries). In one embodiment of the invention each entry in control queue **118** corresponds to the entry (e.g., packet) in packet queue **116** having the same number. FIG. **9** depicts entry **900** having various fields, such as CPU number **902**, No_Assist signal **904**, operation code **906**, payload offset **908**, payload size **910** and other status information **912**. An entry may also include other status or control information (not shown in FIG. **9**). Entries in control queue **118** in alternative embodiments of the invention may comprise different information.

CPU (or processor) number **902**, discussed in a previous section, indicates which one of multiple processors on the host computer system should process the packet's protocol headers. Illustratively, CPU number **902** is six bits in size. No_Assist signal **904**, also described in a preceding section, indicates whether the packet is compatible with (e.g., is formatted according to) any of a set of pre-selected protocols that may be parsed by header parser **106**. No_Assist signal **904** may comprise a single flag (e.g. one bit). In one embodiment of the invention the state or value of No_Assist signal **904** may be used by flow database manager **108** to determine whether a packet's data is re-assembleable and/or whether its headers may be processed with those of related packets. In particular, the FDBM may use the No_Assist signal in determining which operation code to assign to the packet.

Operation code **906** provides information to DMA engine **120** to assist in the re-assembly of the packet's data. As described in a previous section, an operation code may indicate whether a packet includes data or whether a packet's data is suitable for re-assembly. Illustratively, operation code **906** is three bits in size. Payload offset **908** and payload size **910** correspond to the offset and size of the packet's TCP payload (e.g., TCP data), respectively. These fields may be seven and fourteen bits large, respectively.

In the illustrated embodiment, other status information **912** includes diagnostic and/or status information concerning the packet. Status information **912** may include a starting position for a checksum calculation (which may be seven bits in size), an offset of the layer three (e.g., IP) protocol header (which may also be seven bits in size), etc. Status information **912** may also include an indicator as to whether the size of the packet exceeds a first threshold (e.g., whether the packet is greater than 1522 bytes) or falls under a second threshold (e.g., whether the packet is 256 bytes or less). This information may be useful in re-assembling packet data. Illustratively, these indicators comprise single-bit flags.

In one alternative embodiment of the invention, status information **912** includes a packet's flow key and/or flow number (e.g., the index of the packet's flow in flow database **110**). The flow key or flow number may, for example, be used for debugging or other diagnostic purposes. In one embodiment of the invention, the packet's flow number may be stored in status information **912** so that multiple packets in a single flow may be identified. Such related packet may then be collectively transferred to and/or processed by a host computer.

FIG. **9** depicts a read pointer and a write pointer for indexing control queue **118**. Read pointer **914** indicates an entry to be read by DMA engine **120**. Write pointer **916** indicates the entry in which to store information concerning the next packet stored in packet queue **116**.

In an alternative embodiment of the invention, a second read pointer (not shown in FIG. **9**) may be used for indexing control queue **118**. As described in a later section, when a packet is to be transferred to the host computer, information drawn from entries in the control queue is searched to determine whether a related packet (e.g., a packet in the same flow as the packet to be transferred) is also going to be transferred. If so, the host computer is alerted so that protocol headers from the related packets may be processed collectively. In this alternative embodiment of the invention, related packets are identified by matching their flow numbers (or flow keys) in status information **912**. The second read pointer may be used to look ahead in the control queue for packets with matching flow numbers.

In one embodiment of the invention CPU number **902** may be stored in the **10** control queue by load distributor **112** and No_Assist signal **904** may be stored by header parser **106**. Operation code **906** may be stored by flow database manager **108**, and payload offset **908** and payload size **910** may be stored by header parser **106**. Portions of other status information may be written by the preceding modules and/or others, such as IPP module **104** and checksum generator **114**. In one particular embodiment of the invention, however, many of these items of information are stored by IPP module **104** or some other module acting in somewhat of a coordinator role.

One Embodiment of a DMA Engine

FIG. **10** is a block diagram of DMA (Direct Memory Access) engine **120** in one embodiment of the invention. One purpose of DMA engine **120** in this embodiment is to transfer packets from packet queue **116** into buffers in host computer memory. Because related packets (e.g., packets that are part of one flow) can be identified by their flow numbers or flow keys, data from the related packets may be transferred together (e.g., in the same buffer). By using one buffer for data from one flow, the data can be provided to an application program or other destination in a highly efficient manner. For example, after the host computer receives the data, a page-flip operation may be performed to transfer the data to an application's memory space rather than performing numerous copy operations.

With reference back to FIGS. **1A–B**, a packet that is to be transferred into host memory by DMA engine **120** is stored in packet queue **116** after being received from network **102**. Header parser **106** parses a header portion of the packet and generates a flow key, and flow database manager **108** assigns an operation code to the packet. In addition, the communication flow that includes the packet is registered in flow database **110**. The packet's flow may be identified by its flow key or flow number (e.g., the index of the flow in flow database **110**). Finally, information concerning the packet (e.g., operation code, a packet size indicator, flow number) is stored in control queue **118** and, possibly, other portions or modules of NIC **100**, and the packet is transferred to the host computer by DMA engine **120**. During the transfer process, the DMA engine may draw upon information stored in the control queue to copy the packet into an appropriate buffer, as described below. Dynamic packet batching module **122** may also use information stored in the control queue, as discussed in detail in a following section.

With reference now to FIG. **10**, one embodiment of a DMA engine is presented. In this embodiment, DMA man-

ager **1002** manages the transfer of a packet, from packet queue **116**, into one or more buffers in host computer memory. Free ring manager **1012** identifies or receives empty buffers from host memory and completion ring manager **1014** releases the buffers to the host computer, as described below. The free ring manager and completion ring managers may be controlled with logic contained in DMA manager **1002**. In the illustrated embodiment, flow re-assembly table **1004**, header table **1006**, MTU table **1008** and jumbo table **1010** store information concerning buffers used to store different types of packets (as described below). Information stored in one of these tables may include a reference to, or some other means of identifying, a buffer. In FIG. **10**, DMA engine **120** is partially or fully implemented in hardware.

Empty buffers into which packets may be stored are identified via a free descriptor ring that is maintained in host memory. As one skilled in the art is aware, a descriptor ring is a data structure that is logically arranged as a circular queue. A descriptor ring contains descriptors for storing information (e.g., data, flag, pointer, address). In one embodiment of the invention, each descriptor stores its index within the free descriptor ring and an identifier (e.g., memory address, pointer) of a free buffer that may be used to store packets. In this embodiment a buffer is identified in a descriptor by its address in memory, although other means of identifying a memory buffer are also suitable. In one embodiment of the invention a descriptor index is thirteen bits large, allowing for a maximum of 8,192 descriptors in the ring, and a buffer address is sixty-four bits in size.

In the embodiment of FIG. **10**, software that executes on a host computer, such as a device driver for NIC **100**, maintains a free buffer array or other data structure (e.g., list, table) for storing references to (e.g., addresses of) the buffers identified in free descriptors. As descriptors are retrieved from the ring their buffer identifiers are placed in the array. Thus, when a buffer is needed for the storage of a packet, it may be identified by its index (e.g., cell, element) in the free buffer array. Then, when the buffer is no longer needed, it may be released to the host computer by placing its array index or reference in a completion descriptor. A packet stored in the buffer can then be retrieved by accessing the buffer identified in the specified element of the array. Thus, in this embodiment of the invention the size of a descriptor index (e.g., thirteen bits) may not limit the number of buffers that may be assigned by free ring manager **1012**. In particular, virtually any number of buffers or descriptors could be managed by the software. For example, in one alternative embodiment of the invention buffer identifiers may be stored in one or more linked lists after being retrieved from descriptors in a free descriptor ring. When the buffer is released to the host computer, a reference to the head of the buffer's linked list may be provided. The list could then be navigated to locate the particular buffer (e.g., by its address).

As one skilled in the art will appreciate, the inclusion of a limited number of descriptors in the free descriptor ring (e.g., 8,192 in this embodiment) means that they may be re-used in a round-robin fashion. In the presently described embodiment, a descriptor is just needed long enough to retrieve its buffer identifier (e.g., address) and place it in the free buffer array, after which it may be re-used relatively quickly. In other embodiments of the invention free descriptor rings having different numbers of free descriptors may be used, thus allowing some control over the rate at which free descriptors must be re-used.

In one alternative embodiment of the invention, instead of using a separate data structure to identify a buffer for storing a packet, a buffer may be identified within DMA engine **120** by the index of the free descriptor within the free descriptor ring that referenced the buffer. One drawback to this scheme when the ring contains a limited number of descriptors, however, is that a particular buffer's descriptor may need to be re-used before its buffer has been released to the host computer. Thus, either a method of avoiding or skipping the re-use of such a descriptor must be implemented or the buffer referenced by the descriptor must be released before the descriptor is needed again. Or, in another alternative, a free descriptor ring may be of such a large size that a lengthy or even virtually infinite period of time may pass from the time a free descriptor is first used until it needs to be re-used.

Thus, in the illustrated embodiment of the invention free ring manager **1012** retrieves a descriptor from the free descriptor ring, stores its buffer identifier (e.g., memory address) in a free buffer array, and provides the array index and/or buffer identifier to flow re-assembly table **1004**, header table **1006**, MTU table **1008** or jumbo table **1010**.

Free ring manager **1012** attempts to ensure that a buffer is always available for a packet. Thus, in one embodiment of the invention free ring manager **1012** includes descriptor cache **1012***a* configured to store a number of descriptors (e.g., up to eight) at a time. Whenever there are less than a threshold number of entries in the cache (e.g., five), additional descriptors may be retrieved from the free descriptor ring. Advantageously, the descriptors are of such a size (e.g., sixteen bytes) that some multiple (e.g., four) of them can be efficiently retrieved in a sixty-four byte cache line transfer from the host computer.

Returning now to the illustrated embodiment of the invention, each buffer in host memory is one memory page in size. However, buffers and the packets stored in the buffers may be divided into multiple categories based on packet size and whether a packet's data is being re-assembled. Re-assembly refers to the accumulation of data from multiple packets of a single flow into one buffer for efficient transfer from kernel space to user or application space within host memory. In particular, re-assembleable packets may be defined as packets that conform to a pre-selected protocol (e.g., a protocol that is parseable by header parser **106**). By filling a memory page with data for one destination, page-flipping may be performed to provide a page in kernel space to the application or user space. A packet's category (e.g., whether re-assembleable or non-re-assembleable) may be determined from information retrieved from the control queue or flow database manager. In particular, and as described previously, an operation code may be used to determine whether a packet contains a re-assembleable portion of data.

In the illustrated embodiment of the invention, data portions of related, re-assembleable, packets are placed into a first category of buffers—which may be termed re-assembly buffers. A second category of buffers, which may be called header buffers, stores the headers of those packets whose data portions are being re-assembled and may also store small packets (e.g., those less than or equal to 256 bytes in size). A third category of buffers, MTU buffers, stores non-re-assembleable packets that are larger than 256 bytes, but no larger than MTU size (e.g., 1522 bytes). Finally, a fourth category of buffers, jumbo buffers, stores jumbo packets (e.g., large packets that are greater than 1522 bytes in size) that are not being re-assembled. Illustratively, a jumbo packet may be stored intact (e.g., its headers and data portions kept together in one buffer) or its headers may be stored in a header buffer while its data portion is stored in an appropriate (e.g., jumbo) non-re-assembly buffer.

In one alternative embodiment of the invention, no distinction is made between MTU and jumbo packets. Thus, in this alternative embodiment, just three types of buffers are used: re-assembly and header buffers, as described above, plus non-re-assembly buffers. Illustratively, all non-small packets (e.g., larger than 256 bytes) that are-not re-assembled are placed in a non-re-assembly buffer.

In another alternative embodiment, jumbo packets may be re-assembled in jumbo buffers. In particular, in this embodiment data portions of packets smaller than a predetermined size (e.g., MTU) are re-assembled in normal re-assembly buffers while data portions of jumbo packets (e.g., packets greater in size than MTU) are re-assembled in jumbo buffers. Re-assembly of jumbo packets may be particularly effective for a communication flow that comprises jumbo frames of a size such that multiple frames can fit in one buffer. Header portions of both types of packets may be stored in one type of header buffer or, alternatively, different header buffers may be used for the headers of the different types of re-assembleable packets.

In yet another alternative embodiment of the invention buffers may be of varying sizes and may be identified in different descriptor rings or other data structures. For example, a first descriptor ring or other mechanism may be used to identify buffers of a first size for storing large or jumbo packets. A second ring may store descriptors referencing buffers for MTU-sized packets, and another ring may contain descriptors for identifying page-sized buffers (e.g., for data re-assembly).

A buffer used to store portions of more than one type of packet—such as a header buffer used to store headers and small packets, or a non-re-assembly buffer used to store MTU and jumbo packets—may be termed a "hybrid" buffer.

Illustratively, each time a packet or a portion of a packet is stored in a buffer, completion ring manager **1014** populates a descriptor in a completion descriptor ring with information concerning the packet. Included in the information stored in a completion descriptor in this embodiment is a number or reference identifying the free buffer array cell or element in which an identifier (e.g., memory address) of a buffer in which a portion of the packet is stored. The information may also include an offset into the buffer (e.g., to the beginning of the packet portion), the identity of another free buffer array entry that stores a buffer identifier for a buffer containing another portion of the packet, a size of the packet, etc. A packet may be stored in multiple buffers, for example, if the packet data and header are stored separately (e.g., the packet's data is being re-assembled in a re-assembly buffer while the packet's header is placed in a header buffer). In addition, data portions of a jumbo packet or a re-assembly packet may span two or more buffers, depending on the size of the data portion.

A distinction should be kept in mind between a buffer identifier (e.g., the memory address of a buffer) and the entry in the free buffer array in which the buffer identifier is stored. In particular, it has been described above that when a memory buffer is released to a host computer it is identified to the host computer by its position within a free buffer array (or other suitable data structure) rather than by its buffer identifier. The host computer retrieves the buffer identifier from the specified array element and accesses the specified buffer to locate a packet stored in the buffer. As one skilled in the art will appreciate, identifying memory buffers in completion descriptors by the buffers' positions in a free buffer array can be more efficient than identifying them by their memory addresses. In particular, in FIG. **10** buffer identifiers are sixty-four bits in size while an index in a free buffer array or similar data structure will likely be far smaller. Using array positions thus saves space compared to using buffer identifiers. Nonetheless, buffer identifiers may be used to directly identify buffers in an alternative embodiment of the invention, rather than filtering access to them through a free buffer array. However, completion descriptors would have to be correspondingly larger in order to accommodate them.

A completion descriptor may also include one or more flags indicating the type or size of a packet, whether the packet data should be re-assembled, whether the packet is the last of a datagram, whether the host computer should delay processing the packet to await a related packet, etc. As described in a following section, in one embodiment of the invention dynamic packet batching module **122** determines, at the time a packet is transferred to the host computer, whether a related packet will be sent shortly. If so, the host computer may be advised to delay processing the transferred packet and await the related packet in order to allow more efficient processing.

A packet's completion descriptor may be marked appropriately when the buffer identified by its buffer identifier is to be released to the host computer. For example, a flag may be set in the descriptor to indicate that the packet's buffer is being released from DMA engine **120** to the host computer or software operating on the host computer (e.g., a driver associated with NIC **100**). In one embodiment of the invention, completion ring manager **1014** includes completion descriptor cache **1014***a*. Completion descriptor cache **1014***a* may store one or more completion descriptors for collective transfer from DMA engine **120** to the host computer.

Thus, empty buffers are retrieved from a free ring and used buffers are released to the host computer through a completion ring. One reason that a separate ring is employed to release used buffers to the host computer is that buffers may not be released in the order in which they were taken. In one embodiment of the invention, a buffer (especially a flow re-assembly buffer) may not be released until it is full. Alternatively, a buffer may be released at virtually any time, such as when the end of a communication flow is detected. Free descriptors and completion descriptors are further described below in conjunction with FIG. **12**.

Another reason that separate rings are used for free and completion descriptors is that the number of completion descriptors that are required in an embodiment of the invention may exceed the number of free descriptors provided in a free descriptor ring. For example, a buffer provided by a free descriptor may be used to store multiple headers and/or small packets. Each time a header or small packet is stored in the header buffer, however, a separate completion descriptor is generated. In an embodiment of the invention in which a header buffer is eight kilobytes in size, a header buffer may store up to thirty-two small packets. For each packet stored in the header buffer, another completion descriptor is generated.

FIG. **11** includes diagrams of illustrative embodiments of flow re-assembly table **1004**, header table **1006**, MTU table **1008** and jumbo table **1010**. One alternative embodiment of the invention includes a non-re-assembly table in place of MTU table **1008** and jumbo table **1010**, corresponding to a single type of non-re-assembly buffer for both MTU and jumbo packets. Jumbo table **1010** may also be omitted in another alternative embodiment of the invention in which jumbo buffers are retrieved or identified only when needed. Because a jumbo buffer is used only once in this alternative embodiment, there is no need to maintain a table to track its use.

Flow re-assembly table **1004** in the illustrated embodiment stores information concerning the re-assembly of packets in one or more communication flows. For each flow that is active through DMA engine **120**, separate flow re-assembly buffers may be used to store the flow's data. More than one buffer may be used for a particular flow, but each flow has one entry in flow re-assembly table **1004** with which to track the use of a buffer. As described in a previous section, one embodiment of the invention supports the interleaving of up to sixty-four flows. Thus, flow re-assembly buffer table **1004** in this embodiment maintains up to sixty-four entries. A flow's entry in the flow re-assembly table may match its flow number (e.g., the index of the flow's flow key in flow database **110**) or, in an alternative embodiment, an entry may be used for any flow.

In FIG. **11**, an entry in flow re-assembly table **1004** includes flow re-assembly buffer index **1102**, next address **1104**. and validity indicator **1106**. Flow re-assembly buffer index **1102** comprises the index, or position, within a free buffer array or other data structure for storing buffer identifiers identified in free descriptors, of a buffer for storing data from the associated flow. Illustratively, this value is written into each completion descriptor associated with a packet whose data portion is stored in the buffer. This value may be used by software operating on the host computer to access the buffer and process the data. Next address **1104** identifies the location within the buffer (e.g., a memory address) at which to store the next portion of data. Illustratively, this field is updated each time data is added to the buffer. Validity indicator **1106** indicates whether the entry is valid. Illustratively, each entry is set to a valid state (e.g., stores a first value) when a first portion of data is stored in the flow's re-assembly buffer and is invalidated (e.g., stores a second value) when the buffer is full. When an entry is invalidated, the buffer may be released or returned to the host computer (e.g., because it is full).

Header table **1006** in the illustrated embodiment stores information concerning one or more header buffers in which packet headers and small packets are stored. In the illustrated embodiment of the invention, only one header buffer is active at a time. That is, headers and small packets are stored in one buffer until it is released, at which time a new buffer is used. In this embodiment, header table **1006** includes header buffer index **1112**, next address **1114** and validity indicator **1116**. Similar to flow re-assembly table **1004**, header buffer index **1112** identifies the cell or element in the free buffer array that contains a buffer identifier for a header buffer. Next address **1114** identifies the location within the header buffer at which to store the next header or small packet. This identifier, which may be a counter, may be updated each time a header or small packet is stored in the header buffer. Validity indicator **116** indicates whether the header buffer table and/or the header buffer is valid. This indicator may be set to valid when a first packet or header is stored in a header buffer and may be invalidated when it is released to the host computer.

MTU table **1008** stores information concerning one or more MTU buffers for storing MTU packets (e.g., packets larger than 256 bytes but less than 1523 bytes) that are not being re-assembled. MTU buffer index **1122** identifies the free buffer array element that contains a buffer identifier (e.g., address) of a buffer for storing MTU packets. Next address **1124** identifies the location in the current MTU buffer at which to store the next packet. Validity indicator **1126** indicates the validity of the table entry. The validity indicator may be set to a valid state when a first packet is stored in the MTU buffer and an invalid state when the buffer is to be released to the host computer.

Jumbo table **1010** stores information concerning one or more jumbo buffers for storing jumbo packets (e.g., packets larger than 1522 bytes) that are not being re-assembled. Jumbo buffer index **1132** identifies the element within the free buffer array that stores a buffer identifier corresponding to a jumbo buffer. Next address **1134** identifies the location in the jumbo buffer at which to store the next packet. Validity indicator **1136** indicates the validity of the table entry. Illustratively, the validity indicator is set to a valid state when a first packet is stored in the jumbo buffer and is set to an invalid state when the buffer is to be released to the host computer.

In the embodiment of the invention depicted in FIG. **11**, a packet larger than a specified size (e.g., 256 bytes) is not re-assembled if it is incompatible with the pre-selected protocols for NIC **100** (e.g., TCP, IP, Ethernet) or if the packet is too large (e.g., greater than 1522 bytes). Although two types of buffers (e.g., MTU and jumbo) are used for non-re-assembleable packets in this embodiment, in an alternative embodiment of the invention any number may be used, including one. Packets less than the specified size are generally not re-assembled. Instead, as described above, they are stored intact in a header buffer.

In the embodiment of the invention depicted in FIG. **11**, next address fields may store a memory address, offset, pointer, counter or other means of identifying a position within a buffer. Advantageously, the next address field of a table or table entry is initially set to the address of the buffer assigned to store packets of the type associated with the table (and, for re-assembly table **1004**, the particular flow). As the buffer is populated, the address is updated to identify the location in the buffer at which to store the next packet or portion of a packet.

Illustratively, each validity indicator stores a first value (e.g., one) to indicate validity, and a second value (e.g., zero) to indicate invalidity. In the illustrated embodiment of the invention, each index field is thirteen bits, each address field is sixty-four bits and the validity indicators are each one bit in size.

Tables **1004**, **1006**, **1008** and **1010** may take other forms and remain within the scope of the invention as contemplated. For example, these data structures may take the form of arrays, lists, databases, etc., and may be implemented in hardware or software. In the illustrated embodiment of the invention, header table **1006**, MTU table **1008** and jumbo table **1010** each contain only one entry at a time. Thus, only one header buffer, MTU buffer and jumbo buffer are active (e.g., valid) at a time in this embodiment. In an alternative embodiment of the invention, multiple header buffers, MTU buffers and/or jumbo buffers may be used (e.g., valid) at once.

In one embodiment of the invention, certain categories of buffers (e.g., header, non-re-assembly) may store a predetermined number of packets or packet portions. For example, where the memory page size of a host computer processor is eight kilobytes, a header buffer may store a maximum of thirty-two entries, each of which is 256 bytes. Illustratively, even when one packet or header is less than 256 bytes, the next entry in the buffer is stored at the next 256-byte boundary. A counter may be associated with the buffer and decremented (or incremented) each time a new entry is stored in the buffer. After thirty-two entries have been made, the buffer may be released.

In one embodiment of the invention, buffers other than header buffers may be divided into fixed-size regions. For example, in an eight-kilobyte MTU buffer, each MTU packet may be allocated two kilobytes. Any space remaining

in a packet's area after the packet is stored may be left unused or may be padded.

In one alternative embodiment of the invention, entries in a header buffer and/or non-re-assembly buffer (e.g., MTU, jumbo) are aligned for more efficient transfer. In particular, two bytes of padding (e.g., random bytes) are stored at the beginning of each entry in such a buffer. Because a packet's layer two Ethernet header is fourteen bytes long, by adding two pad bytes each packet's layer three protocol header (e.g., IP) will be aligned with a sixteen-byte boundary. Sixteen-byte alignment, as one skilled in the art will appreciate, allows efficient copying of packet contents (such as the layer three header). The addition of two bytes may, however, decrease the size of the maximum packet that may be stored in a header buffer (e.g., to 254 bytes).

As explained above, counters and/or padding may also be used with non-re-assembly buffers. Some non-re-assembleable packets (e.g., jumbo packets) may, however, be split into separate header and data portions, with each portion being stored in a separate buffer—similar to the re-assembly of flow packets. In one embodiment of the invention padding is only used with header portions of split packets. Thus, when a non-re-assembled (e.g., jumbo) packet is split, padding may be applied to the header/small buffer in which the packet's header portion is stored but not to the non-re-assembly buffer in which the packet's data portion is stored. When, however, a non-re-assembly packet is stored with its header and data together in a non-re-assembly buffer, then padding may be applied to that buffer.

In another alternative embodiment of the invention, a second level of padding may be added to each entry in a buffer that stores non-re-assembled packets that are larger than 256 bytes (e.g., MTU packets and jumbo packets that are not split). In this alternative embodiment, a cache line of storage (e.g., sixty-four bytes for a Solaris™ workstation) is skipped in the buffer before storing each packet. The extra padding area may be used by software that processes the packets and/or their completion descriptors. The software may use the extra padding area for routing or as temporary storage for information needed in a secondary or later phase of processing.

For example, before actually processing the packet, the software may store some data that promotes efficient multi-tasking in the padding area. The information is then available when the packet is finally extracted from the buffer. In particular, in one embodiment of the invention a network interface may generate one or more data values to identify multicast or alternate addresses that correspond to a layer two address of a packet received from a network. The multicast or alternate addresses may be stored in a network interface memory by software operating on a host computer (e.g., a device driver). By storing the data value(s) in the padding, enhanced routing functions can be performed when the host computer processes the packet.

Reserving sixty-four bytes at the beginning of a buffer also allows header information to be modified or prepended if necessary. For example, a regular Ethernet header of a packet may, because of routing requirements, need to be replaced with a much larger FDDI (Fiber Distributed Data Interface) header. One skilled in the art will recognize the size disparity between these headers. Advantageously, the reserved padding area may be used for the FDDI header rather than allocating another block of memory.

In a present embodiment of the invention DMA engine **120** may determine which category a packet belongs in, and which type of buffer to store the packet in, by examining the packet's operation code. As described in a previous section, an operation code may be stored in control queue **118** for each packet stored in packet queue **116**. Thus, when DMA engine **120** detects a packet in packet queue **116**, it may fetch the corresponding information in the control queue and act appropriately.

An operation code may indicate whether a packet is compatible with the protocols pre-selected for NIC **100**. In an illustrative embodiment of the invention, only compatible packets are eligible for data re-assembly and/or other enhanced operations offered by NIC **100** (e.g., packet batching or load distribution). An operation code may also reflect the size of a packet (e.g., less than or greater than a predetermined size), whether a packet contains data or is a control packet, and whether a packet initiates, continues or ends a flow. In this embodiment of the invention, eight different operation codes are used. In alternative embodiments of the invention more or less than eight codes may be used. TABLE 1 lists operation codes that may be used in one embodiment of the invention.

FIGS. **12A–12B** illustrate descriptors from a free descriptor ring and a completion descriptor ring in one embodiment of the invention. FIG. **12A** also depicts a free buffer array for storing buffer identifiers retrieved from free descriptors.

Free descriptor ring **1200** is maintained in host memory and is populated with descriptors such as free descriptor **1202**. Illustratively, free descriptor **1202** comprises ring index **1204**, the index of descriptor **1202** in free ring **1200**, and buffer identifier **1206**. A buffer identifier in this embodiment is a memory address, but may, alternatively, comprise a pointer or any other suitable means of identifying a buffer in host memory.

In the illustrated embodiment, free buffer array **1210** is constructed by software operating on a host computer (e.g., a device driver). An entry in free buffer array **1210** in this embodiment includes array index field **1212**, which may be used to identify the entry, and buffer identifier field **1214**. Each entry's buffer identifier field thus stores a buffer identifier retrieved from a free descriptor in free descriptor ring **1200**.

In one embodiment of the invention, free ring manager **1012** of DMA engine **120** retrieves descriptor **1202** from the ring and stores buffer identifier **1206** in free buffer array **1210**. The free ring manager also passes the buffer identifier to flow re-assembly table **1004**, header table **1006**, MTU table **1008** or jumbo table **1010** as needed. In another embodiment the free ring manager extracts descriptors from the free descriptor ring and stores them in a descriptor cache until a buffer is needed, at which time the buffer's buffer identifier is stored in the free buffer array. In yet another embodiment, a descriptor may be used (e.g., the buffer that it references may be used to store a packet) while still in the cache.

In one embodiment of the invention descriptor **1202** is sixteen bytes in length. In this embodiment, ring index **1204** is thirteen bits in size, buffer identifier **1206** (and buffer identifier field **1214** in free buffer array **1210**) is sixty-four bits, and the remaining space may store other information or may not be used.

The size of array index field **1212** depends upon the dimensions of array **1210**; in one embodiment the field is thirteen bits in size.

Completion descriptor ring **1220** is also maintained in host memory. Descriptors in completion ring **1220** are written or configured when a packet is transferred to the host computer by DMA engine **120**. The information written to a descriptor, such as descriptor **1222**, is used by software operating on the host computer (e.g., a driver associated with

NIC **100**) to process the packet. Illustratively, an ownership indicator (described below) in the descriptor indicates whether DMA engine **120** has finished using the descriptor. For example, this field may be set to a particular value (e.g., zero) when the DMA engine finishes using the descriptor and a different value (e.g., one) when it is available for use by the DMA engine. However, in another embodiment of the invention, DMA engine **120** issues an interrupt to the host computer when it releases a completion descriptor. Yet another means of alerting the host computer may be employed in an alternative embodiment. Descriptor **1222**, in one embodiment of the invention, is thirty-two bytes in length.

In the illustrated embodiment of the invention, information stored in descriptor **1222** concerns a transferred packet and/or the buffer it was stored in, and includes the following fields. Data size **1230** reports the amount of data in the packet (e.g., in bytes). The data size field may contain a zero if there is no data portion in the packet or no data buffer (e.g., flow re-assembly buffer, non-re-assembly buffer, jumbo buffer, MTU buffer) was used. Data buffer index **1232** is the index, within free buffer array **1210**, of the buffer identifier for the flow re-assembly buffer, non-re-assembly buffer, jumbo buffer or MTU buffer in which the packet's data was stored. When the descriptor corresponds to a small packet fully stored in a header buffer, this field may store a zero or remain unused. Data offset **1234** is the offset of the packet's data within the flow re-assembly buffer, non-re-assembly buffer, jumbo buffer or MTU buffer (e.g., the location of the first byte of data within the data buffer).

In FIG. **12**B, flags field **1236** includes one or more flags concerning a buffer or packet. For example, if a header buffer or data is being released (e.g., because it is full), a release header or release data flag, respectively, is set. A release flow flag may be used to indicate whether a flow has, at least temporarily, ended. In other words, if a release flow flag is set (e.g., stores a value of one), this indicates that there are no other packets waiting in the packet queue that are in the same flow as the packet associated with descriptor **1222**. Otherwise, if this flag is not set (e.g., stores a value of zero), software operating on the host computer may queue this packet to await one or more additional flow packets so that they may be processed collectively. A split flag may be included in flags field **1236** to identify whether a packet's contents (e.g., data) spans multiple buffers. Illustratively, if the split flag is set, there will be an entry in next data buffer index **1240**, described below.

Descriptor type **1238**, in the presently described embodiment of the invention, may take any of three values. A first value (e.g., one) indicates that DMA engine **120** is releasing a flow buffer for a flow that is stale (e.g., no packet has been received in the flow for some period of time). A second value (e.g., two) may indicate that a non-re-assembleable packet was stored in a buffer. A third value (e.g., three) may be used to indicate that a flow packet (e.g., a packet that is part of a flow through NIC **100**) was stored in a buffer.

Next buffer index **1240** stores an index, in free buffer array **1210**, of an entry containing a buffer identifier corresponding to a buffer storing a subsequent portion of a packet if the entire packet, or its data, could not fit into the first assigned buffer. The offset in the next buffer may be assumed to be zero. Header size **1242** reports the length of the header (e.g., in bytes). The header size may be set to zero if the header buffer was not used for this packet (e.g., the packet is not being re-assembled and is not a small packet). Header buffer index **1244** is the index, in free buffer array **1210**, of the buffer identifier for the header buffer used to store this packet's header. Header offset **1246** is the offset of the packet's header within the buffer (e.g., header buffer) in which the header was stored. The header offset may take the form of a number of bytes into the buffer at which the header can be found. Alternatively, the offset may be an index value, reporting the index position of the header. For example, in one embodiment of the invention mentioned above, entries in a header buffer are stored in 256-byte units. Thus, each entry begins at a 256-byte boundary regardless of the actual size of the entries. The 256-byte entries may be numbered or indexed within the buffer.

In the illustrated embodiment, flow number **1250** is the packet's flow number (e.g., the index in flow database **110** of the packet's flow key). Flow number **1250** may be used to identify packets in the same flow. Operation code **1252** is a code generated by flow database manager **108**, as described in a previous section, and used by DMA engine **120** to process the packet and transfer it into an appropriate buffer. Methods of transferring a packet depending upon its operation code are described in detail in the following section. No_Assist signal **1254**, also described in a previous section, may be set or raised when the packet is not compatible with the protocols pre-selected for NIC **100**. One result of incompatibility is that header parser **106** may not extensively parse the packet, in which case the packet will not receive the subsequent benefits. Processor identifier **1256**, which may be generated by load distributor **112**, identifies a host computer system processor for processing the packet. As described in a previous section, load distributor **112** attempts to share or distribute the load of processing network packets among multiple processors by having all packets within one flow processed by the same processor. Layer three header offset **1258** reports an offset within the packet of the first byte of the packet's layer three protocol (e.g., IP) header. With this value, software operating on the host computer may easily strip off one or more headers or header portions.

Checksum value **1260** is a checksum computed for this packet by checksum generator **114**. Packet length **1262** is the length (e.g., in bytes) of the entire packet.

Ownership indicator **1264** is used in the presently described embodiment of the invention to indicate whether NIC **100** or software operating on the host computer "owns" completion descriptor **1222**. In particular, a first value (e.g., zero) is placed in the ownership indicator field when NIC **100** (e.g., DMA engine **120**) has completed configuring the descriptor. Illustratively, this first value is understood to indicate that the software may now process the descriptor. When finished processing the descriptor, the software may store a second value (e.g., one) in the ownership indicator to indicate that NIC **100** may now use the descriptor for another packet.

One skilled in the art will recognize that there are numerous methods that may be used to inform host software that a descriptor has been used by, or returned to, DMA engine **120**. In one embodiment of the invention, for example, one or more registers, pointers or other data structures are maintained to indicate which completion descriptors in a completion descriptor ring have or have not been used. In particular, a head register may be used to identify a first of a series of descriptors that are owned by host software, while a tail register identifies the last descriptor in the series. DMA engine **120** may update these registers as it configures and releases descriptors. Thus, by examining these registers the host software and the DMA engine can determine how many descriptors have or have not been used.

Finally, other information, flags and indicators may be stored in other field **1266**. Other information that may be

stored in one embodiment of the invention includes the length and/or offset of a TCP payload, flags indicating a small packet (e.g., less than 257 bytes) or a jumbo packet (e.g., more than 1522 bytes), a flag indicating a bad packet (e.g., CRC error), a checksum starting position, etc.

In alternative embodiments of the invention only information and flags needed by the host computer (e.g., driver software) are included in descriptor **1222**. Thus, in one alternative embodiment one or more fields other than the following may be omitted: data size **1230**, data buffer index **1232**, data offset **1234**, a split flag, next data buffer index **1240**, header size **1242**, header buffer index **1244**, header offset **1246** and ownership indicator **1264**.

In addition, a completion descriptor may be organized in virtually any form; the order of the fields of descriptor **1222** in FIG. **12** is merely one possible configuration. It is advantageous, however, to locate ownership indicator **1264** towards the end of a completion descriptor since this indicator may be used to inform host software when the DMA engine has finished populating the descriptor. If the ownership indicator were placed in the beginning of the descriptor, the software may read it and attempt to use the descriptor before the DMA engine has finished writing to it.

One skilled in the art will recognize that other systems and methods than those described in this section may be implemented to identify storage areas in which to place packets being transferred from a network to a host computer without exceeding the scope of the invention.

Methods of Transferring a Packet into a Memory Buffer by a DMA Engine

FIGS. **13–20** are flow charts describing procedures for transferring a packet into a host memory buffer. In these procedures, a packet's operation code helps determine which buffer or buffers the packet is stored in. An illustrative selection of operation codes that may be used in this procedure are listed and explained in TABLE 1.

The illustrated embodiments of the invention employ four categories of host memory buffers, the sizes of which are programmable. The buffer sizes are programmable in order to accommodate various host platforms, but are programmed to be one memory page in size in present embodiments in order to enhance the efficiency of handling and processing network traffic. For example, the embodiments discussed in this section are directed to the use of a host computer system employing a SPARC™ processor, and so each buffer is eight kilobytes in size. These embodiments are easily adjusted, however, for host computer systems employing memory pages having other dimensions.

One type of buffer is for re-assembling data from a flow, another type is for headers of packets being re-assembled and for small packets (e.g., those less than or equal to 256 bytes in size) that are not re-assembled. A third type of buffer stores packets up to MTU size (e.g., 1522 bytes) that are not re-assembled, and a fourth type stores jumbo packets that are greater than MTU size and which are not re-assembled. These buffers are called flow re-assembly, header, MTU and jumbo buffers, respectively.

The procedures described in this section make use of free descriptors and completion descriptors as depicted in FIG. **12**. In particular, in these procedures free descriptors retrieved from a free descriptor ring store buffer identifiers (e.g., memory addresses, pointers) for identifying buffers in which to store a portion of a packet. A used buffer may be returned to a host computer by identifying the location within a free buffer array or other data structure used to store the buffer's buffer identifier. One skilled in the art will recognize that these procedures may be readily adapted to work with alternative methods of obtaining and returning buffers for storing packets.

FIG. **13** is a top-level view of the logic controlling DMA engine **120** in this embodiment of the invention. State **1300** is a start state.

In state **1302**, a packet is stored in packet queue **116** and associated information is stored in control queue **118**. One embodiment of a packet queue is depicted in FIG. **8** and one embodiment of a control queue is depicted in FIG. **9**. DMA engine **120** may detect the existence of a packet in packet queue **116** by comparing the queue's read and write pointers. As long as they do not reference the same entry, then it is understood that a packet is stored in the queue. Alternatively, DMA engine **120** may examine control queue **118** to determine whether an entry exists there, which would indicate that a packet is stored in packet queue **116**. As long as the control queue's read and write pointers do not reference the same entry, then an entry is stored in the control queue and a packet must be stored in the packet queue.

In state **1304**, the packet's associated entry in the control queue is read. Illustratively, the control queue entry includes the packet's operation code, the status of the packet's No_Assist signal (e.g., indicating whether or not the packet is compatible with a pre-selected protocol), one or more indicators concerning the size of the packet (and/or its data portion), etc.

In state **1306**, DMA engine **120** retrieves the packet's flow number. As described previously, a packet's flow number is the index of the packet's flow in flow database **110**. A packet's flow number may, as described in a following section, be provided to and used by dynamic packet batching module **122** to enable the collective processing of headers from related packets. In one embodiment of the invention, a packet's flow number may be provided to any of a number of NIC modules (e.g., IPP module **104**, packet batching module **122**, DMA engine **120**, control queue **118**) after being generated by flow database manager **108**. The flow number may also be stored in a separate data structure (e.g., a register) until needed by dynamic packet batching module **122** and/or DMA engine **120**. In one embodiment of the invention DMA engine **120** retrieves a packet's flow number from dynamic packet batching module **122**. In an alternative embodiment of the invention, the flow number may be retrieved from a different location or module.

Then, in states **1308–1318**, DMA engine **120** determines the appropriate manner of processing the packet by examining the packet's operation code. The operation code may, for example, indicate which buffer the engine should transfer the packet into and whether a flow is to be set up or torn down in flow re-assembly buffer table **1004**.

The illustrated procedure continues at state **1400** (FIG. **14**) if the operation code is 0, state **1500** (FIG. **15**) for operation code 1, state **1600** (FIG. **16**) for operation code 2, state **1700** (FIG. **17**) for operation code 3, state **1800** (FIG. **18**) for operation code 4, state **1900** (FIG. **19**) for operation code 5 and state **2000** (FIG. **20**) for operation codes 6 and 7.

A Method of Transferring a Packet with Operation Code 0

FIG. **14** depicts an illustrative procedure in which DMA engine **120** transfers a packet associated with operation code 0 to a host memory buffer. As reflected in TABLE 1, operation code 0 indicates in this embodiment that the packet is compatible with the protocols that may be parsed by NIC **100**. As explained above, compatible packets are eligible for re-assembly, such that data from multiple packets of one flow may be stored in one buffer that can then be efficiently provided (e.g., via a page-flip) to a user or

small packet. The processing associated with a packet having operation code 0 then ends with end state **1499**. In one embodiment of the invention, the ownership indicator field of a descriptor that is written in state **1406** is not changed, or an interrupt is not issued, until end state **1499**. Delaying the notification of the host computer allows the descriptor to be updated or modified for as long as possible before turning it over to the host.

A Method of Transferring a Packet with Operation Code 1

FIG. **15** depicts an illustrative procedure in which DMA engine **120** transfers a packet associated with operation code 1 to a host memory buffer. As reflected in TABLE 1, in this embodiment operation code 1 indicates that the packet is compatible with the protocols that may be parsed by NIC **100**. A packet having operation code 1, however, may be a control packet having a particular flag set. No new flow is set up, but a flow should already exist and is to be torn down; there is no data to re-assemble and the entire packet may be stored in a header buffer.

In state **1500**, DMA engine **120** (e.g., DMA manager **1002**) determines whether there is a valid (e.g., active) header buffer. Illustratively, this determination is made by examining validity indicator **1116** of header buffer table **1006**, which manages the active header buffer. If the validity indicator is set, then there is a header buffer ready to receive this packet and the procedure continues at state **1504**.

Otherwise, in state **1502** a new header buffer is prepared or initialized for storing small packets and headers of re-assembled packets. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, its buffer identifier (e.g., pointer, address, index) is stored in free buffer array **1210** and its initial storage location (e.g., address or cell location) is stored in next address field **1114** of header buffer table **1006**. The index or position of the buffer identifier within the free buffer array is stored in header buffer index **1112**. Finally, validity indicator **1116** is set to a valid state.

In state **1504** the packet is copied into the header buffer at the address or location specified in the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes are inserted before the packet in order to align the beginning of the packet's layer three (e.g., IP) header with a sixteen-byte boundary. And, the packet (with or without padding) may be placed into a pre-defined area or cell of the buffer.

In the illustrated embodiment, operation code 1 indicates that the packet's existing flow is to be torn down. Thus, in state **1506** it is determined whether a flow re-assembly buffer is valid (e.g., active) for this flow by examining the flow's validity indicator in flow re-assembly buffer table **1004**. If, for example, the indicator is valid, then there is an active buffer storing data from one or more packets in this flow. Illustratively, the flow is torn down by invalidating the flow re-assembly buffer and releasing it to the host computer. If there is no valid flow re-assembly buffer for this flow, the illustrated procedure continues at state **1512**. Otherwise, the procedure proceeds to state **1508**.

In state **1508**, a completion descriptor is configured to release the flow's re-assembly buffer and to provide information to the host computer for processing the current packet. In particular, the header buffer index and the offset of the first byte of the packet (or location of the packet's cell) within the header buffer are placed in the descriptor. The index within the free buffer array of the entry containing the re-assembly buffer's buffer identifier is stored in a data index field of the descriptor. The size of the packet is stored in a header size field and a data size field is set to zero to indicate that no separate buffer was used for storing this packet's data. A release header flag is set in the descriptor if the header buffer is full and a release data flag is set to indicate that no more data will be placed in this flow's present re-assembly buffer (e.g., it is being released). In addition, a release flow flag is set to indicate that DMA engine **120** is tearing down the packet's flow. The header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release header flag may be set at that time.

In state **1510**, the flow's entry in flow re-assembly buffer table **1004** is invalidated. After state **1510**, the procedure continues at state **1514**.

In state **1512**, a completion descriptor is configured with information somewhat different than that of state **1508**. In particular, the header buffer index, the offset to this packet within the header buffer and the packet size are placed within the same descriptor fields as above. The data size field is set to zero, as above, but no data index needs to be stored and no release data flag is set (e.g., because there is no flow re-assembly buffer to release). A release header flag is still set in the descriptor if the header buffer is full and a release flow flag is again set to indicate that DMA engine **120** is tearing down the packet's flow. Also, the descriptor type field is changed to a value indicating that DMA engine **120** transferred a flow packet into host memory.

In state **1514**, it is determined whether the header buffer is now full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, a counter is used to keep track of entries placed into each new header buffer. The buffer is considered full when thirty-two entries are stored.

If the buffer is full, in state **1516** the header buffer is invalidated.

Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer via the descriptor configured in state **1508** or state **1512**. In this embodiment of the invention a release header flag in the descriptor is set to indicate that the header buffer is full.

If the header buffer is not full, then in state **1518** the next address field of header buffer table **1006** is updated to indicate the address at which to store the next header or small packet.

The processing associated with a packet having operation code 1 then ends with end state **1599**. In this end state, the descriptor used for this packet is turned over to the host computer by changing its ownership indicator field (e.g., from one to zero), issuing an interrupt, or some other mechanism.

One skilled in the art will appreciate that in an alternative embodiment of the invention a change in the descriptor type field to any value other than the value (e.g., zero) it had when DMA engine **120** was using it, may constitute a surrender of "ownership" of the descriptor to the host computer or software operating on the host computer. The host computer will detect the change in the descriptor type field and subsequently use the stored information to process the packet.

A Method of Transferring a Packet with Operation Code 2

FIGS. **16A–16F** illustrate a procedure in which DMA engine **120** transfers a packet associated with operation code

2 to a host memory buffer. As reflected in TABLE 1, operation code 2 may indicate that the packet is compatible with the protocols that may be parsed by NIC **100**, but that it is out of sequence with another packet in the same flow. It may also indicate an attempt to re-establish a flow, but that no more data is likely to be received after this packet. For operation code 2, no new flow is set up and any existing flow with the packet's flow number is to be torn down. The packet's data is not to be re-assembled with data from other packets in the same flow.

Because an existing flow is to be torn down (e.g., the flow's re-assembly buffer is to be invalidated and released to the host computer), in state **1600** it is determined whether a flow re-assembly buffer is valid (e.g., active) for the flow having the flow number that was read in state **1306**. This determination may be made by examining the validity indicator in the flow's entry in flow re-assembly buffer table **1004**. Illustratively, if the indicator is valid then there is an active buffer storing data from one or more packets in the flow. If there is a valid flow re-assembly buffer for this flow, the illustrated procedure continues at state **1602**. Otherwise, the procedure proceeds to state **1606**.

In state **1602**, a completion descriptor is written or configured to release the existing flow re-assembly buffer. In particular, the flow re-assembly buffer's index (e.g., the location within the free buffer array that contains the buffer identifier corresponding to the flow re-assembly buffer) is written to the descriptor. In this embodiment of the invention, no offset needs to be stored in the descriptor's data offset field and the data size field may be set to zero because no new data was stored in the re-assembly buffer. Similarly, the header buffer is not yet being released, therefore the header index and header offset fields of the descriptor need not be used and a zero may be stored in the header size field.

Illustratively, the descriptor's release header flag is cleared (e.g., a zero is stored in the flag) because the header buffer is not to be released. The release data flag is set (e.g., a one is stored in the flag), however, because no more data will be placed in the released flow re-assembly buffer. Further, a release flow flag in the descriptor is also set, to indicate that the flow associated with the released flow re-assembly buffer is being torn down.

The descriptor type field may be changed to a value indicating that DMA engine **120** is releasing a stale flow buffer (e.g., a flow re-assembly buffer that has not been used for some time). Finally, the descriptor is turned over to the host computer by changing its ownership indicator field or by issuing an interrupt or using some other mechanism. In one embodiment of the invention, however, the descriptor is not released to the host computer until end state **1699**.

Then, in state **1604**, the flow re-assembly buffer is invalidated by modifying validity indicator **1106** in the flow's entry in flow re-assembly buffer table **1004** appropriately.

In state **1606**, it is determined whether the present packet is a small packet (e.g., less than or equal to 256 bytes in size), suitable for storage in a header buffer. If so, the illustrated procedure proceeds to state **1610**. Information stored in packet queue **116** and/or control queue **118** may be used to make this determination.

In state **1608**, it is determined whether the present packet is a jumbo packet (e.g., greater than 1522 bytes in size), such that it should be stored in a jumbo buffer. If so, the illustrated procedure proceeds to state **1650**. If not, the procedure continues at state **1630**.

In state **1610** (reached from state **1606**), it has been determined that the present packet is a small packet suitable for storage in a header buffer. Therefore, DMA engine **120** (e.g., DMA manager **1002**) determines whether there is a valid (e.g., active) header buffer. Illustratively, this determination is made by examining validity indicator **1116** of header buffer table **1006**, which manages the active header buffer. If the validity indicator is set, then there should be a header buffer ready to receive this packet and the procedure continues at state **1614**.

Otherwise, in state **1612** a new header buffer is prepared or initialized for storing small packets and headers of re-assembled packets. This initialization process may involve obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or some other indicator of the first storage location in the buffer is placed in next address field **1114** of header buffer table **1006**. The buffer identifier's position or index within the free buffer array is stored in header buffer index **1112**, and validity indicator **1116** is set to a valid state.

In state **1614** the packet is copied or transferred (e.g., via a DMA operation) into the header buffer at the address or location specified in the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes are inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In addition, the packet may be positioned within a cell of predetermined size (e.g., 256 bytes) within the header buffer.

In state **1616**, a completion descriptor is written or configured to provide necessary information to the host computer (e.g., a software driver) for processing the packet. In particular, the header buffer index (e.g. the position within the free buffer array of the header buffer's buffer identifier) and the packet's offset within the header buffer are placed in the descriptor. Illustratively, this offset may serve to identify the first byte of the packet, the first pad byte before the packet or the beginning of the packet's cell within the buffer. The size of the packet is also stored in the descriptor in a header size field. A data size field within the descriptor may be set to zero to indicate that the entire packet was placed in the header buffer (e.g., no separate data portion was stored). A release header flag is set in the descriptor if the header buffer is full. However, the header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release header flag may be set (or cleared) at that time. A release data flag is cleared (e.g., set to a value of zero), because there is no separate data portion being conveyed to the host computer.

Also, the descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. And, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention the ownership indicator field is not changed until end state **1699** below. In one alternative embodiment of the invention, DMA engine **120** issues an interrupt or other signal to alert the host computer that a descriptor is being released.

In state **1618**, it is determined whether the header buffer is full. In this embodiment of the invention, where each

buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, a counter may be used to keep track of entries placed into each new header buffer. The buffer is considered full when thirty-two entries are stored.

If the buffer is full, in state **1620** the header buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to an invalid state and communicating this status to the host computer. In this embodiment of the invention, a release header flag in the descriptor is set. The illustrated procedure then ends with end state **1699**.

If the header buffer is not full, then in state **1622** the next address field of header buffer table **1006** is updated to indicate the address or cell boundary at which to store the next header or small packet. The illustrated procedure then ends with end state **1699**.

In state **1630** (reached from state **1608**), it has been determined that the packet is not a small packet or a jumbo packet. The packet may, therefore, be stored in a non-re-assembly buffer (e.g., an MTU buffer) used to store packets that are up to MTU in size (e.g., 1522 bytes). Thus, in state **1630** DMA engine **120** determines whether a valid (e.g., active) MTU buffer exists. Illustratively, this determination is made by examining validity indicator **1126** of MTU buffer table **1008**, which manages an active MTU buffer. If the validity indicator is set, then there is an MTU buffer ready to receive this packet and the procedure continues at state **1634**.

Otherwise, in state **1632** a new MTU buffer is prepared or initialized for storing non-re-assembleable packets up to 1522 bytes in size. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer (e.g., a buffer identifier). If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in the free buffer array. The buffer's initial address or some other indication of the first storage location in the buffer is placed in next address field **1124** of MTU buffer table **1008**. Further, the position of the buffer identifier within the free buffer array is stored in MTU buffer index **1122** and validity indicator **1126** is set to a valid state.

In state **1634** the packet is copied or transferred (e.g., via a DMA operation) into the MTU buffer at the address or location specified in the next address field. As described above, in one embodiment of the invention pad bytes may be inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In another embodiment of the invention packets may be aligned in an MTU buffer in cells of predefined size (e.g., two kilobytes), similar to entries in a header buffer.

In state **1636**, a completion descriptor is written or configured to provide necessary information to the host computer (e.g., a software driver) for processing the packet. In particular, the MTU buffer index (e.g. the free buffer array element that contains the buffer identifier for the MTU buffer) and offset (e.g., the offset of the first byte of this packet within the MTU buffer) are placed in the descriptor in data index and data offset fields, respectively. The size of the packet is also stored in the descriptor, illustratively within a data size field. A header size field within the descriptor is set to zero to indicate that the entire packet was placed in the MTU buffer (e.g., no separate header portion was stored in a header buffer). A release data flag is set in the descriptor if the MTU buffer is full. However, the MTU buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release data flag may be set (or cleared) at that time. A release header flag is cleared (e.g., set to zero), because there is no separate header portion being conveyed to the host computer.

Further, the descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. Also, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In a present embodiment of the invention the ownership field is not set until end state **1699** below. In one alternative embodiment of the invention, DMA engine **120** issues an interrupt or other signal to alert the host computer that a descriptor is being released, or communicates this event to the host computer through the descriptor type field.

In state **1638**, it is determined whether the MTU buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the MTU buffer are allotted two kilobytes, a counter may be used to keep track of entries placed into each new header buffer. The buffer may be considered full when a predetermined number of entries (e.g., four) are stored. In an alternative embodiment of the invention DMA engine **120** determines how much storage space within the buffer has yet to be used. If no space remains, or if less than a predetermined amount of space is still available, the buffer may be considered full.

If the MTU buffer is full, in state **1640** it is invalidated to ensure that it is not used again. Illustratively, this involves setting the MTU buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention, a release data flag in the descriptor is set. The illustrated procedure then ends with end state **1699**.

If the MTU buffer is not full, then in state **1642** the next address field of MTU buffer table **1008** is updated to indicate the address or location (e.g., cell boundary) at which to store the next packet. The illustrated procedure then ends with end state **1699**.

In state **1650** (reached from state **1608**), it has been determined that the packet is a jumbo packet (e.g., that it is greater than 1522 bytes in size). In this embodiment of the invention jumbo packets are stored in jumbo buffers and, if splitting of jumbo packets is enabled (e.g., as determined in state **1654** below), headers of jumbo packets are stored in a header buffer. DMA engine **120** determines whether a valid (e.g., active) jumbo buffer exists. Illustratively, this determination is made by examining validity indicator **1136** of jumbo buffer table **1010**, which manages the active jumbo buffer. If the validity indicator is set, then there is a jumbo buffer ready to receive this packet and the procedure continues at state **1654**. As explained above, a jumbo buffer table may not be used in an embodiment of the invention in which a jumbo buffer is used only once (e.g., to store just one, or just part of one, jumbo packet).

Otherwise, in state **1652** a new jumbo buffer is prepared or initialized for storing a non-re-assembleable packet that is larger than 1522 bytes. This initialization process may involve obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer (e.g., a buffer identifier). If the cache is

empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, its buffer identifier (e.g., pointer, address, index) is stored in a free buffer array (or other data structure). The buffer's initial address or other indication of the first storage location in the buffer is placed in next address field **1134** of jumbo buffer table **1010**. Also, the location of the buffer identifier within the free buffer array is stored in jumbo buffer index **1132** and validity indicator **1136** is set to a valid state.

Then, in state **1654** DMA engine **120** determines whether splitting of jumbo buffers is enabled. If enabled, the header of a jumbo packet is stored in a header buffer while the packet's data is stored in one or more jumbo buffers. If not enabled, the entire packet will be stored in one or more jumbo buffers. Illustratively, splitting of jumbo packets is enabled or disabled according to the configuration of a programmable indicator (e.g., flag, bit, register) that may be set by software operating on the host computer (e.g., a device driver). If splitting is enabled, the illustrated procedure continues at state **1670**. Otherwise, the procedure continues with state **1656**.

In state **1656**, DMA engine **120** determines whether the packet will fit into one jumbo buffer. For example, in an embodiment of the invention using eight kilobyte pages, if the packet is larger than eight kilobytes a second jumbo buffer will be needed to store the additional contents. If the packet is too large, the illustrated procedure continues at state **1662**.

In state **1658**, the packet is copied or transferred (e.g., via a DMA operation) into the current jumbo buffer, at the location specified in the next address field **1134** of jumbo buffer table **1010**. When the packet is transferred intact like this, padding may be added to align a header portion of the packet with a sixteen-byte boundary. One skilled in the art will appreciate that the next address field may not need to be updated to account for this new packet because the jumbo buffer will be released. In other words, in one embodiment of the invention a jumbo buffer may be used just once (e.g., to store one packet or a portion of one packet).

In state **1660**, a completion descriptor is written or configured to release the jumbo buffer and to provide information to the host computer for processing the packet. The jumbo buffer index (e.g., the position within the free buffer array of the buffer identifier for the jumbo buffer) and the offset of the packet within the jumbo buffer are placed in the descriptor. Illustratively, these values are stored in data index and data offset fields, respectively. The size of the packet (e.g., the packet length) may be stored in a data size field.

A header size field is cleared (e.g., a zero is stored) to indicate that the header buffer was not used (e.g., the header was not stored separately from the packet's data). Because there is no separate packet header, header index and header offset fields are not used or are set to zero (e.g., the values stored in their fields do not matter). A release header flag is cleared and a release data flag is set to indicate that no more data will be placed in this jumbo buffer (e.g., because it is being released).

Also, the descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-reassembleable packet into host memory. And, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In an alternative embodiment, the descriptor may be released by issuing an interrupt or other alert. In yet another embodiment, changing the descriptor type field (e.g., to a non-zero value) may signal the release of the descriptor. In one embodiment of the invention the ownership indicator is not set until end state **1699** below. After state **1660**, the illustrated procedure resumes at state **1668**.

In state **1662**, a first portion of the packet is stored in the present (e.g., valid) jumbo buffer, at the location identified in the buffer's next address field **1134**. Then, because the fill packet will not fit into this buffer, in state **1664** a new jumbo buffer is prepared and the remainder of the packet is stored in that buffer.

In state **1666**, a completion descriptor is written or configured. The contents are similar to those described in state **1660** but this descriptor must reflect that two jumbo buffers were used to store the packet.

Thus, the jumbo buffer index (e.g., the index, within the free buffer array, of the buffer identifier that identifies the header buffer) and the offset of the packet within the first jumbo buffer are placed in the descriptor, as above. The size of the packet (e.g., the packet length) is stored in a data size field.

A header size field is cleared (e.g., a zero is stored) to indicate that the header buffer was not used (e.g., the header was not stored separately from the packet's data). Because there is no separate packet header, header index and header offset fields are not used (e.g., the values stored in their fields do not matter).

A release header flag is cleared and a release data flag is set to indicate that no more data will be placed in these jumbo buffers (e.g., because they are being released). Further, a split packet flag is set to reflect the use of a second jumbo buffer, and the index (within the free buffer array) of the buffer identifier for the second buffer is stored in a next index field.

Further, the descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-reassembleable packet into host memory. Finally, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field, or some other mechanism is employed, to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention, the descriptor is not released to the host computer until end state **1699** below.

In state **1668**, the jumbo buffer entry or entries in jumbo buffer table **1010** are invalidated (e.g., validity indicator **1136** is set to invalid) to ensure that they are not used again. In the procedure described above a jumbo packet was stored in, at most, two jumbo buffers. In an alternative embodiment of the invention a jumbo buffer may be stored across any number of buffers. The descriptor(s) configured to report the transfer of such a packet is/are constructed accordingly, as will be obvious to one skilled in the art.

After state **1668**, the illustrated procedure ends with end state **1699**.

In state **1670** (reached from state **1654**), it has been determined that the present jumbo packet will be split to store the packet header in a header buffer and the packet data in one or more jumbo buffers. Therefore, DMA engine **120** (e.g., DMA manager **1002**) determines whether there is a valid (e.g., active) header buffer. Illustratively, this determination is made by examining validity indicator **1116** of header buffer table **1006**, which manages the active header buffer. If the validity indicator is set, then there is a header buffer ready to receive this packet and the procedure continues at state **1674**.

Otherwise, in state **1672** a new header buffer is prepared or initialized for storing small packets and headers of other packets. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or some other indication of the first storage location or cell in the buffer is placed in next address field **1114** of header buffer table **1006**. Also, the index of the buffer identifier within the free buffer array is stored in header buffer index **1112** and validity indicator **1116** is set to a valid state.

In state **1674** the packet's header is copied or transferred (e.g., via a DMA operation) into the header buffer at the address or location specified in the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes are inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In addition, the packet's header may be positioned within a cell of predetermined size (e.g., 256 bytes) within the buffer.

In state **1676**, DMA engine **120** determines whether the packet's data (e.g., the TCP payload) will fit into one jumbo buffer. If the packet is too large, the illustrated procedure continues at state **1682**.

In state **1678**, the packet's data is copied or transferred (e.g., via a DMA operation) into the current jumbo buffer, at the location specified in the next address field **1134** of jumbo buffer table **1010**. One skilled in the art will appreciate that the next address field may not need to be updated to account for this new packet because the jumbo buffer will be released. In other words, in one embodiment of the invention a jumbo buffer may be used just once (e.g., to store one packet or a portion of one packet).

In state **1680**, a completion descriptor is written or configured to release the jumbo buffer and to provide information to the host computer for processing the packet. The header buffer index (e.g. the index of the header buffer's buffer identifier within the free buffer array) and offset of the packet's header within the buffer are placed in the descriptor in header index and header offset fields, respectively. Illustratively, this offset may serve to identify the first byte of the header, the first pad byte before the header or the location of the cell in which the header is stored. The jumbo buffer index (e.g., the position or index within the free buffer array of the buffer identifier that identifies the jumbo buffer) and the offset of the first byte of the packet's data within the jumbo buffer are placed in data index and data offset fields, respectively. Header size and data size fields are used to store the size of the packet's header (e.g., the offset of the payload within the packet) and data (e.g., payload size), respectively.

A release header flag is set in the descriptor if the header buffer is full. However, the header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release header flag may be set (or cleared) at that time. A release data flag is also set, because no more data will be placed in the jumbo buffer (e.g., it is being released to the host computer).

The descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. Also, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention the ownership indicator is not changed until end state **1699** below. In an alternative embodiment, the descriptor may be released by issuing an interrupt or other alert. In yet another alternative embodiment, changing the descriptor type value may signal the release of the descriptor.

After state **1680**, the illustrated procedure proceeds to state **1688**.

In state **1682**, a first portion of the packet's data is stored in the present (e.g., valid) jumbo buffer, at the location identified in the buffer's next address field **1134**.

Because all of the packet's data will not fit into this buffer, in state **1684** a new jumbo buffer is prepared and the remainder of the packet is stored in that buffer.

In state **1686**, a completion descriptor is written or configured. The contents are similar to those described in states **1680** but this descriptor must reflect that two jumbo buffers were used to store the packet. The header buffer index (e.g. the index of the free buffer array element containing the header buffer's buffer identifier) and offset (e.g., the location of this packet's header within the header buffer) are placed in the descriptor in header index and header offset fields, respectively. The jumbo buffer index (e.g., the index, within the free buffer array, of the buffer identifier that references the jumbo buffer) and the offset of the first byte of the packet's data within the jumbo buffer are placed in data index and data offset fields, respectively. Header size and data size fields are used to store the size of the packet's header (e.g., as measured by the offset of the packet's payload from the start of the packet) and data (e.g., payload size), respectively.

A release header flag is set in the descriptor if the header buffer is full. However, the header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release header flag may be set (or cleared) at that time. A release data flag is also set, because no more data will be placed in the jumbo buffer (e.g., it is being released to the host computer). Further, a split packet flag is set to indicate that a second jumbo buffer was used, and the location (within the free buffer array or other data structure) of the second buffer's buffer identifier is stored in a next index field

The descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. Finally, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention the ownership indicator is not changed until end state **1699** below.

In state **1688**, the jumbo buffer's entry in jumbo buffer table **1010** is invalidated (e.g., validity indicator **1136** is set to invalid) to ensure that it is not used again. In the procedure described above, a jumbo packet was stored in, at most, two jumbo buffers. In an alternative embodiment of the invention a jumbo packet may be stored across any number of buffers. The descriptor that is configured to report the transfer of such a packet is constructed accordingly, as will be obvious to one skilled in the art.

In state **1690**, it is determined whether the header buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, a counter may be used

to keep track of entries placed into each new header buffer. The buffer may be considered full when thirty-two entries are stored.

If the buffer is full, in state **1692** the header buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention, a release header flag in the descriptor is set. The illustrated procedure then ends with end state **1699**.

If the header buffer is not full, then in state **1694** the next address field of header buffer table **1006** is updated to indicate the address at which to store the next header or small packet. The illustrated procedure then ends with end state **1699**.

In end state **1699**, a descriptor may be turned over to the host computer by changing a value in the descriptor's descriptor type field (e.g., from one to zero), as described above. Illustratively, the host computer (or software operating on the host computer) detects the change and understands that DMA engine **120** is returning ownership of the descriptor to the host computer.

A Method of Transferring a Packet with Operation Code 3

FIGS. **17A–17C** illustrate one procedure in which DMA engine **120** transfers a packet associated with operation code 3 to a host memory buffer. As reflected in TABLE 1, operation code 3 may indicate that the packet is compatible with a protocol that can be parsed by NIC **100** and that it carries a final portion of data for its flow. No new flow is set up, but a flow should already exist and is to be torn down. The packet's data is to be re-assembled with data from previous flow packets. Because the packet is to be re-assembled, the packet's header should be stored in a header buffer and its data in the flow's re-assembly buffer. The flow's active re-assembly buffer may be identified by the flow's entry in flow re-assembly buffer table **1004**.

In state **1700**, DMA engine **120** (e.g., DMA manager **1002**) determines whether there is a valid (e.g., active) header buffer. Illustratively, this determination is made by examining validity indicator **1116** of header buffer table **1006**, which manages the active header buffer. If the validity indicator is set (e.g., equal to one), then it is assumed that there is a header buffer ready to receive this packet and the procedure continues at state **1704**.

Otherwise, in state **1702** a new header buffer is prepared or initialized for storing small packets and headers of re-assembled packets. This initialization process may involve obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its buffer identifier (e.g., a reference to an available memory buffer). If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

Illustratively, when a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or some other indication of the first storage location or cell in the buffer is placed in next address field **1114** of header buffer table **1006**. Further, the index of the buffer identifier within the free buffer array is stored in header buffer index **1112** and validity indicator **1116** is set to a valid state.

In state **1704** the packet's header is copied or transferred into the header buffer at the address or location specified in the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes may be inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In addition, the header may be positioned within a cell of predetermined size (e.g., 256 bytes) within the header buffer.

In the illustrated embodiment, operation code 3 indicates that an existing flow is to be torn down (e.g., the flow re-assembly buffer is to be invalidated and released to the host computer). Thus, in state **1706** it is determined whether a flow re-assembly buffer is valid (e.g., active) for this flow by examining the validity indicator in the flow's entry in flow re-assembly buffer table **1004**. Illustratively, if the indicator is valid then there should be an active buffer storing data from one or more packets in this flow. If there is a valid flow re-assembly buffer for this flow, the illustrated procedure continues at state **1712**. Otherwise, the procedure proceeds to state **1708**.

In state **1708**, a new flow re-assembly buffer is prepared to store this packet's data. Illustratively, a free ring descriptor is obtained from a cache maintained by free ring manager **1012** and its reference to an empty buffer is retrieved. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or other indication of its first storage location is placed in next address field **1104** of the flow's entry in flow re-assembly buffer table **1004**. The flow's entry in the re-assembly buffer table may be recognized by its flow number. The location within the free buffer array of the buffer identifier is stored in re-assembly buffer index **1102**, and validity indicator **1106** is set to a valid state.

In state **1710**, the packet's data is copied or transferred (e.g., via a DMA operation) into the address or location specified in the next address field of the flow's entry in flow re-assembly buffer table **1004**.

In state **1712**, a completion descriptor is written or configured to release the flow's re-assembly buffer and to provide information to the host computer for processing the packet. In particular, the header buffer index (e.g., the index, within the free buffer array, of the header buffer's identifier) and the offset of the packet's header within the header buffer are placed in the descriptor. Illustratively, this offset serves to identify the first byte of the header, the first pad byte preceding the header or the cell in which the header is stored. The flow re-assembly buffer index (e.g., the index, within the free buffer array, of the flow re-assembly buffer's identifier) and the offset of the packet's data within that buffer are also stored in the descriptor.

The size of the packet's data (e.g., the size of the packet's TCP payload) and header (e.g., the offset of the TCP payload within the packet) portions are stored in data size and header size fields, respectively. The descriptor type field is given a value that indicates that a flow packet has been transferred to host memory. A release header flag may be set if the header buffer is full and a release data flag may be set to indicate that no more data will be placed in this flow re-assembly buffer (e.g., because it is being released). In addition, a release flow flag is set to indicate that DMA engine **120** is tearing down the packet's flow. The header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment, the release header flag may be set (or cleared) at that time.

Then, in state **1714**, the flow re-assembly buffer is invalidated by modifying validity indicator **1106** in the flow's entry in flow re-assembly buffer table **1004** appropriately. After state **1714**, the procedure continues at state **1730**.

In state **1716**, DMA engine **120** determines whether the packet's TCP payload (e.g., the packet's data portion) will fit into the valid flow re-assembly buffer. If not, the illustrated procedure continues at state **1722**.

In state **1718**, the packet data is copied or transferred (e.g., via a DMA operation) into the flow's re-assembly buffer, at the location specified in the next address field **1104** of the flow's entry in flow re-assembly table **1004**. One skilled in the art will appreciate that the next address field may or may not be updated to account for this new packet because the re-assembly buffer is being released.

In state **1720**, a completion descriptor is written or configured to release the flow's re-assembly buffer and to provide information to the host computer for processing the packet. The header buffer index (e.g., the location or index, within the free buffer array, of the header buffer's identifier) and the offset of the packet's header within the header buffer are placed in the descriptor. The flow re-assembly buffer index (e.g., the location or index within the free buffer array of the flow re-assembly buffer's identifier) and the offset of the packet's data within that buffer are also stored in the descriptor.

The size of the packet's data (e.g., the size of the packet's TCP payload) and header (e.g., the offset of the TCP payload within the packet) are stored in data size and header size fields, respectively. The descriptor type field is given a value that indicates that a flow packet has been transferred to host memory. A release header flag is set if the header buffer is full and a release data flag is set to indicate that no more data will be placed in this flow re-assembly buffer (e.g., because it is being released). As explained above, the header buffer may not be tested to see if it is full until a later state of this procedure, at which time the release header flag may be set. Finally, a release flow flag is set to indicate that DMA engine **120** is tearing down the packet's flow. After state **1720**, the illustrated procedure resumes at state **1728**.

In state **1722**, a first portion of the packet's payload (e.g., data) is stored in the flow's present (e.g., valid) re-assembly buffer, at the location identified in the buffer's next address field **1104**.

Because the full payload will not fit into this buffer, in state **1724** a new flow re-assembly buffer is prepared and the remainder of the payload is stored in that buffer. In one embodiment of the invention information concerning the first buffer is stored in a completion descriptor. This information may include the position within the free buffer array of the first buffer's buffer identifier and the offset of the first portion of data within the buffer. The flow's entry in flow re-assembly buffer table **1004** may then be updated for the second buffer (e.g., store a first address in next address field **1104** and the location of buffer's identifier in the free buffer array in re-assembly buffer index **1102**).

In state **1726**, a completion descriptor is written or configured. The contents are similar to those described for states **1712** and **1720** but this descriptor must reflect that two re-assembly buffers were used.

Thus, the header buffer index (e.g., the position within the free buffer array of the buffer identifier corresponding to the header buffer) and the offset of the packet's header within the header buffer are placed in the descriptor, as above. The first flow re-assembly buffer index (e.g., the position, within the free buffer array, of the buffer identifier corresponding to the first flow re-assembly buffer used to store this packet's payload) and the offset of the packet's first portion of data within that buffer are also stored in the descriptor.

The size of the packet's data (e.g., the size of the packet's TCP payload) and header (e.g., the offset of the TCP payload within the packet) are stored in data size and header size fields, respectively. The descriptor type field is given a value that indicates that a flow packet has been transferred to host memory. A release header flag is set if the header buffer is full and a release data flag is set to indicate that no more data will be placed in this flow re-assembly buffer. A release flow flag is set to indicate that DMA engine **120** is tearing down the packet's flow.

Because two re-assembly buffers were used, a split packet flag is set and the index, within the free buffer array, of the re-assembly buffer's buffer identifier is stored in a next index field. Additionally, because the packet contains the final portion of data for the flow, a release next data buffer flag may also be set to indicate that the second flow re-assembly buffer is being released.

In state **1728**, the flow's entry in flow re-assembly buffer table **1004** is invalidated to ensure that it is not used again.

In state **1730**, it is determined whether the header buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, a counter is used to keep track of entries placed into each new header buffer. The buffer is considered full when thirty-two entries are stored.

If the buffer is full, in state **1732** the header buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention a release header flag in the descriptor is set.

If the header buffer is not full, then in state **1734** the next address field of header buffer table **1006** is updated to indicate the address at which to store the next header or small packet.

The processing associated with a packet having operation code 3 then ends with end state **1799**. In this end state, the descriptor used for this packet is turned over to the host computer by changing its ownership indicator field (e.g., from one to zero). Alternatively, some other mechanism may be used, such as issuing an interrupt or changing the descriptor' descriptor type field. Illustratively, the descriptor type field would be changed to a value indicating that DMA engine **120** transferred a flow packet into host memory.

In one alternative embodiment of the invention an optimization may be performed when processing a packet with operation code 3. This optimization takes advantage of the knowledge that the packet contains the last portion of data for its flow. In particular, instead of loading a descriptor into flow re-assembly buffer table **1004** the descriptor may be used where it is—in a descriptor cache maintained by free ring manager **1012**.

For example, instead of retrieving a buffer identifier from a descriptor and storing it in an array in state **1708** above, only to store one packet's data in the identified buffer before releasing it, it may be more efficient to use the descriptor without removing it from the cache. In this embodiment, when a completion descriptor is written the values stored in its data index and data offset fields are retrieved from a descriptor in the descriptor cache. Similarly, when the first portion of a code 3 packet's data fits into the flow's active buffer but a new one is needed just for the remaining data, a descriptor in the descriptor cache may again be used without first loading it into a free buffer array and the flow re-assembly buffer table. In this situation, the completion descriptor's next index field is retrieved from the descriptor in the descriptor cache.

A Method of Transferring a Packet with Operation Code 4

FIGS. **18A–18D** depict an illustrative procedure in which DMA engine **120** transfers a packet associated with opera-

tion code 4 to a host memory buffer. As reflected in TABLE 1, operation code 4 in this embodiment indicates that the packet is compatible with the protocols that may be parsed by NIC **100** and continues a flow that is already established. No new flow is set up, the existing flow is not to be torn down, and the packet's data is to be re-assembled with data from other flow packets. Because the packet is to be re-assembled, the packet's header should be stored in a header buffer and its data in the flow's re-assembly buffer.

In state **1800**, DMA engine **120** determines whether there is a valid (e.g., active) header buffer. Illustratively, this determination is made by examining validity indicator **116** of header buffer table **1006**, which manages the active header buffer. If the validity indicator is set, then there should be a header buffer ready to receive this packet and the procedure continues at state **1804**.

Otherwise, in state **1802** a new header buffer is prepared or initialized for storing small packets and headers of re-assembled packets. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or some other indication of the first storage location in the buffer is place in next address field **1114** of header buffer table **1006**. Also, the position or index of the buffer identifier within the free buffer array is stored in header buffer index **1112** and validity indicator **1116** is set to a valid state.

In state **1804** the packet's header is copied or transferred into the header buffer at the address or location specified in the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes are inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In addition, the packet's header may be positioned within a cell of predetermined size (e.g., 256 bytes) within the buffer.

In the illustrated embodiment, operation code 4 indicates that an existing flow is to be continued. Thus, in state **1806** it is determined whether a flow re-assembly buffer is valid (e.g., active) for this flow by examining the validity indicator in the flow's entry in flow re-assembly buffer table **1004**. Illustratively, if the indicator is valid then there is an active buffer storing data from one or more packets in this flow. If there is a valid flow re-assembly buffer for this flow, the illustrated procedure continues at state **1808**. Otherwise, the procedure proceeds to state **1810**.

In state **1808**, it is determined whether the packet's data (e.g., its TCP payload) portion is too large for the current flow re-assembly buffer. If the data portion is too large, two flow re-assembly buffers will be used and the illustrated procedure proceeds to state **1830**. Otherwise, the procedure continues at state **1820**.

In state **1810**, because it was found (in state **1806**) that there was no valid flow re-assembly buffer for this packet, a new flow re-assembly buffer is prepared. Illustratively, a free ring descriptor is obtained from a cache maintained by free ring manager **1012** and its reference to an empty buffer is retrieved. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or other indicator of its first storage location is placed in next address field **1104** of the flow's entry in flow re-assembly buffer table **1004**. The flow's entry in the table may be recognized by its flow number. The location of the buffer identifier in the free buffer array is stored in re-assembly buffer index **1102**, and validity indicator **1106** is set to a valid state.

In state **1812**, the packet's data is copied or transferred (e.g., via a DMA operation) into the address or location specified in the next address field of the flow's entry in flow re-assembly buffer table **1004**.

In state **1814**, a completion descriptor is written or configured to provide information to the host computer for processing the packet. In particular, the header buffer index (e.g., the index within the free buffer array of the buffer identifier that identifies the header buffer) and the offset of the packet's header within the header buffer are placed in the descriptor. Illustratively, this offset may serve to identify the first byte of the header, the first pad byte preceding the header or the header's cell within the header buffer. The flow re-assembly buffer index (e.g., the index within the free buffer array of the buffer identifier that identifies the flow re-assembly buffer) and the offset of the packet's data within that buffer are also stored in the descriptor.

The size of the packet's data (e.g., the size of the packet's TCP payload) and header (e.g., the offset of the TCP payload within the packet) are stored in data size and header size fields, respectively. The descriptor type field is given a value indicating that a flow packet has been transferred to host memory. A release header flag is set if the header buffer is full but a release data flag is not set, because more data will be placed in this flow re-assembly buffer. The header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment, the release header flag may be set (or cleared) at that time.

In one embodiment of the invention a release flow flag may also be set, depending upon dynamic packet batching module **122**. For example, if the packet batching module determines that another packet in the same flow will soon be transferred to the host computer, the release flow flag will be cleared (e.g., a zero will be stored). This indicates that the host computer should await the next flow packet before processing this one. By collectively processing multiple packets from a single flow, the packets can be processed more efficiently and less processor time is required. If, however, no other packets in the same flow are identified, the release flow flag may be set (e.g., a one is stored) to indicate that the host computer should process the flow packets it has received so far, without waiting for more.

In state **1816**, the flow's entry in flow re-assembly buffer table **1004** is updated. In particular, next address field **1104** is updated to identify the. location in the re-assembly buffer at which the next flow packet's data should be stored. After state **1816**, the illustrated procedure continues at state **1838**.

In state **1820** (reached from state **1808**), it is known that the packet's data, or TCP payload, will fit within the flow's current re-assembly buffer. Thus, the packet data is copied or transferred into the buffer at the location identified in next address field **1104** of the flow's entry in flow re-assembly buffer table **1004**.

In state **1822**, a completion descriptor is written or configured to provide information to the host computer for processing the packet. In particular, the header buffer index (e.g., the index within the free buffer array of the buffer identifier that identifies the header buffer) and the offset of the packet's header within the header buffer are placed in the

descriptor. The flow re-assembly buffer index (e.g., the index within the free buffer array of the buffer identifier that identifies the flow re-assembly buffer) and the offset of the packet's data within that buffer are also stored in the descriptor.

The size of the packet's data (e.g., the size of the packet's TCP payload) and header (e.g., the offset of the TCP payload within the packet) are stored in data size and header size fields, respectively. The descriptor type field is given a value indicating that a flow packet has been transferred to host memory. A release header flag is set if the header buffer is full but a release data flag is set only if the flow re-assembly buffer is now full. The header and flow re-assembly buffers may not be tested to see if they are full until a later state of this procedure. In such an embodiment, the flags may be set (or cleared) at that time.

In one embodiment of the invention a release flow flag may also be set, depending upon dynamic packet batching module **122**. For example, if the packet batching module determines that another packet in the same flow will soon be transferred to the host computer, the release flow flag will be cleared. This indicates that the host computer should await the next flow packet before processing this one. By collectively processing multiple packets from a single flow, the packets can be processed more efficiently and less processor time is required. If, however, no other packets in the same flow are identified, the release flow flag may be set to indicate that the host computer should process the flow packets received so far, without waiting for more.

In state **1824**, the flow re-assembly buffer is examined to determine if it is full. In the presently described embodiment of the invention this test is conducted by first determining how much data (e.g., how many bytes) has been stored in the buffer. Illustratively, the flow's next address field and the amount of data stored from this packet are summed. Then, the initial buffer address (e.g., before any data was stored in it) is subtracted from this sum. This value, representing how much data is now stored in the buffer, is then compared to the size of the buffer (e.g., eight kilobytes).

If the amount of data currently stored in the buffer equals the size of the buffer, then it is full. In the presently described embodiment of the invention it is desirable to completely fill flow re-assembly buffers. Thus, a flow re-assembly buffer is not considered full until its storage space is completely populated with flow data. This scheme enables the efficient processing of network packets.

If the flow re-assembly buffer is full, in state **1826** the buffer is invalidated to ensure it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention, a release data flag in the descriptor is set. After state **1826**, the procedure continues at state **1838**.

If the flow re-assembly buffer is not full, then in state **1828** next address field **1104** in the flow's entry in flow re-assembly buffer table **1004** is updated to indicate the address at which to store the next portion of flow data. After state **1828**, the procedure continues at state **1838**.

In state **1830** (reached from state **1808**), it is known that the packet's data will not fit into the flow's current re-assembly buffer. Therefore, some of the data is stored in the current buffer and the remainder in a new buffer. In particular, in state **1830** a first portion of data (e.g., an amount sufficient to fill the buffer) is copied or transferred into the current flow re-assembly buffer.

In state **1832**, a new descriptor is loaded from a descriptor cache maintained by free ring manager **1012**. Its identifier of a new buffer is retrieved and the remaining data from the packet is stored in the new buffer. In one embodiment of the invention, after the first portion of data is stored information from the flow's entry in flow re-assembly table **1004** is stored in a completion descriptor. Illustratively, this information includes re-assembly buffer index **1102** and the offset of the first portion of data within the full buffer. Then the new descriptor can be loaded—its index is stored in re-assembly buffer index **1102** and an initial address is stored in next address **1104**.

In state **1834**, a completion descriptor is written or configured to provide information to the host computer for processing the packet. In particular, the header buffer index (e.g., the location of the header buffer's buffer identifier within the free buffer array) and the offset of the packet's header within the header buffer are placed in the descriptor. The flow re-assembly buffer index (e.g., the location of the flow re-assembly buffer's buffer identifier within the free buffer array) and the offset of the packet's data within that buffer are also stored in the descriptor.

The size of the packet's data (e.g., the size of the packet's TCP payload) and header (e.g., the offset of the TCP payload within the packet) are stored in data size and header size fields, respectively. The descriptor type field is given a value indicating that a flow packet has been transferred to host memory. A release header flag is set if the header buffer is full and a release data flag is set because the first flow re-assembly buffer is being released. The header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment, the release header flag may be set (or cleared) at that time.

Because two re-assembly buffers were used, a split packet flag in the descriptor is set and the index, within the free descriptor ring, of the descriptor that references the second re-assembly buffer is stored in a next index field.

In one embodiment of the invention a release flow flag may also be set, depending upon dynamic packet batching module **122**. For example, if the packet batching module determines that another packet in the same flow will soon be transferred to the host computer, the release flow flag will be cleared. This indicates that the host computer should await the next flow packet before processing this one. By collectively processing multiple packets from a single flow, the packets can be processed more efficiently and less processor time is required. If, however, no other packets in the same flow are identified, the release flow flag may be set to indicate that the host computer should process the flow packets received so far, without waiting for more.

In state **1836**, next address field **1104** in the flow's entry in flow re-assembly buffer table **1004** is updated to indicate the address in the new buffer at which to store the next portion of flow data.

In state **1838**, it is determined whether the header buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, a counter may be used to keep track of entries placed into each new header buffer. The buffer is considered full when thirty-two entries are stored.

If the buffer is full, in state **1840** the header buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention, a release header flag in the descriptor is set.

If the header buffer is not full, then in state **1842** the next address field of header buffer table **1006** is updated to indicate the address at which to store the next header or small packet.

The processing associated with a packet having operation code 4 then ends with end state **1899**. In this end state, the descriptor used for this packet is turned over to the host computer by changing its ownership indicator field (e.g., from one to zero). In one alternative embodiment of the invention, DMA engine **120** issues an interrupt or uses other means to alert the host computer that a descriptor is being released.

In one alternative embodiment of the invention the optimization described above for packets associated with operation code 3 may be performed when processing a packet with operation code 4. This optimization is useful, for example, when a code 4 packet's data is too large to fit in the current flow re-assembly buffer. Instead of loading a new descriptor for the second portion of data, the descriptor may be used where it is—in a descriptor cache maintained by free ring manager **1012**. This allows DMA engine **120** to finish transferring the packet and turn over the completion descriptor before adjusting flow re-assembly buffer table **1004** to reflect a new buffer.

In particular, instead of loading information from a new descriptor in state **1832** above, it may be more efficient to use the descriptor without removing it from the cache. In this embodiment a new buffer for storing a remainder of the packet's data is accessed by retrieving its buffer identifier from a descriptor in the free ring manager's descriptor cache. The data is stored in the buffer and, after the packet's completion descriptor is configured and released, the necessary information is loaded into the flow re-assembly table as described above.

Illustratively, re-assembly buffer index **1102** stores the buffer identifier's index within the free buffer array, and an initial memory address of the buffer, taking into account the newly stored data, is placed in next address **1104**.

A Method of Transferring a Packet with Operation Code 5

FIGS. **19A–19E** depict a procedure. in which DMA engine **120** transfers a packet associated with operation code 5 to a host memory buffer. As reflected in TABLE 1, operation code 5 in one embodiment of the invention may indicate that a packet is incompatible with the protocols that may be parsed by NIC **100**. It may also indicate that a packet contains all of the data for a new flow (e.g., no more data will be received for the packet's flow). Therefore, for operation code 5, no new flow is set up and there should not be any flow to tear down. The packet's data, if there is any, is not to be re-assembled.

In state **1900**, it is determined whether the present packet is a small packet (e.g., less than or equal to 256 bytes in size) suitable for storage in a header buffer. If so, the illustrated procedure proceeds to state **1920**.

Otherwise, in state **1902** it is determined whether the present packet is a jumbo packet (e.g., greater than 1522 bytes in size), such that it should be stored in a jumbo buffer. If so, the illustrated procedure proceeds to state **1940**. If not, the procedure continues at state **1904**.

In state **1904**, it has been determined that the packet is not a small packet or a jumbo packet. The packet may, therefore, be stored in a non-re-assembly buffer used to store packets that are no greater in size than MTU (Maximum Transfer Unit) in size, which is 1522 bytes in a present embodiment. This buffer may be called an MTU buffer. Therefore, DMA engine **120** determines whether a valid (e.g., active) MTU buffer exists. Illustratively, this determination is made by examining validity indicator **1126** of MTU buffer table **1008**, which manages the active MTU buffer. If the validity indicator is set, then there should be a MTU buffer ready to receive this packet and the procedure continues at state **1908**.

Otherwise, in state **1906** a new MTU buffer is prepared or initialized for storing non-re-assembleable packets up to 1522 bytes in size. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its buffer identifier (e.g., a reference to an empty host memory buffer). If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or some other indication of the first storage location in the buffer is placed in next address field **1124** of MTU buffer table **1008**. The buffer identifier's index or position within the free buffer array is stored in MTU buffer index **1122**, and validity indicator **1126** is set to a valid state.

In state **1908** the packet is copied or transferred (e.g., via a DMA operation) into the MTU buffer at the address or location specified in the next address field of MTU buffer table **1008**. As described above, in one embodiment of the invention pad bytes may be inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In addition, the packet may be positioned within a cell of predetermined size (e.g., two kilobytes) within the MTU buffer.

In state **1910**, a completion descriptor is written or configured to provide necessary information to the host computer for processing the packet. In particular, the MTU buffer index (e.g. the location within the free buffer array of the buffer identifier for the MTU buffer) and offset (e.g., the offset to the packet or the packet's cell within the buffer) are placed in the descriptor in data index and data offset fields, respectively. The size of the packet is stored in a data size field. A header size field within the descriptor may be set to zero to indicate that the entire packet was placed in the MTU buffer (e.g., no separate header portion was stored in a header buffer). A release data flag is set in the descriptor if the MTU buffer is full. The MTU buffer may not, however, be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release data flag may be set (or cleared) at that time. A release header flag may be cleared (e.g., not set), because there is no separate header portion being conveyed to the host computer.

Further, the descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. Also, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention, the ownership indicator is not set until end state **1999** below. In an alternative embodiment of the invention, the descriptor may be released by issuing an interrupt or other alert. In yet another alternative embodiment, changing the descriptor's descriptor type field may signal the descriptor's release.

In state **1912**, DMA engine **120** determines whether the MTU buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size, each entry in the MTU buffer may be allotted two kilobytes of space and a counter may be used to keep track of entries placed into an MTU buffer. The buffer may be considered full when a predetermined number of entries (e.g., four) are stored. In an alternative embodiment of the invention entries in an MTU

buffer may or may not be allocated a certain amount of space, in which case DMA engine **120** may calculate how much storage space within the buffer has yet to be used. If no space remains, or if less than a predetermined amount of space is still available, the buffer may be considered full.

If the MTU buffer is full, in state **1914** the buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the MTU buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention a release data flag in the descriptor is set. The illustrated procedure then ends with end state **1999**.

If the MTU buffer is not full, then in state **1916** the next address field of MTU buffer table **1008** is updated to indicate the address at which to store the next packet. The illustrated procedure then ends with end state **1999**.

In state **1920** (reached from state **1900**), it has been determined that the present packet is a small packet suitable for storage in a header buffer. Therefore, DMA engine **120** (e.g., DMA manager **1002**) determines whether there is a valid (e.g., active) header buffer. Illustratively, this determination is made by examining validity indicator **1116** of header buffer table **1006**, which manages the active header buffer. If the validity indicator is set, then there is a header buffer ready to receive this packet and the procedure continues at state **1924**.

Otherwise, in state **1922** a new header buffer is prepared or initialized for storing small packets and headers of re-assembled packets. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or some other indicator of the first storage location or cell in the buffer is placed in next address field **1114** of header buffer table **1006**. Further, the buffer identifier's position within the free buffer array is stored in header buffer index **1112** and validity indicator **1116** is set to a valid state.

In state **1924** the packet is copied or transferred (e.g., via a DMA operation) into the header buffer at the address or location specified in the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes may be inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In addition, the packet may be positioned within a cell of predetermined size (e.g., 256 bytes) within the buffer.

In state **1926**, a completion descriptor is written or configured to provide necessary information to the host computer (e.g., a software driver) for processing the packet. In particular, the header buffer index (e.g. the index of the free buffer array element that contains the header buffer's identifier) and offset are placed in the descriptor, in header index and header offset fields, respectively. Illustratively, this offset serves to identify the first byte of the packet, the first pad byte preceding the packet or the location of the packet's cell within the buffer. The size of the packet is also stored in the descriptor, illustratively within a header size field. A data size field within the descriptor may be set to zero to indicate that the entire packet was placed in the header buffer (e.g., no separate data portion was stored in another buffer). A release header flag may be set in the descriptor if the header buffer is full. However, the header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release header flag may be set (or cleared) at that time. A release data flag may be cleared (e.g., not set), because there is no separate data portion being conveyed to the host computer.

The descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. Also, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention the ownership indicator is not set until end state **1999** below.

In state **1928** it is determined whether the header buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, a counter is used to keep track of entries placed into each new header buffer. The buffer is considered full when thirty-two entries are stored.

If the buffer is full, in state **1930** the header buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention a release header flag in the descriptor is set. The illustrated procedure then ends with end state **1999**.

If the header buffer is not full, then in state **1932** the next address field of header buffer table **1006** is updated to indicate the address at which to store the next header or small packet. The illustrated procedure then ends with end state **1999**.

In state **1940** (reached from state **1902**), it has been determined that the packet is a jumbo packet (e.g., that it is greater than 1522 bytes in size). In this embodiment of the invention a jumbo packet's data portion is stored in a jumbo buffer. Its header is also stored in the jumbo buffer unless splitting of jumbo packets is enabled, in which case its header is stored in a header buffer. DMA engine **120** thus determines whether a valid (e.g., active) jumbo buffer exists. Illustratively, this determination is made by examining validity indicator **1136** of jumbo buffer table **1010**, which manages an active jumbo buffer. If the validity indicator is set, then there is a jumbo buffer ready to receive this packet and the procedure continues at state **1944**.

Otherwise, in state **1942** a new jumbo buffer is prepared or initialized for storing a non-re-assembleable packet that is larger than 1522 bytes. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or other indication of the first storage location within the buffer is placed in next address field **1134** of jumbo buffer table **1010**. The position of the buffer identifier within the free buffer array is stored in jumbo buffer index **1132**, and validity indicator **1136** is set to a valid state.

Then, in state **1944**, DMA engine **120** determines whether splitting of jumbo buffers is enabled. If enabled, the header of a jumbo packet is stored in a header buffer while the packet's data is stored in one or more jumbo buffers. If not

enabled, the entire packet will be stored in one or more jumbo buffers. Illustratively, splitting of jumbo packets is enabled or disabled according to the configuration of a programmable indicator (e.g., flag, bit, register) that is set by software operating on the host computer (e.g., a device driver). If splitting is enabled, the illustrated procedure continues at state **1960**. Otherwise, the procedure proceeds to state **1946**.

In state **1946**, DMA engine **120** determines whether the packet will fit into one jumbo buffer. For example, in an embodiment of the invention using eight kilobyte pages, if the packet is larger than eight kilobytes a second jumbo buffer will be needed to store the additional contents. If the packet is too large, the illustrated procedure continues at state **1952**.

Otherwise, in state **1948** the packet is copied or transferred (e.g., via a DMA operation) into the current jumbo buffer, at the location specified in the next address field **1134** of jumbo buffer table **1010**. When the packet is transferred intact like this, padding may be added to align a header portion of the packet with a sixteen-byte boundary. One skilled in the art will appreciate that the next address field may not need to be updated to account for this new packet because the jumbo buffer will be released. In other words, in one embodiment of the invention a jumbo buffer is only used once (e.g., to store one packet or a portion of one packet). In an alternative embodiment of the invention a jumbo buffer may store portions of two or more packets, in which case next address field **1134** may need to be updated.

In state **1950**, a completion descriptor is written or configured to release the jumbo buffer and to provide information to the host computer for processing the packet. The jumbo buffer index (e.g., the index, within the free buffer array, of the buffer identifier that corresponds to the jumbo buffer) and the offset of the first byte of the packet within the jumbo buffer are placed in the descriptor, in data index and data size fields, respectively. The size of the packet (e.g., the packet length) is stored in a data size field.

A header size field may be cleared (e.g., a zero is stored) to indicate that the header buffer was not used (e.g., the header was not stored separately from the packet's data). Because the packet was stored intact, header index and header offset fields may or may not be used (e.g., the values stored in their fields do not matter). A release header flag is cleared and a release data flag is set to indicate that no more data will be placed in this jumbo buffer (e.g., because it is being released).

The descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. Finally, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention, the ownership indicator is not changed until end state **1999** below.

After state **1950**, the illustrated procedure resumes at state **1958**. In one alternative embodiment of the invention, DMA engine **120** issues an interrupt or uses some other means, possibly not until end state **1999**, to alert the host computer that a descriptor is being released.

In state **1952**, a first portion of the packet is stored in the present (e.g., valid) jumbo buffer, at the location identified in the buffer's next address field **134**. Because the whole packet will not fit into this buffer, in state **1954** a new jumbo buffer is prepared and the remainder of the packet is stored in that buffer.

In state **1956**, a completion descriptor is written or configured. The contents are similar to those described in state **1950** but this descriptor must reflect that two jumbo buffers were used to store the packet. Thus, the jumbo buffer index (e.g., the index, within the free buffer array, of the array element containing the header buffer's buffer identifier) and the offset of the first byte of the packet within the first jumbo buffer are placed in the descriptor, as above. The size of the packet (e.g., the packet length) is stored in a data size field.

A header size field may be cleared (e.g., a zero is stored) to indicate that the header buffer was not used (e.g., the header was not stored separately from the packet's data). Because there is no separate packet header, header index and header offset fields may or may not be used (e.g., the values stored in their fields do not matter).

A release header flag is cleared and a release data flag is set to indicate that no more data will be placed in these jumbo buffers (e.g., because they are being released). Further, a split packet flag is set to indicate that a second jumbo buffer was used, and the index (within the free buffer array) of the buffer identifier for the second buffer is stored in a next index field.

The descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. And, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention the ownership indicator is not changed until end state **1999** below.

In state **1958**, the jumbo buffer's entry in jumbo buffer table **1010** is invalidated (e.g., validity indicator **1136** is set to invalid) to ensure that it is not used again. In the procedure described above, a jumbo packet was stored in, at most, two jumbo buffers. In an alternative embodiment of the invention, a jumbo buffer may be stored across any number of buffers. The descriptor that is configured to report the transfer of such a packet is constructed accordingly, as will be obvious to one skilled in the art.

After state **1958**, the illustrated procedure ends at end state **1999**.

In state **1960** (reached from state **1944**), it has been determined that the present jumbo packet will be split to store the packet header in a header buffer and the packet data in one or more jumbo buffers. Therefore, DMA engine **120** (e.g., DMA manager **1002**) first determines whether there is a valid (e.g., active) header buffer. Illustratively, this determination is made by examining validity indicator **1116** of header buffer table **1006**, which manages the active header buffer. If the validity indicator is set, then there is a header buffer ready to receive this packet and the procedure continues at state **1964**.

Otherwise, in state **1962** a new header buffer is prepared or initialized for storing small packets and headers of other packets. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or some other indication of the first storage location or cell in the buffer is placed in next address field **1114** of header buffer table **1006**. The index or position of the buffer identifier within the free buffer array is stored in header buffer index **1112**, and validity indicator **1116** is set to a valid state.

In state **1964** the packet's header is copied or transferred (e.g., via a DMA operation) into the header buffer at the address or location specified in the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes may be inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In addition, the header may be positioned within a cell of predetermined size (e.g., 256 bytes) in the buffer.

In state **1966**, DMA engine **120** determines whether the packet's data (e.g., the TCP payload) will fit into one jumbo buffer. If the packet is too large to fit into one (e.g., the current jumbo buffer), the illustrated procedure continues at state **1972**.

In state **1968**, the packet's data is copied or transferred (e.g., via a DMA operation) into the current jumbo buffer, at the location specified in the next address field **1134** of jumbo buffer table **1010**. One skilled in the art will appreciate that the next address field may not need to be updated to account for this new packet because the jumbo buffer will be released. In other words, in one embodiment of the invention a jumbo buffer is only used once (e.g., to store one packet or a portion of one packet).

In state **1970**, a completion descriptor is written or configured to release the jumbo buffer and to provide information to the host computer for processing the packet. The header buffer index (e.g. the free buffer array position of the buffer identifier corresponding to the header buffer) and offset of the packet's header are placed in the descriptor in header index and header offset fields, respectively. Illustratively, this offset serves to identify the first byte of the header, the first pad byte preceding the header or the cell in which the header is stored. The jumbo buffer index (e.g., the index within the free buffer array of the buffer identifier that references the jumbo buffer) and the offset of the first byte of the packet's data within the jumbo buffer are placed in data index and data offset fields, respectively. Header size and data size fields are used to store the size of the packet's header (e.g., the offset of the payload within the packet) and data (e.g., payload size), respectively.

A release header flag may be set in the descriptor if the header buffer is full. However, the header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release header flag may be set (or cleared) at that time. A release data flag is also set, because no more data will be placed in the jumbo buffer (e.g., it is being released to the host computer).

The descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. Also, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention the ownership indicator is not set until end state **1999** below.

After state **1970**, the illustrated procedure proceeds to state **1978**.

In state **1972**, a first portion of the packet's data is stored in the present (e.g., valid) jumbo buffer, at the location identified in the buffer's next address field **1134**. Because all of the packet's data will not fit into this buffer, in state **1974** a new jumbo buffer is prepared and the remainder of the packet is stored in that buffer.

In state **1976**, a completion descriptor is written or configured. The contents are similar to those described in states **1970** but this descriptor must reflect that two jumbo buffers were used to store the packet. The header buffer index (e.g. the free buffer array element that contains the header buffer's identifier) and offset of the header are placed in the descriptor in header index and header offset fields, respectively. The jumbo buffer index (e.g., the free buffer array element containing the jumbo buffer's buffer identifier) and the offset of the first byte of the packet's data within the jumbo buffer are placed in data index and data offset fields, respectively. Header size and data size fields are used to store the size of the packet's header (e.g., the offset of the payload within the packet) and data (e.g., payload size), respectively.

A release header flag is set in the descriptor if the header buffer is full. However, the header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment of the invention, the release header flag may be set (or cleared) at that time. A release data flag is also set, because no more data will be placed in the jumbo buffer (e.g., it is being released to the host computer). Further, a split packet flag is set to indicate that a second jumbo buffer was used, and the position or index within the free buffer array of the second buffer's buffer identifier is stored in a next index field.

The descriptor type field is changed to a value indicating that DMA engine **120** transferred a non-re-assembleable packet into host memory. Finally, a predetermined value (e.g., zero) is stored in the descriptor's ownership indicator field to indicate that DMA engine **120** is releasing a packet to the host computer and turning over ownership of the descriptor. In one embodiment of the invention the ownership indicator is not set until end state **1999** below. In an alternative embodiment of the invention DMA engine **120** issues an interrupt or uses some other signal to alert the host computer that a descriptor is being released.

In state **1978**, the jumbo buffer's entry in jumbo buffer table **1010** is invalidated (e.g., validity indicator **1136** is set to invalid) to ensure that it is not used again. In the procedure described above, a jumbo packet was stored in, at most, two jumbo buffers. In an alternative embodiment of the invention a jumbo buffer may be stored across any number of buffers. The descriptor that is configured to report the transfer of such a packet is constructed accordingly, as will be obvious to one skilled in the art.

In state **1980**, it is determined whether the header buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, a counter may be used to keep track of entries placed into each new header buffer. The buffer is considered full when thirty-two entries are stored.

If the buffer is full, in state **1982** the header buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention a release header flag in the descriptor is set. The illustrated procedure then ends with end state **1999**.

If the header buffer is not full, then in state **1984** the next address field of header buffer table **1006** is updated to indicate the address at which to store the next header or small packet. The illustrated procedure then ends with end state **1999**.

In end state **1999**, a descriptor may be turned over to the host computer by storing a particular value (e.g., zero) in the descriptor's ownership indicator field as described above. Illustratively, the host computer (or software operating on the host computer) detects the change and understands that DMA engine **120** is returning ownership of the descriptor to the host computer.

A Method of Transferring a Packet with Operation Code 6 or Operation Code 7

FIGS. **20A–20B** depict an illustrative procedure in which DMA engine **120** transfers a packet associated with operation code 6 or 7 to a host memory buffer. As reflected in TABLE 1, operation codes 6 and 7 may indicate that a packet is compatible with the protocols pre-selected for NIC **100** and is the first packet of a new flow. The difference between these operation codes in this embodiment of the invention is that operation code 7 is used when an existing flow is to be replaced (e.g., in flow database **110** and/or flow re-assembly buffer table **1004**) by the new flow. With operation code 6, in contrast, no flow needs to be torn down. For both codes, however, a new flow is set up and the associated packet's data may be re-assembled with data from other packets in the newly established flow. Because the packet data is to be re-assembled, the packet's header should be stored in a header buffer and its data in a new flow re-assembly buffer.

As described in a previous section, the flow that is torn down to make room for a new flow (in the case of operation code 7) may be the least recently used flow. Because flow database **110** and flow re-assembly buffer table **1004** contain only a limited number of entries in the presently described embodiment of the invention, when they are full and a new flow arrives an old one must be torn down. Choosing the least recently active flow for replacement is likely to have the least impact on network traffic through NIC **100**. In one embodiment of the invention DMA engine **120** tears down the flow in flow re-assembly buffer table **1004** that has the same flow number as the flow that has been replaced in flow database **110**.

In state **2000**, DMA engine **120** determines whether there is a valid (e.g., active) header buffer. Illustratively, this determination is made by examining validity indicator **1116** of header buffer table **1006**, which manages the active header buffer. If the validity indicator is set, then there is a header buffer ready to receive this packet and the procedure continues at state **2004**.

Otherwise, in state **2002** a new header buffer is prepared or initialized for storing small packets and headers of re-assembled packets. Illustratively, this initialization process involves obtaining a free ring descriptor from a cache maintained by free ring manager **1012** and retrieving its reference to an empty buffer. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or some other indication of the first storage location or cell in the buffer is placed in next address field **1114** of header buffer table **1006**. The position or index of the buffer identifier within the free buffer array is stored in header buffer index **1112**, and validity indicator **1116** is set to a valid state.

In state **2004** the packet's header is copied or transferred into the header buffer at the address or location specified in the next address field of header buffer table **1006**. As described above, in one embodiment of the invention pad bytes may be inserted before the header in order to align the beginning of the packet's layer three protocol (e.g., IP) header with a sixteen-byte boundary. In addition, the packet's header may be positioned in a cell of predetermined size (e.g., 256 bytes) within the buffer.

As discussed above, operation code 7 indicates that an old flow is to be torn down in flow re-assembly buffer table **1004** to make room for a new flow. This requires the release of any flow re-assembly buffer that may be associated with the flow being torn down.

Thus, in state **2006** it is determined whether a flow re-assembly buffer is valid (e.g., active) for a flow having the flow number that was read from control queue **118** for this packet. As explained in a previous section, for operation code 7 the flow number represents the entry in flow database **110** (and flow re-assembly buffer table **1004**) that is being replaced with the new flow. DMA engine **120** thus examines the validity indicator in the flow's entry in flow re-assembly buffer table **1004**. Illustratively, if the indicator is valid then there is an active buffer storing data from one or more packets in the flow that is being replaced. If there is a valid flow re-assembly buffer for this flow, the illustrated procedure continues at state **2008**. Otherwise, the procedure proceeds to state **2010**. It will be understood that the illustrated procedure will normally proceed to state **2008** for operation code 7 and state **2010** for operation code 6.

In state **2008**, a completion descriptor is written or configured to release the replaced flow's re-assembly buffer. In particular, the flow re-assembly buffer index (e.g., the index within the free buffer array of the flow re-assembly buffer's buffer identifier) is written to the descriptor. In this embodiment of the invention, no offset needs to be stored in the descriptor's data offset field and the data size field is set to zero because no new data was stored in the buffer that is being released. Similarly, the header buffer is not yet being released, and therefore the header index and header offset fields of the descriptor need not be used and a zero may be stored in the header size field.

The descriptor's release header flag is cleared (e.g., a zero is stored in the flag) because the header buffer is not being released. The release data flag is set (e.g., a one is stored in the flag), however, because no more data will be placed in the released flow re-assembly buffer. Further, a release flow flag in the descriptor is set to indicate that the flow associated with the released flow re-assembly buffer is being torn down.

The descriptor type field is changed to a value indicating that DMA engine **120** is releasing a stale flow buffer (e.g., a flow re-assembly buffer that has not been used for some time). Finally, the descriptor used to release the replaced flow's re-assembly buffer and terminate the associated flow is turned over to the host computer by changing its ownership indicator field (e.g., from one to zero). In one alternative embodiment of the invention, DMA engine **120** issues an interrupt or employs some other means of alerting the host computer that a descriptor is being released.

In state **2010**, a new flow re-assembly buffer is prepared for the flow that is being set up. Illustratively, a free ring descriptor is obtained from a cache maintained by free ring manager **1012** and its buffer identifier (e.g., a reference to an empty memory buffer) is retrieved. If the cache is empty, new descriptors may be retrieved from the free descriptor ring in host memory to replenish the cache.

When a new descriptor is obtained from the cache or from the free descriptor ring, the buffer identifier (e.g., pointer, address, index) contained in the descriptor is stored in a free buffer array. The buffer's initial address or other indication of the first storage location in the buffer is placed in next address field **1104** of the flow's entry in flow re-assembly buffer table **1004**. The flow's entry in the table may be recognized by its flow number. The position or index of the buffer identifier within the free buffer array is stored in re-assembly buffer index **1102**, and validity indicator **1106** is set to a valid state.

In state **2012**, the packet's data is copied or transferred (e.g., via a DMA operation) into the address or location specified in the next address field of the flow's entry in flow re-assembly buffer table **1004**.

In state **2014**, a completion descriptor is written or configured to provide information to the host computer for processing the packet. In particular, the header buffer index (e.g., the location or position within the free buffer array of the buffer identifier that references the header buffer) and the offset of the packet's header within the header buffer are placed in the descriptor. Illustratively, the offset identifies the first byte of the header, the first pad byte preceding the header or the location of the header's cell in the header buffer.

The flow re-assembly buffer index (e.g., the location or position, within the free buffer array, of the buffer identifier that references the flow re-assembly buffer) and the offset of the packet's data within that buffer are also stored in the descriptor. It will be recognized, however, that the offset reported for this packet's data may be zero, because the packet data is stored at the very beginning of the new flow re-assembly buffer.

The size of the packet's data (e.g., the size of the packet's TCP payload) and header (e.g., the offset of the TCP payload within the packet) are stored in data size and header size fields, respectively. The descriptor type field is changed to a value indicating that DMA engine **120** transferred a flow packet into host memory. A release header flag is set if the header buffer is full but a release data flag is not set, because more data will be placed in this flow re-assembly buffer. The header buffer may not be tested to see if it is full until a later state of this procedure. In such an embodiment, the release header flag may be set (or cleared) at that time.

In one embodiment of the invention a release flow flag may also be set, depending upon dynamic packet batching module **122**. For example, if the packet batching module determines that another packet in the same flow will soon be transferred to the host computer, the release flow flag will be cleared (e.g., a zero will be stored). This indicates that the host computer should await the next flow packet before processing this one. By collectively processing multiple packets from a single flow, the packets can be processed more efficiently and less processor time will be required for network traffic. If, however, no other packets in the same flow are identified, the release flow flag may be set to indicate that the host computer should process the flow packets received so far, without waiting for more.

In state **2016**, the flow's entry in flow re-assembly buffer table **1004** is updated. In particular, next address field **1104** is updated to identify the location in the re-assembly buffer at which the next flow packet's data should be stored.

In state **2018**, it is determined whether the header buffer is full. In this embodiment of the invention, where each buffer is eight kilobytes in size and entries in the header buffer are no larger than 256 bytes, a counter may be used to keep track of entries placed into each new header buffer. The buffer is considered full when thirty-two entries are stored.

If the buffer is full, in state **2020** the header buffer is invalidated to ensure that it is not used again. Illustratively, this involves setting the header buffer table's validity indicator to invalid and communicating this status to the host computer. In this embodiment of the invention, a release header flag in the descriptor is set.

If the header buffer is not full, then in state **2022** the next address field of header buffer table **1006** is updated to indicate the address at which to store the next header or small packet.

The processing associated with a packet having operation codes 6 and 7 then ends with end state **2099**. In this end state, the descriptor used for this packet (e.g., the descriptor that was configured in state **2014**) is turned over to the host computer by changing its ownership indicator field (e.g., from one to zero). In one alternative embodiment of the invention, DMA engine **120** issues an interrupt or employs other means (e.g., such as the descriptor's descriptor type field) to alert the host computer that a descriptor is being released.

One Embodiment of a Packet Batching Module

FIG. **21** is a diagram of dynamic packet batching module **122** in one embodiment of the invention. In this embodiment, packet batching module **122** alerts a host computer to the transfer, or impending transfer, of multiple packets from one communication flow. The related packets may then be processed through an appropriate protocol stack collectively, rather than processing one at a time. As one skilled in the art will recognize, this increases the efficiency with which network traffic may be handled by the host computer.

In the illustrated embodiment, a packet is transferred from NIC **100** to the host computer by DMA engine **120** (e.g., by copying its payload into an appropriate buffer). When a packet is transferred, packet batching module **122** determines whether a related packet (e.g., a packet in the same flow) will soon be transferred as well. In particular, packet batching module **122** examines packets that are to be transferred after the present packet. One skilled in the art will appreciate that the higher the rate of packet arrival at NIC **100**, the more packets that are likely to await transfer to a host computer at a given time. The more packets that await transfer, the more packets that may be examined by the dynamic packet batching module and the greater the benefit it may provide. In particular, as the number of packets awaiting transfer increases, packet batching module **122** may identify a greater number of related packets for collective processing. As the number of packets processed together increases, the amount of host processor time required to process each packet decreases.

Thus, if a related packet is found the packet batching module alerts the host computer so that the packets may be processed as a group. As described in a previous section, in one embodiment of the invention dynamic packet batching module **122** alerts the host computer to the availability of a related packet by clearing a release flow flag in a completion descriptor associated with a transferred packet. The flag may, for example, be cleared by DMA engine **120** in response to a signal or alert from dynamic packet batching module **122**.

In contrast, in an alternative embodiment of the invention dynamic packet batching module **122** or DMA engine **120** may alert the host computer when no related packets are found or when, for some other reason, the host processor should not delay processing a transferred packet. In particular, a release flow flag may be set when the host computer is not expected to receive a packet related to a transferred packet in the near future (e.g., thus indicating that the associated flow is being released or torn down). For example, it may be determined that the transferred packet is the last packet in its flow or that a particular packet doesn't even belong to a flow (e.g., this may be reflected in the packet's associated operation code).

With reference now to FIG. **21**, packet batching module **122** in one embodiment of the invention includes memory **2102** and controller **2104**. Illustratively, each entry in memory **2102**, such as entry **2106**, comprises two fields:

flow number **2108** and validity indicator **2110**. In alternative embodiments of the invention, other information may be stored in memory **2102**. Read pointer **2112** and write pointer **2114** serve as indices into memory **2102**.

In the illustrated embodiment, memory **2102** is an associative memory (e.g., a CAM) configured to store up to 256 entries. Each entry corresponds to and represents a packet stored in packet queue **116**. As described in a previous section, packet queue **116** may also contain up to 256 packets in one embodiment of the invention. When a packet is, or is about to be transferred, by DMA engine **120** from packet queue **116** to the host computer, memory **2102** may be searched for an entry having a flow number that matches the flow number of the transferred packet. Because memory **2102** is a CAM in this embodiment, all entries in the memory may be searched simultaneously or nearly simultaneously. In this embodiment, memory **2102** is implemented in hardware, with the entries logically arranged as a ring. In alternative embodiments, memory **2102** may be virtually any type of data structure (e.g., array, table, list, queue) implemented in hardware or software. In one particular alternative embodiment, memory **2102** is implemented as a RAM, in which case the entries may be examined in a serial manner.

The maximum of 256 entries in the illustrated embodiment matches the maximum number of packets that may be stored in a packet queue. Because the depth of memory **2102** matches the depth of the packet queue, when a packet is stored in the packet queue its flow number may be automatically stored in memory **2102**. Although the same number of entries are provided for in this embodiment, in an alternative embodiment of the invention memory **2102** may be configured to hold a smaller or greater number of entries than the packet queue. And, as discussed in a previous section, for each packet stored in the packet queue, related information may also be stored in the control queue.

In the illustrated embodiment of the invention, flow number **2108** is the index into flow database **110** of the flow comprising the corresponding packet. As described above, in one embodiment of the invention a flow includes packets carrying data from one datagram sent from a source entity to a destination entity. Illustratively, each related packet has the same flow key and the same flow number. Flow number **2108** may comprise the index of the packet's flow key in flow database **110**.

Validity indicator **2110** indicates whether the information stored in the entry is valid or current. In this embodiment, validity indicator **2110** may store a first value (e.g., one) when the entry contains valid data, and a second value (e.g., zero) when the data is invalid. For example, validity indicator **2110** in entry **2106** may be set to a valid state when the corresponding entry in packet queue **116** contains a packet awaiting transfer to the host computer and belongs to a flow (e.g., which may be indicated by the packet's operation code). Similarly, validity indicator **2110** may be set to an invalid state when the entry is no longer needed (e.g., when the corresponding packet is transferred to the host computer).

Flow validity indicator **2110** may also be set to an invalid state when a corresponding packet's operation code indicates that the packet does not belong to a flow. It may also be set to an invalid state when the corresponding packet is a control packet (e.g., contains no data) or is otherwise non-re-assembleable (e.g., because it is out of sequence, incompatible with a pre-selected protocol, has an unexpected control flag set). Validity indicator **2110** may be managed by controller **2104** during operation of the packet batching module.

In the illustrated embodiment of the invention, an entry's flow number is received from a register in which it was placed for temporary storage. A packet's flow number may be temporarily stored in a register, or other data structure, in order to facilitate its timely delivery to packet batching module **122**. Temporary storage of the flow number also allows the flow database manager to turn its attention to a later packet. A flow number may, for example, be provided to dynamic packet batching module **122** at nearly the same time that the associated packet is stored in packet queue **116**. Illustratively, the flow number may be stored in the register by flow database manager **108** or by IPP module **104**. In an alternative embodiment, the flow number is received from control queue **118** or some other module of NIC **100**.

In the illustrated embodiment of the invention, memory **2102** contains an entry corresponding to each packet in packet queue **116**. When a packet in the packet queue is transferred to a host computer (e.g., when it is written to a re-assembly buffer), controller **2104** invalidates the memory entry that corresponds to that packet. Memory **2102** is then searched for another entry having the same flow number as the transferred packet. Afterwards, when a new packet is stored in packet queue **116**, perhaps in place of the transferred packet, a new entry is stored in memory **2102**.

In an alternative embodiment of the invention, memory **2102** may be configured to hold entries for only a subset of the maximum number of packets stored in packet queue **116** (e.g., just re-assembleable packets). Entries in memory **2102** may still be populated when a packet is stored in the packet queue. However, if memory **2102** is full when a new packet is received, then creation of an entry for the new packet must wait until a packet is transferred and its entry in memory **2102** invalidated. Therefore, in this alternative embodiment entries in memory **2102** may be created by extracting information from entries in control queue **118** rather than packet queue **116**. Controller **2104** would therefore continually attempt to copy information from entries in control queue **118** into memory **2102**. The function of populating memory **2102** may be performed independently or semi-independently of the function of actually comparing the flow numbers of memory entries to the flow number of a packet being transferred to the host computer.

In this alternative embodiment a second read pointer may be used to index control queue **118** to assist in the population of memory **2102**. In particular, the second read pointer may be used by packet batching module **122** to find and fetch entries for memory **2102**. Illustratively, if the second, or "lookahead" read pointer references the same entry as the control queue's write pointer, then it could be determined that no new entries were added to control queue **118** since the last check by controller **2104**. Otherwise, as long as there is an empty (e.g., invalid) entry in memory **2102**, the necessary information (e.g., flow number) may be copied into memory **2102** for the packet corresponding to the entry referenced by the lookahead read pointer. The lookahead read pointer would then be incremented.

Returning now to FIG. **21**, read pointer **2112** of dynamic packet batching module **122** identifies the current entry in memory **2102** (e.g., the entry corresponding to the packet at the front of the packet queue or the next packet to be transferred). Illustratively, this pointer is incremented each time a packet is transferred to the host computer. Write pointer **2114** identifies the position at which the next entry in memory **2102** is to be stored. Illustratively, the write pointer is incremented each time an entry is added to memory **2102**. One manner of collectively processing headers from related packets is to form them into one "super-

"header. In this method, the packets' data portions are stored separately (e.g., in a separate memory page or buffer) from the super-header.

Illustratively, a super-header comprises one combined header for each layer of the packets' associated protocol stack (e.g., one TCP header and one IP header). To form each layer's portion of a super-header, the packet's individual headers may be merged to make a regular-sized header whose fields accurately reflect the assembled data and combined headers. For example, merged header fields relating to payload or header length would indicate the size of the aggregated data or aggregated headers, the sequence number of a merged TCP header would be set appropriately, etc. The super-header portion may then be processed through its protocol stack similar to the manner in which a single packet's header is processed.

This method of collectively processing related packets' headers (e.g., with "super-"headers) may require modification of the instructions for processing packets (e.g., a device driver). For example, because multiple headers are merged for each layer of the protocol stack, the software may require modification to recognize and handle the super-headers. In one embodiment of the invention the number of headers folded or merged into a super-header may be limited. In an alternative embodiment of the invention the headers of all the aggregated packets, regardless of number, may be combined.

In another method of collectively processing related packets' header portions, packet data and headers may again be stored separately (e.g., in separate memory pages). But, instead of combining the packets' headers for each layer of the appropriate protocol stack to form a super-header, they may be submitted for individual processing in quick succession. For example, all of the packets' layer two headers may be processed in a rapid sequence—one after the other—then all of the layer three headers, etc. In this manner, packet processing instructions need not be modified, but headers are still processed more efficiently. In particular, a set of instructions (e.g., for each protocol layer) may be loaded once for all related packets rather than being separately loaded and executed for each packet.

As discussed in a previous section, data portions of related packets may be transferred into storage areas of predetermined size (e.g., memory pages) for efficient transfer from the host computer's kernel space into application or user space. Where the transferred data is of memory page size, the data may be transferred using highly efficient "page-flipping," wherein a full page of data is provided to application or user memory space.

FIGS. **22A–22B** present one method of dynamic packet batching with packet batching module **122**. In the illustrated method, memory **2102** is populated with flow numbers of packets stored in packet queue **116**. In particular, a packet's flow number and operation code are retrieved from control queue **118**, IPP module **104**, flow database manager **108** or other module(s) of NIC **100**. The packet's flow number is stored in the flow number portion of an entry in memory **2102**, and validity indicator **2110** is set in accordance with the operation code. For example, if the packet is not re-assembleable (e.g., codes 2 and 5 in TABLE 1), the validity indicator may be set to zero; otherwise it may be set to one.

The illustrated method may operate in parallel to the operation of DMA engine **120**. In other words, dynamic packet batching module **122** may search for packets related to a packet in the process of being transferred to a host memory buffer. Alternatively, a search may be conducted shortly after or before the packet is transferred. Because memory **2102** may be associative in nature, the search operation may be conducted quickly, thus introducing little, if any, delay into the transfer process.

FIG. **22A** may be considered a method of searching for a related packet, while FIG. **22B** may be considered a method of populating the dynamic packet batching module's memory.

FIGS. **22A–22B** each reflect one "cycle" of a dynamic packet batching operation (e.g., one search and creation of one new memory entry). Illustratively, however, the operation of packet batching module **122** runs continuously. That is, at the end of one cycle of operation another cycle immediately begins. In this manner, controller **2104** strives to ensure memory **2102** is populated with entries for packets as they are stored in packet queue **116**. If memory **2102** is not large enough to store an entry for each packet in packet queue **116**, then controller **2104** attempts to keep the memory as full as possible and to quickly replace an invalidated entry with a new one.

State **2200** is a start state for a memory search cycle. In state **2202**, it is determined whether a packet (e.g., the packet at the front of the packet queue) is being transferred to the host computer. This determination may, for example, be based on the operation of DMA engine **120** or the status of a pointer in packet queue **116** or control queue **118**. Illustratively, state **2202** is initiated by DMA engine **120** as a packet is copied into a buffer in the host computer. One purpose of state **2202** is simply to determine whether memory **2102** should be searched for a packet related to one that was, will be, or is being transferred. Until a packet is transferred, or about to be transferred, the illustrated procedure continues in state **2202**.

When, however, it is time for a search to be conducted (e.g., a packet is being transferred), the method continues at state **2204**. In state **2204**, the entry in memory **2102** corresponding to the packet being transferred is invalidated. Illustratively, this consists of storing a predetermined value (e.g., zero) in validity indicator **2110** for the packet's entry. In a present embodiment of the invention read pointer **2112** identifies the entry corresponding to the packet to be transferred. As one skilled in the art will recognize, one reason for invalidating a transferred packet's entry is so that when memory **2102** is searched for an entry associated with a packet related to the transferred packet, the transferred packet's own entry will not be identified.

In one embodiment of the invention the transferred packet's flow number is copied into a register (e.g., a hardware register) when dynamic packet batching module **122** is to search for a related packet. This may be particularly helpful (e.g., to assist in comparing the flow number to flow numbers of other packets) if memory **2102** is implemented as a RAM instead of a CAM.

In state **2206**, read pointer **2112** is incremented to point to the next entry in memory **2102**. If read pointer is incremented to the same entry that is referenced by write pointer **2114**, and that entry is also invalid (as indicated by validity indicator **2110**), it may be determined that memory **2102** is now empty.

Then, in state **2208**, memory **2102** is searched for a packet related to the packet being transferred (e.g., the memory is searched for an entry having the same flow number). As described above, entries in memory **2102** are searched associatively in one embodiment of the invention. Thus, the result of the search operation may be a single signal indicating whether or not a match was found.

In the illustrated embodiment of the invention, only valid entries (e.g., those having a value of one in their validity

indicators) are searched. As explained above, an entry may be marked invalid (e.g., its validity indicator stores a value of zero) if the associated packet is considered incompatible. Entries for incompatible packets may be disregarded because their data is not ordinarily reassembled and their headers are not normally batched. In an alternative embodiment of the invention, all entries may be searched but a match is reported only if a matching entry is valid.

In state **2210**, the host computer is alerted to the availability or non-availability of a related packet. In this embodiment of the invention, the host computer is alerted by storing a predetermined value in a specific field of the transferred packet's completion descriptor (described in a previous section). As discussed in the previous section, when a packet is transferred a descriptor in a descriptor ring in host memory is populated with information concerning the packet (e.g., an identifier of its location in host memory, its size, an identifier of a processor to process the packet's headers). In particular, a release flow flag or indicator is set to a first value (e.g., zero) if a related packet is found, and a second value if no related packet is found. Illustratively, DMA engine **120** issues the alert or stores the necessary information to indicate the existence of a related packet in response to notification from dynamic packet batching module **122**. Other methods of notifying the host computer of the presence of a related packet are also suitable (e.g., an indicator, flag, key), as will be appreciated by one skilled in the art.

In FIG. 22B, state **2220** is a start state for a memory population cycle.

In state **2222**, it is determined whether a new packet has been received at the network interface. Illustratively, a new entry is made in the packet batching module's memory for each packet received from the network. The receipt of a new packet may be signaled by IPP module **104**. For example, the receipt of a new packet may be indicated by the storage of the packet's flow number, by IPP module **104**, in a temporary location (e.g., a register). Until a new packet is received, the illustrated procedure waits. When a packet is received, the procedure continues at state **2224**.

In state **2224**, if memory **2102** is configured to store fewer entries than packet queue **116** (and, possibly, control queue **118**), memory **2102** is examined to determine if it is full.

In one embodiment of the invention memory **2102** may be considered fill if the validity indicator is set (e.g., equal to one) for each entry or for the entry referenced by write pointer **2114**. If the memory is full, the illustrated procedure waits until the memory is not full. As one skilled in the art will recognize, memory **2102** and other data structures in NIC **100** may be tested for saturation (e.g., whether they are filled) by comparing their read and write pointers.

In state **2226**, a new packet is represented in memory **2102** by storing its flow number in the entry identified by write pointer **2114** and storing an appropriate value in the entry's validity indicator field. If, for example, the packet is not re-assembleable (e.g., as indicated by its operation code), the entry's validity indicator may be set to an invalid state. For purposes of the operation of dynamic packet batching module **122**, a TCP control packet may or may not be considered re-assembleable. Thus, depending upon the implementation of a particular embodiment the validity indicator for a packet that is a TCP control packet may be set to a valid or invalid state.

In an alternative embodiment of the invention an entry in memory **2102** is populated with information from the control queue entry identified by the second read pointer described above. This pointer may then be incremented to the next entry in control queue **118**.

In state **2228**, write pointer **2114** is incremented to the next entry of memory **2102**, after which the illustrated method ends at end state **2230**. If write pointer **2114** references the same entry as read pointer **2112**, it may be determined that memory **2102** is full. One skilled in the art will recognize that many other suitable methods of managing pointers for memory **2102** may be employed.

As mentioned above, in one embodiment of the invention one or both of the memory search and memory population operations run continuously. Thus, end state **2230** may be removed from the procedure illustrated in FIG. 22B, in which case the procedure would return to state **2222** after state **2228**.

Advantageously, in the illustrated embodiment of the invention the benefits provided to the host computer by dynamic packet batching module **122** increase as the host computer becomes increasingly busy. In particular, the greater the load placed on a host processor, the more delay that will be incurred until a packet received from NIC **100** may be processed. As a result, packets may queue up in packet queue **116** and, the more packets in the packet queue, the more entries that can be maintained in memory **2102**.

The more entries that are stored in memory **2102**, the further ahead dynamic packet batching module can look for a related packet. The further ahead it scans, the more likely it is that a related packet will be found. As more related packets are found and identified to the host computer for collective processing, the amount of processor time spent on network traffic decreases and overall processor utilization increases.

One skilled in the art will appreciate that other systems and methods may be employed to identify multiple packets from a single communication flow or connection without exceeding the scope of the present invention.

Early Random Packet Discard in One Embodiment of the Invention

Packets may arrive at a network interface from a network at a rate faster than they can be transferred to a host computer. When such a situation exists, the network interface must often drop, or discard, one or more packets. Therefore, in one embodiment of the present invention a system and method for randomly discarding a packet are provided. Systems and methods discussed in this section may be applicable to other communication devices as well, such as gateways, routers, bridges, modems, etc.

As one skilled in the art will recognize, one reason that a packet may be dropped is that a network interface is already storing the maximum number of packets that it can store for transfer to a host computer. In particular, a queue that holds packets to be transferred to a host computer, such as packet queue **116** (shown in FIG. 1A), may be fully populated when another packet is received from a network. Either the new packet or a packet already stored in the queue may be dropped.

Partly because of the bursty nature of much network traffic, multiple packets may often be dropped when a network interface is congested. And, in some network interfaces, if successive packets are dropped one particular network connection or flow (e.g., a connection or flow that includes all of the dropped packets) may be penalized even if it is not responsible for the high rate of packet arrival. If a network connection or flow is penalized too heavily, the network entity generating the traffic in that connection or flow may tear it down in the belief that a "broken pipe" has been encountered. As one skilled in the art will recognize, a broken pipe occurs when a network entity interprets a communication problem as indicating that a connection has been severed.

For certain network traffic (e.g., TCP traffic), the dropping of a packet may initiate a method of flow control in which a network entity's window (e.g., number of packets it transmits before waiting for an acknowledgement) shrinks or is reset to a very low number. Thus, every time a packet from a TCP communicant is dropped by a network interface at a receiving entity, the communicant must re-synchronize its connection with the receiving entity. If one or a subset of communicants are responsible for a large percentage of network traffic received at the entity, then it seems fair that those communicants should be penalized in proportion to the amount of traffic that it is responsible for.

In addition, it may be wise to prevent certain packets or types of packets from being discarded. For example, discarding a small control packet may do very little to alleviate congestion in a network interface and yet have a drastic and negative effect upon a network connection or flow. Further, if a network interface is optimized for packets adhering to a particular protocol, it may be more efficient to avoid dropping such packets. Even further, particular connections, flows or applications may be prioritized, in which case higher priority traffic should not be dropped.

Thus, in one embodiment of a network interface according to the present invention, a method is provided for randomly discarding a packet when a communication device's packet queue is full or is filled to some threshold level. Intelligence may be added to such a method by selecting certain types of packets for discard (e.g., packets from a particular flow, connection or application) or excepting certain types of packets from being discarded (e.g., control packets, packets conforming to a particular protocol or set of protocols).

A provided method is random in that discarded packets are selected randomly from those packets that are considered discardable. Applying a random discard policy may be sufficient to avoid broken pipes by distributing the impact of dropped packets among multiple connections or flows. In addition, if a small number of transmitting entities are responsible for a majority of the traffic received at a network interface, dropping packets randomly may ensure that the offending entities are penalized proportionately. Different embodiments of the invention that are discussed below provide various combinations of randomness and intelligence, and one of these attributes may be omitted in one or more embodiments.

FIG. **24** depicts a system and method for randomly discarding packets in a present embodiment of the invention. In this embodiment, packet queue **2400** is a hardware FIFO (e.g., first-in first-out) queue that is 16 KB in size. In other embodiments of the invention the packet queue may be smaller or larger or may comprise another type of data structure (e.g., list, array, table, heap) implemented in hardware or software.

Similar to packet queue **116** discussed in a previous section, packet queue **2400** receives packets from a network and holds them for transfer to a host computer. Packets arriving from a network may arrive from the network at a high rate and may be processed or examined by one or more modules (e.g., header parser **106**, flow database manager **108**) prior to being stored in packet queue **2400**. For example, where the network is capable of transmitting one gigabit of traffic per second, packets conforming to one set of protocols (e.g., Ethernet, IP and TCP) may be received at a rate of approximately 1.48 million packets per second. After being stored in packet queue **2400**, packets are transferred to a host computer at a rate partially dependent upon events and conditions internal to the host computer. Thus, the network interface may not be able to control the rate of packet transmittal to the host computer.

In the illustrated embodiment, packet queue **2400** is divided into a plurality of zones or regions, any of which may overlap or share a common boundary. Packet queue **2400** may be divided into any number of regions, and the invention is not limited to the three regions depicted in FIG. **24**. Illustratively, region zero (represented by the numeral **2402**) encompasses the portion of packet queue **2400** from 0 KB (e.g., no packets are stored in the queue) to 8 KB (e.g., half full). Region one (represented by the numeral **2404**) encompasses the portion of the packet queue from 8 KB to 12 KB. Region two (represented by the numeral **2406**) encompasses the remaining portion of the packet queue, from 12 KB to 16 KB. In an alternative embodiment, regions may only be defined for a portion of packet queue **2400**. For example, only the upper half (e.g., above 8 KB) may be divided into one or more regions.

The number and size of the different regions and the location of boundaries between the regions may vary according to several factors. Among the factors are the type of packets received at the network interface (e.g., the protocols according to which the packets are configured), the size of the packets, the rate of packet arrival (e.g., expected rate, average rate, peak rate), the rate of packet transfer to the host computer, the size of the packet queue, etc. For example, in another embodiment of the invention, packet queue **2400** is divided into five regions. A first region extends from 0 KB to 8 KB; a second region ranges from 8 KB to 10 KB; a third from 10 KB to 12 KB; a fourth from 12 KB to 14 KB;, and a final region extends from 14 KB to 16 KB.

During operation of a network interface according to a present embodiment, traffic indicator **2408** indicates how full packet queue **2400** is. Traffic indicator **2408**, in one embodiment of the invention, comprises read pointer **810** and/or write pointer **812** (shown in FIG. **8**). In the presently discussed embodiment in which packet queue **2400** is fully partitioned, traffic indicator **2408** will generally be located in one of the regions into which the packet queue was divided or at a dividing boundary. Thus, during operation of a network interface appropriate action may be taken, as described below, depending upon how full the packet queue is (e.g., depending upon which region is identified by traffic indicator **2408**).

In FIG. **24**, counter **2410** is incremented as packets arrive at packet queue **2400**. In the illustrated embodiment, counter **2410** continuously cycles through a limited range of values, such as zero through seven. In one embodiment of the invention, each time a new packet is received the counter is incremented by one. In an alternative embodiment, counter **2410** may not be incremented when certain "non-discardable" packets are received. Various illustrative criteria for identifying non-discardable packets are presented below.

For one or more regions of packet queue **2400**, an associated programmable probability indicator indicates the probability that a packet will be dropped when traffic indicator **2408** indicates that the level of traffic in the packet queue has reached the associated region. Therefore, in the illustrated embodiment probability indicator **2412** indicates the probability that a packet will be dropped while the packet queue is less than half full (e.g., when traffic indicator **2408** is located in region zero). Similarly, probability indicators **2414** and **2416** specify the probability that a new packet will be dropped when traffic indicator **2408** identifies regions one and two, respectively.

In the illustrated embodiment, probability indicators **2412**, **2414** and **2416** each comprise a set, or mask, of sub-indicators such as bits or flags. Illustratively, the number of sub-indicators in a probability indicator matches the range of counter values—in this case, eight. In one embodiment of the invention, each sub-indicator may have one of two values (e.g., zero or one) indicating whether a packet is dropped. Thus, the sub-elements of a probability indicator may be numbered from zero to seven (illustratively, from right to left) to correspond to the eight possible values of counter **2410**. For each position in a probability indicator that stores a first value (e.g., one), when the value of counter **2410** matches the number of that bit, the next discardable packet received for packet queue **2400** will be dropped. As discussed above, certain types of packets (e.g., control packets) may not be dropped. Illustratively, counter **2410** is only incremented for discardable packets.

In FIG. **24**, probability indicator **2412** (e.g., 00000000) indicates that no packets are to be dropped as long as the packet queue is less than half full (e.g., as long as traffic indicator **2408** is in region zero). Probability indicator **2414** (e.g., 00000001) indicates that every eighth packet is to be dropped when there is at least 8 KB stored in the packet queue. In other words, when traffic indicator **2408** is located in region one, there is a 12.5% probability that a discardable packet will be dropped. In particular, when counter **2410** equals zero the next discardable packet, or a packet already stored in the packet queue, is discarded. Probability indicator **2416** (e.g., 01010101) specifies that every other discardable packet is to be dropped. There is thus a 50% probability that a discardable packet will be dropped when the queue is more than three-quarters full. Illustratively, when a packet is dropped, counter **2410** is still incremented.

As another example, in the alternative embodiment described above in which the packet queue is divided into five regions, suitable probability indicators may include the following. For regions zero and one, 00000000; for region two, 00000001; for region three, 00000101; and for region four, 01111111. Thus, in this alternative embodiment, region one is treated as an extension to region zero. Further, the probability of dropping a packet has a wider range, from 0% to 87.5%.

In one alternative embodiment described above, only a portion of a packet queue is partitioned into regions. In this alternative embodiment, a default probability or null probability (e.g., 00000000) of dropping a packet may be associated with the un-partitioned portion. Illustratively, this ensures that no packets are dropped before the level of traffic stored in the queue reaches a first threshold. Even in an embodiment where the entire queue is partitioned, a default or null probability may be associated with a region that encompasses or borders a 0 KB threshold.

Just as a packet queue may be divided into any number of regions for purposes of the present invention, probability indicators may comprise bit masks of any size or magnitude, and need not be of equal size or magnitude. Further, probability indicators are programmable in a present embodiment, thus allowing them to be altered even during the operation of a network interface.

One skilled in the art will recognize that discarding packets on the basis of a probability indicator injects randomness into the discard process. A random early discard policy may be sufficient to avoid the problem of broken pipes discussed above. In particular, in one embodiment of the invention, all packets are considered discardable, such that all packets are counted by counter **2410** and all are candidates for being dropped. As already discussed, however, in another embodiment of the invention intelligence is added in the process of excluding certain types of packets from being discarded.

It will be understood that probability indicators and a counter simply constitute one system for enabling the random discard of packets in a network interface. Other mechanisms are also suitable. In one alternative embodiment, a random number generator may be employed in place of a counter and/or probability indicators to enable a random discard policy. For example, when a random number is generated, such as M, the Mth packet (or every Mth packet) after the number is generated may be dropped. Or, the random number may specify a probability of dropping a packet. The random number may thus be limited to (e.g., hashed into) a certain range of values or probabilities. As another alternative, a random number generator may be used in tandem with multiple regions or thresholds within a packet queue. In this alternative embodiment a programmable value, represented here as N, may be associated with a region or queue threshold. Then, when a traffic indicator reaches that threshold or region, the Nth packet (or every Nth packet) may be dropped until another threshold or boundary is reached.

In yet another alternative embodiment of the invention, the probability of dropping a packet is expressed as a binary fraction. As one skilled in the art will recognize, a binary fraction consists of a series of bits in which each bit represents one half of the magnitude of its more significant neighbor. For example, a binary fraction may use four digits in one embodiment of the invention. From left to right, the bits may represent 0.5, 0.25, 0.125 and 0.0625, respectively. Thus, a binary fraction of 1010 would be interpreted as indicating a 62.5% probability of dropping a packet (e.g., 50% plus 12.5%). The more positions (e.g., bits) used in a binary fraction, the greater precision that may be attained.

In one implementation of this alternative embodiment a separate packet counter is associated with each digit. The counter for the leftmost bit increments at twice the rate of the next counter, which increments twice as fast as the next counter, etc. In other words, when the counter for the most significant (e.g., left) bit increments from 0 to 1 the other counters do not change. When the most significant counter increments again, from 1 back to 0, then the next counter increments from 0 to 1. Likewise, the counter for the third bit does not increment from 0 to 1 until the second counter returns to 0. In summary, the counter for the most significant bit changes (i.e., increments) each time a packet is received. The counter for the next most significant bit maintains each value (i.e., 0 or 1) for two packets before incrementing. Similarly, the counter for the third most significant bit maintains each counter value for four packets before incrementing and the counter for the least significant bit maintains its values for eight packets before incrementing.

Each time a packet is received or a counter is incremented the counters are compared to the probability indicator (e.g., the specified binary fraction). In one embodiment the determination of whether a packet is dropped depends upon which of the fraction's bits are equal to one. Illustratively, for each fraction bit equal to one a random packet is dropped if the corresponding counter is equal to one and the counters for any bits of higher significance are equal to zero. Thus for the example fraction 1010, whenever the most significant bit's counter is equal to one a random packet is dropped. In addition, a random packet is also dropped whenever the counter for the third bit is equal to one and the counters for the first two bits are equal to zero.

A person skilled in the art may also derive other suitable mechanisms for specifying and enforcing a probability of

dropping a packet received at a network interface without exceeding the scope of the present invention.

As already mentioned, intelligence may be imparted to a random discard policy in order to avoid discarding certain types of packets. In a previous section, methods of parsing a packet received from a network were described. In particular, in a present embodiment of the invention a packet received from a network is parsed before it is placed into a packet queue such as packet queue **2400**. During the parsing procedure various information concerning the packet may be gleaned. This information may be used to inject intelligence into a random discard policy. In particular, one or more fields of a packet header may be copied, an originating or destination entity of the packet may be identified, a protocol may be identified, etc.

Thus, in various embodiments of the invention, certain packets or types of packets may be immune from being discarded. In the embodiment illustrated in FIG. **24**, for example, control packets are immune. As one skilled in the art will appreciate, control packets often contain information essential to the establishment, re-establishment or maintenance of a communication connection. Dropping a control packet may thus have a more serious and damaging effect than dropping a packet that is not a control packet. In addition, because control packets generally do not contain data, dropping a control packet may save very little space in the packet queue.

Many other criteria for immunizing packets are possible. For example, when a packet is parsed according to a procedure described in a previous section, a No_Assist flag or signal may be associated with the packet to indicate whether the packet is compatible with a set of pre-selected communication protocols. Illustratively, if the flag is set to a first value (e.g., one) or the signal is raised, the packet is considered incompatible and is therefore ineligible for certain processing enhancements (e.g., re-assembly of packet data, batch processing of packet headers, load-balancing). Because a packet for which a No_Assist flag is set to the first value may be a packet conforming to an unexpected protocol or unique format, it may be better not to drop such packets. For example, a network manager may want to ensure receipt of all such packets in order to determine whether a parsing procedure should be augmented with the ability to parse additional protocols.

Another reason for immunizing a No_Assist packet (e.g., packets that are incompatible with a set of selected protocols) from being discarded concerns the reaction to dropping the packet. Because the packet's protocols were not identified, it may not be known how the packet's protocols respond to the loss of a packet. In particular, if the sender of the packet does not lower its transmission rate in response to the dropped packet (e.g., as a form of congestion control), then there is no benefit to dropping it.

A packet's flow number may be used to immunize certain packets in another alternative embodiment of the invention. As discussed in a previous section, a network interface may include a flow database and flow database manager to maintain a record of multiple communication flows received by the network interface. It may be efficacious to prevent packets from one or more certain flows from being discarded. Immunized flows may include a flow involving a high-priority network entity, a flow involving a particular application, etc. For example, it may be considered relatively less damaging to discard packets from an animated or streaming graphics application in which a packet, or a few packets, may be lost without seriously affecting the destination entity and the packets may not even need to be retransmitted. In contrast, the consequences may be more severe if a few packets are dropped from a file transfer connection. The packets will likely need to be retransmitted, and the transmitting entity's window may be shrunk as a result—thus decreasing the rate of file transfer.

In yet another alternative embodiment of the invention, a probability indicator may comprise a bit mask in which each bit corresponds to a separate, specific flow through the network interface. In particular, the bits may correspond to the flows maintained in the flow database described in a previous section.

Although embodiments of the invention discussed thus far in this section involve discarding packets as they arrive at a packet queue, in an alternative embodiment packets may be discarded from within the packet queue. In particular, as the packet queue is filled (e.g., as a traffic indicator reaches pre-defined regions or thresholds), packets already stored in the queue may be discarded at random according to one or more probability indicators. In the embodiment illustrated in FIG. **24**, for example, when traffic indicator **2408** reaches a certain threshold, such as the boundary between regions one and two or the end of the queue, packets may be deleted in one or more regions according to related probability indicators. Such probability indicators would likely have different values than those indicated in FIG. **24**.

In a present embodiment of the invention, probability indicators and/or the specifications (e.g., boundaries) into which a packet queue is partitioned are programmable and may be adjusted by software operating on a host computer (e.g., a device driver). Criteria for immunizing packets may also be programmable. Methods of discarding packets in a network interface or other communication device may thus be altered in accordance with the embodiments described in this section, even during continued operation of such a device. Various other embodiments and criteria for randomly discarding packets and/or applying criteria for the intelligent discard of packets will be apparent to those skilled in the art.

FIGS. **25A–25B** comprise a flow chart demonstrating one method of implementing a policy for randomly discarding packets in a network interface according to the embodiment of the invention substantially similar to the embodiment illustrated in FIG. **24**. In this embodiment, a packet is received while packet queue **2400** is not yet full. As one skilled in the will appreciate, this embodiment provides a method of determining whether to discard the packet. Once packet queue **2400** is full, when another packet is received the network interface generally must drop a packet—either the one just received or one already stored in the queue—in which case the only decision is which packet to drop.

In FIG. **25A**, state **2500** is a start state. State **2500** may reflect the initialization of the network interface (and packet queue **2400**) or may reflect a point in the operation of the network interface at which one or more parameters or aspects concerning the packet queue and the random discard policy are to be modified.

In state **2502**, one or more regions are identified in packet queue **2400**, perhaps by specifying boundaries such as the 8 KB and 12 KB boundaries depicted in FIG. **24**. Although the regions depicted in FIG. **24** fully encompass packet queue **2400** when viewed in unison, regions in an alternative embodiment of the invention may encompass less than the entire queue.

In state **2504**, one or more probability indicators are assigned and configured. In the illustrated embodiment, one probability indicator is associated with each region. Alternatively, multiple regions may be associated with one

probability indicator. Even further, one or more regions may not be explicitly associated with a probability indicator, in which case a default or null probability indicator may be assumed. As described above, a probability indicator may take the form of a multi-bit mask, whereby the number of bits in the mask reflect the range of possible values maintained by a packet counter. In another embodiment of the invention, a probability indicator may take the form of a random number or a threshold value against which a randomly generated number is compared when a decision must be whether to discard a packet.

In state **2506**, if certain types of packets are to be prevented from being discarded, criteria are expressed to identify the exempt packets. Some packets that may be exempted are control packets, packets conforming to unknown or certain known protocols, packets belonging to a particular network connection or flow, etc. In one embodiment of the invention, no packets are exempt from being discarded.

In state **2508**, a packet or traffic counter is initialized. As described above, the counter may be incremented, possibly through a limited range of values, when a discardable packet is received for storage in packet queue **2400**. The limited range of counter values may correspond to the number of bits in a mask form of a probability indicator. Alternatively, the counter may be configured to increment through a greater range, in which case a counter value may be filtered through a modulus or hash function prior to being compared to a probability indicator as described below.

In state **2510**, a packet is received from a network and may be processed through one or more modules (e.g., a header parser, an IPP module) prior to its arrival at packet queue **2400**. Thus, in state **2510** the packet is ready to be stored in the packet queue. One or more packets may already be stored in the packet queue and a traffic indicator (e.g., a pointer or index) identifies the level of traffic stored in the queue (e.g., by a storage location and/or region in the queue).

In state **2512**, it may be determined whether the received packet is discardable. For example, if the random discard policy that is in effect allows for the exemption of some packets from being discarded, in state **2512** it is determined whether the received packet meets any of the exemption criteria. If so, the illustrated procedure continues at state **2522**. Otherwise, the procedure continues at state **2514**.

In state **2514**, an active region of packet queue **2400** is identified. In particular, the region of the packet queue to which the queue is presently populated with traffic is determined. The level of traffic stored in the queue depends upon the number and size of packets that have been stored in the queue to await transfer to a host computer. The slower the transfer process, the higher the level of traffic may reach in the queue. Although the level of traffic stored in the queue rises and falls as packets are stored and transferred, the level may be identified at a given time by examining the traffic indicator. The traffic indicator may comprise a pointer identifying the position of the last or next packet to be stored in the queue. Such a pointer may be compared to another pointer that identifies the next packet to be transferred to the host computer in order to reveal how much traffic is stored in the queue.

In state **2516**, the counter value (e.g., a value between zero and seven in the embodiment of FIG. **24**) is compared to the probability indicator associated with the active region. As previously described, the counter is incremented as discardable packets are received at the queue. This comparison is conducted so as to determine whether the received packet should be discarded. As explained above, in the embodiment of FIG. **24** the setting of the probability indicator bit corresponding to the counter value is examined. For example, if the counter has a value of N, then bit number N of the probability indicator mask is examined. If the bit is set to a first state (e.g., one) the packet is to be discarded; otherwise it is not to be discarded.

In state **2518**, the counter is incremented to reflect the receipt of a discardable packet, whether or not the packet is to be discarded. In the presently discussed embodiment of the invention, if the counter contains its maximum value (e.g., seven) prior to being incremented, incrementing it entails resetting it to its minimum value (e.g., zero).

In state **2520**, if the packet is to be discarded the illustrated procedure continues at state **2524**. Otherwise, the procedure continues at state **2522**. In state **2522**, the packet is stored in packet queue **2400** and the illustrated procedure ends with end state **2526**. In state **2524**, the packet is discarded and the illustrated procedure ends with end state **2526**.

The foregoing descriptions of embodiments of the invention have been presented for purposes of illustration and description only. They are not intended to be exhaustive or to limit the invention to the forms disclosed. Many modifications and variations will be apparent to practitioners skilled in the art. Accordingly, the above disclosure is not intended to limit the invention; the scope of the invention is defined by the appended claims.

What is claimed is:

1. A method of managing a communication flow comprising one or more packets received by a network interface, the method comprising:

identifying a flow index of a packet received at a network interface, wherein said flow index identifies a communication flow comprising said packet;

searching a flow database for a flow record comprising said flow index;

at a flow database manager, generating an operation code representing an eligibility of said packet for one or more predetermined processes; and

forwarding said operation code to a reassembly engine configured to reassemble a data portion of the packet with a data portion of another packet in the communication flow.

2. The method of claim **1**, further comprising storing said operation code.

3. The method of claim **1**, further comprising receiving packet information extracted from a header of said packet.

4. The method of claim **3**, wherein said packet information comprises a sequence number of said packet.

5. The method of claim **3**, wherein said packet information comprises an indicator configured to indicate whether said packet includes a data portion.

6. The method of claim **3**, wherein said packet information comprises an identifier of a source of said packet and an identifier of a destination of said packet.

7. The method of claim **1**, further comprising updating said flow record.

8. The method of claim **7**, wherein said updating comprises incrementing a flow activity indicator in said flow record.

9. The method of claim **8**, wherein said updating further comprises modifying a flow sequence number in said flow record.

10. The method of claim **1**, further comprising adding a flow record to said flow database comprising said flow index

if a flow record comprising said flow index is not found in said flow database.

11. The method of claim 1, further comprising replacing said flow record.

12. The method of claim 1, wherein said identifying comprises receiving said flow index from a network interface module configured to examine a header portion of said packet.

13. The method of claim 1, wherein said generating comprises:

determining whether said packet is suitable for a function that said network interface is configured to perform; and

assigning an operation code for said packet to indicate whether said function is to be performed.

14. The method of claim 13, wherein said generating further comprises determining whether said packet includes a data portion.

15. The method of claim 14, wherein said generating further comprises determining whether said data portion exceeds a pre-determined size.

16. The method of claim 13, wherein said generating further comprises determining whether said packet was received out of order.

17. The method of claim 13, wherein said generating further comprises determining whether said flow database is full.

18. The method of claim 1, further comprising:

determining whether said flow database is full;

for each flow record in said flow database, examining a flow activity indicator configured to indicate a recency of traffic in an associated communication flow;

selecting an aged flow record having a flow activity indicator indicating least recent traffic among said associated communication flows; and

replacing said aged flow record with a new flow record comprising said flow index.

19. A method of managing a communication flow comprising a collection of data directed from a source entity to a destination entity, the method comprising:

receiving a first packet at a network interface, said first packet comprising a first portion of a collection of data;

identifying a first flow key, said first flow key comprising an identifier of a source of said first packet and an identifier of a destination of said first packet;

setting up a first communication flow for said collection of data, wherein said first communication flow is identifiable by said first flow key; and

assigning an operation code to said first packet, said operation code indicating whether said first portion of data is reassembleable with another portion of data in said collection of data;

wherein said first communication flow is configured to be terminated after said collection of data is received at said network interface.

20. The method of claim 19, further comprising:

receiving a second packet at said network interface, said second packet comprising a second portion of said collection of data;

determining whether said second portion of said collection of data comprises a final portion of said collection of data; and

terminating said first communication flow if said second portion comprises said final portion.

21. The method of claim 20, wherein said setting up comprises:

storing said first flow key in a database; and

indicating that said first communication flow is valid.

22. The method of claim 21, wherein said indicating comprises configuring a validity indicator in said database.

23. The method of claim 22, wherein said terminating comprises modifying said validity indicator to indicate that said first communication flow is invalid.

24. The method of claim 21, wherein said terminating comprises removing said first flow key from said database.

25. The method of claim 19, further comprising:

receiving a second packet at said network interface; and

associating an operation code with said second packet to indicate whether said first communication flow is to be terminated.

26. The method of claim 25, wherein said associating comprises:

receiving information extracted from a header portion of said second packet; and

examining said information to determine whether said first communication flow is to be terminated.

27. The method of claim 26, wherein said associating further comprises examining said information to determine whether a second communication flow is to be established and whether a data portion of said second packet is to be re-assembled with a data portion of another packet.

28. A method of managing a network flow received at a network interface, comprising:

parsing a packet received at a network interface;

assembling a flow identifier configured to identify a network flow comprising said packet;

searching a flow database on said network interface for said flow identifier;

updating a flow sequence number in a flow database record comprising said flow identifier;

setting a flow activity indicator in said flow database record to reflect receipt of said packet; and

setting a flow validity indicator in said flow database to indicate said network flow is valid.

29. A method of processing a packet received at a network interface, comprising:

receiving a packet at a network interface, wherein said packet was sent from a source entity to a destination entity;

parsing said packet to identify a flow between said source entity and said destination entity that comprises said packet;

receiving a status indicator extracted from said packet;

searching a flow database for said flow;

generating an operation code based on said status indicator, wherein said operation code is configured to:

indicate whether said packet is a control packet; and

indicate whether a header portion of said packet conforms to one of a set of pre-selected communication protocols; and

updating said flow database by:

updating a flow activity indicator associated with said flow to reflect receipt of said packet; and

updating a flow validity indicator associated with said flow to indicate said flow is valid.

30. The method of claim 29, wherein said generating comprises determining whether said status indicator has a predetermined value.

31. The method of claim 29, wherein said generating comprises determining whether said packet includes a data portion.

32. The method of claim 29, wherein said generating comprises determining whether a data portion of said packet exceeds a predetermined size.

33. The method of claim 29, wherein said generating comprises determining whether a sequence number of said packet correlates with a sequence number associated with said flow in said flow database.

34. The method of claim 29, wherein said generating comprises determining whether said packet comprises a request to reset a flow.

35. The method of claim 29, further comprising determining whether said flow database is full.

36. The method of claim 29, wherein said parsing comprises assembling a flow key configured to identify a communication flow between said source entity and said destination entity.

37. The method of claim 36, wherein said searching comprises searching a flow database for said flow key.

38. A network interface for receiving a communication flow from a network, comprising:

a parser for examining a header portion of a first packet received from a network, said first packet comprising a first portion of data transmitted from a source entity to a destination entity;

a data structure comprising:

a flow key for identifying said communication flow, wherein said flow key comprises identifiers of said source entity and said destination entity;

an activity indicator for indicating a recency with which a packet in said communication flow has been received; and

a validity indicator for indicating whether said communication flow is valid;

a data manager for managing said data structure; and

a generator configured to generate an operation code for every packet in said communication flow, to facilitate forwarding of said data toward the destination entity from the network interface;

wherein said data manager establishes said communication flow and stores said flow key in said data structure upon receipt of said first portion of data, and terminates said communication flow upon receipt of a final portion of said data.

39. A network interface, comprising:

a database configured to facilitate management of a network flow, said network flow comprising one or more packets sent from a source entity to a destination entity, said database comprising:

a flow key configured to identify said network flow; and

a validity indicator configured to indicate whether said network flow is valid;

a database manager configured to manage said database; and

an operation code generator configured to generate an operation code for every packet within said network flow, wherein said operation code is configured to specify an operation to be performed with said packet;

wherein said database manager receives said flow key and updates said database when said packet is received.

40. The network interface of claim 39, further comprising:

a control memory;

wherein said database manager further stores said operation code in said control memory when said packet is received.

41. The network interface of claim 39, wherein said database manager comprises said operation code generator.

42. A computer readable storage medium storing instructions that, when executed by a computer, cause the computer to perform a method of managing a network flow database storing information relating to a network flow received by a network interface, the method comprising:

identifying a flow index of a packet received at a network interface, wherein said flow index identifies a communication flow comprising said packet;

searching a flow database for a flow record comprising said flow index;

at a flow database manager, generating an operation code representing an eligibility of said packet for one or more predetermined processes; and

forwarding said operation code to a reassembly engine configured to reassemble a data portion of the packet with a data portion of another packet in the communication flow.

43. The network interface of claim 39, wherein said database further comprises an activity indicator for indicating a recency with which a packet in said network flow has been received.

44. The method of claim 1, wherein said one or more predetermined processes include reassembly of a data portion of the packet with a data portion of another packet in the communication flow.

45. The method of claim 1, wherein said one or more predetermined processes include batch processing of headers of multiple packets in the communication flow, including the packet.

46. The method of claim 1, wherein said one or more predetermined processes include distributing packets of different communication flows to different host computer processors.

47. The method of claim 1, further comprising forwarding said operation code to a packet batching module configured to facilitate batch processing of headers of multiple packets in the communication flow.

48. A computer readable storage medium storing instructions that, when executed by a computer, cause the computer to perform a method of managing a communication flow comprising a collection of data directed from a source entity to a destination entity, the method comprising:

receiving a first packet at a network interface, said first packet comprising a first portion of a collection of data;

identifying a first flow key, said first flow key comprising an identifier of a source of said first packet and an identifier of a destination of said first packet;

setting up a first communication flow for said collection of data, wherein said first communication flow is identifiable by said first flow key; and

assigning an operation code to said first packet, said operation code indicating whether said first portion of data is reassembleable with another portion of data in said collection of data;

wherein said first communication flow is configured to be terminated after said collection of data is received at said network interface.

49. A computer readable storage medium storing instructions that, when executed by a computer, cause the computer

to perform a method of processing a packet received at a network interface, the method comprising:

receiving a packet at a network interface, wherein said packet was sent from a source entity to a destination entity;

parsing said packet to identify a flow between said source entity and said destination entity that comprises said packet;

receiving a status indicator extracted from said packet;

searching a flow database for said flow;

generating an operation code based on said status indicator, wherein said operation code is configured to:

indicate whether said packet is a control packet; and

indicate whether a header portion of said packet conforms to one of a set of pre-selected communication protocols; and

updating said flow database by:

updating a flow activity indicator associated with said flow to reflect receipt of said packet; and

updating a flow validity indicator associated with said flow to indicate said flow is valid.

50. The method of claim **28**, wherein said flow sequence number comprises a sequence number of said packet.

51. The method of claim **28**, wherein said flow activity indicator is configured to indicate how recently said network flow was active.

52. The method of claim **28**, wherein said setting a flow activity indicator comprises incrementing said flow activity indicator.

53. The method of claim **28**, further comprising:

generating an operation code configured to identify a status of said packet.

54. The method of claim **53**, wherein said operation code is configured to indicate whether said packet includes a data portion.

55. The method of claim **53**, wherein said operation code is configured to indicate whether said packet includes a data portion larger than a pre-determined size.

56. The method of claim **53**, wherein said operation code is configured to indicate whether said packet was received out of order.

57. The method of claim **53** wherein said operation code is configured to indicate whether said packet is reassembleable with another packet in said network flow.

58. The method of claim **53**, said operation code is configured to indicate whether said network flow is to be terminated.

59. A computer readable storage medium storing instructions that, when executed by a computer, cause the computer to perform a method of managing a network flow received at a network interface, the method comprising:

parsing a packet received at a network interface;

assembling a flow identifier configured to identify a network flow comprising said packet;

searching a flow database on said network interface for said flow identifier;

updating a flow sequence number in a flow database record comprising said flow identifier;

setting a flow activity indicator in said flow database record to reflect receipt of said packet; and

setting a flow validity indicator in said flow database to indicate said network flow is valid.

* * * * *

US006665725B1

(12) **United States Patent**
Dietz et al.

(10) **Patent No.: US 6,665,725 B1**
(45) **Date of Patent: Dec. 16, 2003**

(54) **PROCESSING PROTOCOL SPECIFIC INFORMATION IN PACKETS SPECIFIED BY A PROTOCOL DESCRIPTION LANGUAGE**

(75) Inventors: **Russell S. Dietz**, San Jose, CA (US); **Andrew A. Koppenhaver**, Littleton, CO (US); **James F. Torgerson**, Andover, MN (US)

(73) Assignee: **Hi/fn, Inc.**, Los Gatos, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 537 days.

(21) Appl. No.: **09/609,179**

(22) Filed: **Jun. 30, 2000**

Related U.S. Application Data

(60) Provisional application No. 60/141,903, filed on Jun. 30, 1999.

(51) **Int. Cl.**[7] **G06F 13/00**

(52) **U.S. Cl.** **709/230**; 709/246; 709/228; 370/389

(58) **Field of Search** 709/203, 206, 709/216, 217, 222, 246, 225, 228, 230, 232; 703/26; 370/489, 13, 17

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,736,320	A	4/1988	Bristol	364/300
4,891,639	A	1/1990	Nakamura	340/825.5
5,101,402	A	3/1992	Chui et al.	370/17
5,247,517	A	9/1993	Ross et al.	370/85.5
5,247,693	A	9/1993	Bristol	709/203
5,315,580	A	5/1994	Phaal	370/13
5,339,268	A	8/1994	Machida	365/49
5,351,243	A	9/1994	Kalkunte et al.	370/92
5,365,514	A	11/1994	Hershey et al.	370/17
5,375,070	A	12/1994	Hershey et al.	364/550
5,394,394	A	2/1995	Crowther et al.	370/60
5,414,650	A	5/1995	Hekhuis	364/715.02
5,414,704	A	5/1995	Spinney	370/60

(List continued on next page.)

OTHER PUBLICATIONS

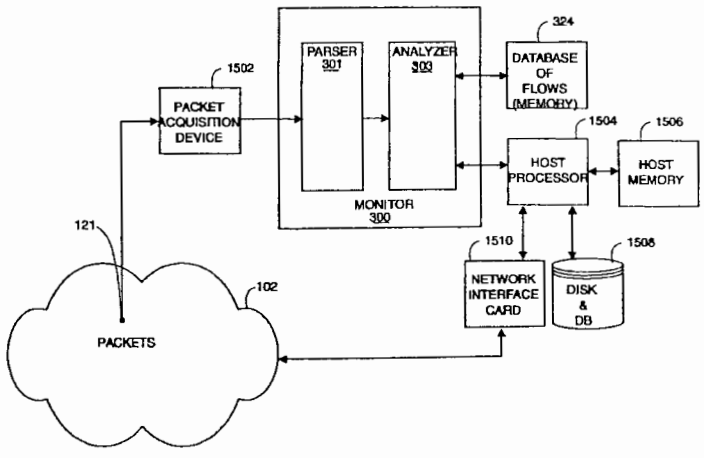
"Technical Note: the Narus System," Downloaded Apr. 29, 1999 from www.narus.com, Narus Corporation, Redwood City California.

Primary Examiner—Hosain T. Alam
Assistant Examiner—Khanh Quang Dinh
(74) *Attorney, Agent, or Firm*—Dov Rosenfeld; Inventek

(57) **ABSTRACT**

A method of performing protocol specific operations on a packet passing through a connection point on a computer network. The packet contents conform to protocols of a layered model wherein the protocol at a at a particular layer level may include one or a set of child protocols defined for that level. The method includes receiving the packet and receiving a set of protocol descriptions for protocols may be used in the packet. A protocol description for a particular protocol at a particular layer level includes any child protocols of the particular protocol, and for any child protocol, where in the packet information related to the particular child protocol may be found. A protocol description also includes any protocol specific operations to be performed on the packet for the particular protocol at the particular layer level. The method includes performing the protocol specific operations on the packet specified by the set of protocol descriptions based on the base protocol of the packet and the children of the protocols used in the packet. A particular embodiment includes providing the protocol descriptions in a high-level protocol description language, and compiling to the descriptions into a data structure. The compiling may further include compressing the data structure into a compressed data structure. The protocol specific operations may include parsing and extraction operations to extract identifying information. The protocol specific operations may also include state processing operations defined for a particular state of a conversational flow of the packet.

17 Claims, 20 Drawing Sheets



U.S. PATENT DOCUMENTS

5,430,709	A		7/1995	Galloway 370/13
5,432,776	A		7/1995	Harper 370/17
5,493,689	A		2/1996	Waclawsky et al. 709/206
5,500,855	A		3/1996	Hershey et al. 370/17
5,511,215	A		4/1996	Terasaka et al. 709/246
5,568,471	A		10/1996	Hershey et al. 370/17
5,574,875	A		11/1996	Stansfield et al. 395/403
5,586,266	A		12/1996	Hershey et al. 709/216
5,606,668	A		2/1997	Shwed 709/216
5,608,662	A		3/1997	Large et al. 364/724.01
5,634,009	A		5/1997	Iddon et al. 709/206
5,651,002	A		7/1997	Van Seters et al. 370/392
5,680,585	A	*	10/1997	Bruell 703/26
5,684,954	A		11/1997	Kaiserswerth et al. 709/203
5,703,877	A		12/1997	Nuber et al. 370/395
5,721,827	A	*	2/1998	Logan et al. 709/217
5,732,213	A		3/1998	Gessel et al. 709/216
5,740,355	A		4/1998	Watanabe et al. 395/183.21
5,761,424	A		6/1998	Adams et al. 709/232
5,764,638	A		6/1998	Ketchum 370/401
5,781,735	A		7/1998	Southard 709/238
5,784,298	A		7/1998	Hershey et al. 364/557
5,787,253	A		7/1998	McCreery et al. 709/227
5,805,808	A		9/1998	Hansani et al. 709/203
5,812,529	A		9/1998	Czarnik et al. 370/245
5,819,028	A		10/1998	Manghirmalani et al. ... 709/203
5,825,774	A		10/1998	Ready et al. 370/401
5,826,017	A		10/1998	Holzmann 709/206
5,835,726	A		11/1998	Shwed et al. 709/228
5,838,919	A		11/1998	Schwaller et al. 709/208
5,841,895	A		11/1998	Huffman 382/155
5,850,386	A		12/1998	Anderson et al. 370/241
5,850,388	A		12/1998	Anderson et al. 370/252
5,862,335	A		1/1999	Welch, Jr. et al. 709/232
5,878,420	A		3/1999	de la Salle 707/10
5,893,155	A		4/1999	Cheriton 711/144
5,903,754	A		5/1999	Pearson 709/238
5,917,821	A		6/1999	Gobuyan et al. 370/392
6,014,380	A		1/2000	Hendel et al. 370/392
6,272,151	B1	*	8/2001	Gupta et al. 370/489
6,430,409	B1	*	8/2002	Rossmann 455/422.1
6,516,337	B1	*	2/2003	Tripp et al. 709/202
6,519,568	B1	*	2/2003	Harvey et al. 705/1

* cited by examiner

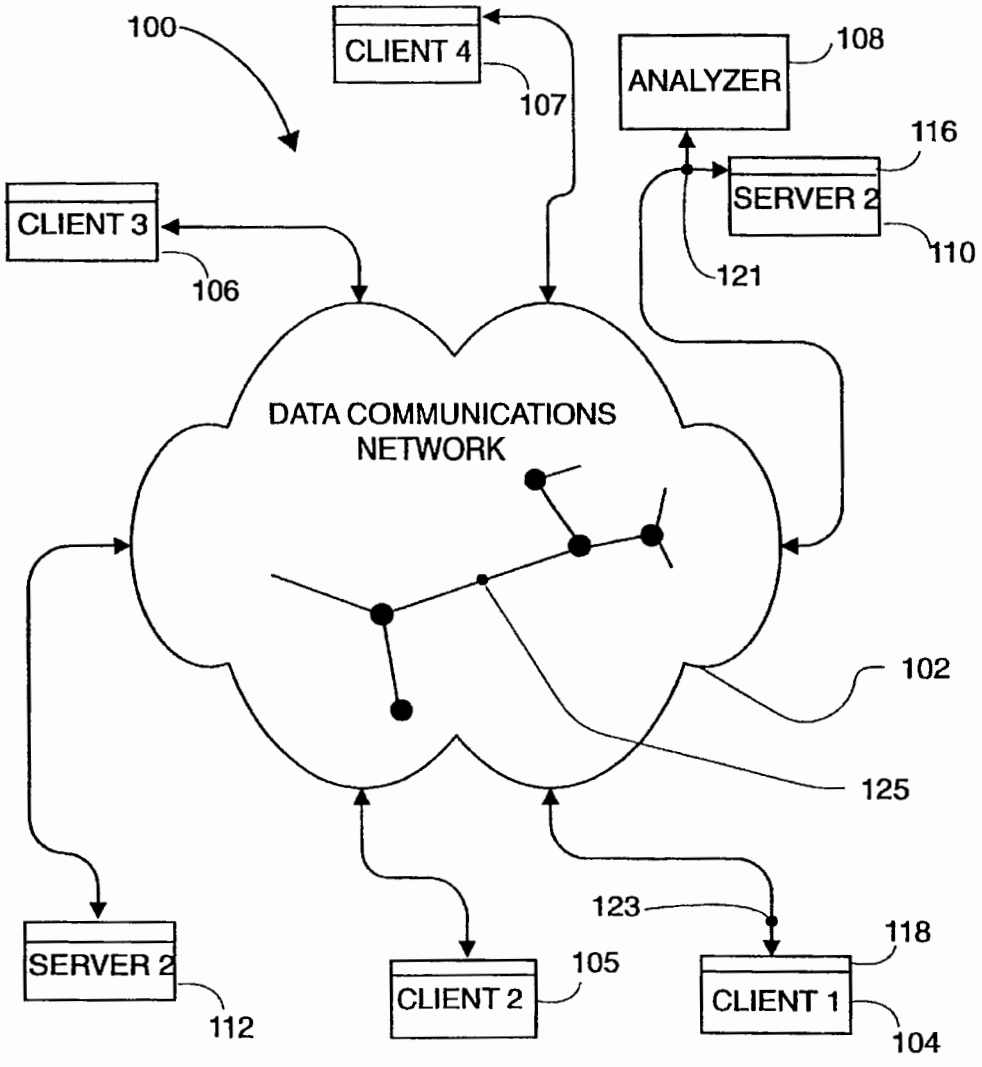

100
CLIENT 4
107
ANALYZER
108
116
SERVER 2
110
121
CLIENT 3
106
DATA COMMUNICATIONS NETWORK
102
125
123
118
SERVER 2
112
CLIENT 2
105
CLIENT 1
104

FIG. 1

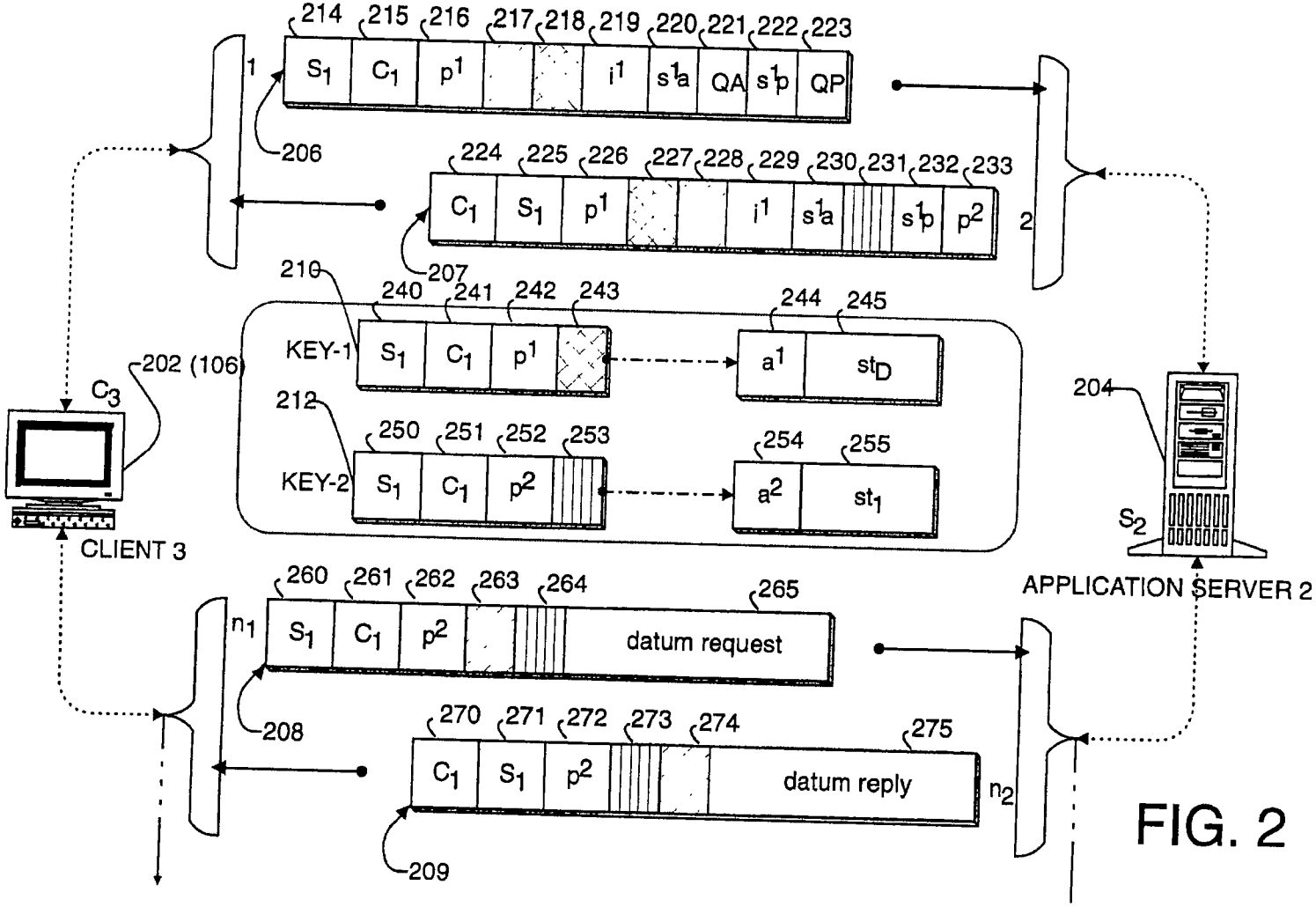

214 215 216 217 218 219 220 221 222 223
S1 C1 p1 i1 s1a QA s1p QP
1
206
224 225 226 227 228 229 230 231 232 233
C1 S1 p1 i1 s1a s1p p2
2
207
210
240 241 242 243 244 245
KEY-1 S1 C1 p1 a1 stD
212
250 251 252 253 254 255
KEY-2 S1 C1 p2 a2 st1
202 (106)
C3
CLIENT 3
204
S2
APPLICATION SERVER 2
260 261 262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

FIG. 2

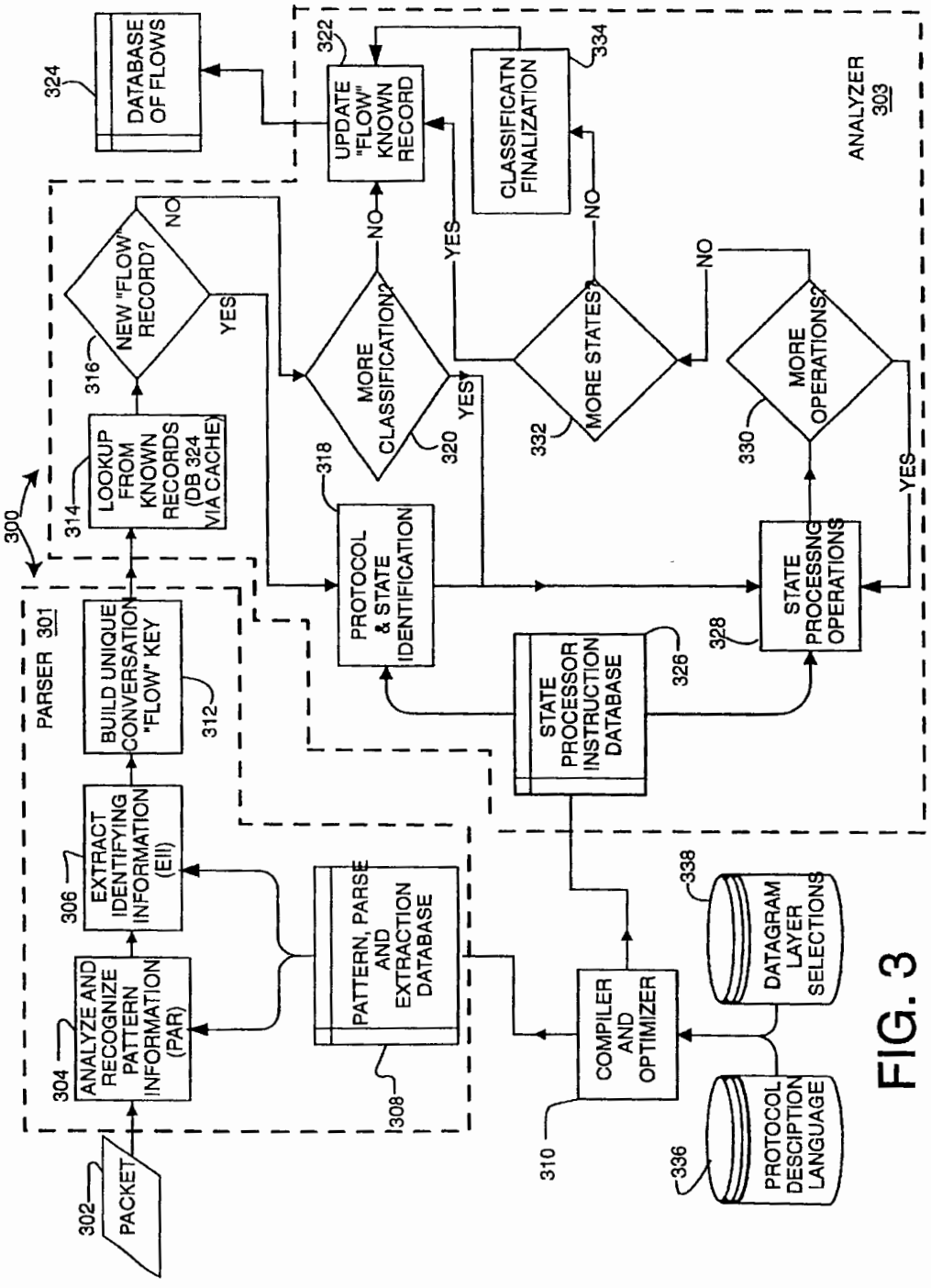

300
PARSER 301
ANALYZER 303
302
PACKET
304
ANALYZE AND RECOGNIZE PATTERN INFORMATION (PAR)
306
EXTRACT IDENTIFYING INFORMATION (EII)
308
PATTERN, PARSE AND EXTRACTION DATABASE
310
COMPILER AND OPTIMIZER
312
BUILD UNIQUE CONVERSATION "FLOW" KEY
314
LOOKUP FROM KNOWN RECORDS (DB 324 VIA CACHE)
316
NEW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

FIG. 3

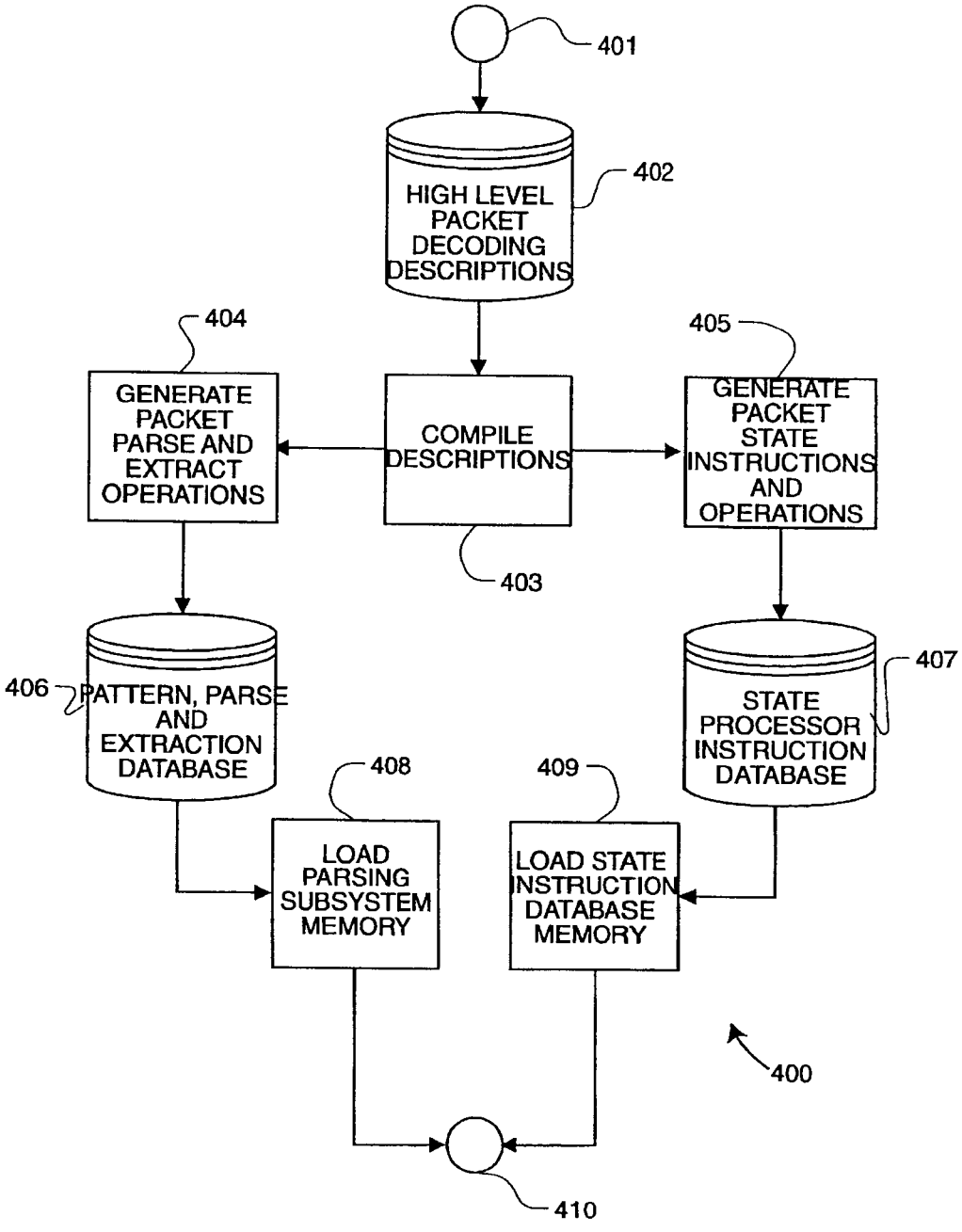

401
HIGH LEVEL PACKET DECODING DESCRIPTIONS
402
404
405
GENERATE PACKET PARSE AND EXTRACT OPERATIONS
COMPILE DESCRIPTIONS
GENERATE PACKET STATE INSTRUCTIONS AND OPERATIONS
403
406
PATTERN, PARSE AND EXTRACTION DATABASE
407
STATE PROCESSOR INSTRUCTION DATABASE
408
409
LOAD PARSING SUBSYSTEM MEMORY
LOAD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

FIG. 4

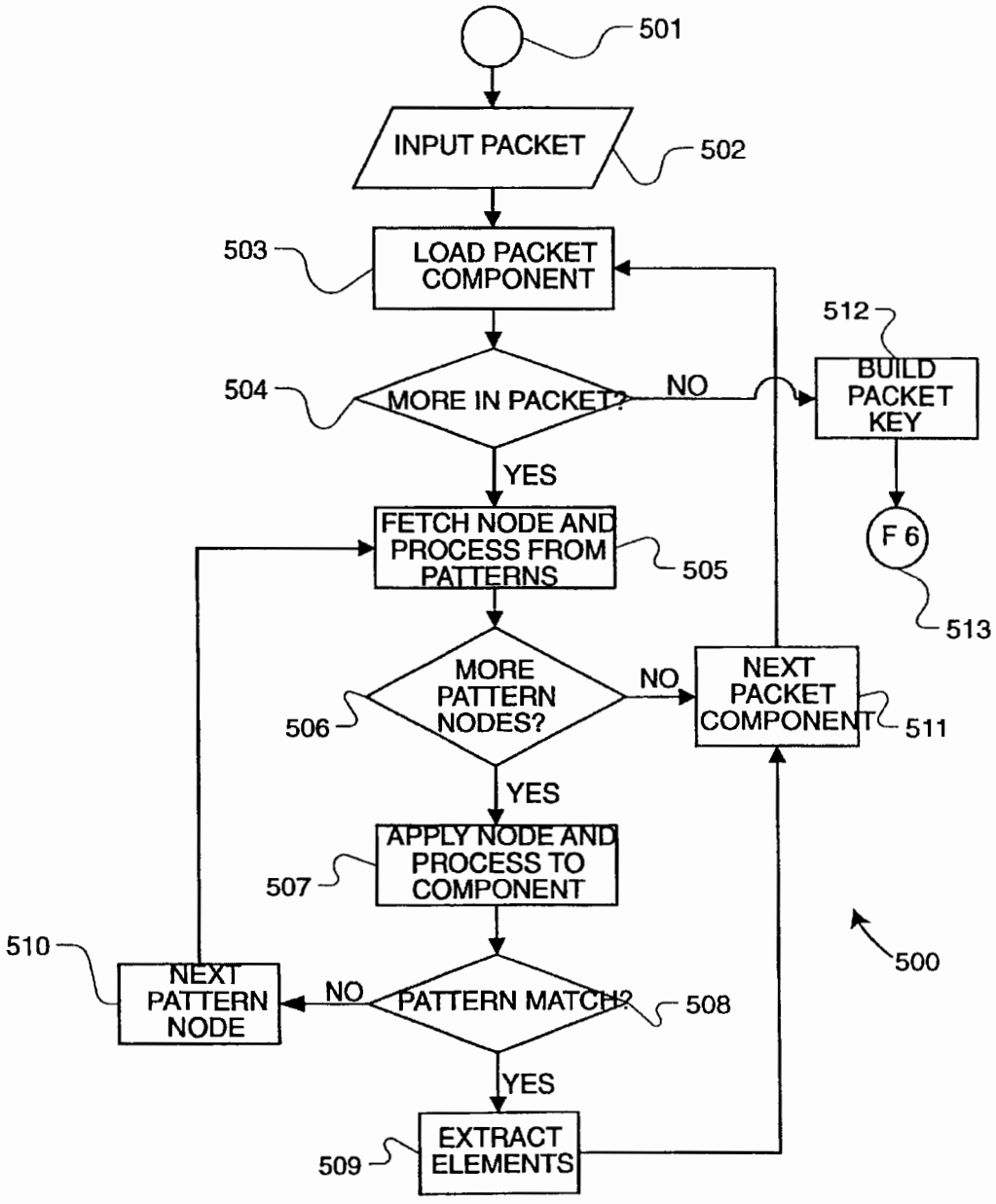

501
INPUT PACKET
502
503
LOAD PACKET COMPONENT
504
MORE IN PACKET?
NO
512
BUILD PACKET KEY
F 6
513
YES
FETCH NODE AND PROCESS FROM PATTERNS
505
MORE PATTERN NODES?
506
NO
NEXT PACKET COMPONENT
511
YES
APPLY NODE AND PROCESS TO COMPONENT
507
500
510
NEXT PATTERN NODE
NO
PATTERN MATCH?
508
YES
EXTRACT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

FIG. 5

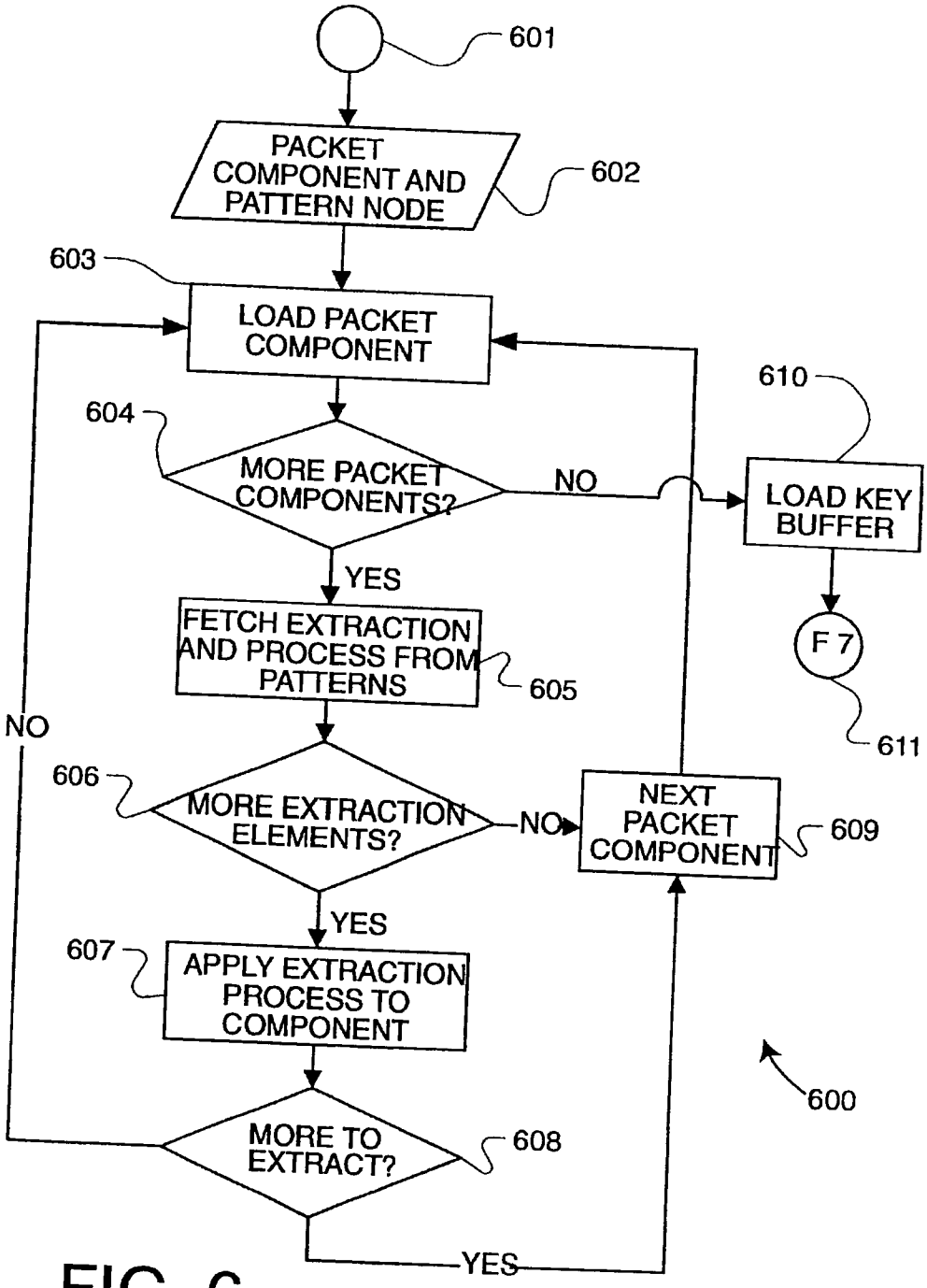

601
PACKET COMPONENT AND PATTERN NODE
602
603
LOAD PACKET COMPONENT
604
MORE PACKET COMPONENTS?
NO
610
LOAD KEY BUFFER
F 7
611
YES
FETCH EXTRACTION AND PROCESS FROM PATTERNS
605
NO
606
MORE EXTRACTION ELEMENTS?
NO
NEXT PACKET COMPONENT
609
YES
607
APPLY EXTRACTION PROCESS TO COMPONENT
600
MORE TO EXTRACT?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

FIG. 6

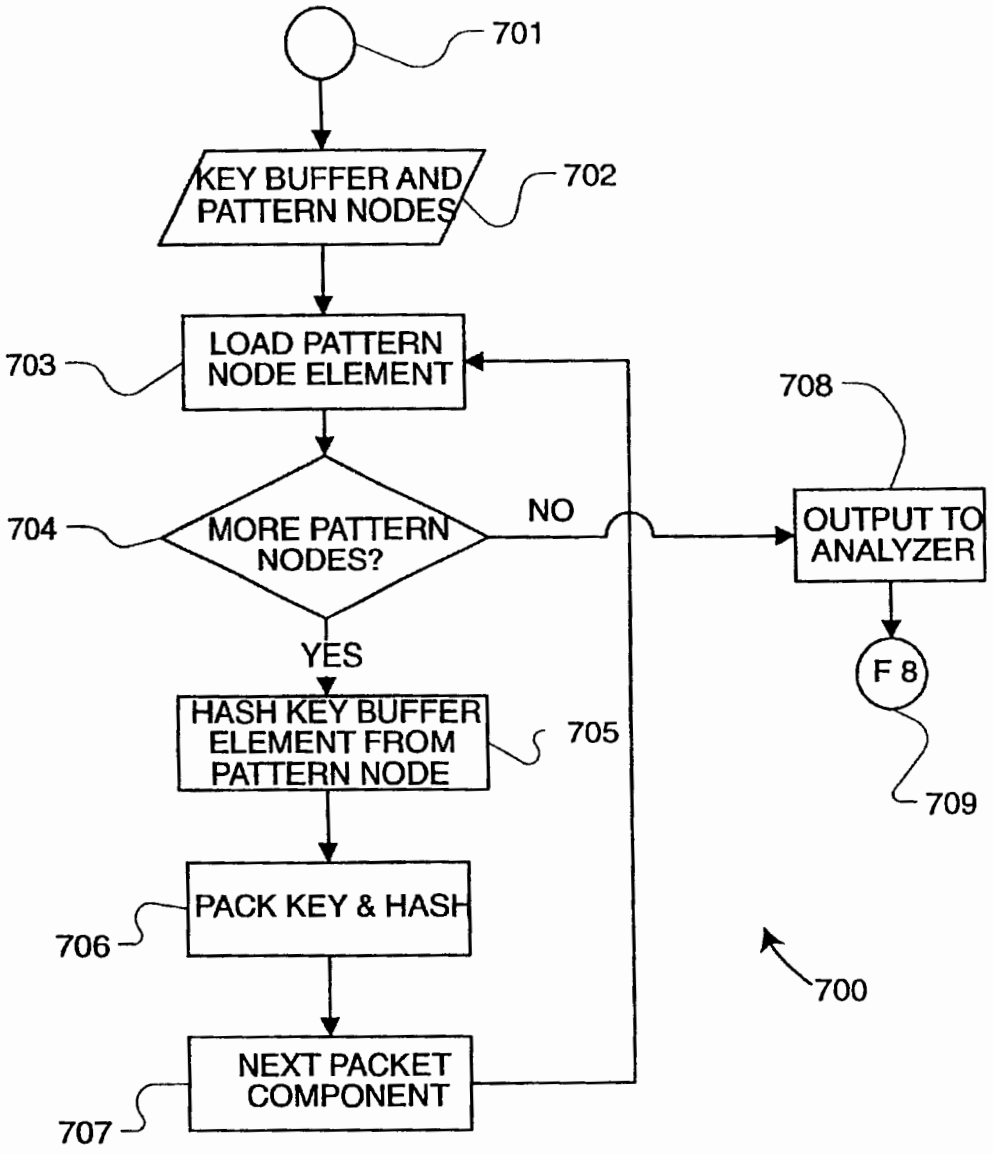

701
KEY BUFFER AND PATTERN NODES
702
LOAD PATTERN NODE ELEMENT
703
MORE PATTERN NODES?
704
NO
YES
HASH KEY BUFFER ELEMENT FROM PATTERN NODE
705
PACK KEY & HASH
706
NEXT PACKET COMPONENT
707
OUTPUT TO ANALYZER
708
F 8
709
700

FIG. 7

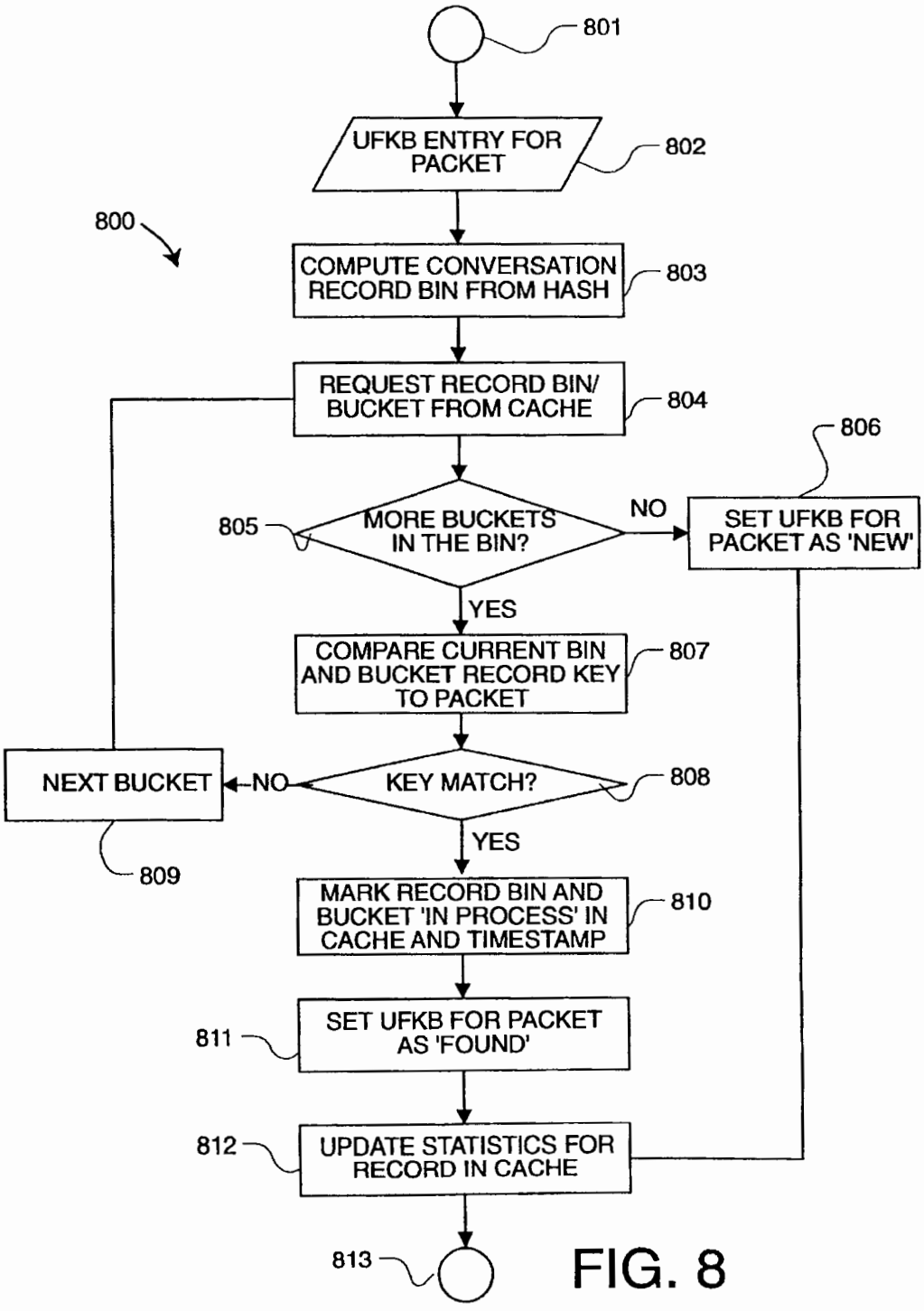

801
UFKB ENTRY FOR PACKET
802
800
COMPUTE CONVERSATION RECORD BIN FROM HASH
803
REQUEST RECORD BIN/ BUCKET FROM CACHE
804
806
805
MORE BUCKETS IN THE BIN?
NO
SET UFKB FOR PACKET AS 'NEW'
YES
COMPARE CURRENT BIN AND BUCKET RECORD KEY TO PACKET
807
NEXT BUCKET
NO
KEY MATCH?
808
YES
809
MARK RECORD BIN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

FIG. 8

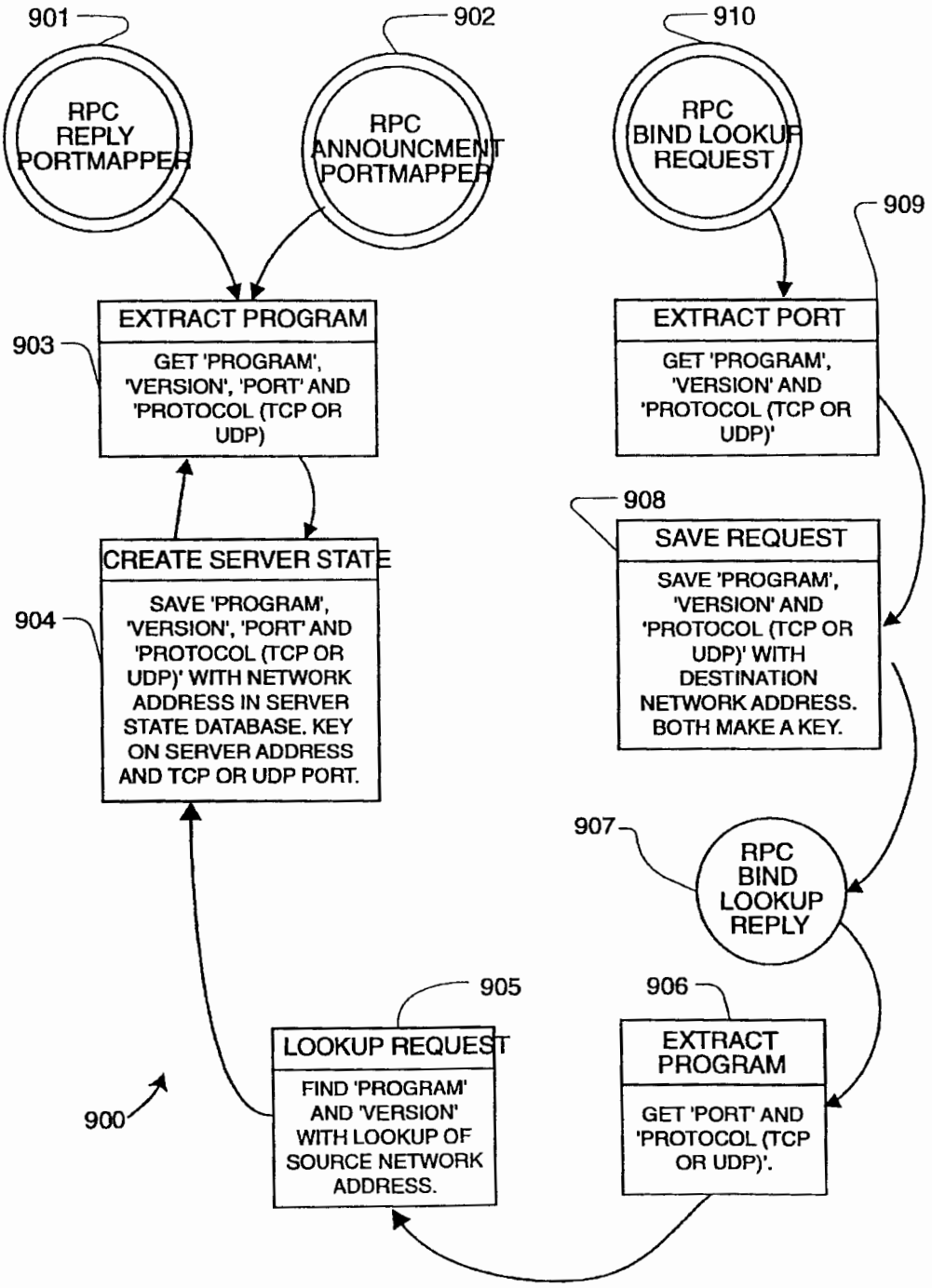

901
902
910
RPC REPLY PORTMAPPER
RPC ANNOUNCMENT PORTMAPPER
RPC BIND LOOKUP REQUEST
909
EXTRACT PROGRAM
GET 'PROGRAM', 'VERSION', 'PORT' AND 'PROTOCOL (TCP OR UDP)
903
EXTRACT PORT
GET 'PROGRAM', 'VERSION' AND 'PROTOCOL (TCP OR UDP)'
908
SAVE REQUEST
SAVE 'PROGRAM', 'VERSION' AND 'PROTOCOL (TCP OR UDP)'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

FIG. 9

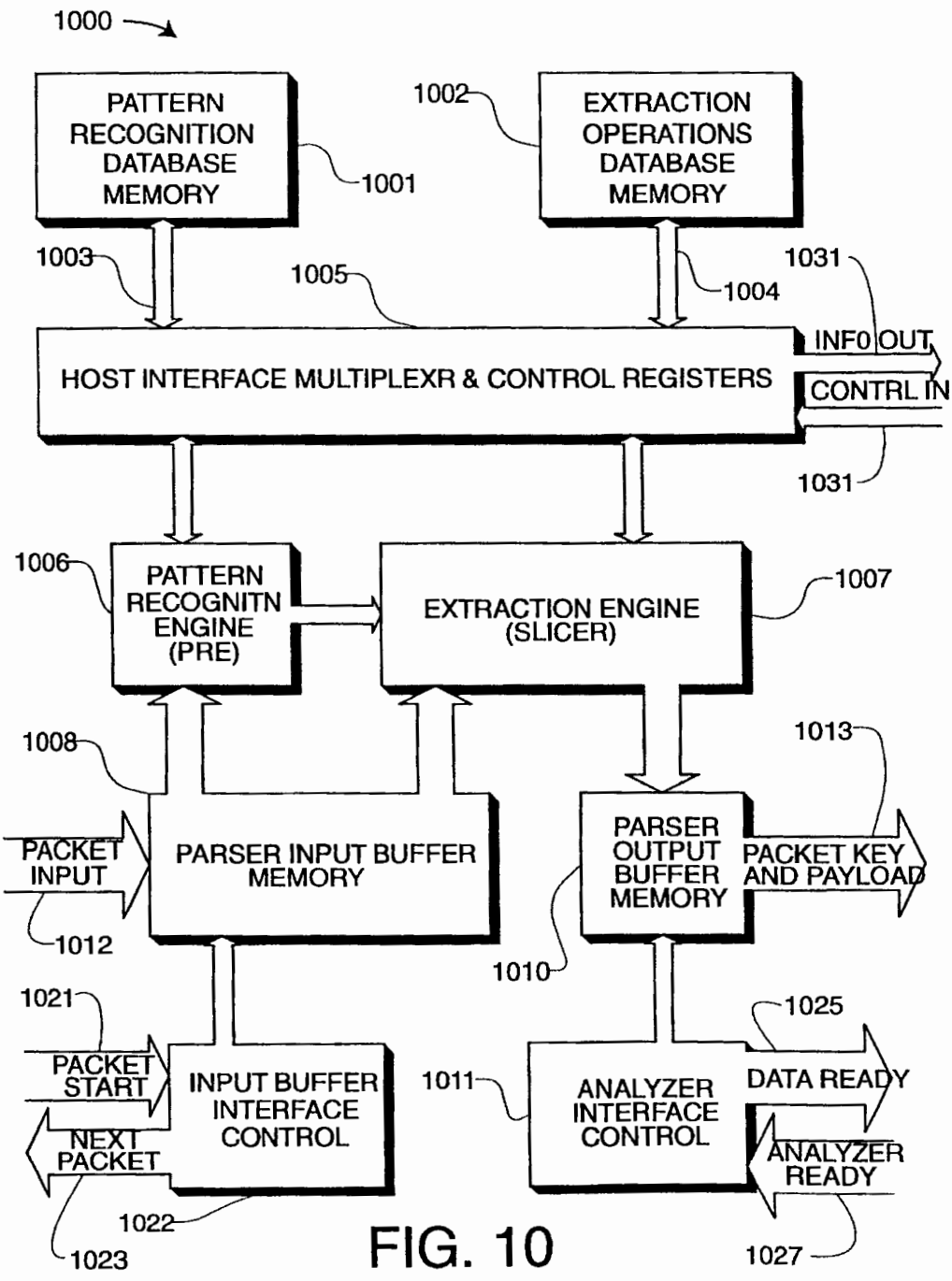

1000
PATTERN RECOGNITION DATABASE MEMORY
1001
1002
EXTRACTION OPERATIONS DATABASE MEMORY
1003
1005
1004
1031
INFO OUT
CONTRL IN
HOST INTERFACE MULTIPLEXR & CONTROL REGISTERS
1031
1006
PATTERN RECOGNITN ENGINE (PRE)
EXTRACTION ENGINE (SLICER)
1007
1008
1013
PACKET INPUT
PARSER INPUT BUFFER MEMORY
PARSER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

FIG. 10

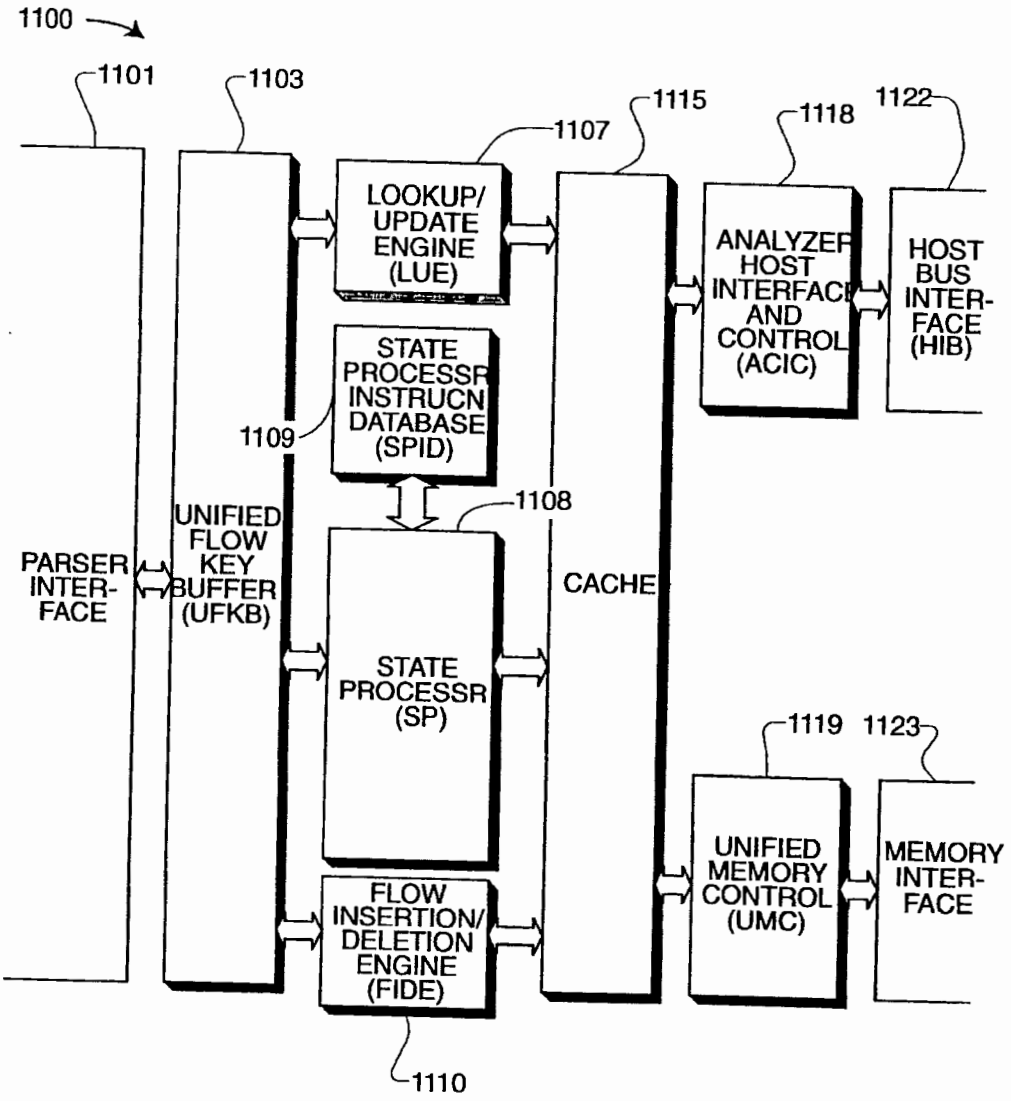

1100
1101
1103
1107
1115
1118
1122
LOOKUP/ UPDATE ENGINE (LUE)
ANALYZER HOST INTERFACE AND CONTROL (ACIC)
HOST BUS INTER-FACE (HIB)
STATE PROCESSR INSTRUCN DATABASE (SPID)
1109
1108
PARSER INTER-FACE
UNIFIED FLOW KEY BUFFER (UFKB)
CACHE
STATE PROCESSR (SP)
1119
1123
UNIFIED MEMORY CONTROL (UMC)
MEMORY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

FIG. 11

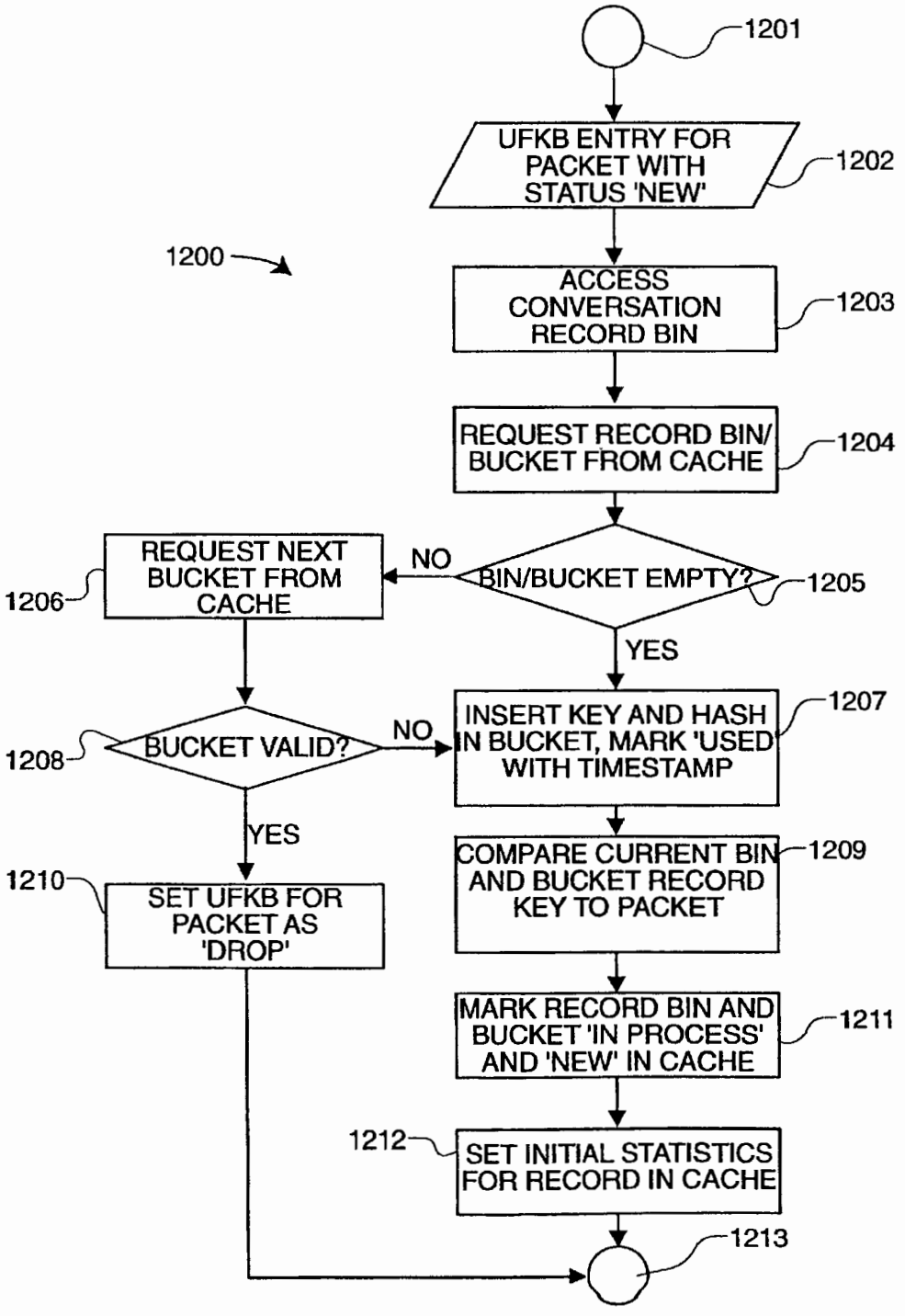

1201
UFKB ENTRY FOR PACKET WITH STATUS 'NEW'
1202
1200
ACCESS CONVERSATION RECORD BIN
1203
REQUEST RECORD BIN/ BUCKET FROM CACHE
1204
REQUEST NEXT BUCKET FROM CACHE
NO
BIN/BUCKET EMPTY?
1205
1206
YES
INSERT KEY AND HASH IN BUCKET, MARK 'USED' WITH TIMESTAMP
1207
BUCKET VALID?
NO
1208
YES
COMPARE CURRENT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

FIG. 12

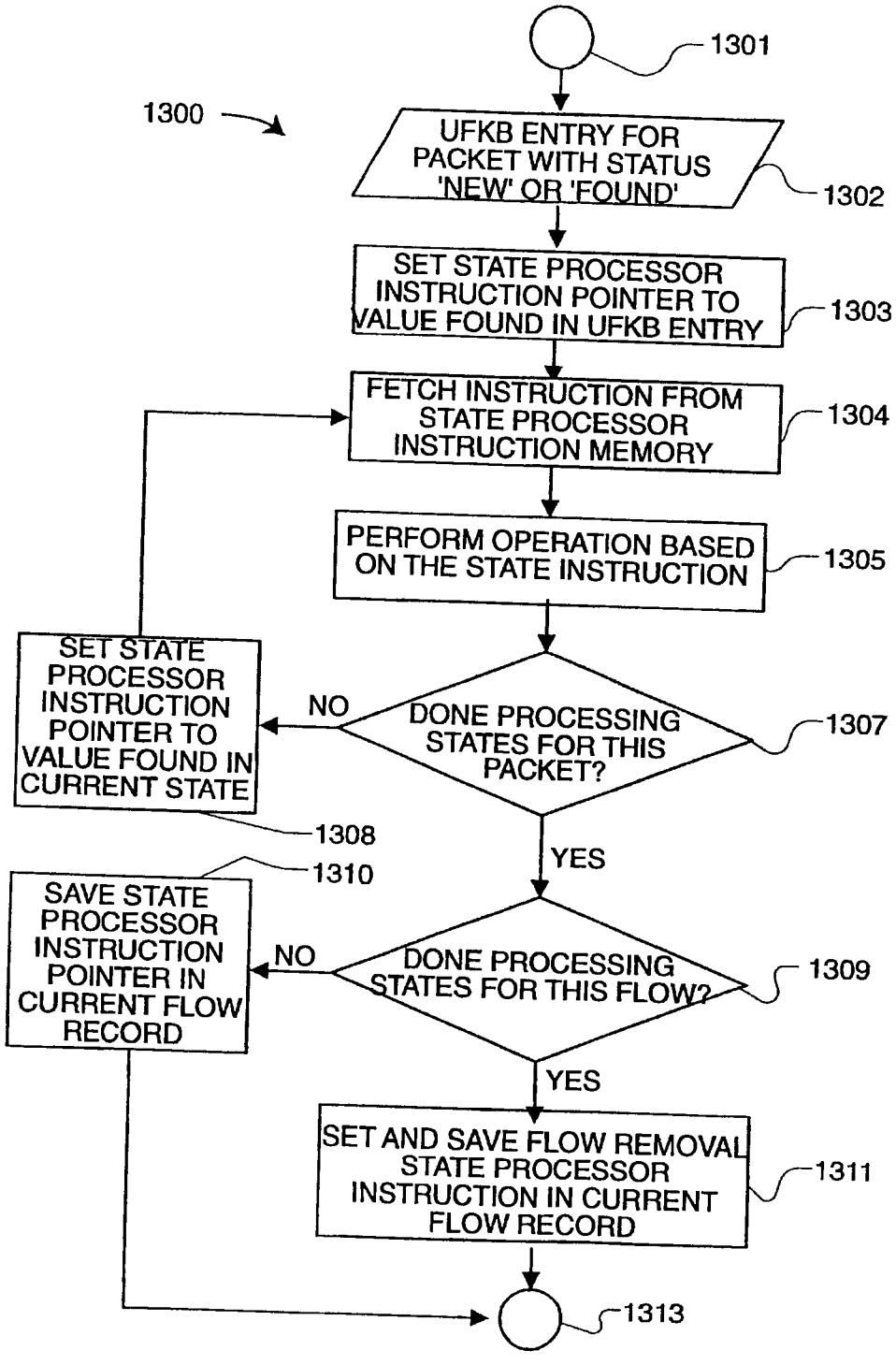

1301
1300
UFKB ENTRY FOR PACKET WITH STATUS 'NEW' OR 'FOUND'
1302
SET STATE PROCESSOR INSTRUCTION POINTER TO VALUE FOUND IN UFKB ENTRY
1303
FETCH INSTRUCTION FROM STATE PROCESSOR INSTRUCTION MEMORY
1304
PERFORM OPERATION BASED ON THE STATE INSTRUCTION
1305
DONE PROCESSING STATES FOR THIS PACKET?
1307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

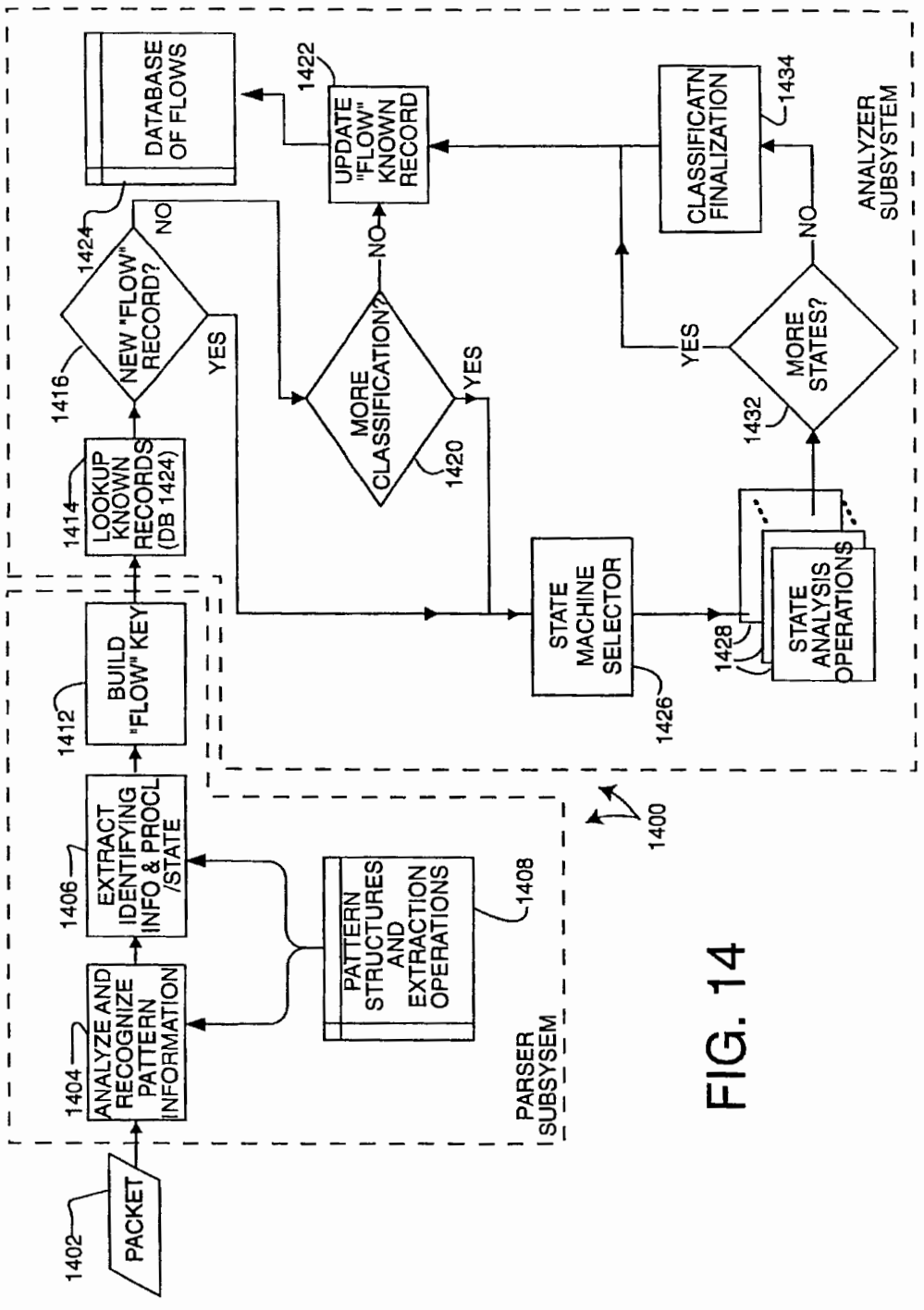

PACKET
1402
PARSER SUBSYSEM
ANALYZE AND RECOGNIZE PATTERN INFORMATION
1404
EXTRACT IDENTIFYING INFO & PROCL /STATE
1406
PATTERN STRUCTURES AND EXTRACTION OPERATIONS
1408
1400
BUILD "FLOW" KEY
1412
LOOKUP KNOWN RECORDS (DB 1424)
1414
NEW "FLOW" RECORD?
1416
DATABASE OF FLOWS
1424
NO
YES
MORE CLASSIFICATION?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

FIG. 14

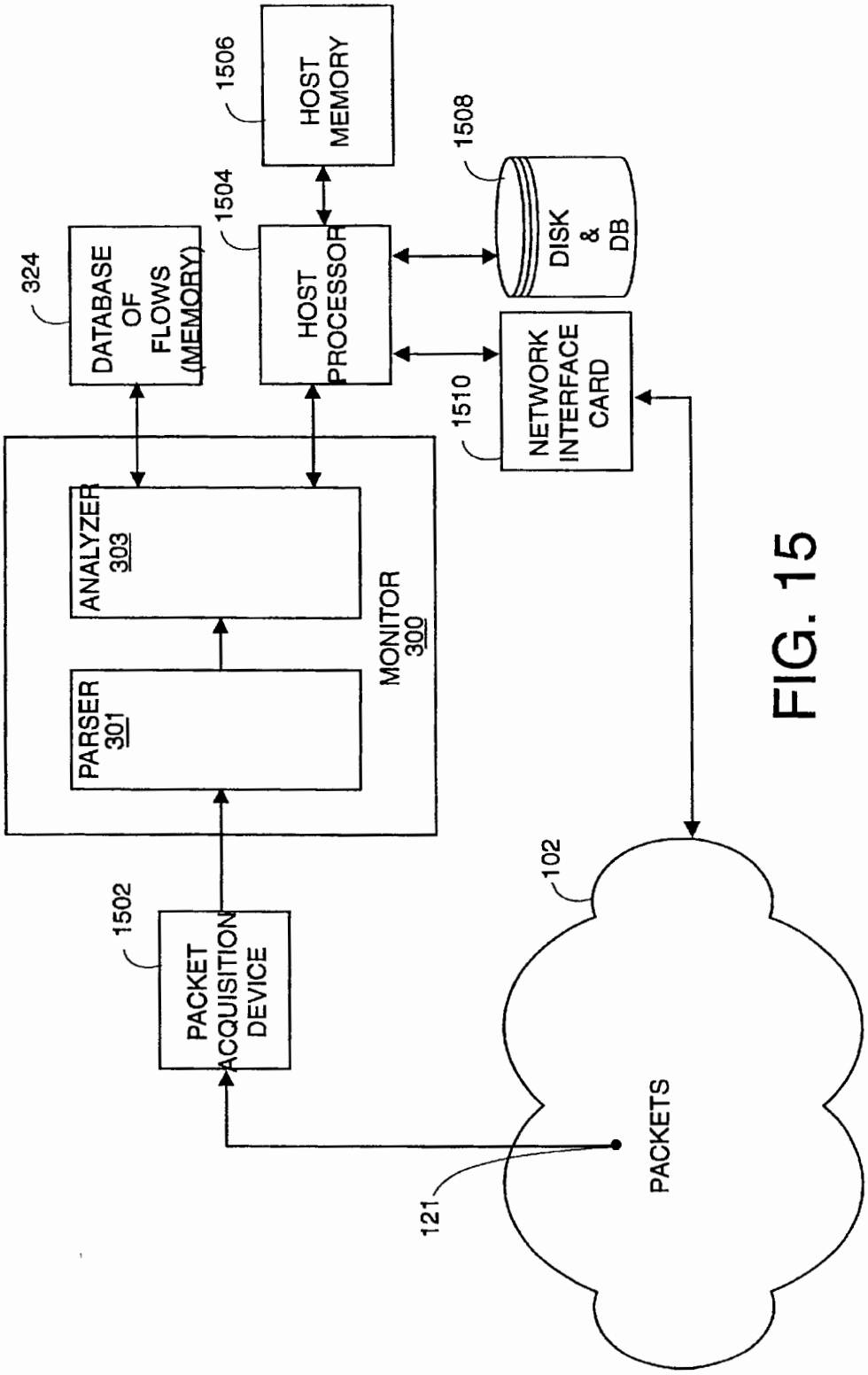

HOST MEMORY
1506
1508
DISK & DB
1504
HOST PROCESSOR
324
DATABASE OF FLOWS (MEMORY)
1510
NETWORK INTERFACE CARD
ANALYZER 303
MONITOR 300
PARSER 301
1502
PACKET ACQUISITION DEVICE
102
PACKETS
121

FIG. 15

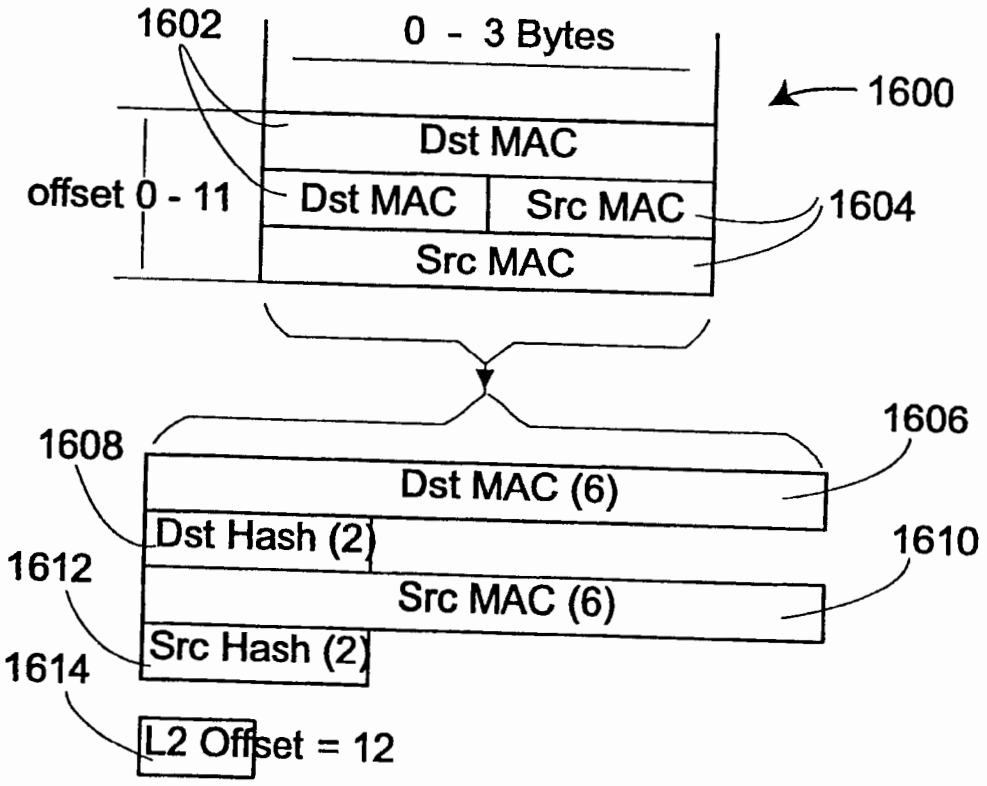

1602
0 - 3 Bytes
1600
Dst MAC
offset 0 - 11
Dst MAC
Src MAC
1604
Src MAC
1606
1608
Dst MAC (6)
Dst Hash (2)
1610
1612
Src MAC (6)
Src Hash (2)
1614
L2 Offset = 12

FIG. 16

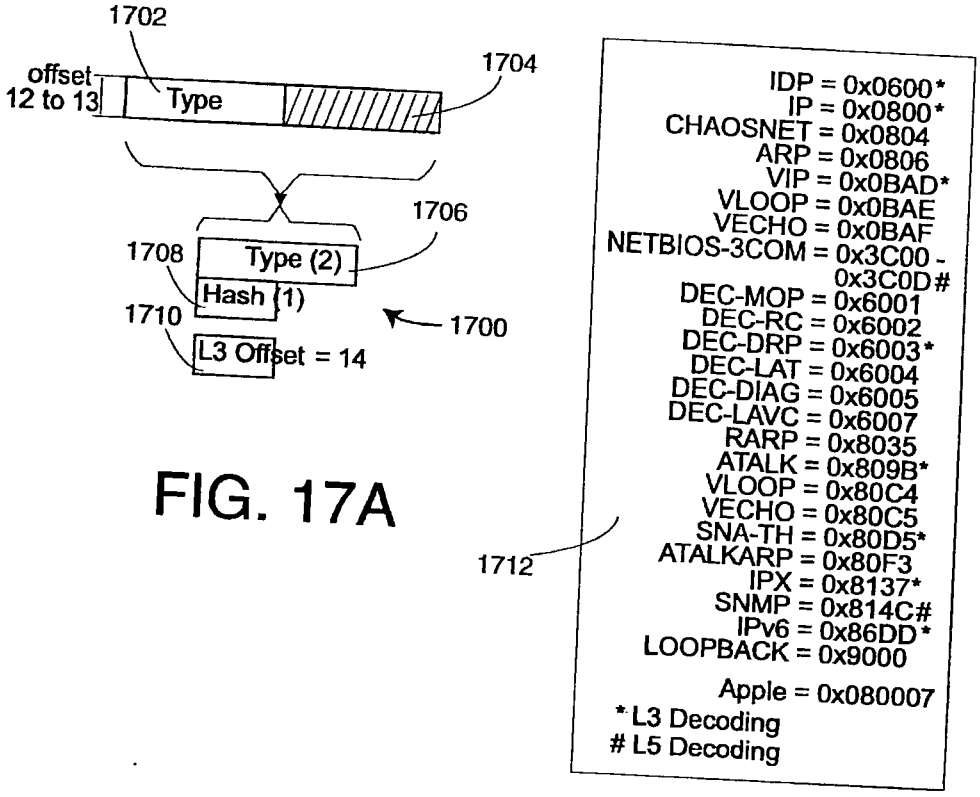


FIG. 17A

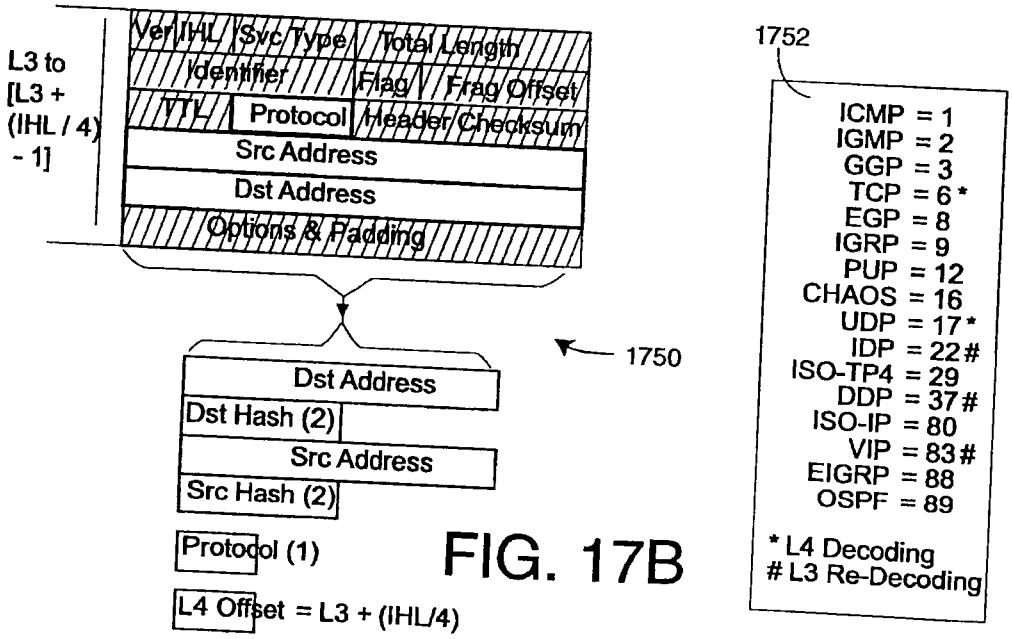


FIG. 17B

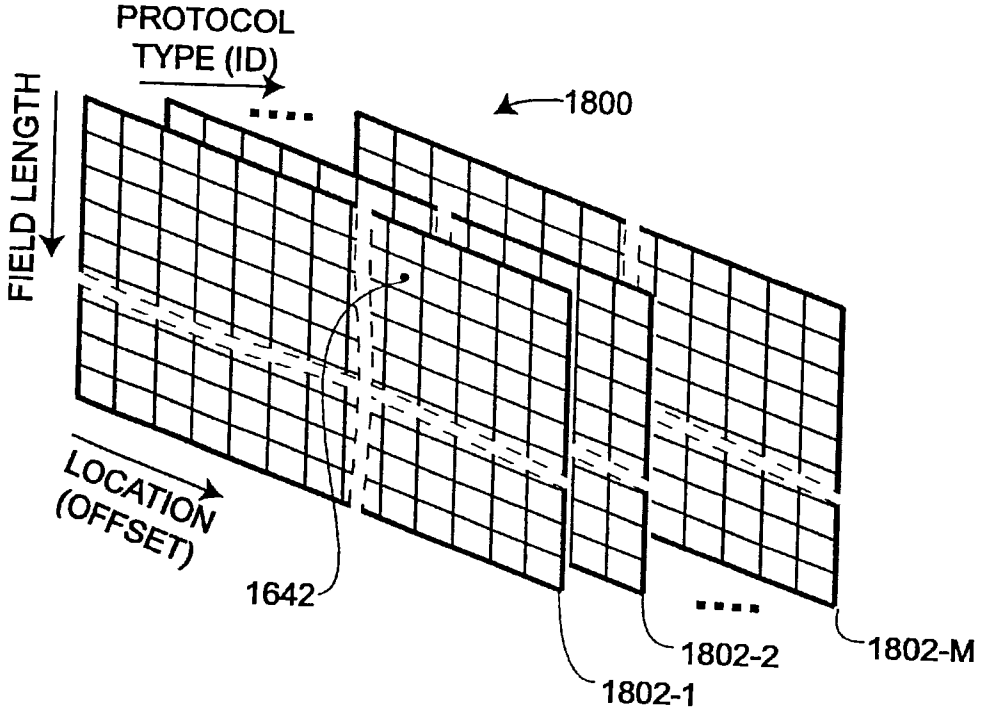


FIG. 18A

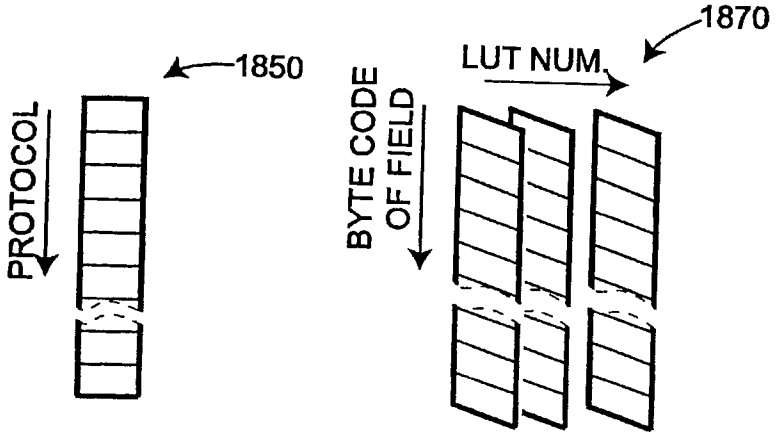


FIG. 18B

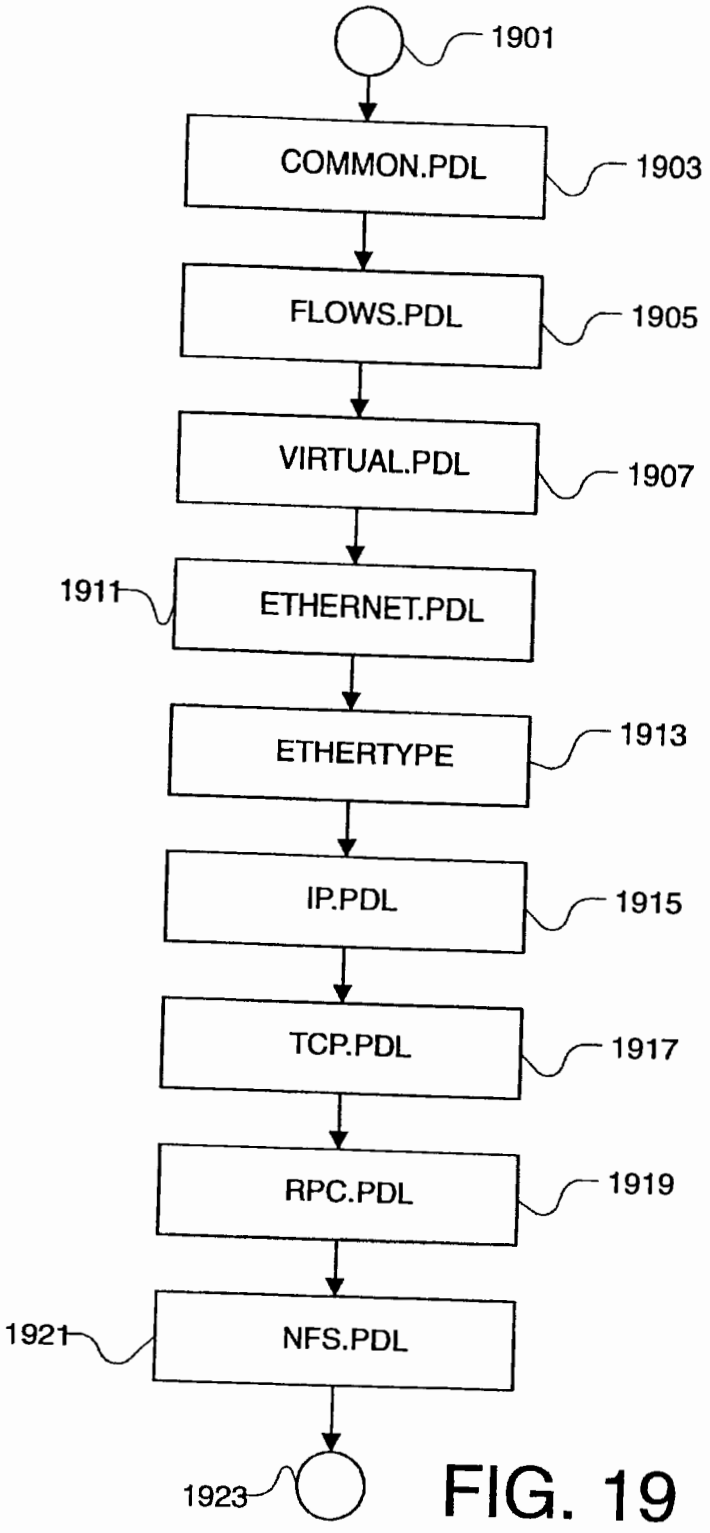

1901
COMMON.PDL
1903
FLOWS.PDL
1905
VIRTUAL.PDL
1907
1911
ETHERNET.PDL
ETHERTYPE
1913
IP.PDL
1915
TCP.PDL
1917
RPC.PDL
1919
1921
NFS.PDL
1923

FIG. 19

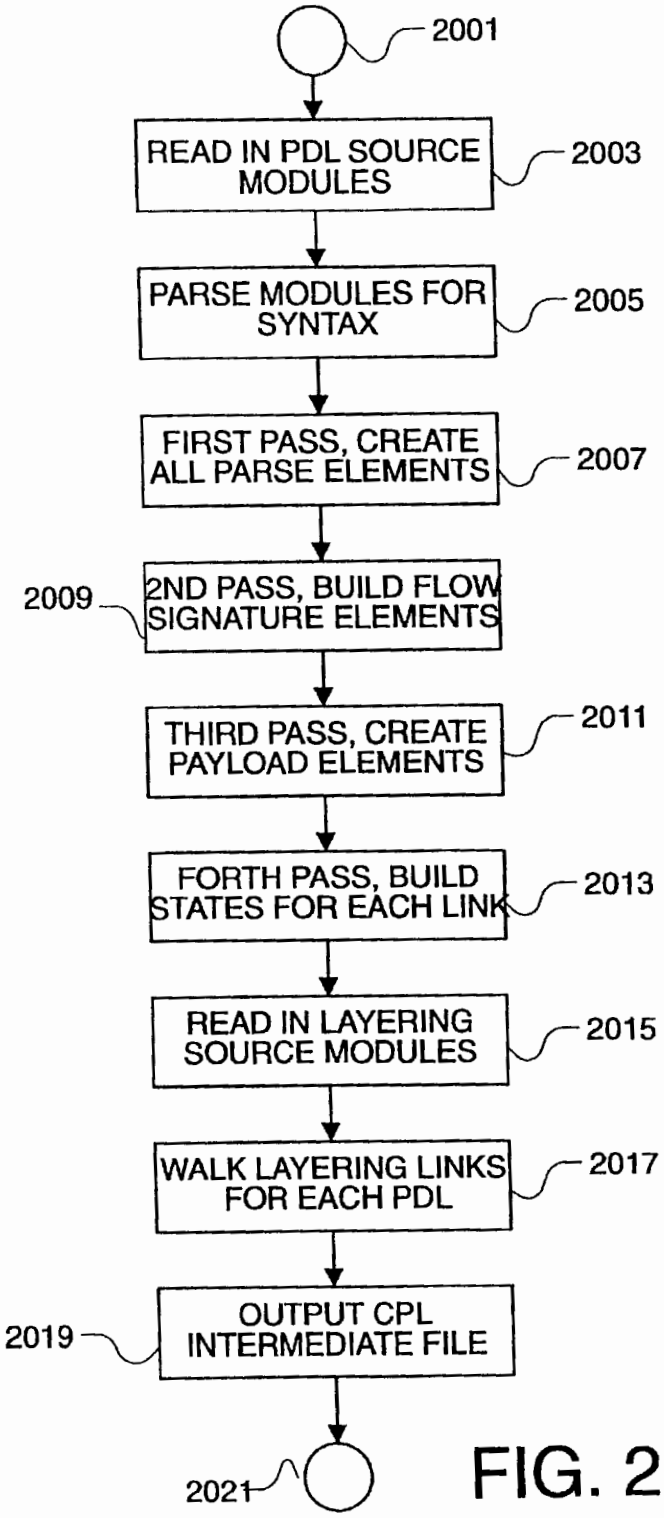

2001
READ IN PDL SOURCE MODULES
2003
PARSE MODULES FOR SYNTAX
2005
FIRST PASS, CREATE ALL PARSE ELEMENTS
2007
2009
2ND PASS, BUILD FLOW SIGNATURE ELEMENTS
THIRD PASS, CREATE PAYLOAD ELEMENTS
2011
FORTH PASS, BUILD STATES FOR EACH LINK
2013
READ IN LAYERING SOURCE MODULES
2015
WALK LAYERING LINKS FOR EACH PDL
2017
2019
OUTPUT CPL INTERMEDIATE FILE
2021

FIG. 20

PROCESSING PROTOCOL SPECIFIC INFORMATION IN PACKETS SPECIFIED BY A PROTOCOL DESCRIPTION LANGUAGE

CROSS-REFERENCE TO RELATED APPLICATION

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

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

- U.S. patent application Ser. No. 09/608,237 for METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK, to inventors Dietz, et al., filed Jun. 30, 2000, and incorporated herein by reference.
- U.S. patent application Ser. No. 09/608,126 for RE-USING INFORMATION FROM DATA TRANSACTIONS FOR MAINTAINING STATISTICS IN NETWORK MONITORING, to inventors Dietz, et al., filed Jun. 30, 2000, and incorporated herein by reference.
- U.S. patent application Ser. No. 09/608,266 for ASSOCIATIVE CACHE STRUCTURE FOR LOOKUPS AND UPDATES OF FLOW RECORDS IN A NETWORK MONITOR, to inventors Sarkissian, et al., filed Jun. 30, 2000, and incorporated herein by reference.
- U.S. patent application Ser. No. 09/608,267 for STATE PROCESSOR FOR PATTERN MATCHING IN A NETWORK MONITOR DEVICE, to inventors Sarkissian, et al., filed Jun. 30, 2000, and incorporated herein by reference.

FIELD OF INVENTION

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

COPYRIGHT NOTICE

BACKGROUND

There has long been a need for network activity monitors. This need has become especially acute, however, given the recent popularity of the Internet and other interconnected networks. In particular, there is a need for a real-time network monitor that can provide details as to the application programs being used. Such a monitor should enable non-intrusive, remote detection, characterization, analysis, and capture of all information passing through any point on the network (i.e., of all packets and packet streams passing through any location in the network). Not only should all the packets be detected and analyzed, but for each of these packets the network monitor should determine the protocol (e.g., http, ftp, H.323, VPN, etc.), the application/use within the protocol (e.g., voice, video, data, real-time data, etc.), and an end user's pattern of use within each application or the application context (e.g., options selected, service delivered, duration, time of day, data requested, etc.). Also, the network monitor should not be reliant upon server resident information such as log files. Rather, it should allow a user such as a network administrator or an Internet service provider (ISP) the means to measure and analyze network activity objectively; to customize the type of data that is collected and analyzed; to undertake real time analysis; and to receive timely notification of network problems.

The recognizing and classifying in such a network monitor should be at all protocol layer levels in conversational flows that pass in either direction at a point in a network. Furthermore, the monitor should provide for properly analyzing each of the packets exchanged between a client and a server, maintaining information relevant to the current state of each of these conversational flows.

Related and incorporated by reference U.S. patent application Ser. No. 09/608,237 for METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK, to inventors Dietz, et al, describes a network monitor that includes carrying out protocol specific operations on individual packets including extracting information from header fields in the packet to use for building a signature for identifying the conversational flow of the packet and for recognizing future packets as belonging to a previously encountered flow. A parser subsystem includes a parser for recognizing different patterns in the packet that identify the protocols used. For each protocol recognized, a slicer extracts important packet elements from the packet. These form a signature (i.e., key) for the packet. The slicer also preferably generates a hash for rapidly identifying a flow that may have this signature from a database of known flows.

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

Each flow-entry includes one or more statistical measures, e.g., the packet count related to the flow, the time of arrival of a packet, the time differential.

In the preferred hardware embodiment, each of the LUE, state processor, and FIDE operate independently from the other two engines. The state processor performs one or more operations specific to the state of the flow.

A network analyzer should be able to 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), T1, and 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. 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 in a conversational flow that uses the frame-relay standard or the Ethernet protocol.

In order for a network monitor to be able to analyze different packet or frame formats, the monitor needs to be able to perform protocol specific operations on each packet with each packet carrying information conforming to different protocols and related to different applications. For example, the monitor needs to be able to parse packets of different formats into fields 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.

Prior art network monitors exist that parse individual packets and look for information at different fields to use for building a signature for identifying packets. Chiu, et al., describe a method for collecting information at the session level in a computer network in U.S. Pat. No. 5,101,402, titled "APPARATUS AND METHOD FOR REAL-TIME MONITORING OF NETWORK SESSIONS AND A LOCAL AREA NETWORK." In this patent, there are fixed locations specified for particular types of packets. For example, if a DECnet packet appears, the Chiu system looks at six specific fields (at 6 locations) in the packet in order to identify the session of the packet. If, on the other hand, an IP packet appears, a different set of six locations are examined. The system looks only at the lowest levels up to the protocol layer. There are fixed locations for each of the fields that specified the next level. With the proliferation of protocols, clearly the specifying of all the possible places to look to determine the session becomes more and more difficult. Likewise, adding a new protocol or application is difficult.

It is desirable to be able to adaptively determine the locations and the information extracted from any packet for the particular type of packet. In this way, an optimal signature may be defined using a protocol-dependent and packet-content-dependent definition of what to look for and where to look for it in order to form a signature.

There thus is also a need for a network monitor that can be tailored or adapted for different protocols and for different application programs. There thus is also a need for a network monitor that can accommodate new protocols and for new application programs. There also is a need for means for specifying new protocols and new levels, including new applications. There also is a need for a mechanism to describe protocol specific operations, including, for example, what information is relevant to packets and packets that need to be decoded, and to include specifying parsing operations and extraction operations. There also is a need for a mechanism to describe state operations to perform on packets that are at a particular state of recognition of a flow in order to further recognize the flow.

SUMMARY

One embodiment of the invention is a method of performing protocol specific operations on a packet passing through a connection point on a computer network. The packet contents conform to protocols of a layered model wherein the protocol at a particular layer level may include one or a set of child protocols defined for that level. The method includes receiving the packet and receiving a set of protocol descriptions for protocols may be used in the packet. A protocol description for a particular protocol at a particular layer level includes any child protocols of the particular protocol, and for any child protocol, where in the packet information related to the particular child protocol may be found. A protocol description also includes any protocol specific operations to be performed on the packet for the particular protocol at the particular layer level. The method includes performing the protocol specific operations on the packet specified by the set of protocol descriptions based on the base protocol of the packet and the children of the protocols used in the packet. A particular embodiment includes providing the protocol descriptions in a high-level protocol description language, and compiling to the descriptions into a data structure. The compiling may further include compressing the data structure into a compressed data structure. The protocol specific operations may include parsing and extraction operations to extract identifying information. The protocol specific operations may also include state processing operations defined for a particular state of a conversational flow of the packet.

BRIEF DESCRIPTION OF THE DRAWINGS

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

FIG. **19** shows various PDL file modules to be compiled together by the compiling process illustrated in FIG. **20** as an example, in accordance with a compiling aspect of the invention.

FIG. **20** is a flowchart of the process of compiling high-level language files according to an 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 however the signal names are not needed to understand the operation and practice of the invention.

Operation in a Network

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

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

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

ISO MODEL

Layer	Functionality	Example
7	Application	Telnet, NFS, Novell NCP, HTTP, H.323
6	Presentation	XDR
5	Session	RPC, NBTBIOS, 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)

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

In some communication arts, the term "frame" generally refers to encapsulated data at OSI layer 2, including a destination address, control bits for flow control, the data or payload, and CRC (cyclic redundancy check) data for error checking. The term "packet" generally refers to encapsulated data at OSI layer 3. In the TCP/IP world, the term

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

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

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

In real-world uses of the monitor **108**, the number of packets on the network **102** passing by the monitor **108**'s connection point can exceed a million per second. Consequently, the monitor has very little time available to analyze and type each packet and identify and maintain the state of the flows passing through the connection point. The monitor **108** therefore masks out all the unimportant parts of each packet that will not contribute to its classification. However, the parts to mask-out will change with each packet depending on which flow it belongs to and depending on the state of the flow.

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

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

It may be that the application is now recognized, or recognition may need to proceed to a third level of analysis using the second level signature. For each packet, therefore, the monitor parses the packet and generates a signature to determine if this signature identified a previously encountered flow, or shall be used to recognize future packets belonging to the same conversational flow. In real time, the packet is further analyzed in the context of the sequence of previously encountered packets (the state), and of the possible future sequences such a past sequence may generate in conversational flows associated with different applications. A new signature for recognizing future packets may also be generated. This process of analysis continues until the applications are identified. The last generated signature may then be used to efficiently recognize future packets associated with the same conversational flow. Such an arrangement makes it possible for the monitor **108** to cope with millions of packets per second that must be inspected.

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

The Network Monitor

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

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

Aspects shown here include: (1) the initialization of the monitor to generate what operations need to occur on packets of different types—accomplished by compiler and optimizer **310**, (2) the processing—parsing and extraction of

selected portions—of packets to generate an identifying signature—accomplished by parser subsystem **301**, and (3) the analysis of the packets—accomplished by analyzer **303**.

The purpose of compiler and optimizer **310** is to provide protocol specific information to parser subsystem **301** and to analyzer subsystem **303**. The initialization occurs prior to operation of the monitor, and only needs to re-occur when new protocols are to be added.

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

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

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

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

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

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

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

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

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

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

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

The parser record enters a buffer called the unified flow key buffer (UFKB). The UFKB stores the data on flows in

a data structure that is similar to the parser record, but that includes a field that can be modified. In particular, one or the UFKB record fields stores the packet sequence number, and another is filled with state information in the form of a program counter for a state processor that implements state processing **328**.

The determination (**316**) of whether a record with the same signature already exists is carried out by a lookup engine (LUE) that obtains new UFKB records and uses the hash in the UFKB record to lookup if there is a matching known flow. In the particular embodiment, the database of known flows **324** is in an external memory. A cache is associated with the database **324**. A lookup by the LUE for a known record is carried out by accessing the cache using the hash, and if the entry is not already present in the cache, the entry is looked up (again using the hash) in the external memory.

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

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

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

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

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

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

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

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

Thus, state processor **328** applies the first state operation to the packet for this particular flow-entry. A process **330** decides if more operations need to be performed for this state. If so, the analyzer continues looping between block **330** and **328** applying additional state operations to this particular packet until all those operations are completed—that is, there are no more operations for this packet in this state. A process **332** decides if there are further states to be analyzed for this type of flow according to the state of the flow and the protocol, in order to fully characterize the flow. If not, the conversational flow has now been fully characterized and a process **334** finalizes the classification of the conversational flow for the flow.

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

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

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

The flow-entry also is updated after classification finalization so that any further packets belonging to this flow will

be readily identified from their signature as belonging to this fully analyzed conversational flow.

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

Thus, the embodiment of present invention shown in FIG. **3** automatically maintains flow-entries, which in one aspect includes storing states. The monitor of FIG. **3** also generates characteristic parts of packets—the signatures—that can be used to recognize flows. The flow-entries may be identified and accessed by their signatures. Once a packet is identified to be from a known flow, the state of the flow is known and this knowledge enables state transition analysis to be performed in real time for each different protocol and application. In a complex analysis, state transitions are traversed as more and more packets are examined. Future packets that are part of the same conversational flow have their state analysis continued from a previously achieved state. When enough packets related to an application of interest have been processed, a final recognition state is ultimately reached, i.e., a set of states has been traversed by state analysis to completely characterize the conversational flow. The signature for that final state enables each new incoming packet of the same conversational flow to be individually recognized in real time.

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

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

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

Detailed Operation

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

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

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

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

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

An alternate embodiment of the data structure used in database **308** is illustrated in FIG. **18B**. Thus, like the 3-D structure of FIG. **18A**, the data structure permits rapid searches to be performed by the pattern recognition process **304** by indexing locations in a memory rather than performing address link computations. In this alternate embodiment, the PRD **308** includes two parts, a single protocol table **1850** (PT) which has an entry for each protocol known for the monitor, and a series of Look Up Tables **1870** (LUT's) that are used to identify known protocols and their children. The protocol table includes the parameters needed by the pattern analysis and recognition process **304** (implemented by PRE **1006**) to evaluate the header information in the packet that is associated with that protocol, and parameters needed by extraction process **306** (implemented by slicer **1007**) to process the packet header. When there are children, the PT describes which bytes in the header to evaluate to determine the child protocol. In particular, each PT entry contains the header length, an offset to the child, a slicer command, and some flags.

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

Compilation process is described in FIG. **4**. The source-code information in the form of protocol description files is shown as **402**. In the particular embodiment, the high level decoding descriptions includes a set of protocol description files **336**, one for each protocol, and a set of packet layer

selections **338**, which describes the particular layering (sets of trees of protocols) that the monitor is to be able to handle.

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

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

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

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

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

FIG. **5** shows a flowchart of how actual parser subsystem **301** functions. Starting at **501**, the packet **302** is input to the packet buffer in step **502**. Step **503** loads the next (initially the first) packet component from the packet **302**. The packet components are extracted from each packet **302** one element at a time. A check is made (**504**) to determine if the load-packet-component operation **503** succeeded, indicating that there was more in the packet to process. If not, indicating all components have been loaded, the parser subsystem **301** builds the packet signature (**512**)—the next stage (FIG. **6**).

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

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

Once all the packet components have been the loaded and processed from the input packet **302**, then the load packet will fail (indicated by test **504**), and the parser subsystem **301** moves to build a packet signature which is described in FIG. **6** FIG. **6** is a flow chart for extracting the information from which to build the packet signature. The flow starts at **601**, which is the exit point **513** of FIG. **5**. At this point parser subsystem **301** has a completed packet component and a pattern node available in a buffer (**602**). Step **603** loads the packet component available from the pattern analysis process of FIG. **5**. If the load completed (test **604**), indicating that there was indeed another packet component, the parser subsystem **301** fetches in **605** the extraction and process elements received from the pattern node component in **602**. If the fetch was successful (test **606**), indicating that there are extraction elements to apply, the parser subsystem **301** in step **607** applies that extraction process to the packet component based on an extraction instruction received from that pattern node. This removes and saves an element from the packet component.

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

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

Referring now to FIG. **7**, the process continues at **701**. The signature buffer and the pattern node elements are available (**702**). The parser subsystem **301** loads the next pattern node element. If the load was successful (test **704**) indicating there are more nodes, the parser subsystem **301** in **705**

hashes the signature buffer element based on the hash elements that are found in the pattern node that is in the element database. In **706** the resulting signature and the hash are packed. In **707** the parser subsystem **301** moves on to the next packet component which is loaded in **703**.

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

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

FIG. **8** is a flow diagram describing the operation of the lookup/update engine (LUE) that implements lookup operation **314**. The process starts at **801** from FIG. **7** with the parser record that includes a signature, the hash and at least parts of the payload. In **802** those elements are shown in the form of a UFKB-entry in the buffer. The LUE, the lookup engine **314** computes a "record bin number" from the hash for a flow-entry. A bin herein may have one or more "buckets" each containing a flow-entry. The preferred embodiment has four buckets per bin.

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

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

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

To provide for such analyses, the lookup/update engine updates one or more counters that are part of the flow-entry (in the cache) in step **812**. The process exits at **813**. In our embodiment, the counters include the total packets of the flow, the time, and a differential time from the last timestamp to the present timestamp.

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

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

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

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

The Hardware System

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

FIG. **10** is a description of the parsing subsystem (**301**, shown here as subsystem **1000**) as implemented in hardware. Memory **1001** is the pattern recognition database memory, in which the patterns that are going to be analyzed are stored. Memory **1002** is the extraction-operation database memory, in which the extraction instructions are stored. Both **1001** and **1002** correspond to internal data structure **308** of FIG. **3**. Typically, the system is initialized from a microprocessor (not shown) at which time these memories are loaded through a host interface multiplexor and control register **1005** via the internal buses **1003** and **1004**. Note that the contents of **1001** and **1002** are preferably obtained by compiling process **310** of FIG. **3**.

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

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

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

The PRE **1006** includes of a comparison engine. The comparison engine has a first stage that checks the protocol type field to determine if it is an 802.3 packet and the field should be treated as a length. If it is not a length, the protocol is checked in a second stage. The first stage is the only protocol level that is not programmable. The second stage has two full sixteen bit content addressable memories (CAMs) defined for future protocol additions.

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

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

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

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

Once all the selected parts of the packet used to form the signature are extracted, the hash is loaded into parser output buffer memory **1010**. Any additional payload from the packet that is required for further analysis is also included. The parser output memory **1010** is interfaced with the analyzer subsystem by analyzer interface control **1011**. Once all the information of a packet is in the parser output buffer memory **1010**, a data ready signal **1025** is asserted by analyzer interface control. The data from the parser subsystem **1000** is moved to the analyzer subsystem via **1013** when an analyzer ready signal **1027** is asserted.

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

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

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

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

A new UFKB record is first processed by the LUE **1107**. A record that has been processed by the LUE **1107** may be processed by the state processor **1108**, and a UFKB record data may be processed by the flow insertion/deletion engine **110** after being processed by the state processor **1108** or only by the LUE. Whether or not a particular engine has been applied to any unified flow key buffer entry is determined by status fields set by the engines upon completion. In one embodiment, a status flag in the UFKB-entry indicates whether an entry is new or found. In other embodiments, the LUE issues a flag to pass the entry to the state processor for processing, and the required operations for a new record are included in the SP instructions.

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

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

The cache subsystem **1115** is an associative cache that includes a set of content addressable memory cells (CAMs) each including an address portion and a pointer portion pointing to the cache memory (e.g., RAM) containing the cached flow-entries. The CAMs are arranged as a stack ordered from a top CAM to a bottom CAM. The bottom CAM's pointer points to the least recently used (LRU) cache memory entry. Whenever there is a cache miss, the contents of cache memory pointed to by the bottom CAM are replaced by the flow-entry from the flow-entry database **324**. This now becomes the most recently used entry, so the contents of the bottom CAM are moved to the top CAM and all CAM contents are shifted down. Thus, the cache is an associative cache with a true LRU replacement policy.

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

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

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

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

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

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

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

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

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

FIG. **13** describes the operation of the state processor **1108**. The state processor is entered at **1301** with a unified flow key buffer entry to be processed. The UFKB-entry is new or corresponding to a found flow-entry. This UFKB-entry is retrieved from unified flow key buffer **1103** in **1301**. In **1303**, the protocol identifier for the UFKB-entry is used to set the state processor's instruction counter. The state processor **1108** starts the process by using the last protocol recognized by the parser subsystem **301** as an offset into a jump table. The jump table takes us to the instructions to use for that protocol. Most instructions test something in the unified flow key buffer or the flow-entry if it exists. The state processor **1108** may have to test bits, do comparisons, add or subtract to perform the test.

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

In **1307**, a check is made to determine if there are any more instructions to be performed for the packet. If yes, then

in **1308** the system sets the state processor instruction pointer (SPIP) to obtain the next instruction. The SPIP may be set by an immediate jump vector in the currently decoded instruction, or by a value provided by the SPALU during processing.

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

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

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

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

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

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

The process of flow insertion is now described with the aid of FIG. **12**. Flows are grouped into bins of buckets by the hash value. The engine processes a UFKB-entry that may be new or that the state processor otherwise has indicated needs to be created. FIG. **12** shows the case of a new entry being created. A conversation record bin (preferably containing 4 buckets for four records) is obtained in **1203**. This is a bin that matches the hash of the UFKB, so this bin may already have been sought for the UFKB-entry by the LUE. In **1204** the FIDE **1110** requests that the record bin/bucket be maintained in the cache system **1115**. If in **1205** the cache system **1115** indicates that the bin/bucket is empty, step **1207** inserts the flow signature (with the hash) into the bucket and the bucket is marked "used" in the cache engine of cache **1115** using a timestamp that is maintained throughout the process. In **1209**, the FIDE **1110** compares the bin and bucket record flow signature to the packet to verify that all the elements are in place to complete the record. In **1211** the system marks the record bin and bucket as "in process" and as "new" in the cache system (and hence in the external memory). In **1212**, the initial statistical measures for the flow-record are set in the cache system. This in the preferred embodiment clears the set of counters used to maintain statistics, and may perform other procedures for statistical operations requires by the analyzer for the first packet seen for a particular flow.

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

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

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

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

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

FIG. **10** also includes some "generic" interfaces. There is a packet input interface **1012**—a general interface that works in tandem with the signals of the input buffer interface control **1022**. These are designed so that they can be used with any kind of generic systems that can then feed packet information into the parser. Another generic interface is the

interface of pipes **1031** and **1033** respectively out of and into host interface multiplexor and control registers **1005**. This enables the parsing system to be managed by an external system, for example a microprocessor or another kind of external logic, and enables the external system to program and otherwise control the parser.

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

In accordance with one hardware implementation, the elements shown in FIGS. **10** and **11** are implemented in a set of six field programmable logic arrays (FPGA's). The boundaries of these FPGA's are as follows. The parsing subsystem of FIG. **10** is implemented as two FPGAS; one FPGA, and includes blocks **1006**, **1008** and **1012**, parts of **1005**, and memory **1001**. The second FPGA includes **1002**, **1007**, **1013**, **1011** parts of **1005**. Referring to FIG. **11**, the unified look-up buffer **1103** is implemented as a single FPGA. State processor **1108** and part of state processor instruction database memory **1109** is another FPGA. Portions of the state processor instruction database memory **1109** are maintained in external SRAM's. The lookup/update engine **1107** and the flow insertion/deletion engine **1110** are in another FPGA. The sixth FPGA includes the cache system **1115**, the unified memory control **1119**, and the analyzer host interface and control **1118**.

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

Operation of the Invention

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

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

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

Examples of Packet Elucidation

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

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

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

"Session tracking" also is known as one of the primary processes for tracking applications in client/server packet exchanges. The process of tracking sessions requires an initial connection to a predefined socket or port number. This method of communication is used in a variety of transport layer protocols. It is most commonly seen in the TCP and UDP transport protocols of the IP protocol.

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

One example of session tracking is TFTP (Trivial File Transfer Protocol), a version of the TCP/IP FTP protocol that has no directory or password capability. During the client/server exchange process of TFTP, a specific port (port number **69**) is always used to initiate the packet exchange. Thus, when the client begins the process of communicating, a request is made to UDP port **69**. Once the server receives this request, a new port number is created on the server. The

server then replies to the client using the new port. In this example, it is clear that in order to recognize TFTP; network monitor **300** analyzes the initial request from the client and generates a signature for it. Monitor **300** uses that signature to recognize the reply. Monitor **300** also analyzes the reply from the server with the key port information, and uses this to create a signature for monitoring the remaining packets of this data exchange.

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

Server Announcement Example

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

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

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

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

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

1. A client (CLIENT **3**, **106** in FIG. **1**) sends a TCP packet to SERVER **2** (**110** in FIG. **1**) on port **111**, with an RPC Bind Lookup Request (rpcBindLookup). TCP or UDP port **111** is always associated Sun RPC. This request specifies the program (as a program identifier), version, and might specify the protocol (UDP or TCP).
2. The server SERVER **2** (**110** in FIG. **1**) extracts the program identifier and version identifier from the request. The server also uses the fact that this packet came in using the TCP transport and that no protocol was specified, and thus will use the TCP protocol for its reply.
3. The server **110** sends a TCP packet to port number **111**, with an RPC Bind Lookup Reply. The reply contains the specific port number (e.g., port number 'port') on which future transactions will be accepted for the specific RPC program identifier (e.g., Program 'program') and the protocol (UDP or TCP) for use.

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

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

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

Process **909**: Extract the 'program,' 'version,' and 'protocol' (UDP or TCP). Extract the TCP or UDP port (process **909**) which is **111** indicating Sun RPC.

Process **908**: Decode the Sun RPC packet. Check RPC type field for ID. If value is portMapper, save paired socket (i.e., dest for destination address, src for source

address). Decode ports and mapping, save ports with socket/addr key. There may be more than one pairing per mapper packet. Form a signature (e.g., a key). A flow-entry is created in database **324**. The saving of the request is now complete.

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

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

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

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

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

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

In packet **206**, a fourth field **217** and a fifth field **218** are used to communicate port numbers that are used. The conversation direction determines where the port number field is. The diagonal pattern in field **217** is used to identify source-port pattern, and the hash pattern in field **218** is used to identify the destination-port pattern. The order indicates the client-server message direction. A sixth field denoted "i^1" **219** is an element that is being requested by the client from the server. A seventh field denoted "s_1a" **220** is the service requested by the client from server **110**. The following eighth field "QA" **221** (for question mark) indicates that the client **106** wants to know what to use to access application "s_1a". A tenth field "QP" **223** is used to indicate that the client wants the server to indicate what protocol to use for the particular application.

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

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

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

The flow signature and flow states built up as a result of this exchange are now described. When the packet monitor **300** sees the request packet **206** from the client, a first flow signature **210** is built in the parser subsystem **301** according to the pattern and extraction operations database **308**. This signature **210** includes a destination and a source address **240** and **241**. One aspect of the invention is that the flow keys are built consistently in a particular order no matter what the direction of conversation. Several mechanisms may be used to achieve this. In the particular embodiment, the numerically lower address is always placed before the numerically higher address. Such least to highest order is used to get the best spread of signatures and hashes for the lookup operations. In this case, therefore, since we assume "S_1"<"C_1", the order is address "S_1" followed by client address "C_1". The next field used to build the signature is a protocol field **242** extracted from packet **206**'s field **216**, and thus is the protocol "p^1". The next field used for the signature is field **243**, which contains the destination source port number shown as a crosshatched pattern from the field **218** of the packet **206**. This pattern will be recognized in the payload of packets to derive how this packet or sequence of packets exists as a flow. In practice, these may be TCP port numbers, or a combination of TCP port numbers. In the case of the Sun RPC example, the crosshatch represents a set of port numbers of UDS for p^1 that will be used to recognize this flow (e.g., port **111**). Port **111** indicates this is Sun RPC. Some applications, such as the Sun RPC Bind Lookups, are directly determinable ("known") at the parser level. So in this case, the signature KEY-1 points to a known application denoted "a^1" (Sun RPC Bind Lookup), and a next-state that the state processor should proceed to for more complex recognition jobs, denoted as state "st_D" is placed in the field **245** of the flow-entry.

When the Sun RPC Bind Lookup reply is acquired, a flow signature is again built by the parser. This flow signature is identical to KEY-1. Hence, when the signature enters the analyzer subsystem **303** from the parser subsystem **301**, the complete flow-entry is obtained, and in this flow-entry indicates state "st_D". The operations for state "st_D" in the state processor instruction database **326** instructs the state

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

The two flow signatures 210 and 212 always order the destination and source address fields with server "S_1" followed by client "C_1". Such values are automatically filled in when the addresses are first created in a particular flow signature. Preferably, large collections of flow signatures are kept in a lookup table in a least-to-highest order for the best spread of flow signatures and hashes.

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

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

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

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

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

Although the present invention has been described in terms of the presently preferred embodiments, it is to be understood that the disclosure is not to be interpreted as limiting. Various alterations and modifications will no doubt become apparent to those or ordinary skill in the art after having read the above disclosure. Accordingly, it is intended that the claims be interpreted as covering all alterations and modifications as fall within the true spirit and scope of the present invention.

The Pattern Parse and Extraction Database Format

The different protocols that can exist in different layers may be thought of as nodes of one or more trees of linked nodes. The packet type is the root of a tree (called base level). Each protocol is either a parent node of some other protocol at the next later or a terminal node. A parent node links a protocol to other protocols (child protocols) that can be at higher layer levels. Thus a protocol may have zero or more children.

As an example of the tree structure, consider an Ethernet packet. One of the children nodes may be the IP protocol, and one of the children of the IP protocol may be the TCP protocol. Another child of the IP may be the UDP protocol.

A packet includes at least one header for each protocol used. The child protocol of a particular protocol used in a packet is indicated by the contents at a location within the header of the particular protocol. The contents of the packet that specify the child are in the form of a child recognition pattern.

A network analyzer preferably can analyze many different protocols. At a base level, there are a number of packet types 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 packet types use 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; several such cells may be needed to make up the information that might be included in a single packet of some other type.

Note that the term packet herein 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," "frame format" and "cell format" are generally synonymous.

The packet monitor 300 can analyze different protocols, and thus can perform different protocol specific operations on a packet wherein the protocol headers of any protocol are located at different locations depending on the parent protocol or protocols used in the packet. Thus, the packet monitor adapts to different protocols according to the contents of the packet. The locations and the information extracted from any packet are adaptively determined for the particular type of packet. For example, there is no fixed definition of what to look for or where to look in order to form the flow signature. In some prior art systems, such as that described in U.S. Pat. No. 5,101,402 to Chiu, et al., there are fixed locations specified for particular types of packets. With the proliferation of protocols, the specifying of all the possible places to look to determine the session becomes more and more difficult. Likewise, adding a new protocol or application is difficult. In the present invention, the number of levels is variable for any protocol 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).

Even the same protocol may have different variants. Ethernet packets for example, have several known variants, each having a basic format that remains substantially the same. An Ethernet packet (the root node) may be an Ethertype packet—also called an Ethernet Type/Version 2 and a DIX (DIGITAL-Intel-Xerox packet)—or an IEEE Ethernet (IEEE 803.x) packet. A monitor should be able to handle all types of Ethernet protocols. With the Ethertype protocol, the contents that indicate the child protocol is in one location, while with an IEEE type, the child protocol is specified in a different location. The child protocol is indicated by a child recognition pattern.

FIG. **16** shows the header **1600** (base level **1**) of a complete Ethernet frame (i.e., packet) of information and includes information on the destination media access control address (Dst MAC **1602**) and the source media access control address (Src MAC **1604**). Also shown in FIG. **16** is some (but not all) of the information specified in the PDL files for extraction the signature. Such information is also to be specified in the parsing structures and extraction operations database **308**. This includes all of the header information at this level in the form of 6 bytes of Dst MAC information **1606** and 6 bytes of Src MAC information **1610**. Also specified are the source and destination address components, respectively, of the hash. These are shown as 2 byte Dst Hash **1608** from the Dst MAC address and the 2 byte Src Hash **1612** from the Src MAC address. Finally, information is included (**1614**) on where to the header starts for information related to the next layer level. In this case the next layer level (level **2**) information starts at packet offset **12**.

FIG. **17A** now shows the header information for the next level (level-2) for an Ethertype packet **1700**.

For an Ethertype packet **1700**, the relevant information from the packet that indicates the next layer level is a two-byte type field **1702** containing the child recognition pattern for the next level. The remaining information **1704** is shown hatched because it not relevant for this level. The list **1712** shows the possible children for an Ethertype packet as indicated by what child recognition pattern is found offset **12**.

Also shown is some of the extracted part used for the parser record and to locate the next header information. The signature part of the parser record includes extracted part **1702**. Also included is the 1-byte Hash component **1710** from this information.

An offset field **1710** provides the offset to go to the next level information, i.e., to locate the start of the next layer level header. For the Ethertype packet, the start of the next layer header 14 bytes from the start of the frame.

Other packet types are arranged differently. 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.

FIG. **17B** shows the structure of the header of one of the possible next levels, that of the IP protocol. The possible children of the IP protocol are shown in table **1752**. The header starts at a different location (L3) depending on the parent protocol. Also included in FIG. **17B** are some of the fields to be extracted for the signature, and an indication of where the next level's header would start in the packet.

Note that the information shown in FIGS. **16**, **17A**, and **17B** would be specified to the monitor in the form of PDL files and compiled into the database **308** of pattern structures and extraction operations.

The parsing subsystem **301** performs operations on the packet header data based on information stored in the database **308**. Because data related to protocols can be considered as organized in the form of a tree, it is required in the parsing subsystem to search through data that is originally organized in the form of a tree. Since real time operation is preferable, it is required to carry out such searches rapidly.

Data structures are known for efficiently storing information organized as trees. Such storage-efficient means typically require arithmetic computations to determine pointers to the data nodes. Searching using such storage-efficient data structures may therefore be too time consuming for the present application. It is therefore desirable to store the protocol data in some form that enables rapid searches.

In accordance with another aspect of the invention, the database **308** is stored in a memory and includes a data structure used to store the protocol specific operations that are to be performed on a packet. In particular, a compressed representation is used to store information in the pattern parse and extraction database **308** used by the pattern recognition process **304** and the extraction process **306** in the parser subsystem **301**. The data structure is organized for rapidly locating the child protocol related information by using a set of one or more indices to index the contents of the data structure. A data structure entry includes an indication of validity. Locating and identifying the child protocol includes indexing the data structure until a valid entry is found. Using the data structure to store the protocol information used by the pattern recognition engine (PRE) **1006** enables the parser subsystem **301** to perform rapid searches.

In one embodiment, the data structure is in the form of a three-dimensional structure. Note that this three dimensional structure in turn is typically stored in memory as a set of two-dimensional structures whereby one of the three dimensions of the 3-D structure is used as an index to a particular 2-D array. This forms a first index to the data structure.

FIG. **18A** shows such a 3-D representation **1800** (which may be considered as an indexed set of 2-D representations). The three dimensions of this data structure are:

1. Type identifier [1:M]. This is the identifier that identifies a type of protocol at a particular level. For example, 01 indicates an Ethernet frame. 64 indicates IP, 16 indicates an IEEE type Ethernet packet, etc. Depending on how many protocols the packet parser can handle, M may be a large number; M may grow over time as the capability of analyzing more protocols is added to monitor **300**. When the 3-D structure is considered a set of 2-D structures, the type ID is an index to a particular 2-D structure.
2. Size [1:64]. The size of the field of interest within the packet.
3. Location [1:512]. This is the offset location within the packet, expressed as a number of octets (bytes).

At any one of these locations there may or may not be valid data. Typically, there will not be valid data in most

locations. The size of the 3-D array is M by 64 by 512, which can be large; M for example may be 10,000. This is a sparse 3-D matrix with most entries empty (i.e., invalid).

Each array entry includes a "node code" that indicates the nature of the contents. This node code has one of four values: (1) a "protocol" node code indicating to the pattern recognition process **304** that a known protocol has been recognized as the next (i.e., child) protocol; (2) a "terminal" node code indicating that there are no children for the protocol presently being searched, i.e., the node is a final node in the protocol tree; (3) a "null" (also called "flush") node code indicating that there is no valid entry.

In the preferred embodiment, the possible children and other information are loaded into the data structure by an initialization that includes compilation process **310** based on the PDL files **336** and the layering selections **338**. The following information is included for any entry in the data structure that represents a protocol.

(a) A list of 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 the child recognition patterns that need to be compared. For example, $64{:}0800_{16}$ in the list indicates that the value to look for is 0800 (hex) 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 (hex) 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 in the signature, including what operations (AND, ORs, etc.) may need to be carried out. If there is also a hash key component, for instance, then information on that is included. For example, for an Ethertype packet, the 2-byte type (**1706** in FIG. **17**) is used in the signature. Furthermore, a 1-byte hash (**1708** in FIG. **17A**) of the type is included. . Note furthermore, the child protocol starts at offset **14**.

An additional item may be the "fold." Folding is used to reduce the storage requirements for the 3-D structure. Since each 2-D array for each protocol ID may be sparsely populated, multiple arrays may be combined into a single 2-D array as long as the individual entries do not conflict with each other. A fold number is then used to associate each element. For a given lookup, the fold number of the lookup must match the fold number entry. Folding is described in more detail below.

In the case of the Ethernet, the next protocol field may indicate a length, which tells the parser that this is a IEEE type packet, and that the next protocol is elsewhere. Normally, the next protocol field contains a value which identifies the next, i.e., child protocol.

The entry point for the parser subsystem is called the virtual base layer and contains the possible first children, i.e., the packet types. An example set of protocols written in a high level protocol description language (PDL) is included herein. The set includes PDL files, and the file describing all the possible entry points (i.e., the virtual base) is called virtual.pdl. There is only one child, 01, indicating the Ethernet, in this file. Thus, the particular example can only handle Ethernet packets. In practice, there can be multiple entry points.

In one embodiment, the packet acquisition device provides a header for every packet acquired and input into monitor **300** indicating the type of packet. This header is used to determine the virtual base layer entry point to the parser subsystem. Thus, even at the base layer, the parser subsystem can identify the type of packet.

Initially, the search starts at the child of the virtual base, as obtained in the header supplied by the acquisition device. In the case of the example, this has ID value 01, which is the 2-D array in the overall 3-D structure for Ethernet packets.

Thus hardware implementing pattern analysis process **304** (e.g., pattern recognition engine (PRE) **1006** of FIG. **10**) searches to determine the children (if any) for the 2-D array that has protocol ID 01. In the preferred embodiment that uses the 3-D data structure, the hardware PRE **1006** searches up to four lengths (i.e., sizes) simultaneously. Thus, the process **304** searches in groups of four lengths. Starting at protocol ID 01, the first two sets of 3-D locations searched are

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

At each stage of a search, the analysis process **304** examines the packet and the 3-D data structure to see if there is a match (by looking at the node code). If no valid data is found, e.g., using the node code, the size is incremented (to maximum of 4) and the offset is then incremented as well.

Continuing with the example, suppose the pattern analysis process **304** finds something at 1, 2, 12. By this, we mean that the process **304** has found that for protocol ID value 01 (Ethernet) at packet offset **12**, there is information in the packet having a length of 2 bytes (octets) that may relate to the next (child) protocol. The information, for example, may be about a child for this protocol expressed as a child recognition pattern. The list of possible child recognition patterns that may be in that part of the packet is obtained from the data structure.

The Ethernet packet structure comes in two flavors, the Ethertype packet and newer IEEE types, and the packet location that indicates the child is different for both. The location that for the Ethertype packet indicates the child is a "length" for the IEEE type, so a determination is made for the Ethernet packet whether the "next protocol" location contains a value or a length (this is called a "LENGTH" operation). A successful LENGTH operation is indicated by contents less than or equal to $05DC_{16}$, then this is an IEEE type Ethernet frame. In such a case, the child recognition pattern is looked for elsewhere. Otherwise, the location contains a value that indicates the child.

Note that while this capability of the entry being a value (e.g., for a child protocol ID) or a length (indicating further analysis to determine the child protocol) is only used for Ethernet packets, in the future, other packets may end up being modified. Accordingly, this capability in the form of a macro in the PDL files still enables such future packets to be decoded.

Continuing with the example, suppose that the LENGTH operation fails. In that case, we have an Ethertype packet, and the next protocol field (containing the child recognition pattern) is 2 bytes long starting at offset **12** as shown as packet field **1702** in FIG. **17A**. This will be one of the children of the Ethertype shown in table **1712** in FIG. **17A**. The PRE uses the information in the data structure to check what the ID code is for the found 2-byte child recognition pattern. For example, if the child recognition pattern is 0800

(Hex), then the protocol is IP. If the child recognition pattern is 0BAD (Hex) the protocol is VIP (VINES).

Note that an alternate embodiment may keep a separate table that includes all the child recognition patterns and their corresponding protocol ID's

To follow the example, suppose the child recognition pattern at **1, 2, 12** is 0800_{16}, indicating IP. The ID code for the IP protocol is 64_{10}). To continue with the Ethertype example, once the parser matches one of the possible children for the protocl—in the example, the protocol type is IP with an ID of 64—then the parser continues the search for the next level. The ID is 64, the length is unknown, and offset is 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 from location **(64, 1)** at packet offset **14**. A populated node is found at **(64, 2)** at packet offset **14**. Heading details are shown as **1750** in FIG. **17**B. The possible children are shown in table **1752**.

Alternatively, suppose that at **(1, 2, 12)** there was a length 1211_{10}. This indicates that this is an IEEE type Ethernet frame, which stores its type elsewhere. The PRE now continues its search at the same level, but for a new ID, that of an IEEE type Ethernet frame. An IEEE Ethernet packet has protocol ID 16, so the PRE continues its search of the three-dimensional space with ID 16starting at packet offset **14**.

In our example, suppose there is a "protocol" node code found at **(16, 2)** at packet offset **14**, and the next protocol is specified by child recognition pattern 0800_{16}. This indicates that the child is the IP protocol, which has type ID 64. Thus the search continues, starting at **(64, 1)** at packet offset **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=10,000, then this is about 10 gigabytes. It is not practical to include 10 Gbyte of memory in the parser subsystem for storing the database **308**. Thus a compressed form of storing the data is used in the preferred embodiment. The compression is preferably carried out by an optimizer component of the compilation process **310**.

Recall that the data structure is sparse. Different embodiments may use different compression schemes that take advantage of the sparseness of the data structure. One embodiment uses a modification of multi-dimensional run length encoding.

Another embodiment uses a smaller number two-dimensional structures to store the information that otherwise would be in one large three-dimensional structure. The second scheme is used in the preferred embodiment.

FIG. **18**A illustrated how the 3-D array **1800** can be considered a set of 2-D arrays, one 2-D array for each protocol (i.e., each value of the protocol ID). The 2-D structures are shown as **1802-1, 1802-2, . . . , 1802**-M for up to M protocol ID's. One table entry is shown as **1804**. Note that the gaps in table are used to illustrate that each 2-D structure table is typically large.

Consider the set of trees that represent the possible protocols. Each node represents a protocol, and a protocol may have a child or be a terminal protocol. The base (root) of the tree has all packet types as children. The other nodes form the nodes in the tree at various levels from level 1 to the final terminal nodes of the tree. Thus, one element in the base node may reference node ID **1**, another element in the base node may reference node ID **2** and so on. As the tree is traversed from the root, there may be points in the tree where the same node is referenced next. This would occur, for example, when an application protocol like Telnet can run on several transport connections like TCP or UDP. Rather than repeating the Telnet node, only one node is represented in the patterns database **308** which can have several parents. This eliminates considerable space explosion.

Each 2-D structure in FIG. **18**A represents a protocol. To enable saving space by using only one array per protocol which may have several parents, in one embodiment, the pattern analysis subprocess keeps a "current header" pointer. Each location (offset) index for each protocol 2-D array in the 3-D structure is a relative location starting with the start of header for the particular protocol.

Each of the two-dimensional arrays is sparse. The next step of the optimization, is checking all the 2-D arrays against all the other 2-D arrays to find out which ones can share memory. Many of these 2-D arrays are often sparsely populated in that they each have only a small number of valid entries. So, a process of "folding" is next used to combine two or more 2-D arrays together into one physical 2-D array without losing the identity of any of the original 2-D arrays (i.e., all the 2-D arrays continue to exist logically). Folding can occur between any 2-D arrays irrespective of their location in the tree as long as certain conditions are met.

Assume two 2-D arrays are being considered for folding. Call the first 2-D arrays A and the second 2-D array B. Since both 2-D arrays are partially populated, 2-D array B can be combined with 2-D arrays A if and only if none of the individual elements of these two 2-D arrays that have the same 2-D location conflict. If the result is foldable, then the valid entries of 2-D array B are combined with the valid entries of 2-D array A yielding one physical 2-D array. However, it is necessary to be able to distinguish the original 2-D array A entries from those of 2-D array B. For example, if a parent protocol of the protocol represented by 2-D array B wants to reference the protocol ID of 2-D array B, it must now reference 2-D array A instead. However, only the entries that were in the original 2-D array B are valid entries for that lookup. To accomplish this, each element in any given 2-D array is tagged with a fold number. When the original tree is created, all elements in all the 2-D arrays are initialized with a fold value of zero. Subsequently, if 2-D array B is folded into 2-D array A, all valid elements of 2-D array B are copied to the corresponding locations in 2-D array A and are given different fold numbers than any of the elements in 2-D array A. For example, if both 2-D array A and 2-D array B were original 2-D arrays in the tree (i.e., not previously folded) then, after folding, all the 2-D array A entries would still have fold **0** and the 2-D array B entries would now all have a fold value of 1. After 2-D array B is folded into 2-D array A, the parents of 2-D array B need to be notified of the change in the 2-D array physical location of their children and the associated change in the expected fold value.

This folding process can also occur between two 2-D arrays that have already been folded, as long as none of the individual elements of the two 2-D arrays conflict for the same 2-D array location. As before, each of the valid elements in 2-D array B must have fold numbers assigned to them that are unique from those of 2-D array A. This is accomplished by adding a fixed value to all the 2-D array B fold numbers as they are merged into 2-D array A. This fixed value is one larger than the largest fold value in the original 2-D array A. It is important to note that the fold number for any given 2-D array is relative to that 2-D array only and does not span across the entire tree of 2-D arrays.

This process of folding can now be attempted between all combinations of two 2-D arrays until there are no more candidates that qualify for folding. By doing this, the total number of 2-D arrays can be significantly reduced.

Whenever a fold occurs, the 3-D structure (i.e., all 2-D arrays) must be searched for the parents of the 2-D array being folded into another array. The matching pattern which previously was mapped to a protocol ID identifying a single 2-D array must now be replaced with the 2-D array ID and the next fold number (i.e., expected fold).

Thus, in the compressed data structure, each entry valid entry includes the fold number for that entry, and additionally, the expected fold for the child.

An alternate embodiment of the data structure used in database **308** is illustrated in FIG. **18**B. Thus, like the 3-D structure described above, it permits rapid searches to be performed by the pattern recognition process **304** by indexing locations in a memory rather than performing address link computations. The structure, like that of FIG. **18**A, is suitable for implementation in hardware, for example, for implementation to work with the pattern recognition engine (PRE) **1006** of FIG. **10**.

A table **1850**, called the protocol table (PT) has an entry for each protocol known by the monitor **300**, and includes some of the characteristics of each protocol, including a description of where the field that specifies next protocol (the child recognition pattern) can be found in the header, the length of the next protocol field, flags to indicate the header length and type, and one or more slicer commands, the slicer can build the key components and hash components for the packet at this protocol at this layer level.

For any protocol, there also are one or more lookup tables (LUTs). Thus database **308** for this embodiment also includes a set of LUTs **1870**. Each LUT has 256 entries indexed by one byte of the child recognition pattern that is extracted from the next protocol field in the packet. Such a protocol specification may be several bytes long, and so several of LUTs **1870** may need to be looked up for any protocol.

Each LUT's entry includes a 2-bit "node code" that indicates the nature of the contents, including its validity. This node code has one of four values: (1) a "protocol" node code indicating to the pattern recognition engine **1006** that a known protocol has been recognized; (2) an "intermediate" node code, indicating that a multi-byte protocol code has been partially recognized, thus permitting chaining a series of LUTs together before; (3) a "terminal" node code indicating that there are no children for the protocol presently being searched, i.e., the node is a final node in the protocol tree; (4) a "null" (also called "flush" and "invalid") node code indicating that there is no valid entry.

In addition to the node code, each LUT entry may include the next LUT number, the next protocol number (for looking up the protocol table **1850**), the fold of the LUT entry, and the next fold to expect. Like in the embodiment implementing a compressed form of the 3-D representation, folding is used to reduce the storage requirements for the set of LUTs. Since the LUTs **1870** may be sparsely populated, multiple LUTs may be combined into a single LUT as long as the individual entries do not conflict with each other. A fold number is then used to associate each element with its original LUT.

For a given lookup, the fold number of the lookup must match the fold number in the lookup table. The expected fold is obtained from the previous table lookup (the "next fold to expect" field). The present implementation uses 5-bits to describe the fold and thus allows up to 32 tables to be folded into one table.

When using the data structure of FIG. **18**B, when a packet arrives at the parser, the virtual base has been pre-pended or is known. The virtual base entry tells the packet recognition engine where to find the first child recognition pattern in the packet. The pattern recognition engine then extracts the child recognition pattern bytes from the packet and uses them as an address into the virtual base table (the first LUT). If the entry looked up in the specified next LUT by this method matches the expected next fold value specified in the virtual base entry, the lookup is deemed valid. The node code is then examined. If it is an intermediate node then the next table field obtained from the LUT lookup is used as the most significant bits of the address. The next expected fold is also extracted from the entry. The pattern recognition engine **1006** then uses the next byte from the child recognition pattern as the for the next LUT lookup.

Thus, the operation of the PRE continues until a terminal code is found. The next (initially base layer) protocol is looked up in the protocol table **1850** to provide the PRE **1006** with information on what field in the packet (in input buffer memory **1008** of parser subsystem **1000**) to use for obtaining the child recognition pattern of the next protocol, including the size of the field. The child recognition pattern bytes are fetched from the input buffer memory **1008**. The number of bytes making up the child recognition pattern is also now known.

The first byte of the protocol code bytes is used as the lookup in the next LUT. If a LUT lookup results in a node code indicating a protocol node or a terminal node, the Next LUT and next expected fold is set, and the "next protocol" from LUT lookup is used as an index into the protocol table **1850**. This provides the instructions to the slicer **1007**, and where in the packet to obtain the field for the next protocol. Thus, the PRE **1006** continues until it is done processing all the fields (i.e., the protocols), as indicated by the terminal node code reached.

Note that when a child recognition pattern is checked against a table there is always an expected fold. If the expected fold matches the fold information in the table, it is used to decide what to do next. If the fold does not match, the optimizer is finished.

Note also that an alternate embodiment may use different size LUTs, and then index a LUT by a different amount of the child recognition pattern.

The present implementation of this embodiment allows for child recognition patterns of up to four bytes. Child recognition patterns of more than 4 bytes are regarded as special cases.

In the preferred embodiment, the database is generated by the compiler process **310**. The compiler process first builds a single protocol 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 only 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 the LUTs of 256 entries.

Optimization is then carried out. The first step in the optimization is checking all the tables against all the other tables to find out which ones can share a table. This process proceeds the same way as described above for two-dimensional arrays, but now for the sparse lookup tables.

Part of the initialization process (e.g., compiler process **310**) loads a slicer instruction database with data items

including of instruction, source address, destination address, and length. The PRE **1006** when it sends a slicer instruction sends this instruction as an 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 how to determine the connection status of certain protocols.

Note that alternate embodiments may generate the pattern, parse and extraction database other than by compiling PDL files.

The Compilation Process

The compilation process **310** is now described in more detail. This process **310** includes creating the parsing patterns and extractions database **308** that provides the parsing subsystem **301** with the information needed to parse packets and extract identifying information, and the state processing instructions database **326** that provides the state processes that need to be performed in the state processing operation **328**.

Input to the compiler includes a set of files that describe each of the protocols that can occur. These files are in a convenient protocol description language (PDL) which is a high level language. PDL is used for specifying new protocols and new levels, including new applications. The PDL is independent of the different types of packets and protocols that may be used in the computer network. A set of PDL files is used to describe what information is relevant to packets and packets that need to be decoded. The PDL is further used to specify state analysis operations. Thus, the parser subsystem and the analyzer subsystems 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.

There is one file for each packet type and each protocol. Thus there is a PDL file for Ethernet packets and there is a PDL file for frame relay packets. The PDL files are compiled to form one or more databases that enable monitor **300** to perform different protocol specific operations on a packet wherein the protocol headers of any protocol are located at different locations depending on the parent protocol or protocols used in the packet. Thus, the packet monitor adapts to different protocols according to the contents of the packet. In particular, the parser subsystem **301** is able to extract different types of data for different types of packets. For example, the monitor can know how to interpret a Ethernet packet, including decoding the header information, and also how to interpret an frame relay packet, including decoding the header information.

The set of PDL files, for example, may include a generic Ethernet packet file. There also is included a PDL file for each variation Ethernet file, for example, an EEE Ethernet file.

The PDL file for a protocol provides the information needed by compilation process **310** to generate the database **308**. That database in turn tells the parser subsystem how to parse and/or extract information, including one or more of what protocol-specific components of the packet to extract for the flow signature, how to use the components to build the flow signature, where in the packet to look for these components, where to look for any child protocols, and what child recognition patterns to look for. For some protocols, the extracted components may include source and destination addresses, and the PDL file may include the order to use these addresses to build the key. For example, Ethernet frames have end-point addresses that are useful in building a better flow signature. Thus the PDL file for an Ethernet packet includes information on how the parsing subsystem is 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 to identify the flow better, so for those type of packets, the PDL file does not include information that will cause the parser subsystem to extract the end-point addresses.

Some protocols also include information on connections. TCP is an example of such a protocol. Such protocol use connection identifiers that exist in every packet. The PDL file for such a protocol includes information about what those connection identifiers are, where they are, and what their length is. In the example of TCP, for example running over IP, these are port numbers. The PDL file also includes information about whether or not there are states that apply to connections and disconnections and what the possible children are states. So, at each of these levels, the packet monitor **300** learns more about the packet. The packet monitor **300** can identify that a particular packet is part of a particular flow using the connection identifier. Once the flow is identified, the system can determine the current state and what states to apply that deal with connections or disconnections that exist in the next layer up to these particular packets.

For the particular PDL used in the preferred embodiment, a PDL file may include none or more FIELD statement each defining a specific string of bits or bytes (i.e., a field) in the packet. A PDL file may further include none or more GROUP statements each used to tie together several defined fields. A set of such tied together fields is called a group. A PDL file may further include none or more PROTOCOL statements each defining the order of the fields and groups within the header of the protocol. A PDL file may further include none or more FLOW statements each defining a flow by describing where the address, protocol type, and port numbers are in a packet. The FLOW statement includes a description of how children flows of this protocol are determined using state operations. States associated may have state operations that may be used for managing and maintaining new states learned as more packets of a flow are analyzed.

FIG. **19** shows a set of PDL files for a layering structure for an Ethernet packet that runs TCP on top of IP. The contents of these PDL files are attached as an APPENDIX hereto. Common.pdl **(1903)** is a file containing the common protocol definitions, i.e., some field definitions for commonly used fields in various network protocols. Flows.pdl **(1905)** is a file containing general flow definitions. Virtual.pdl **(1907)** is a PDL file containing the definition for the VirtualBase layer used. Ethernet.pdl **(1911)** is the PDL file containing the definition for the Ethernet packet. The decision on Ethertype vs. IEEE type Ethernet file is described herein. If this is Ethertype, the selection is made from the file Ethertype.pdl **(1913)**. In an alternate embodiment, the Ethertype selection definition may be in the same Ethernet file **1911**. In a typical implementation, PDL files for other Ethernet types would be included. IP.pdl **(1915)** is a PDL file containing the packet definitions for the Internet Protocol. TCP.pdl **(1917)** is the PDL file containing the packet definitions for the Transmission Control Protocol, which in this case is a transport service for the IP protocol. In addition to extracting the protocol information the TCP protocol definition file assists in the process of identification of connec-

tions for the processing of states. In a typical set of files, there also would be a file UDP.pdl for the User Datagram Protocol (UDP) definitions. RPC.pdl (**1919**) is a PDL file file containing the packet definitions for Remote Procedure Calls.

NFS.pdl (**1921**) is a PDL file containing the packet definitions for the Network File System. Other PDL files would typically be included for all the protocols that might be encountered by monitor **300**.

Input to the compilation process **310** is the set of PDL files (e.g., the files of FIG. **19**) for all protocols of interest. Input to process **310** may also include layering information shown in FIG. **3** as datagram layer selections **338**. The layer selections information describes the layering of the protocols—what protocol(s) may be on top of any particular protocols. For example, IP may run over Ethernet, and also over many other types of packets. TCP may run on top of IP. UDP also may run on top of IP. When no layering information is explicitly included, it is inherent; the PDL files include the children protocols, and this provides the layering information.

The compiling process **310** is illustrated in FIG. **20**. The compiler loads the PDL source files into a scratch pad memory (step **2003**) and reviews the files for the correct syntax (parse step **2005**). Once completed, the compiler creates an intermediate file containing all the parse elements (step **2007**). The intermediate file in a format called "Compiled Protocol Language" (CPL). CPL instructions have a fixed layer format, and include all of the patterns, extractions, and states required for each layer and for the entire tree for a layer. The CPL file includes the number of protocols and the protocol definitions. A protocol definition for each protocol can include one or more of the protocol name, the protocol ID, a header section, a group identification section, sections for any particular layers, announcement sections, a payload section, a children section, and a states section. The CPL file is then run by the optimizer to create the final databases that will be used by monitor **300**. It would be clear to those in the art that alternate implementations of the compilation process **310** may include a different form of intermediate output, or no intermediate output at all, directly generating the final database(s).

After the parse elements have been created, the compiler builds the flow signature elements (step **2009**). This creates the extraction operations in CPL that are required at each level for each PDL module for the building of the flow signature (and hash key) and for links between layers (**2009**).

With the flow signature operations complete, the PDL compiler creates (step **2011**) 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.

The last pass is to create the state operations required by each PDL module. The state operations are complied from the PDL files and created in CPL form for later use (**2013**).

The CPL file is now run through an optimizer that generates the final databases used by monitor **300**.

PROTOCOL DEFINITION LANGUAGE (PDL) REFERENCE GUIDE (VERSION A0.02)

Included herein is this reference guide (the "guide") for the page description language (PDL) which, 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 capability of the monitor.

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 guide, 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.

1.2 Guide Conventions

The following conventions will be used throughout this guide:

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.

Note that any contents on any line in a PDL file following two hyphen (--) are ignored by the compiler. That is, they are comments.

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:

SAMELAYER	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 DESCRIPITION "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-HEADER	Indicates that the protocol header should be included.
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 0 × 05DC 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"
```

-continued

```
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 " ($ipLength- ($jpHeaderLength*4)) -
        ($tcpHeaderLen*4) "
        ENCAP           tcpport
        DISPLAY-HINT    "HexDump"
tcp     PROTOCOL
```

-continued

```
        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"
tcpOptionDataFIELD
                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
```

-continued

```
                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 /ids?'       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"
mimApplication      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
```

-continued

```
                '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
```

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

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

Although the present invention has been described in terms of the presently preferred embodiments, it is to be understood that the disclosure is not to be interpreted as limiting. Various alterations and modifications will no doubt become apparent to those or ordinary skill in the art after having read the above disclosure. Accordingly, it is intended that the claims be interpreted as covering all alterations and modifications as fall within the true spirit and scope of the present invention.

APPENDIX: 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
```

-continued

```
        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
snn         FLOW
oracleTNS FLOW
        PAYLOAD { INCLUDE-HEADER, LENGTH=256 }
ciscoOUI FLOW
-------------------------------------------------------------
--  IP Protocols
-------------------------------------------------------------
igmp        FLOW
GGP         FLOW
ST          FLOW
UCL         FLOW
```

-continued

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

-continued

```
--  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
rip          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
```

-continued

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

-continued

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

-continued

```
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
ariell          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
```

-continued

```
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,
--       (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, which allows the system
--  to look up multiple buckets with a single HASH. That is, the MeyMatchFlag,
--  StateStatus Flag and MulipacketID may be varied.
--
FLOWKEY {
```

-continued

```
    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(*) { 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),
```

-continued

```
                        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 tyoe 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=11c }
-------------------------------------------------------------------
--
--  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
--
11cSap      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    "ix:"
            DESCRIPTION
                "Service Access Point"
11cControl      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"
```

-continued

```
11cPduType      FIELD
        SYNTAX BITSTRING(2) { 11cInformation(0), 11cSupervisory(1),
            11cInformation(2), 11cUnnumbererd(3) }
11cData         FIELD
        SYNTAX          BYTESTRING(38..1492)
        ENCAP           11cPduType
        FLAGS           SAMELAYER
        DISPLAY-HINT    "HexDump"
11c     PROTOCOL
        SUMMARIZE
            "$11cPduType == 11cUnnumbered" :
              "LLC ($SAP) $Modifier"
            "$11cPduType == 11cSupervisory" :
              "LLC ($SAP) $Function N(R)=$NR"
            "$11cPduType == 0|2" :
              "LLC ($SAP) N(R)=$NR N(S)=$NS"
            "Default"
              "LLC ($SAP) $11cPduType"
        DESCRIPTION
            "IEEE 802.2 LLC frame format"
::= { SAP=11cSap, Control=11cControl, Data=11cData }
11c      FLOW
        HEADER { LENGTH=3 }
        DLC-LAYER { PROTOCOL }
        CHILDREN { DESTINATION=SAP }
11cUnnumberedData FIELD
        SYNTAX          BYTESTRING(0..1500)
        ENCAP           11cSap
        DISPLAY-HINT    "HexDump"
11cUnnumbered PROTOCOL
        SUMMARIZE
            "Default" :
              "LLC ($SAP) $Modifier"
::= { Data=11cUnnumberedData }
11cSupervisoryData      FIELD
        SYNTAX          BYTESTRING(0..1500)
        DISPLAY-HINT    "HexDump"
11cSupervisory          PROTOCOL
        SUMMARIZE
            "Default" :
              "LLC ($SAP) $Function N(R)=$NR"
::= { Data=11cSupervisoryData }
11cInformationData      FIELD
        SYNTAX          BYTESTRING(0..1500)
        ENCAP           11cSap
        DISPLAY-HINT    "HexDump"
11cInformation          PROTOCOL
        SUMMARIZE
            "Default" :
              "LLC ($SAP) N(R)=$NR N(S)=$NS"
::= { Data=11cInformationData }
--
--  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)
```

-continued

```
                    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           =11c
        LENGTH          "$ieee8023Length"
        DISPLAY-HINT    "HexDump"
ieee8023        PROTOCOL
        DESCRIPTION
            "IEEE 802.3 (Ethernet) frame"
        REFERENCE       "RFC 1042"
::= { MacDest=macAddress, Mac:Src=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
```

-continued

```
--  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
                SYNTAXBITSTRING(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
                SYNTAXINT(8) { ipRR(0x07), ipTimestamp(0x44),
                    ipLSRR(0x83), ipSSRR(0x89) }
                DESCRIPTION
                  "IP option code"
```

-continued

```
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
    SYNTAXBITSTRING(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),
```

-continued

```
                                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          PROTCCOL
::= { 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)
        LOOKUPFILE "Udpport.cf"
udpLength FIELD
        SYNTAX          UNSIGNED INT(16)
udpData FIELD
        SYNTAX          BYTESTRING(0..1500)
        ENCAP           udpPort
        DISPLAY-HINT    "HexDump"
udp     PROTOCOL
        SUMARIZE
            "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,
--     (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"
```

-continued

```
            "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"
```

-continued

```
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), ReDectedVerifier(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 Techhically 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
```

-continued

```
        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 }
```

-continued

```
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 akip 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|^[L1]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:    NATCH
                         '\n\r\n'              900 0 0 255 0, NEXT S3
```

-continued

```
                    '\n\n'                  900 0 0 255 0, NEXT S3
                    'POST /tds?'             50 0 0 127 1, CHILD sybaseWebsql
                    '.bts 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.m-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
```

What is claimed is:

1. A method of performing protocol specific operations on a packet passing through a connection point on a computer network, the method comprising:

(a) receiving the packet:

(b) receiving a set of protocol descriptions for a plurality of protocols that conform to a layered model, a protocol description for a particular protocol at a particular layer level including:

(i) if there is at least one child protocol of the protocol at the particular layer level, the-one or more child protocols of the particular protocol at the particular layer level, the packet including for any particular child protocol of the particular protocol at the particular layer level information at one or more locations in the packet related to the particular child protocol,

(ii) the one or more locations in the packet where information is stored related to any child protocol of the particular protocol, and

(iii) if there is at least one protocol specific operation to be performed on the packet for the particular protocol at the particular layer level, the one or more protocol specific operations to be performed on the packet for the particular protocol at the particular layer level; and

(c) performing the protocol specific operations on the packet specified by the set of protocol descriptions based on the base protocol of the packet and the children of the protocols used in the packet,

the method further comprising:

storing a database in a memory, the database generated from the set of protocol descriptions and including a data structure containing information on the possible protocols and organized for locating the child protocol related information for any protocol, the data structure contents indexed by a set of one or more indices, the database entry indexed by a particular set of index values including an indication of validity,

wherein the child protocol related information includes a child recognition pattern,

wherein step (c) of performing the protocol specific operations includes, at any particular protocol layer level starting from the base level, searching the packet at the particular protocol for the child field, the searching including indexing the data structure until a valid entry is found, and

whereby the data structure is configured for rapid searches using the index set.

2. A method according to claim 1, wherein the protocol descriptions are provided in a protocol description language, the method further comprising:

compiling the PDL descriptions to produce the database.

3. A method according to claim 1, wherein the data structure comprises a set of arrays, each array identified by a first index, at least one array for each protocol, each array further indexed by a second index being the location in the packet where the child protocol related information is stored, such that finding a valid entry in the data structure provides the location in the packet for finding the child recognition pattern for an identified protocol.

4. A method according to claim 3, wherein each array is further indexed by a third index being the size of the region in the packet where the child protocol related information is stored, such that finding a valid entry in the data structure provides the location and the size of the region in the packet for finding the child recognition pattern.

5. A method according to claim 4, wherein the data structure is compressed according to a compression scheme that takes advantage of the sparseness of valid entries in the data structure.

6. A method according to claim 5, wherein the compression scheme combines two or more arrays that have no conflicting common entries.

7. A method according to claim 1, wherein the data structure includes a set of tables, each table identified by a first index, at least one table for each protocol, each table further indexed by a second index being the child recognition pattern, the data structure further including a table that for each protocol provides the location in the packet where the child protocol related information is stored, such that finding a valid entry in the data structure provides the location in the packet for finding the child recognition pattern for an identified protocol.

8. A method according to claim 7, wherein the data structure is compressed according to a compression scheme that takes advantage of the sparseness of valid entries in the set of tables.

9. A method according to claim 8, wherein the compression scheme combines two or more tables that have no conflicting common entries.

10. A method of performing protocol specific operations on a packet passing through a connection point on a computer network, the method comprising:

(a) receiving the packet;

(b) receiving a set of protocol descriptions for a plurality of protocols that conform to a layered model, a protocol description for a particular protocol at a particular layer level including:

(i) if there is at least one child protocol of the protocol at the particular layer level, the-one or more child protocols of the particular protocol at the particular layer level, the packet including for any particular child protocol of the particular protocol at the particular layer level information at one or more locations In the packet related to the particular child protocol,

(ii) the one or more locations in the packet where information is stored related to any child protocol of the particular protocol, and

(iii) if there is at least one protocol specific operation to be performed on the packet for the particular protocol at the particular layer level, the one or more protocol specific operations to be performed on the packet for the particular protocol at the particular layer level: and

(c) performing the protocol specific operations on the packet specified by the set of protocol descriptions based on the base protocol of the packet and the children of the protocols used in the packet,

wherein the protocol specific operations include one or more parsing and extraction operations on the packet to extract selected portions of the packet to form a function of the selected portions for identifying the packet as belonging to a conversational flow.

11. A method according to claim 10, wherein step (c) of performing protocol specific operations is performed recursively for any children of the children.

12. A method according to claim 10, wherein which protocol specific operations are performed is step (c) depends on the contents of the packet such that the method adapts to different protocols according to the contents of the packet.

13. A method according to claim 10, wherein the protocol descriptions are provided in a protocol description language.

14. A method according to claim 13, further comprising:

compiling the PDL descriptions to produce a database and store the database in a memory, the database generated from the set of protocol descriptions and including a data structure containing information on the possible protocols and organized for locating the child protocol related information for any protocol, the data structure contents indexed by a set of one or more indices, the database entry indexed by a particular set of index values including an indication of validity,

wherein the child protocol related information includes a child recognition pattern, and

wherein the step of performing the protocol specific operations includes, at any particular protocol layer level starting from the base level, searching the packet at the particular protocol for the child field, the searching including indexing the data structure until a valid entry is found,

whereby the data structure is configured for rapid searches using the index set.

15. A method according to claim 10, further comprising:

looking up a flow-entry database comprising at least one flow-entry for each previously encountered conversational flow, the looking up using at least some of the selected packet portions and determining if the packet matches an flow-entry in the flow-entry database

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

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

wherein for at least one protocol, the parsing and extraction operations depend on the contents of one or more packet headers.

16. A method according to claim 10, wherein the protocol specific operations further include one or more state processing operations that are a function of the state of the flow of the packet.

17. A method of performing protocol specific operations on a packet passing through a connection point on a computer network, the method comprising:

(a) receiving the packet;

(b) receiving a set of protocol descriptions for a plurality of protocols that conform to a layered model, a protocol description for a particular protocol at a particular layer level including:

(i) if there is at least one child protocol of the protocol at the particular layer level, the one or more child protocols of the particular protocol at the particular layer level, the packet including for any particular child protocol of the particular protocol at the particular layer level information at one or more locations in the packet related to the particular child protocol,

(ii) the one or more locations in the packet where information is stored related to any child protocol of the particular protocol, and

(iii) if there is at least one protocol specific operation to be performed on the packet for the particular protocol at the particular layer level, the one or more protocol specific operations to be performed on the packet for the particular protocol at the particular layer level; and

(c) performing the protocol specific operations on the packet specified by the set of protocol descriptions based on the base protocol of the packet and the children of the protocols used in the packet,

wherein the packet belongs to a conversational flow of packets having a set of one or more states, and wherein the protocol specific operations include one or more state processing operations that are a function of the state of the conversational flow of the packet, the state of the conversational flow of the packet being indicative of the sequence of any previously encountered packets of the same conversational flow as the packet.

* * * * *

US006839751B1

(12) **United States Patent**
Dietz et al.

(10) **Patent No.: US 6,839,751 B1**
(45) **Date of Patent: Jan. 4, 2005**

(54) **RE-USING INFORMATION FROM DATA TRANSACTIONS FOR MAINTAINING STATISTICS IN NETWORK MONITORING**

(75) Inventors: **Russell S. Dietz**, San Jose, CA (US); **Joseph R. Maixner**, Aptos, CA (US); **Andrew A. Koppenhaver**, Littleton, CO (US)

(73) Assignee: **Hi/fn, Inc.**, Los Gatos, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 728 days.

(21) Appl. No.: **09/608,126**

(22) Filed: **Jun. 30, 2000**

Related U.S. Application Data

(60) Provisional application No. 60/141,903, filed on Jun. 30, 1999.

(51) **Int. Cl.**[7] **G06F 15/173**

(52) **U.S. Cl.** **709/224**; 709/223; 709/230

(58) **Field of Search** 709/223, 224, 709/231, 232, 230; 370/252, 231; 379/32; 704/43; 714/39; 340/825

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,972,453	A *	11/1990	Daniel et al.	379/9.03
5,535,338	A *	7/1996	Krause et al.	709/222
5,703,877	A	12/1997	Nuber et al.	370/395
5,720,032	A *	2/1998	Picazo, Jr. et al.	709/250
5,761,429	A *	6/1998	Thompson	709/224
5,799,154	A *	8/1998	Kuriyan	709/223
5,802,054	A *	9/1998	Bellenger	370/401
5,850,388	A *	12/1998	Anderson et al.	370/252
5,892,754	A	4/1999	Kompella et al.	370/236
6,097,699	A *	8/2000	Chen et al.	370/231
6,115,393	A *	9/2000	Engel et al.	370/469
6,269,330	B1 *	7/2001	Cidon et al.	704/43
6,279,113	B1 *	8/2001	Vaidya	713/201
6,282,570	B1 *	8/2001	Leung et al.	709/224
6,330,226	B1 *	12/2001	Chapman et al.	370/232
6,363,056	B1 *	3/2002	Beigi et al.	370/252
6,381,306	B1 *	4/2002	Lawson et al.	379/32
6,424,624	B1 *	7/2002	Galand et al	370/231
6,453,345	B2 *	9/2002	Trcka et al.	709/224
6,625,657	B1 *	9/2003	Bullard	709/237
6,651,099	B1 *	11/2003	Dietz et al.	709/224

OTHER PUBLICATIONS

NOV94: Packet Filtering in the SNMP Remote Monitor ; www.skrymir.com/dobbs/articles/1994/9411/9411h/9411h.htm.*

GTrace—A Graphical Traceroute Tool authored by Ram Periakaruppan, Evi Nemeth ; http://www.caida.org/outreach/papers/1999/GTrace/index.xml.*

Advanced Methods for Storage and Retrieval in Image ; http://www.cs.tulane.edu/www/Prototype/proposal.html; 1998.*

Measurement and analysis of the digital DECT propagation channel; IEEE 1998.*

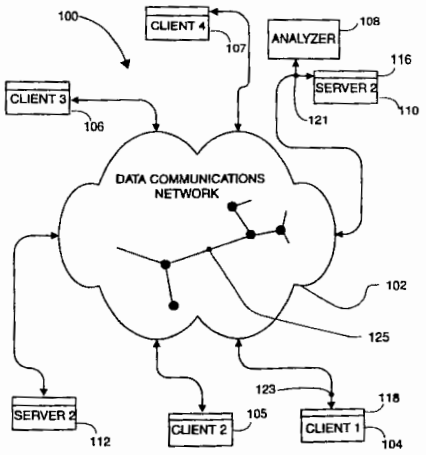
* cited by examiner

Primary Examiner—Thong Vu
(74) *Attorney, Agent, or Firm*—Dov Rosenfeld; Inventek

(57) **ABSTRACT**

A method of and monitor apparatus for analyzing a flow of packets passing through a connection point on a computer network. The method includes receiving a packet from a packet acquisition device, and looking up a flow-entry database containing flow-entries for previously encountered conversational flows. The looking up to determine if the received packet is of an existing flow. Each and every packet is processed. If the packet is of an existing flow, the method updates the flow-entry of the existing flow, including storing one or more statistical measures kept in the flow-entry. If the packet is of a new flow, the method stores a new flow-entry for the new flow in the flow-entry database, including storing one or more statistical measures kept in the flow-entry. The statistical measures are used to determine metrics related to the flow. The metrics may be base metrics from which quality of service metrics are determined, or may be the quality of service metrics.

21 Claims, 18 Drawing Sheets



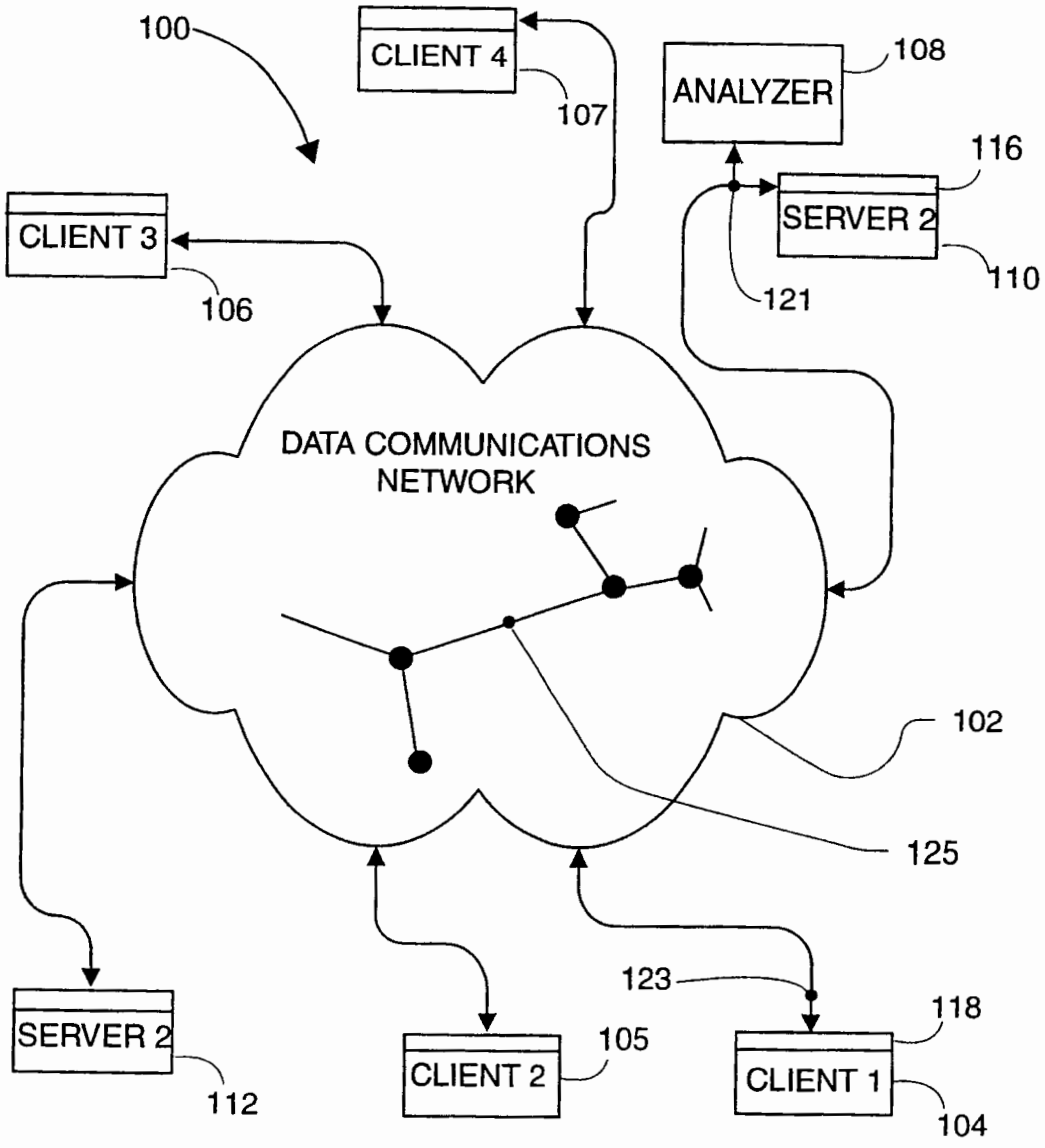

100
CLIENT 4
107
ANALYZER
108
116
SERVER 2
110
121
CLIENT 3
106
DATA COMMUNICATIONS NETWORK
102
125
123
118
SERVER 2
112
CLIENT 2
105
CLIENT 1
104

FIG. 1

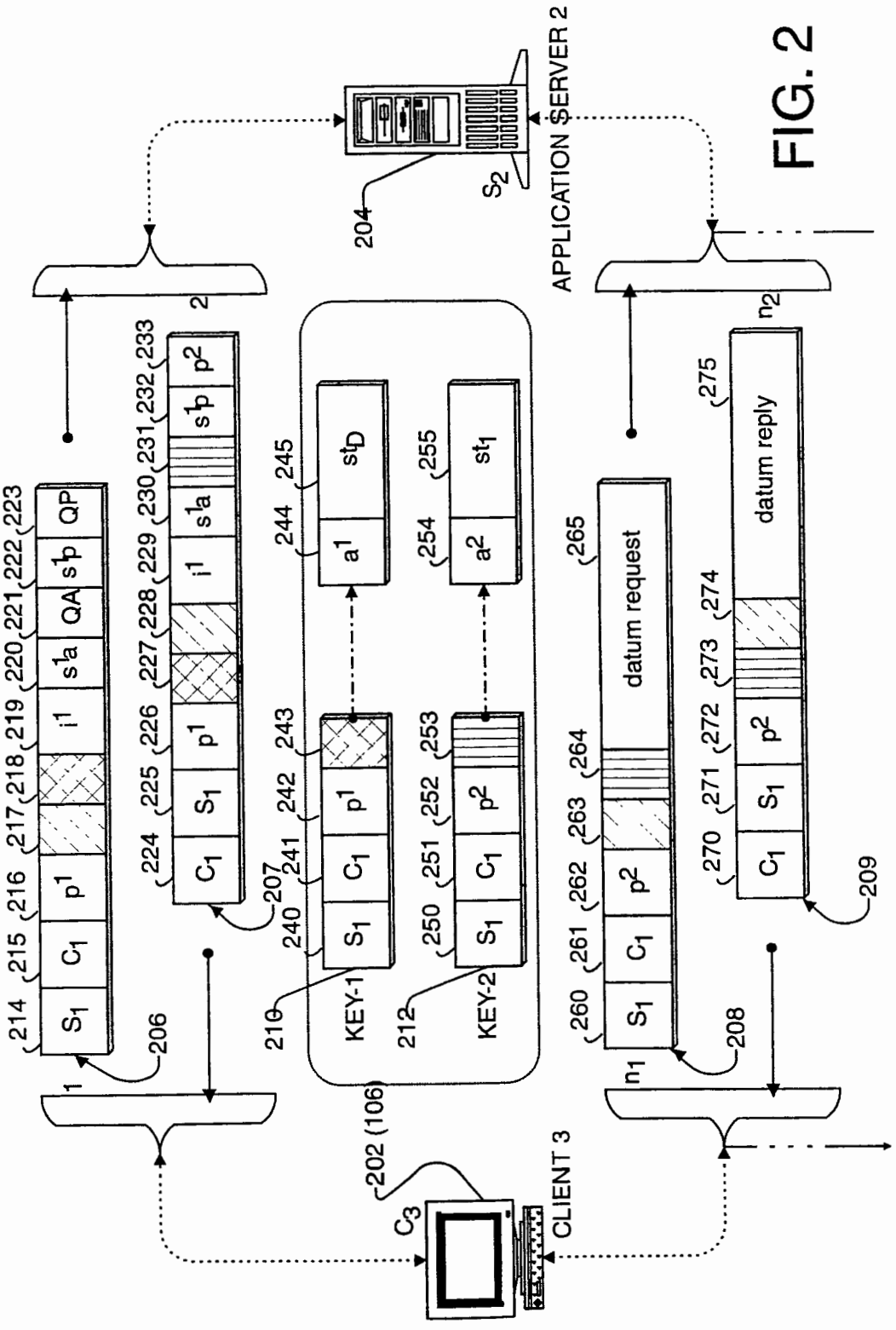


FIG. 2

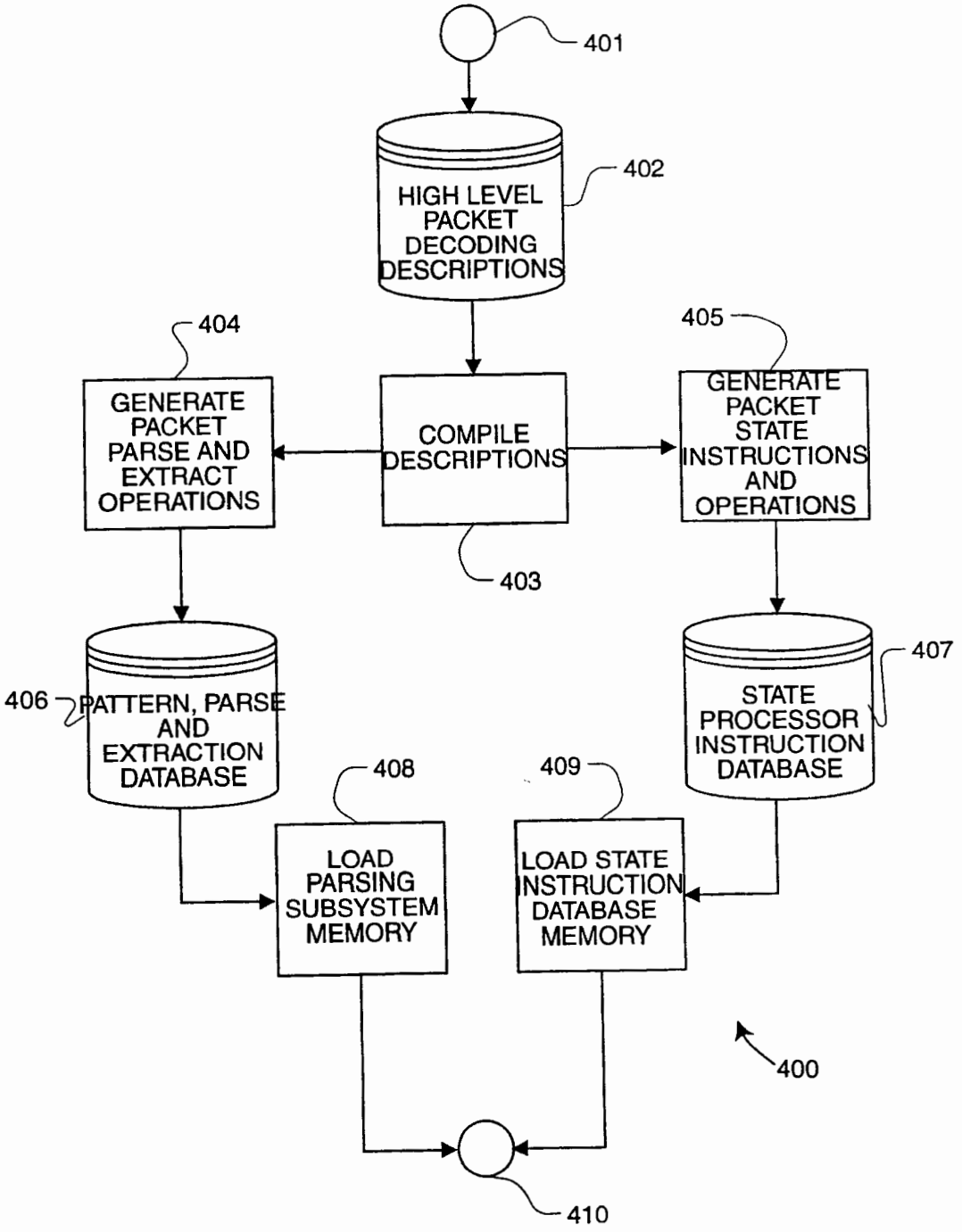

401
HIGH LEVEL PACKET DECODING DESCRIPTIONS
402
404
GENERATE PACKET PARSE AND EXTRACT OPERATIONS
COMPILE DESCRIPTIONS
403
405
GENERATE PACKET STATE INSTRUCTIONS AND OPERATIONS
406
PATTERN, PARSE AND EXTRACTION DATABASE
407
STATE PROCESSOR INSTRUCTION DATABASE
408
LOAD PARSING SUBSYSTEM MEMORY
409
LOAD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

FIG. 4

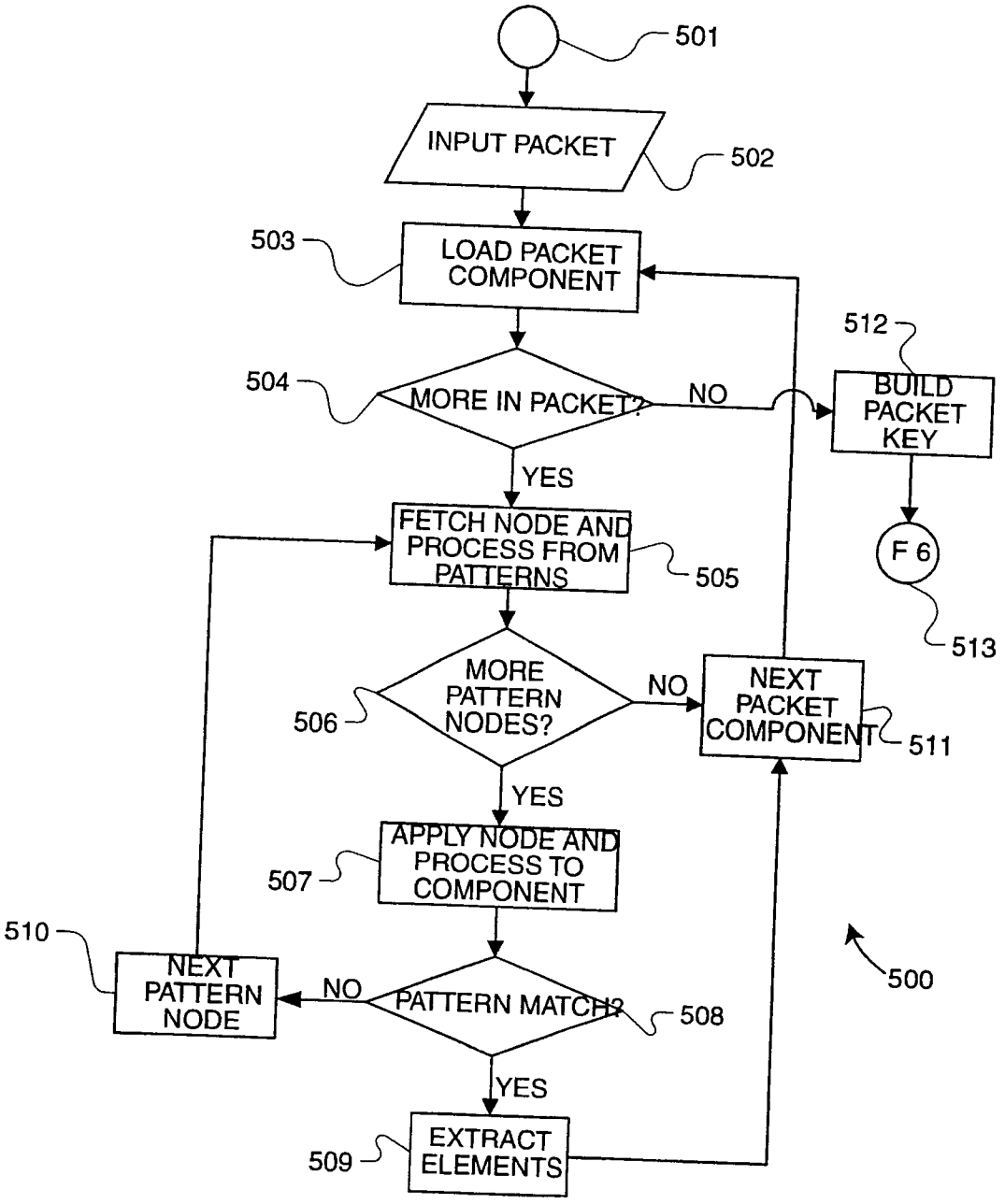

501
INPUT PACKET
502
503
LOAD PACKET COMPONENT
504
MORE IN PACKET?
NO
512
BUILD PACKET KEY
F 6
513
YES
FETCH NODE AND PROCESS FROM PATTERNS
505
MORE PATTERN NODES?
506
NO
NEXT PACKET COMPONENT
511
YES
APPLY NODE AND PROCESS TO COMPONENT
507
510
NEXT PATTERN NODE
NO
PATTERN MATCH?
508
500
YES
EXTRACT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

FIG. 5

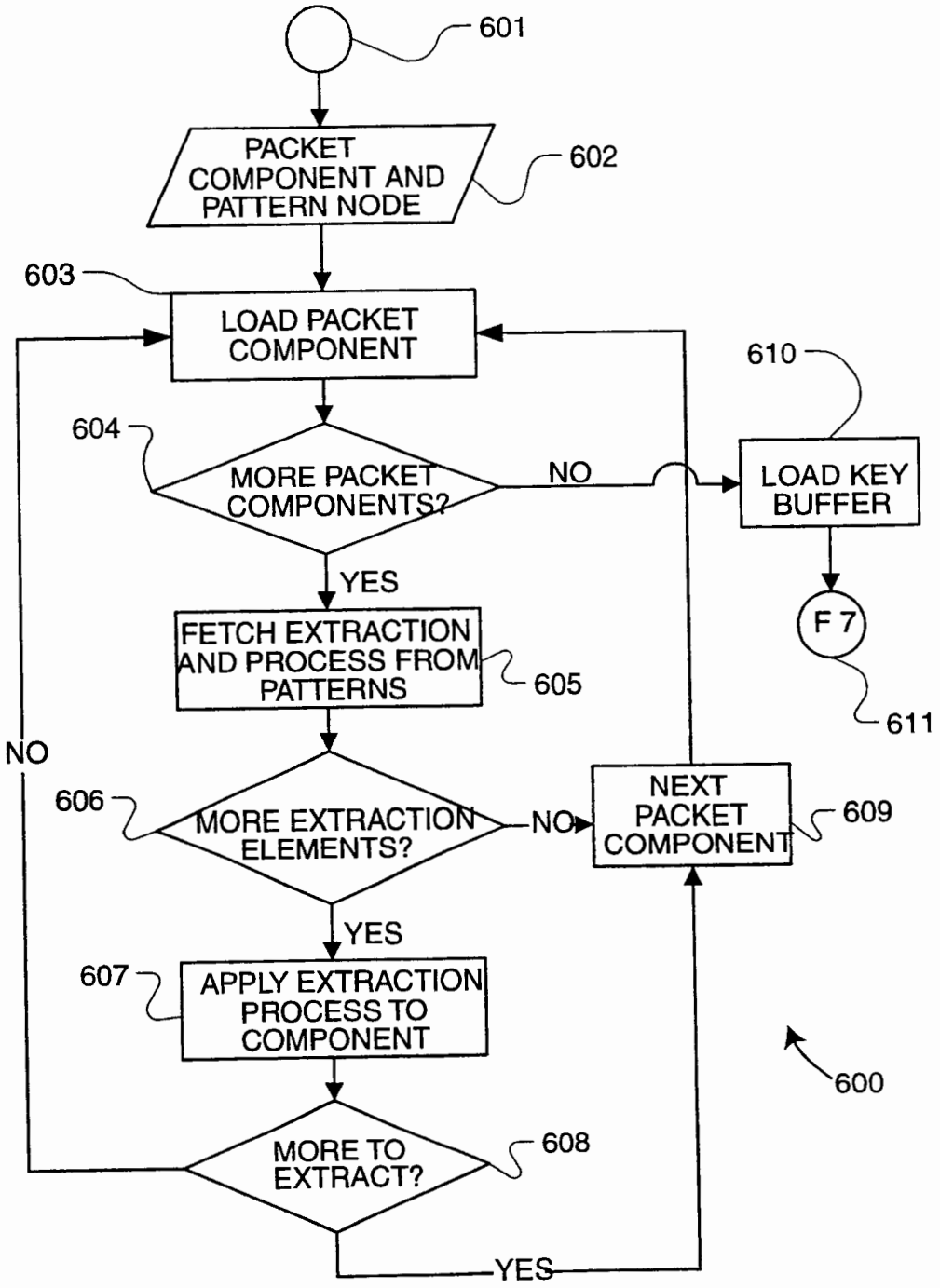

601
PACKET COMPONENT AND PATTERN NODE
602
603
LOAD PACKET COMPONENT
610
604
MORE PACKET COMPONENTS?
NO
LOAD KEY BUFFER
YES
F 7
FETCH EXTRACTION AND PROCESS FROM PATTERNS
605
611
NO
606
MORE EXTRACTION ELEMENTS?
NO
NEXT PACKET COMPONENT
609
YES
607
APPLY EXTRACTION PROCESS TO COMPONENT
600
MORE TO EXTRACT?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

FIG. 6

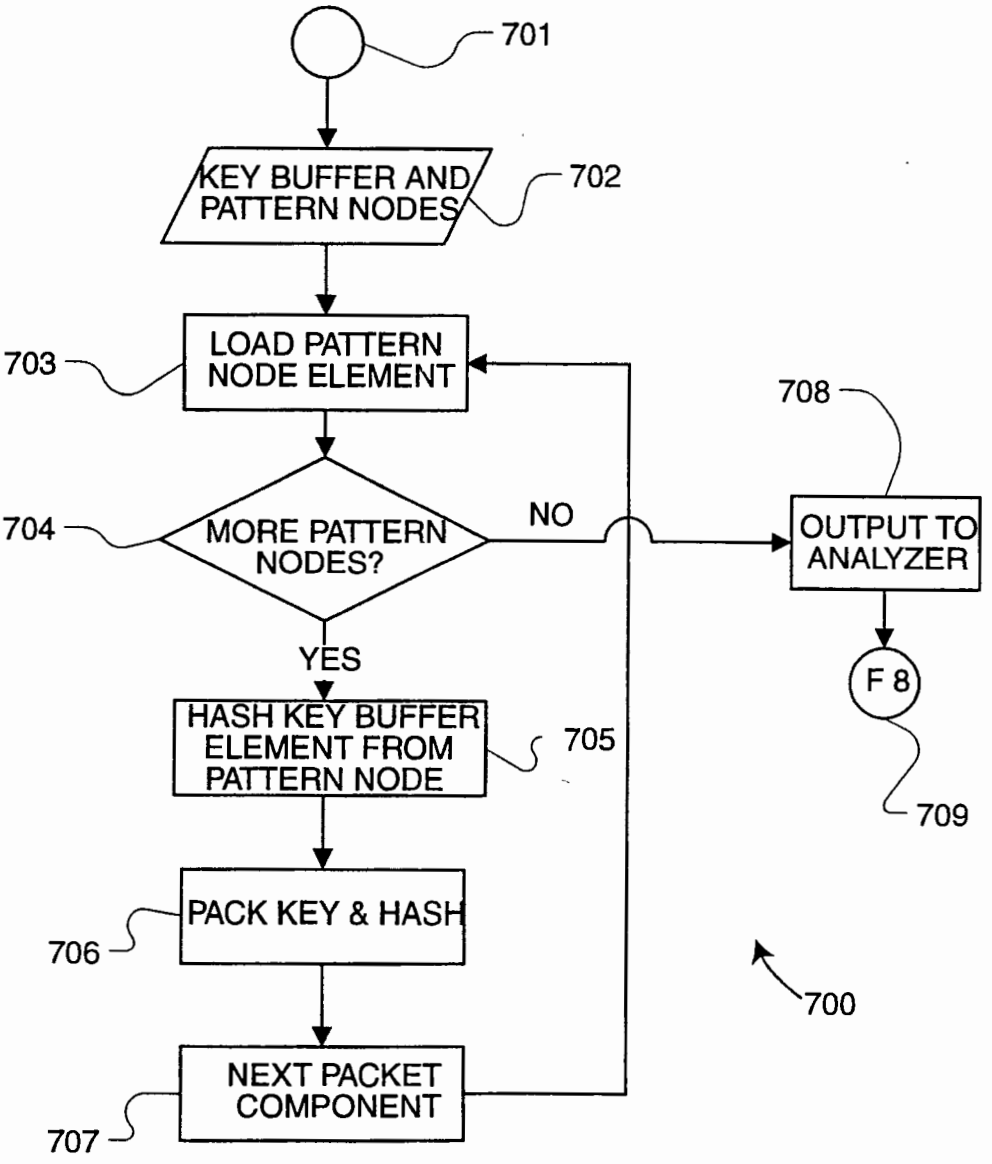

701
KEY BUFFER AND PATTERN NODES
702
LOAD PATTERN NODE ELEMENT
703
MORE PATTERN NODES?
704
NO
YES
OUTPUT TO ANALYZER
708
F 8
709
HASH KEY BUFFER ELEMENT FROM PATTERN NODE
705
PACK KEY & HASH
706
NEXT PACKET COMPONENT
707
700

FIG. 7

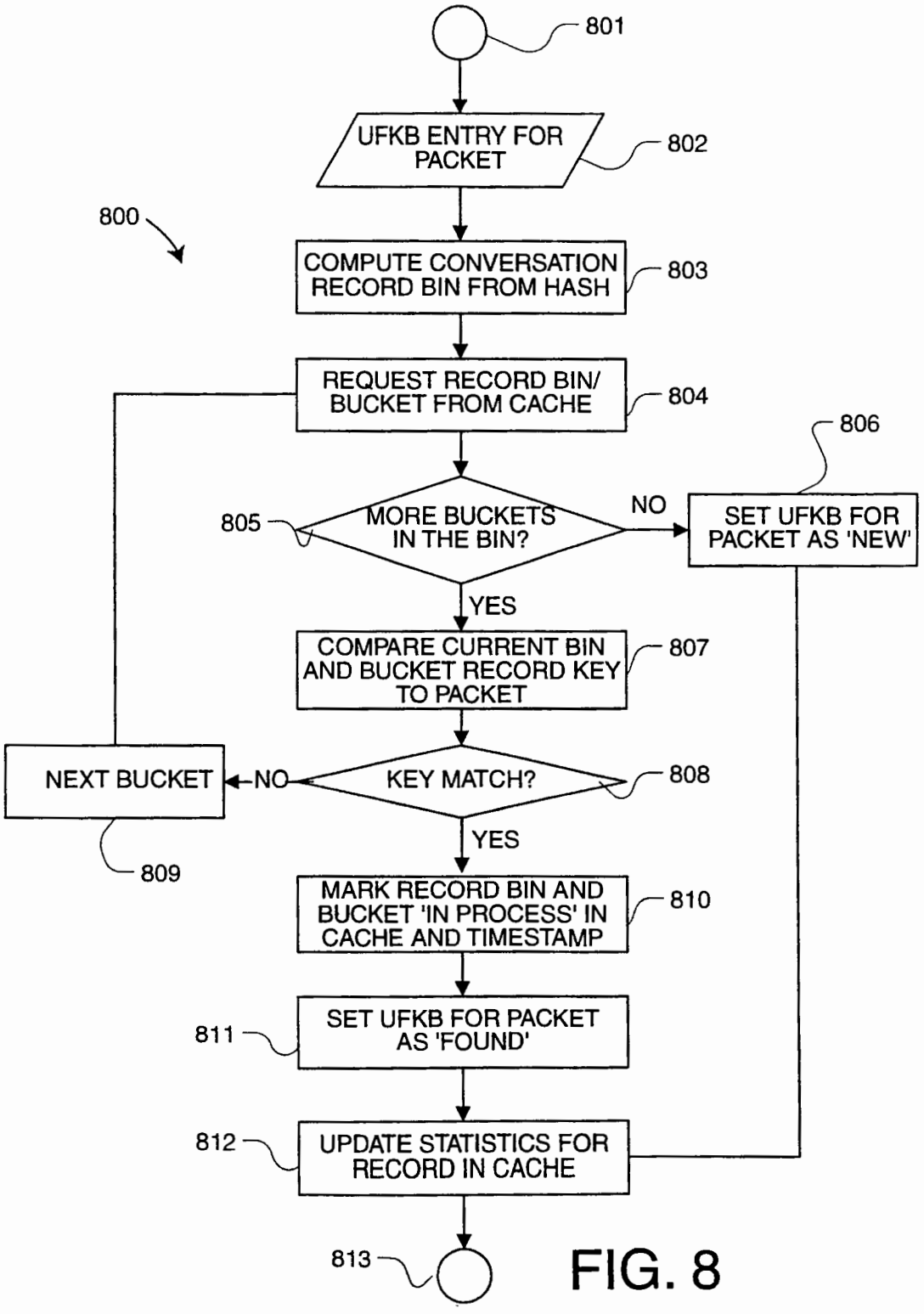

801
UFKB ENTRY FOR PACKET
802
800
COMPUTE CONVERSATION RECORD BIN FROM HASH
803
REQUEST RECORD BIN/ BUCKET FROM CACHE
804
806
805
MORE BUCKETS IN THE BIN?
NO
SET UFKB FOR PACKET AS 'NEW'
YES
COMPARE CURRENT BIN AND BUCKET RECORD KEY TO PACKET
807
NEXT BUCKET
NO
KEY MATCH?
808
YES
809
MARK RECORD BIN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

FIG. 8

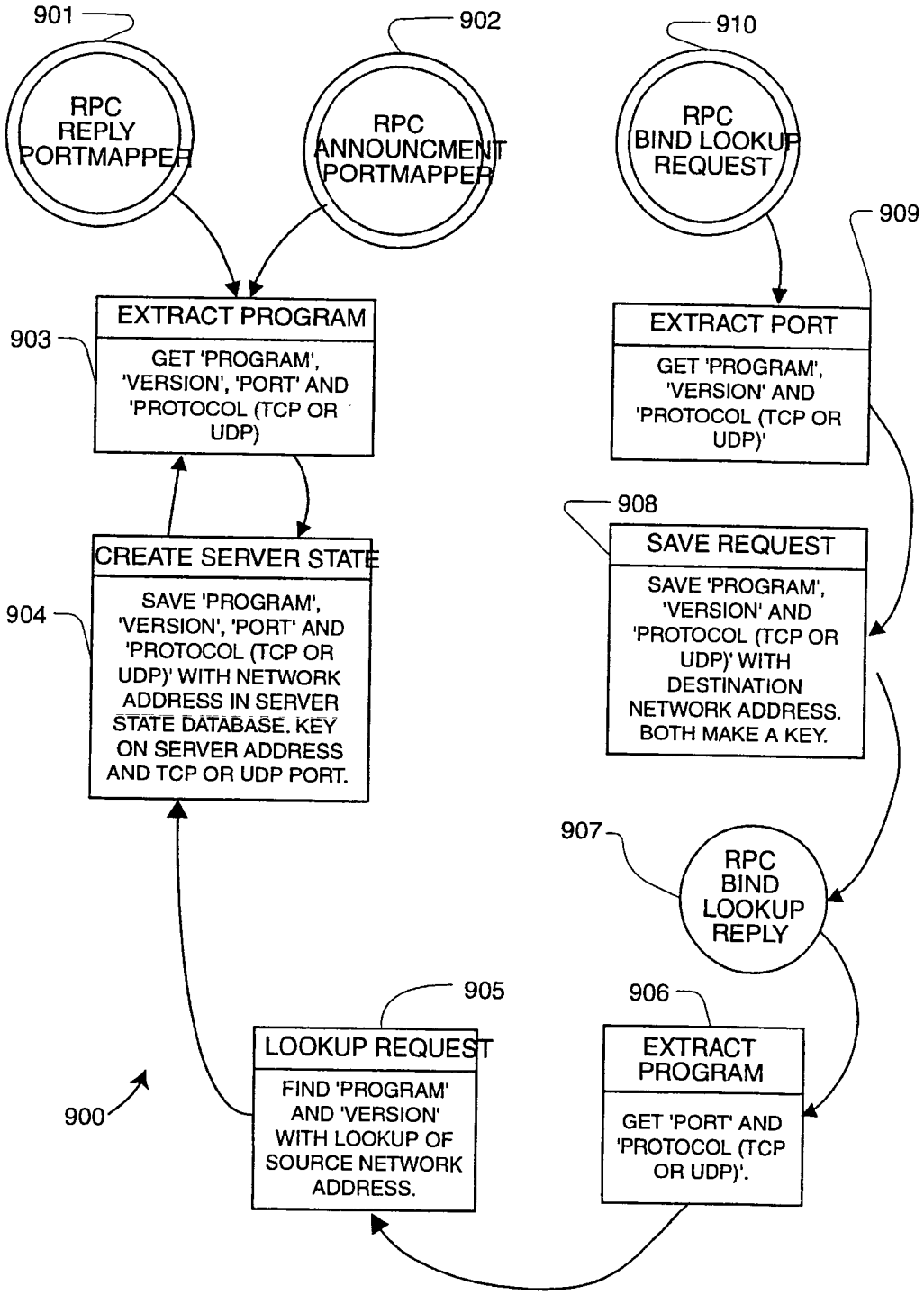

901
RPC REPLY PORTMAPPER
902
RPC ANNOUNCMENT PORTMAPPER
910
RPC BIND LOOKUP REQUEST
909
EXTRACT PROGRAM
GET 'PROGRAM', 'VERSION', 'PORT' AND 'PROTOCOL (TCP OR UDP)
903
EXTRACT PORT
GET 'PROGRAM', 'VERSION' AND 'PROTOCOL (TCP OR UDP)'
908
SAVE REQUEST
SAVE 'PROGRAM', 'VERSION' AND 'PROTOCOL (TCP OR UDP)'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

FIG. 9

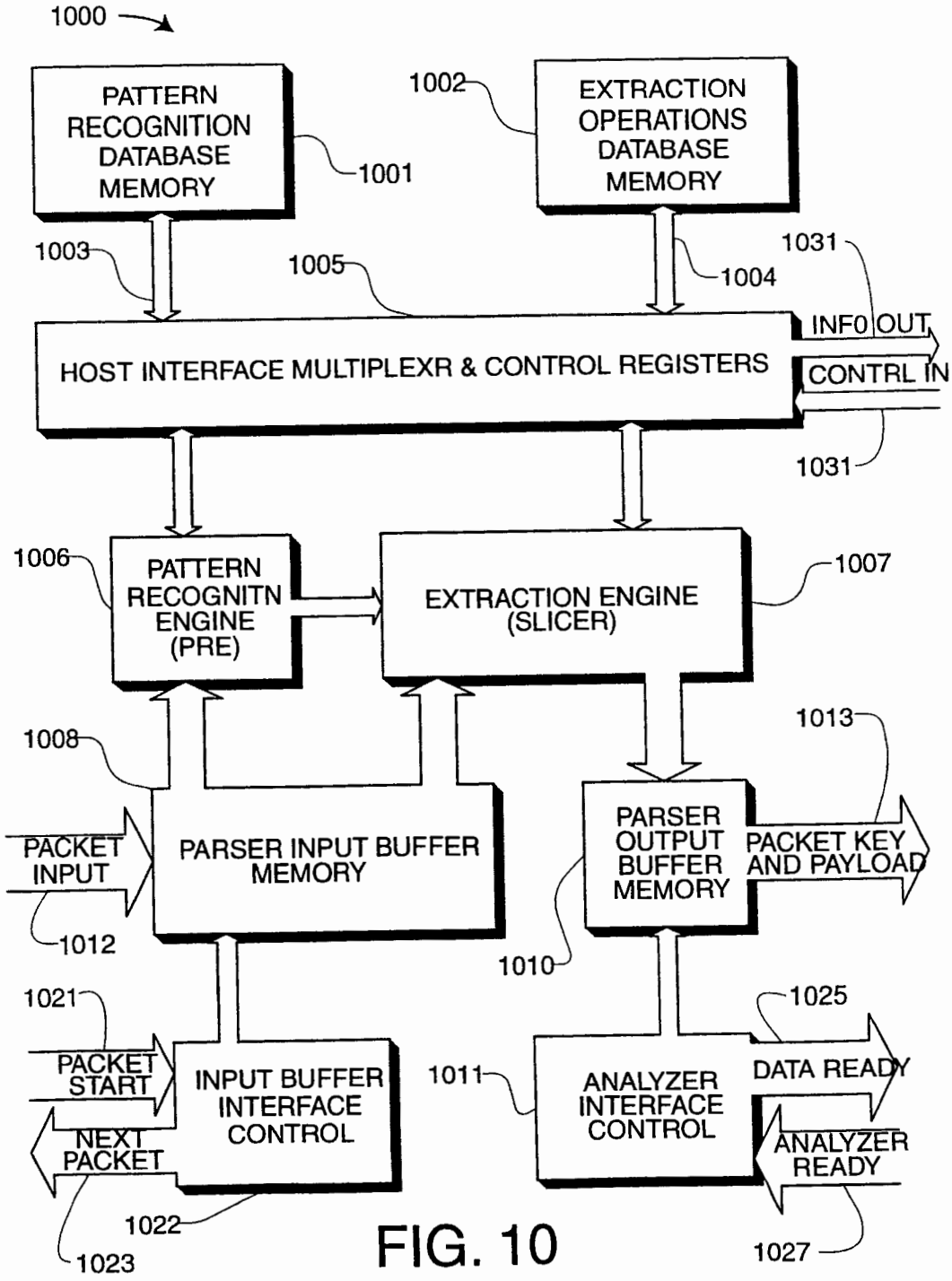

1000
PATTERN RECOGNITION DATABASE MEMORY
1001
1002
EXTRACTION OPERATIONS DATABASE MEMORY
1003
1004
1005
1031
INFO OUT
CONTRL IN
HOST INTERFACE MULTIPLEXR & CONTROL REGISTERS
1031
1006
PATTERN RECOGNITN ENGINE (PRE)
EXTRACTION ENGINE (SLICER)
1007
1008
1013
PACKET INPUT
PARSER INPUT BUFFER MEMORY
PARSER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

FIG. 10

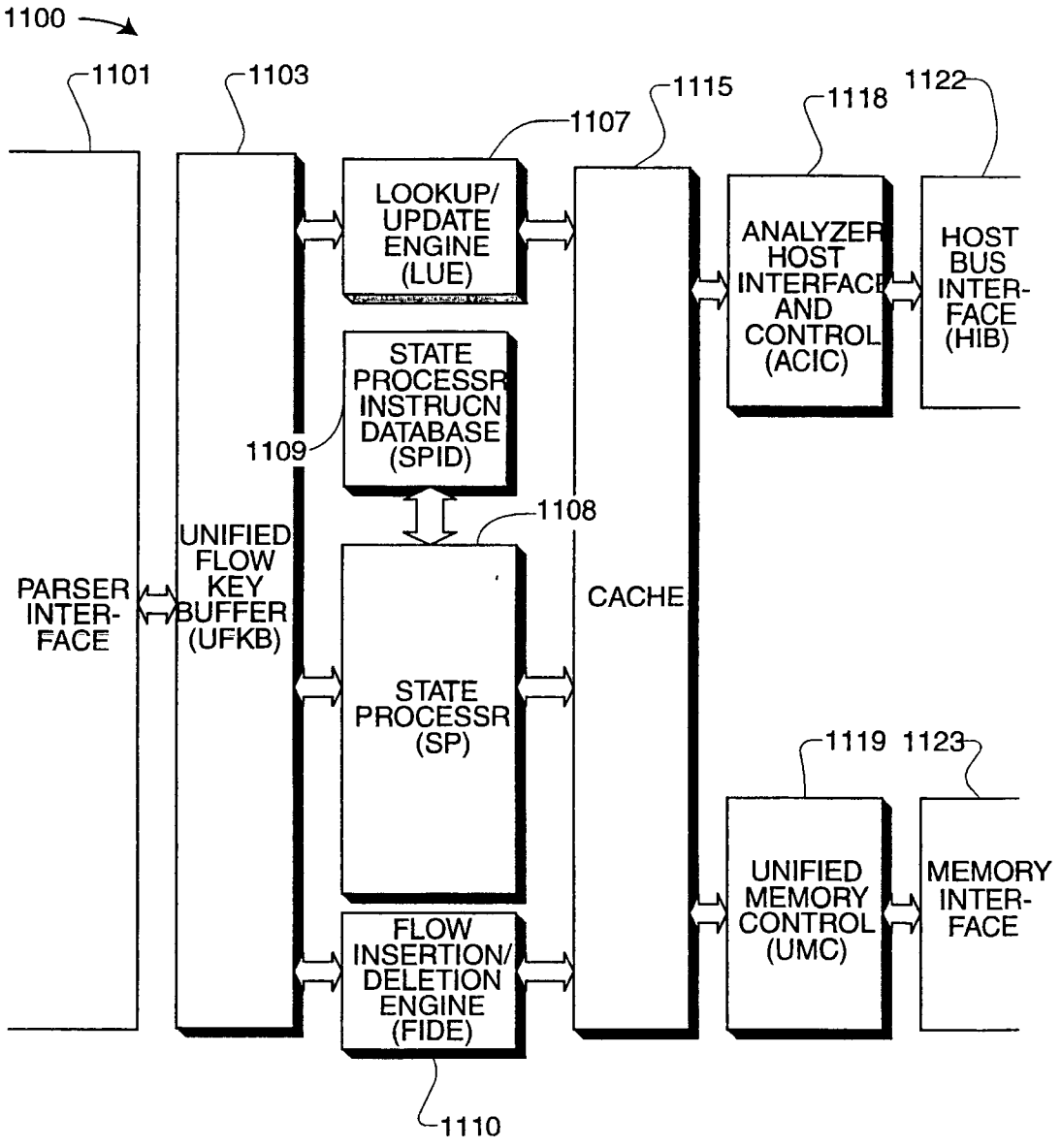

1100
1101
1103
1107
1115
1118
1122
LOOKUP/ UPDATE ENGINE (LUE)
ANALYZER HOST INTERFACE AND CONTROL (ACIC)
HOST BUS INTER-FACE (HIB)
STATE PROCESSR INSTRUCN DATABASE (SPID)
1109
1108
PARSER INTER-FACE
UNIFIED FLOW KEY BUFFER (UFKB)
STATE PROCESSR (SP)
CACHE
1119
1123
UNIFIED MEMORY CONTROL (UMC)
MEMORY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

FIG. 11

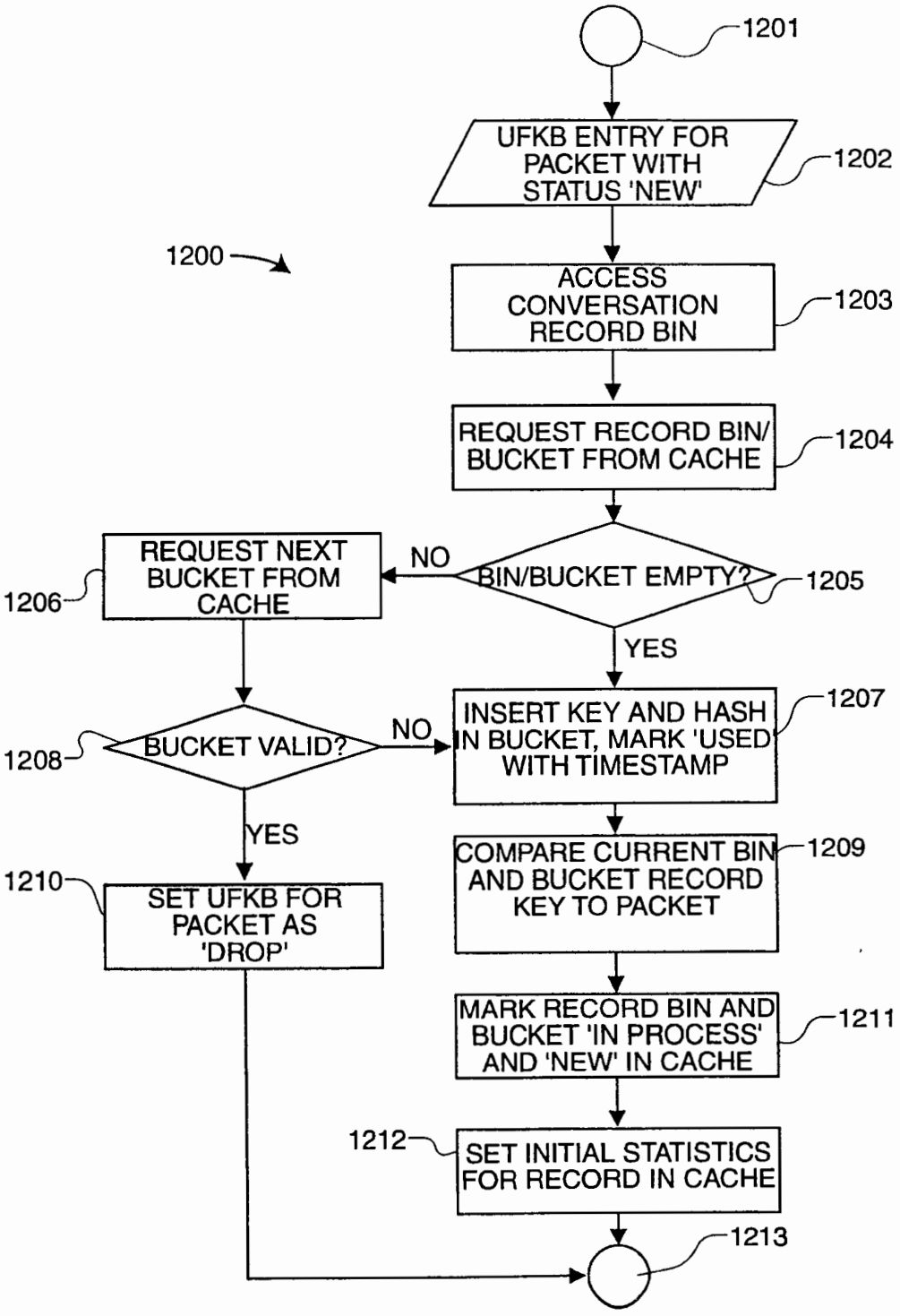

1201
UFKB ENTRY FOR PACKET WITH STATUS 'NEW'
1202
1200
ACCESS CONVERSATION RECORD BIN
1203
REQUEST RECORD BIN/ BUCKET FROM CACHE
1204
REQUEST NEXT BUCKET FROM CACHE
NO
BIN/BUCKET EMPTY?
1205
1206
YES
INSERT KEY AND HASH IN BUCKET, MARK 'USED' WITH TIMESTAMP
1207
BUCKET VALID?
NO
1208
YES
COMPARE CURRENT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

FIG. 12

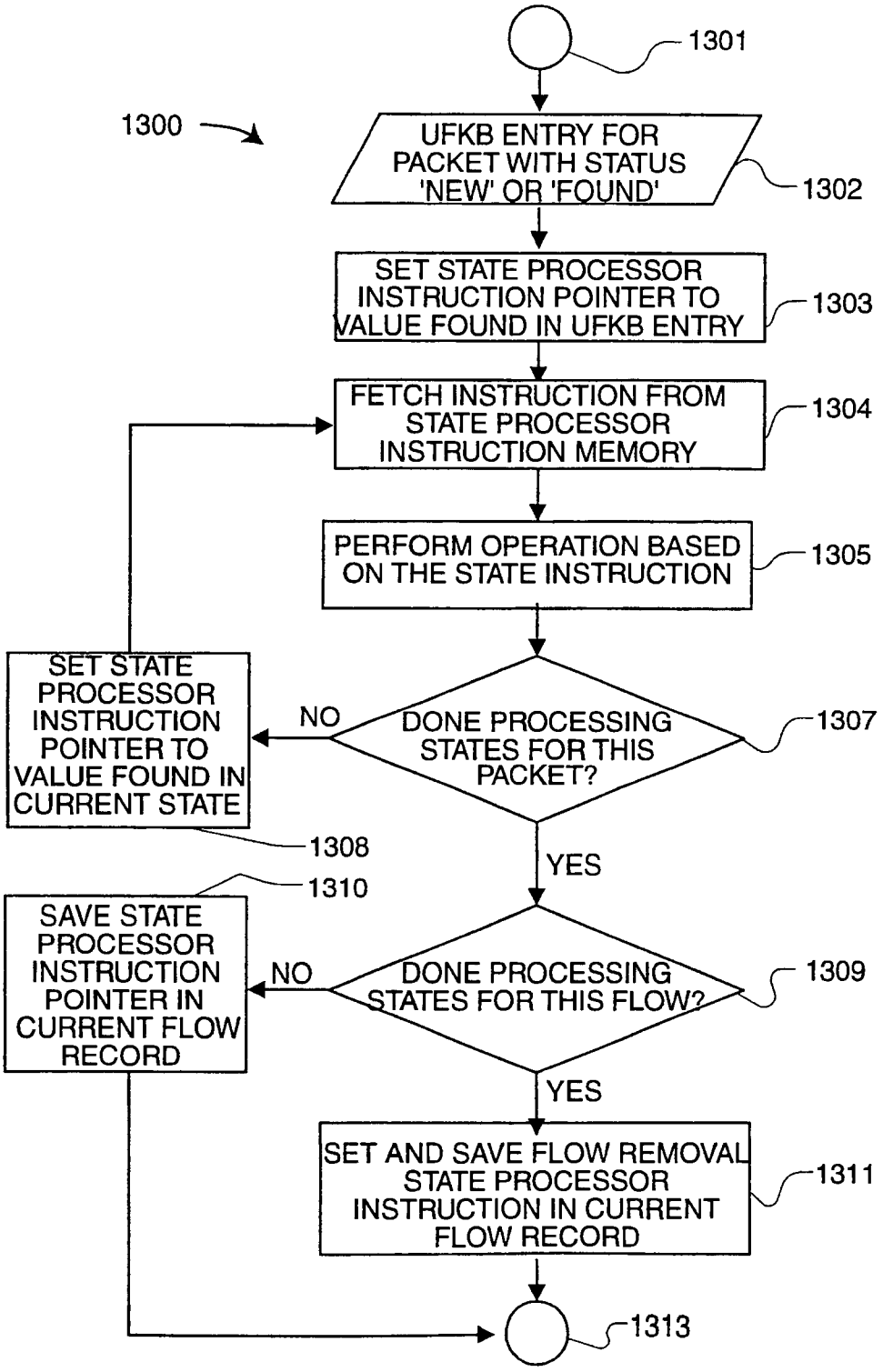

1301
1300
UFKB ENTRY FOR PACKET WITH STATUS 'NEW' OR 'FOUND'
1302
SET STATE PROCESSOR INSTRUCTION POINTER TO VALUE FOUND IN UFKB ENTRY
1303
FETCH INSTRUCTION FROM STATE PROCESSOR INSTRUCTION MEMORY
1304
PERFORM OPERATION BASED ON THE STATE INSTRUCTION
1305
DONE PROCESSING STATES FOR THIS PACKET?
1307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

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

FIG. 14

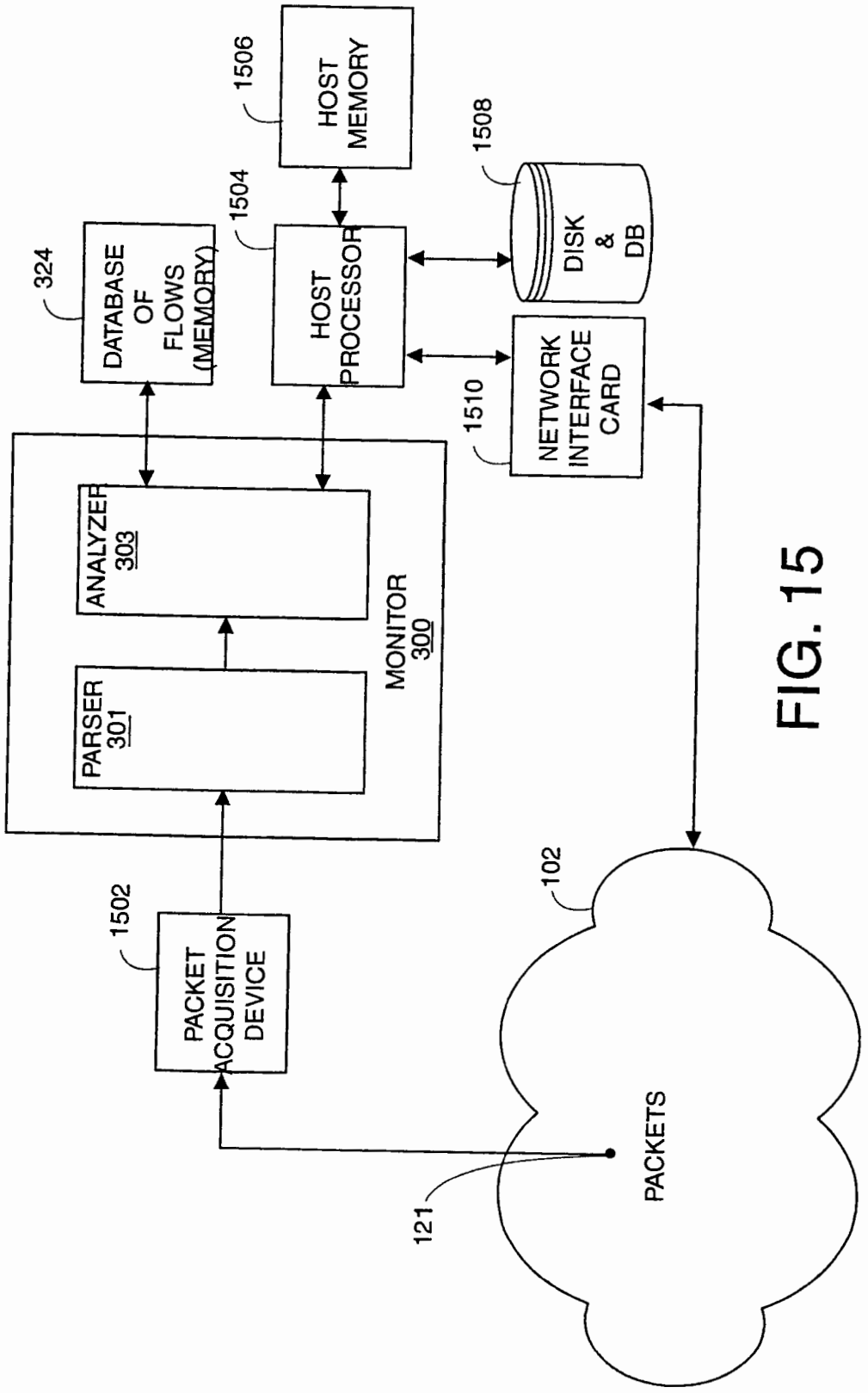

HOST MEMORY
1506
1508
DISK & DB
1504
HOST PROCESSOR
324
DATABASE OF FLOWS (MEMORY)
NETWORK INTERFACE CARD
1510
ANALYZER 303
MONITOR 300
PARSER 301
102
1502
PACKET ACQUISITION DEVICE
PACKETS
121

FIG. 15

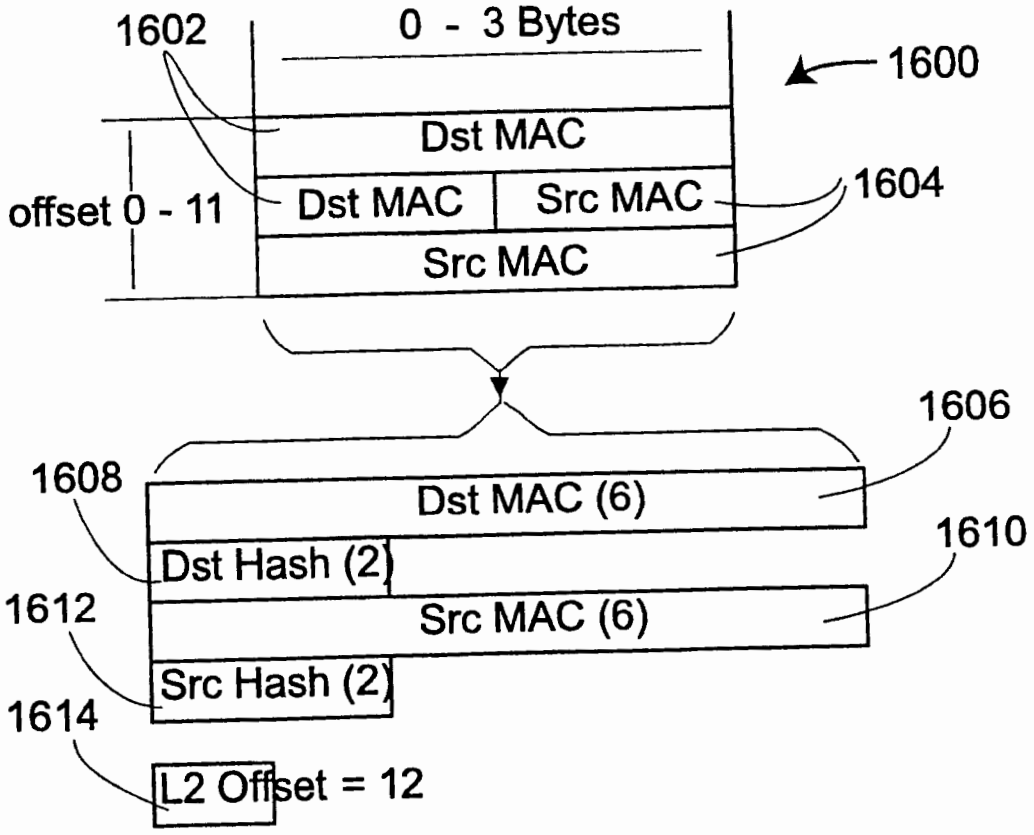

1602
0 - 3 Bytes
1600
Dst MAC
offset 0 - 11
Dst MAC
Src MAC
1604
Src MAC
1606
1608
Dst MAC (6)
Dst Hash (2)
1610
1612
Src MAC (6)
Src Hash (2)
1614
L2 Offset = 12

FIG. 16

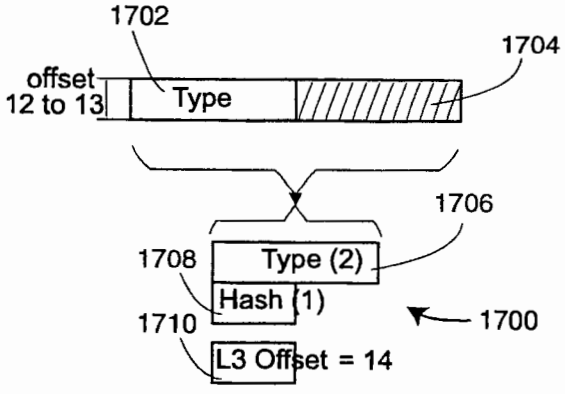


FIG. 17A

1712

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

Apple = 0x080007

* L3 Decoding
# L5 Decoding

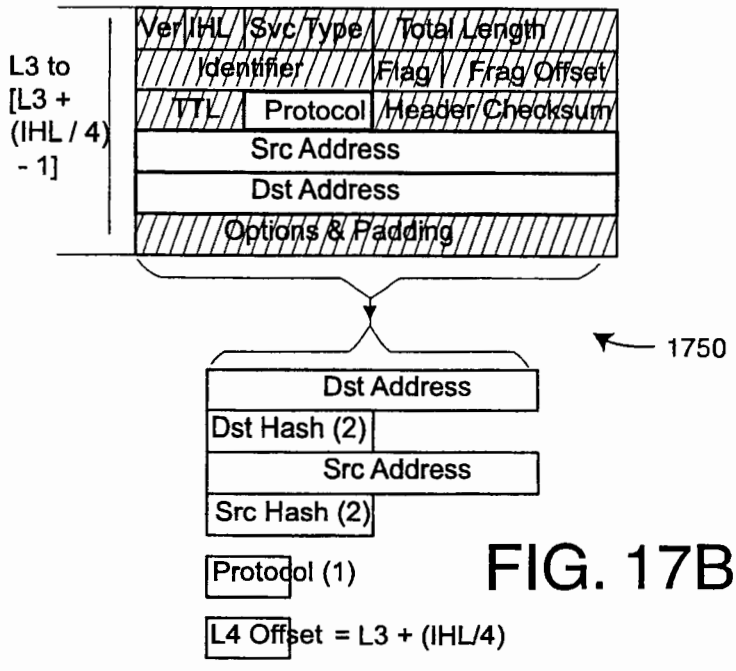


FIG. 17B

1752

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

* L4 Decoding
# L3 Re-Decoding

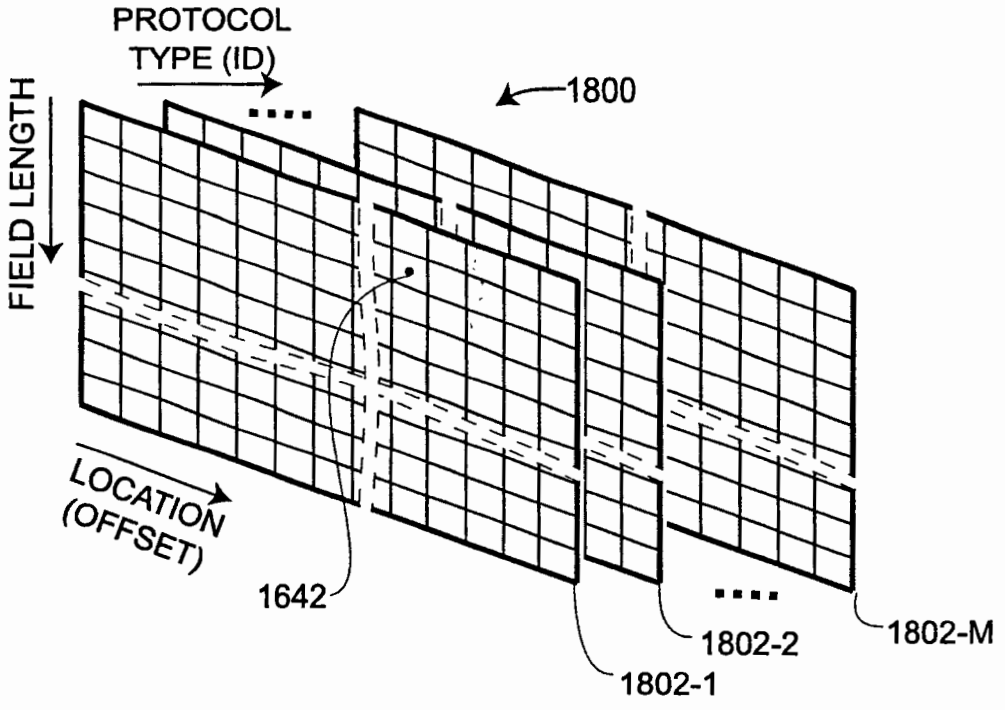


FIG. 18A

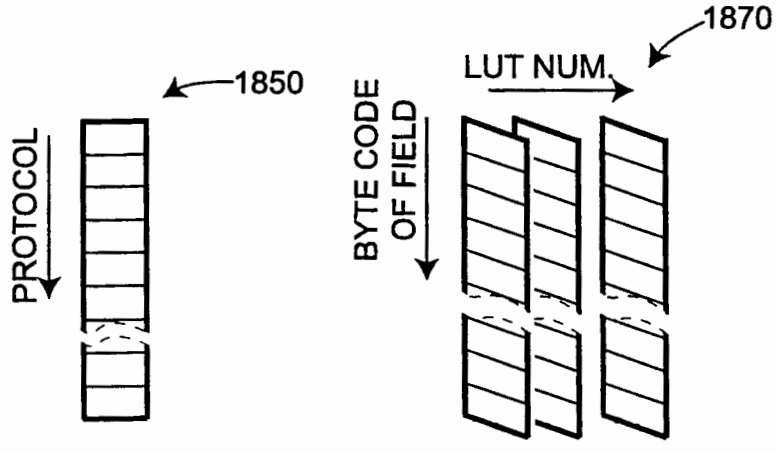


FIG. 18B

RE-USING INFORMATION FROM DATA TRANSACTIONS FOR MAINTAINING STATISTICS IN NETWORK MONITORING

CROSS-REFERENCE TO RELATED APPLICATION

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

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

U.S. patent application Ser. No. 09/608,237 for METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK, to inventors Dietz, et al., filed Jun. 30, 2000, and incorporated herein by reference.

U.S. patent application Ser. No. 09/609,179 for PROCESSING PROTOCOL SPECIFIC INFORMATION IN PACKETS SPECIFIED BY A PROTOCOL DESCRIPTION LANGUAGE, to inventors Koppenhaver, et al., filed Jun. 30, 2000, and incorporated herein by reference.

U.S. patent application Ser. No. 09/608,266 for ASSOCIATIVE CACHE STRUCTURE FOR LOOKUPS AND UPDATES OF FLOW RECORDS IN A NETWORK MONITOR, to inventors Sarkissian, et al., filed Jun. 30, 2000, and incorporated herein by reference.

U.S. patent application Ser. No. 09/608,267 for STATE PROCESSOR FOR PATTERN MATCHING IN A NETWORK MONITOR DEVICE, to inventors Sarkissian, et al., filed Jun. 30, 2000, and incorporated herein by reference.

FIELD OF INVENTION

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

COPYRIGHT NOTICE

BACKGROUND

There has long been a need for network activity monitors. This need has become especially acute, however, given the recent popularity of the Internet and other interconnected networks. In particular, there is a need for a real-time network monitor that can provide details as to the application programs being used. Such a monitor should enable non-intrusive, remote detection, characterization, analysis, and capture of all information passing through any point on the network (i.e., of all packets and packet streams passing through any location in the network). Not only should all the packets be detected and analyzed, but for each of these packets the network monitor should determine the protocol (e.g., http, ftp, H.323, VPN, etc.), the application/use within the protocol (e.g., voice, video, data, real-time data, etc.),

and an end user's pattern of use within each application or the application context (e.g., options selected, service delivered, duration, time of day, data requested, etc.). Also, the network monitor should not be reliant upon server resident information such as log files. Rather, it should allow a user such as a network administrator or an Internet service provider (ISP) the means to measure and analyze network activity objectively; to customize the type of data that is collected and analyzed; to undertake real time analysis; and to receive timely notificatior of network problems.

Related and incorporated by reference U.S. patent application Ser. No. 09/607,237 for METHOD AND APPARATUS FOR MONITORING TRAFFIC IN A NETWORK, to inventors Dietz, et al, describes a network monitor that includes carrying out protocol specific operations on individual packets including extracting information from header fields in the packet to use for building a signature for identifying the conversational flow of the packet and for recognizing future packets as belonging to a previously encountered flow. A parser subsystem includes a parser for recognizing different patterns in the packet that identify the protocols used. For each protocol recognized, a slicer extracts important packet elements from the packet. These form a signature (i.e., key) for the packet. The slicer also preferably generates a hash for rapidly identifying a flow that may have this signature from a database of known flows.

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

Each flow-entry includes one or more statistical measures, e.g., the packet count related to the flow, the time of arrival of a packet, the time differential.

In the preferred hardware embodiment, each of the LUE, state processor, and FIDE operate independently from the other two engines. The state processor performs one or more operations specific to the state of the flow.

It is advantageous to collect statistics on packets passing through a point in a network rather than to simply count each and every packet. By maintaining statistical measures in the flow-entries related to a conversational flow, embodiments of the present invention enable specific metrics to be collected in real-time that otherwise would not be possible. For example, it is desirable to maintain metrics related to bi-directional conversations based on the entire flow for each exchange in the conversation. By maintaining the state of flow, embodiments of the present invention also enable certain metrics related to the states of flows to be determined.

Most prior-art network traffic monitors that use statistical metrics collect only end-point and end-of-session related statistics. Examples of such 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 complete sequence of packets of a flow or to several disjointed sequences of the same flow in a network.

Time based metrics on application data packets are important. Such metrics could be determined if all the timestamps and related data could be stored and forwarded for later analysis. However when faced with thousands or millions 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.

Thus there is a need for maintaining and reporting time-base metrics from statistical measures accumulated from packets in a flow.

Network data is properly modeled as a population and not a sample. Thus, all the data needs to be processed. Because of the nature of application protocols, just sampling some of the packets may not give good measured related to flows. 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 valid data to be lost.

Thus there is also a need for maintaining and reporting time-base metrics from statistical measures accumulated from every packet in a flow.

There also is a need to determine metrics 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 described herein, because the state processor can search for specific data payloads, embodiments of 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 in general is more desirable than metrics related to single packets when evaluating the performance of a system in a network.

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

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 may 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 billing and charge-back for specific traffic flow and events with the traffic flows. Yet other implementations could be programmed to provide metrics useful for troubleshooting and capacity planning and related directly to a focused application and service.

SUMMARY

Another aspect of the invention is determining quality of service metrics based on each and every packet. A method of and monitor apparatus for analyzing a flow of packets passing through a connection point on a computer network are disclosed that may include such quality of service metrics. The method includes receiving a packet from a packet acquisition device, and looking up a flow-entry database containing flow-entries for previously encountered conversational flows. The looking up to determine if the received packet is of an existing flow. Each and every packet is processed. If the packet is of an existing flow, the method updates the flow-entry of the existing flow, including storing one or more statistical measures kept in the flow-entry. If the packet is of a new flow, the method stores a new flow-entry for the new flow in the flow-entry database, including storing one or more statistical measures kept in the flow-entry. The statistical measures are used to determine metrics related to the flow. The metrics may be base metrics from which quality of service metrics are determined, or may be the quality of service metrics.

BRIEF DESCRIPTION OF THE DRAWINGS

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

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

Operation in a Network

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

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

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

ISO MODEL

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

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

In so communication arts, the term "frame" generally refers to encapsulated data at OSI layer 2, including a destination address, control bits for flow control, the data or

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

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

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

In real-world uses of the monitor **108**, the number of packets on the network **102** passing by the monitor **108**'s connection point can exceed a million per second. Consequently, the monitor has very little time available to analyze and type each packet and identify and maintain the state of the flows passing through the connection point. The monitor **108** therefore masks out all the unimportant parts of each packet that will not contribute to its classification. However, the parts to mask-out will change with each packet depending on which flow it belongs to and depending on the state of the flow.

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

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

It may be that the application is now recognized, or recognition may need to proceed to a third level of analysis using the second level signature. For each packet, therefore, the monitor parses the packet and generates a signature to determine if this signature identified a previously encountered flow, or shall be used to recognize future packets belonging to the same conversational flow. In real time, the packet is further analyzed in the context of the sequence of previously encountered packets (the state), and of the possible future sequences such a past sequence may generate in conversational flows associated with different applications. A new signature for recognizing future packets may also be generated. This process of analysis continues until the applications are identified. The last generated signature may then be used to efficiently recognize future packets associated with the same conversational flow. Such an arrangement makes it possible for the monitor **108** to cope with millions of packets per second that must be inspected.

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

The Network Monitor

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

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

Aspects shown here include: (1) the initialization of the monitor to generate what operations need to occur on packets of different types—accomplished by compiler and optimizer **310**, (2) the processing—parsing and extraction of selected portions—of packets to generate an

signature—accomplished by parser subsystem **301**, and (3) the analysis of the packets—accomplished by analyzer **303**.

The purpose of compiler and optimizer **310** is to provide protocol specific information to parser subsystem **301** and to analyzer subsystem **303**. The initialization occurs prior to operation of the monitor, and only needs to re-occur when new protocols are to be added.

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

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

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

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

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

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

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

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

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

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

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

The parser record enters a buffer called the unified flow key buffer (UFKB). The UFKB stores the data on flows in a data structure that is similar to the parser record, but that includes a field that can be modified. In particular, one or the

UFKB record fields stores the packet sequence number, and another is filled with state information in the form of a program counter for a state processor that implements state processing **328**.

The determination (**316**) of whether a record with the same signature already exists is carried out by a lookup engine (LUE) that obtains new UFKB records and uses the hash in the UFKB record to lookup if there is a matching known flow. In the particular embodiment, the database of known flows **324** is in an external memory. A cache is associated with the database **324**. A lookup by the LUE for a known record is carried out by accessing the cache using the hash, and if the entry is not already present in the cache, the entry is looked up (again using the hash) in the external memory.

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

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

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

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

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

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

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

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

Thus, state processor **328** applies the first state operation to the packet for this particular flow-entry. A process **330** decides if more operations need to be performed for this state. If so, the analyzer continues looping between block **330** and **328** applying additional state operations to this particular packet until all those operations are completed—that is, there are no more operations for this packet in this state. A process **332** decides if there are further states to be analyzed for this type of flow according to the state of the flow and the protocol, in order to fully characterize the flow. If not, the conversational flow has now been fully characterized and a process **334** finalizes the classification of the conversational flow for the flow.

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

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

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

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

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

Thus, the embodiment of present invention shown in FIG. **3** automatically maintains flow-entries, which in one aspect includes storing states. The monitor of FIG. **3** also generates characteristic parts of packets—the signatures—that can be used to recognize flows. The flow-entries may be identified and accessed by their signatures. Once a packet is identified to be from a known flow, the state of the flow is known and this knowledge enables state transition analysis to be performed in real time for each different protocol and application. In a complex analysis, state transitions are traversed as more and more packets are examined. Future packets that are part of the same conversational flow have their state analysis continued from a previously achieved state. When enough packets related to an application of interest have been processed, a final recognition state is ultimately reached, i.e., a set of states has been traversed by state analysis to completely characterize the conversational flow. The signature for that final state enables each new incoming packet of the same conversational flow to be individually recognized in real time.

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

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

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

Detailed operation

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

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

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

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

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

An alternate embodiment of the data structure used in database **308** is illustrated in FIG. **18**B. Thus, like the 3-D structure of FIG. **18**A, the data structure permits rapid searches to be performed by the pattern recognition process **304** by indexing locations in a memory rather than performing address link computations. In this alternate embodiment, the PRD **308** includes two parts, a single protocol table **1850** (PT) which has an entry for each protocol known for the monitor, and a series of Look Up Tables **1870** (LUT's) that are used to identify known protocols and their children. The protocol table includes the parameters needed by the pattern analysis and recognition process **304** (implemented by PRE **1006**) to evaluate the header information in the packet that is associated with that protocol, and parameters needed by extraction process **306** (implemented by slicer **1007**) to process the packet header. When there are children, the PT describes which bytes in the header to evaluate to determine the child protocol. In particular, each PT entry contains the header length, an offset to the child, a slicer command, and some flags.

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

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

A compiler **403** compiles the descriptions. The set of packet parse-and-extract operations **406** is generated (**404**), and a set of packet state instructions and operations **407** is generated (**405**) in the form of instructions for the state processor that implements state processing process **328**. Data files for each type of application and protocol to be

recognized by the analyzer are downloaded from the pattern, parse, and extraction database **406** into the memory systems of the parser and extraction engines. (See the parsing process **500** description and FIG. **5**; the extraction process **600** description and FIG. **6**; and the parsing subsystem hardware description and FIG. **10**). Data files for each type of application and protocol to be recognized by the analyzer are also downloaded from the state-processor instruction database **407** into the state processor. (see the state processor **1108** description and FIG. **11**.).

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

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

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

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

FIG. **5** shows a flowchart of how actual parser subsystem **301** functions. Starting at **501**, the packet **302** is input to the packet buffer in step **502**. Step **503** loads the next (initially the first) packet component from the packet **302**. The packet components are extracted from each packet **302** one element at a time. A check is made (**504**) to determine if the load-packet-component operation **503** succeeded, indicating that there was more in the packet to process. If not, indicating all components have been loaded, the parser subsystem **301** builds the packet signature (**512**)—the next stage (FIG. **6**).

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

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

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

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

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

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

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

The **703** to **707** loop continues until there are no more patterns of elements left (test **704**). Once all the patterns of

elements have been hashed, processes **304**, **306** and **312** of parser subsystem **301** are complete. Parser subsystem **301** has generated the signature used by the analyzer subsystem **303**.

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

FIG. **8** is a flow diagram describing the operation of the lookup/update engine (LUE) that implements lookup operation **314**. The process starts at **801** from FIG. **7** with the parser record that includes a signature, the hash and at least parts of the payload. In **802** those elements are shown in the form of a UFKB-entry in the buffer. The LUE, the lookup engine **314** computes a "record bin number" from the hash for a flow-entry. A bin herein may have one or more "buckets" each containing a flow-entry. The preferred embodiment has four buckets per bin.

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

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

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

To provide for such analyses, the lookup/update engine updates one or more counters that are part of the flow-entry (in the cache) in step **812**. The process exits at **813**. In our embodiment, the counters include the total packets of the flow, the time, and a differential time from the last timestamp to the present timestamp.

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

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

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

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

The Hardware System

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

FIG. **10** is a description of the parsing subsystem (**301**, shown here as subsystem **1000**) as implemented in hardware. Memory **1001** is the pattern recognition database memory, in which the patterns that are going to be analyzed are stored. Memory **1002** is the extraction-operation database memory, in which the extraction instructions are stored. Both **1001** and **1002** correspond to internal data structure **308** of FIG. **3**. Typically, the system is initialized from a microprocessor (not shown) at which time these memories are loaded through a host interface multiplexor and control register **1005** via the internal buses **1003** and **1004**. Note that the contents of **1001** and **1002** are preferably obtained by compiling process **310** of FIG. **3**.

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

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

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

The PRE **1006** includes of a comparison engine. The comparison engine has a first stage that checks the protocol type field to determine if it is an 802.3 packet and the field should be treated as a length. If it is not a length, the protocol is checked in a second stage. The first stage is the only protocol level that is not programmable. The second stage has two full sixteen bit content addressable memories (CAMs) defined for future protocol additions.

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

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

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

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

Once all the selected parts of the packet used to form the signature are extracted, the hash is loaded into parser output buffer memory **1010**. Any additional payload from the packet that is required for further analysis is also included. The parser output memory **1010** is interfaced with the analyzer subsystem by analyzer interface control **1011**. Once all the information of a packet is in the parser output buffer memory **1010**, a data ready signal **1025** is asserted by analyzer interface control. The data from the parser subsystem **1000** is moved to the analyzer subsystem via **1013** when an analyzer ready signal **1027** is asserted.

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

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

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

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

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

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

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

The cache subsystem **1115** is an associative cache that includes a set of content addressable memory cells (CAMs) each including an address portion and a pointer portion pointing to the cache memory (e.g., RAM) containing the

cached flow-entries. The CAMs are arranged as a stack ordered from a top CAM to a bottom CAM. The bottom CAM's pointer points to the least recently used (LRU) cache memory entry. Whenever there is a cache miss, the contents of cache memory pointed to by the bottom CAM are replaced by the flow-entry from the flow-entry database **324**. This now becomes the most recently used entry, so the contents of the bottom CAM are moved to the top CAM and all CAM contents are shifted down. Thus, the cache is an associative cache with a true LRU replacement policy.

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

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

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

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

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

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

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

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

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

FIG. **13** describes the operation of the state processor **1108**. The state processor is entered at **1301** with a unified flow key buffer entry to be processed. The UFKB-entry is new or corresponding to a found flow-entry. This UFKB-entry is retrieved from unified flow key buffer **1103** in **1301**. In **1303**, the protocol identifier for the UFKB-entry is used to set the state processor's instruction counter. The state processor **1108** starts the process by using the last protocol recognized by the parser subsystem **301** as an offset into a jump table. The jump table takes us to the instructions to use for that protocol. Most instructions test something in the unified flow key buffer or the flow-entry if it exists. The state processor **1108** may have to test bits, do comparisons, add or subtract to perform the test.

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

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

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

At this stage, a check is made in **1309** if the processing on this particular packet has resulted in a final state. That is, is the analyzer is done processing not only for this particular packet, but for the whole flow to which the packet belongs, and the flow is fully determined. If indeed there are no more states to process for this flow, then in **1311** the processor finalizes the processing. Some final states may need to put

a state in place that tells the system to remove a flow—for example, if a connection disappears from a lower level connection identifier. In that case, in **1311**, a flow removal state is set and saved in the flow-entry. The flow removal state may be a NOP (no-op) instruction which means there are no removal instructions.

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

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

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

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

The process of flow insertion is now described with the aid of FIG. **12**. Flows are grouped into bins of buckets by the hash value. The engine processes a UFKB-entry that may be new or that the state processor otherwise has indicated needs to be created. FIG. **12** shows the case of a new entry being created. A conversation record bin (preferably containing 4 buckets for four records) is obtained in **1203**. This is a bin that matches the hash of the UFKB, so this bin may already have been sought for the UFKB-entry by the LUE. In **1204** the FIDE **1110** requests that the record bin/bucket be maintained in the cache system **1115**. If in **1205** the cache system **1115** indicates that the bin/bucket is empty, step **1207** inserts the flow signature (with the hash) into the bucket and the bucket is marked "used" in the cache engine of cache **1115** using a timestamp that is maintained throughout the process. In **1209**, the FIDE **1110** compares the bin and bucket record flow signature to the packet to verify that all the elements are in place to complete the record. In **1211** the system marks the record bin and bucket as "in process" and as "new" in the cache system (and hence in the external memory). In **1212**, the initial statistical measures for the flow-record are set in the cache system. This in the preferred embodiment clears the set of counters used to maintain statistics, and may perform other procedures for statistical operations requires by the analyzer for the first packet seen for a particular flow.

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

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

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

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

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

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

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

In accordance with one hardware implementation, the elements shown in FIGS. **10** and **11** are implemented in a set of six field programmable logic arrays (FPGA's). The boundaries of these FPGA's are as follows. The parsing

subsystem of FIG. **10** is implemented as two FPGAS; one FPGA, and includes blocks **1006**, **1008** and **1012**, parts of **1005**, and memory **1001**. The second FPGA includes **1002**, **1007**, **1013**, **1011** parts of **1005**. Referring to FIG. **11**, the unified look-up buffer **1103** is implemented as a single FPGA. State processor **1108** and part of state processor instruction database memory **1109** is another FPGA. Portions of the state processor instruction database memory **1109** are maintained in external SRAM's. The lookup/update engine **1107** and the flow insertion/deletion engine **1110** are in another FPGA. The sixth FPGA includes the cache system **1115**, the unified memory control **1119**, and the analyzer host interface and control **1118**.

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

Operation of the Invention

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

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

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

Examples of Packet Elucidation

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

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

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

"Session tracking" also is known as one of the primary processes for tracking applications in client/server packet exchanges. The process of tracking sessions requires an initial connection to a predefined socket or port number. This method of communication is used in a variety of transport layer protocols. It is most commonly seen in the TCP and UDP transport protocols of the IP protocol.

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

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

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

a reply containing the sequence numbers from the request. The states of such a connection-oriented exchange relate to the various types of connection and maintenance messages.

Server Announcement Example

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

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

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

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

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

1. A client (CLIENT **3**, **106** in FIG. **1**) sends a TCP packet to SERVER **2** (**110** in FIG. **1**) on port **111**, with an RPC Bind Lookup Request (rpcBindLookup). TCP or UDP port **111** is always associated Sun RPC. This request specifies the program (as a program identifier), version, and might specify the protocol (UDP or TCP).
2. The server SERVER **2** (**110** in FIG. **1**) extracts the program identifier and version identifier from the request. The server also uses the fact that this packet came in using the TCP transport and that no protocol was specified, and thus will use the TCP protocol for its reply.
3. The server **110** sends a TCP packet to port number **111**, with an RPC Bind Lookup Reply. The reply contains the specific port number (e.g., port number 'port') on which future transactions will be accepted for the specific RPC program identifier (e.g., Program 'program') and the protocol (UDP or TCP) for use.

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

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

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

Process **909**: Extract the 'program,' 'version,' and 'protocol' (UDP or TCP). Extract the TCP or UDP port (process **909**) which is **111** indicating Sun RPC.

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

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

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

The RPC Announcement portMapper announcement **902** is a broadcast. Such causes various clients to execute a

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

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

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

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

In packet **206**, a fourth field **217** and a fifth field **218** are used to communicate port numbers that are used. The conversation direction determines where the port number field is. The diagonal pattern in field **217** is used to identify a source-port pattern, and the hash pattern in field **218** is used to identify the destination-port pattern. The order indicates the client-server message direction. A sixth field denoted "i^1" **219** is an element that is being requested by the client from the server. A seventh field denoted "s_1a" **220** is the service requested by the client from server **110**. The following eighth field "QA" **221** (for question mark) indicates that the client **106** wants to know what to use to access application "s_1a". A tenth field "QP" **223** is used to indicate that the client wants the server to indicate what protocol to use for the particular application.

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

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

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

The flow signature and flow states built up as a result of this exchange are now described. When the packet monitor **300** sees the request packet **206** from the client, a first flow signature **210** is built in the parser subsystem **301** according to the pattern and extraction operations database **308**. This signature **210** includes a destination and a source address **240** and **241**. One aspect of the invention is that the flow keys are built consistently in a particular order no matter what the direction of conversation. Several mechanisms may be used to achieve this. In the particular embodiment, the numerically lower address is always placed before the numerically higher address. Such least to highest order is used to get the best spread of signatures and hashes for the lookup operations. In this case, therefore, since we assume "S_1"<"C_1", the order is address "S_1" followed by client address "C_1". The next field used to build the signature is a protocol field **242** extracted from packet **206**'s field **216**, and thus is the protocol "p^1". The next field used for the signature is field **243**, which contains the destination source port number shown as a crosshatched pattern from the field **218** of the packet **206**. This pattern will be recognized in the payload of packets to derive how this packet or sequence of packets exists as a flow. In practice, these may be TCP port numbers, or a combination of TCP port numbers. In the case of the Sun RPC example, the crosshatch represents a set of port numbers of UDS for p^1 that will be used to recognize this flow (e.g., port **111**). Port **111** indicates this is Sun RPC. Some applications, such as the Sun RPC Bind Lookups, are directly determinable ("known") at the parser level. So in this case, the signature KEY-1 points to a known application denoted "a^1" (Sun RPC Bind Lookup), and a next-state that the state processor should proceed to for more complex recognition jobs, denoted as state "st_D" is placed in the field **245** of the flow-entry.

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

The two flow signatures **210** and **212** always order the destination and source address fields with server "S_1" followed by client "C_1". Such values are automatically filled in when the addresses are first created in a particular flow signature. Preferably, large collections of flow signatures are kept in a lookup table in a least-to-highest order for the best spread of flow signatures and hashes.

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

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

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

Re-Using Information from Flows for Maintaining Metrics

The flow-entry of each flow stores a set of statistical measures for the flow, including the total number of packets in the flow, the time of arrival, and the differential time from the last arrival.

Referring again to FIG. **3**, the state processing process **328** performs operations defined for the state of the flow, for example for the particular protocol so far identified for the flow. One aspect of the invention is that from time to time, a set of one or more metrics related t the flow may be determined using one or more of the statistical measures stored in the flow-entry. Such metric determining may be carried out, for example, by the state processor running instructions in the state processor instruction and pattern database **326**. Such metrics may then be sent by the analyzer subsystem to a host computer connected to the monitor. Alternatively, such metric determining may be carried out by a processor connected to the flow-entry database **324**. In our preferred hardware implementation shown in FIG. **10**, an analyzer host interface and control **1118** may be configured to configured to access flow-entry records via cache system **1115** to output to a processor via the host bus interface. The processor may then do the reporting of the base metrics.

FIG. **15** describes how the monitor system can be set up with a host computer **1504**. The monitor **300** sends metrics from time to time to the host computer **1504**, and the host computer **1504** carries out 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 determined in an embodiment of the invention to support QOS. The base metrics are determined as part of state processing or by a processor connected to monitor **300**, and the QOS metrics are determined from the base metrics by the host computer **1504**. The main reason for the breakdown is that the complete QOS metrics may be computationally complex, involving square roots and other functions requiring more computational resources than may be available in real time. The base functions are chosen to be simple to calculate in real time and from which complete QOS metrics may be determined. Other breakdowns of functions clearly are possible within the scope of the invention.

Such metric determining may be carried out, for example, by the state processor running instructions in the state processor instruction and pattern database **326**. Such base metrics may then be sent by the analyzer subsystem via a microprocessor or logic circuit connected to the monitor. Alternatively, such metric determining may be carried out by a microprocessor (or some other logic) connected to the flow-entry database **324**. In our preferred hardware implementation shown in FIGS. **10** and **11**, such a microprocessor is connected cache system **1115** via an analyzer host interface and control **1118** and host bus interface. These components may be configured to access flow-entry records via cache system **1115** to enable the microprocessor to determine and report the base metrics.

The QOS Metrics may broken into the following Metrics Groups. The names are descriptive. The list is not exhaustive, and other metrics may be used. The QOS metrics below include client-to-server (CS) and server-to-client (SC) metrics.

Traffic Metrics such as CSTraffic and SCTraffic.

Jitter Metrics such as CSTraffic and CS Traffic.

Exchange Response Metrics such as CSExchangeResponseTimeStartToStart, CSExchangeResponseTimeEndToStart, CSExchangeResponseTimeStartToEnd, SCExchangeResponseTimeStartToStart, SCExchangeResponseTimeEndToStart, and SCExchangeResponseTimeStartToEnd.

Transaction Response Metrics such as CSTransactionResponseTimeStartToStart, CSApplicationResponseTimeEndToStart, CSApplicationResponseTimeStartToEnd, SCTransactionResponseTimeStartToStart, SCApplicationResponseTimeEndToStart, and SCApplicationResponseTimeStartToEnd.

Connection Metrics such as ConnectionEstablishment and ConnectionGracefulTermination, and ConnectionTimeoutTermination.

Connection Sequence Metrics such as CSConnectionRetransmissions, SCConnectionRetransmissions, and CSConnectionOutOfOrders, SCConnectionOutOfOrders.

Connection Window Metrics, CSConnectionWindow, SCConnectionWindow, CSConnectionFrozenWindows, SCConnectionFrozenWindows, CSConnectionClosedWindows, and SCConnectionClosedWindows.

QOS Base Metrics

The simplest means of representing a group of data is by frequency distributions in sub-ranges. 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 preferred, alternate embodiments may use variable sub-range sizes.

Determining complete frequency distributions may be computationally expensive. Thus, the preferred embodiment uses metrics determined by summation functions on the individual data elements in a population.

The metrics reporting process provides data that can be used to calculate useful statistical measurements. In one embodiment, the metrics reporting process is part of the state processing that is carried out from time to time according to the state, and in another embodiment, the metrics reporting

process carried out from time to time by a microprocessor having access to flow records. Preferably, the metrics reporting process provides base metrics and the final QOS metrics calculations are carried out by the host computer **1504**. In addition to keeping the real time state processing simple, the partitioning of the tasks in this way provides metrics that are scalable. For example, the base metrics from two intervals may be combined to metrics for larger intervals.

Consider, for example is the arithmetic mean defined as the sum of the data divided by the number of data elements.

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

Two base metrics provided by the metrics reporting process are the sum of the x, and the number of elements N. The host computer **1504** performs the division to obtain the average. Furthermore, two sets base metrics for two intervals may 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.

The base metrics have been chosen to maximize the amount of data available while minimizing the amount of memory needed to store the metric and minimizing the processing requirement needed to generate the metric. The base metrics are provided in a metric data structure that contains five unsigned integer values.

N count of the number of data points for the metric.

ΣX 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.

A metric is used to describe events over a time interval. The base metrics are determined from statistical measures maintained in flow-entries. It is not necessary to cache all the events and then count them at the end of the interval. The base metrics have also been designed to be easily scaleable in terms of combining adjacent intervals.

The following rules are applied when combining base metrics for contiguous time intervals.

N ΣN

ΣX Σ(Σ(X))

$\Sigma(X^2)$ $\Sigma(\Sigma(X^2))$

X_{max} $MAX(X_{max})$

X_{min} $MIN(X_{min})$

In addition to the above five values, a "trend" indicator is included in the preferred embodiment data structure. This is provided by an enumerated type. The reason for this is that the preferred method of generating trend information is by subtract an initial first value for the interval from the final value for the interval. Only the sign of the resulting number may have value, for example, to determine an indication of trend.

Typical operations that may be performed on the base metrics include:

Number N.

$$Frequency \frac{N}{TimeInterval}.$$

Maximum X_{max}.

Minimum X_{min}.

Range $R=X_{max}-X_{min}$.

$$\text{Arithmetic Mean } \bar{X} = \frac{\sum X}{N}.$$

$$\text{Root Mean Square } RMS = \sqrt{\frac{\Sigma (X^2)}{N}}.$$

$$\text{Variance } \sigma^2 = \frac{\sum (X-\bar{X})^2}{N} = \frac{(\sum X^2) - 2\bar{X}(\sum X) + N(\bar{X}^2)}{N}.$$

$$\text{Standard Deviation } \sigma = \sqrt{\frac{\sum ((X-\bar{X})^2)}{N}} = \sqrt{\frac{(\Sigma X^2)) - 2\bar{X}(\Sigma X) + N(\bar{X}^2)}{N}}.$$

Trend information, which may be 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.

Sum of the deltas (i.e., differential values). The trend enumeration can be based on this easy calculation.

Sum of the absolute values of the delta values. This would provide a measurement of the overall movement within an interval.

Sum of positive delta values and sum of the negative delta values. Expanding each of these with an associated count and maximum would give nice information.

The statistical measurement of skew can be obtained by adding $\Sigma(X^3)$ to the existing metric.

The statistical measurement of kurtosis can be obtained by adding $\Sigma(X^3)$ and $\Sigma(X^4)$ to the existing metric.

Data to calculate a slope of a least-squares line through the data.

Various metrics are now described in more detail.

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.

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

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 in our implementaions, 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.

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

-continued

Metric Specification			
Metric	Applicability	Units	Description
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 in our implementaions, ACKnowledgements are not considered within the measurement of this metric.

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.

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.

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 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

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.

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

-continued

Metric Specification

Metric	Applicability	Units	Description
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-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.

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

-continued

Metric Specification			
Metric	Applicability	Units	Description
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.

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 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

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 Client to the Server 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 prior to the 1st data of the next Client to Server Transaction Request. 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.

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.

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)

-continued

Metric Specification

Metric	Applicability	Units	Description
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 a 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.

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.

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 Connection GracefulTermination 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, CSConnectionRetransmissions 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, SCConnectionRetransmissions 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, CSConnectionOutOfOrders 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 CSConnectionRetransmissions 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, SCConnectionOutOfOrders 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 SCConnectionRetransmissions 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, CSConnectionWindow 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, SSConnectionWindow measures number of Transport-level Acknowledges within established connection lifetimes and their relative sizes from the to 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, SCConnectionWindow 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, CSConnectionWindow 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, SCConnectionWindow 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		

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

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

Although the present invention has been described in terms of the presently preferred embodiments, it is to be understood that the disclosure is not to be interpreted as limiting. Various alterations and modifications will no doubt become apparent to those or ordinary skill in the art after having read the above disclosure. Accordingly, it is intended that the claims be interpreted as covering all alterations and modifications as fall within the true spirit and scope of the present invention.

What is claimed is:

1. A method of analyzing a flow of packets passing through a connection point on a computer network, the method comprising:

(a) receiving a packet from a packet acquisition device coupled to the connection point;

(b) for each received packet, looking up a flow-entry database for containing one or more flow-entries for previously encountered conversational flows, the looking up to determine if the received packet is of an existing flow, a conversational flow including an exchange of a sequence of one or more packets in any direction between two network entities as a result of a particular activity using a particular layered set of one or more network protocols, a conversational flow further having a set of one or more states, including an initial state;

(c) if the packet is of an existing flow, identifying the last encountered state of the flow, performing any state operations specified for the state of the flow, and updating the flow-entry of the existing flow including storing one or more statistical measures kept in the flow-entry; and

d) if the packet is of a new flow, performing any state operations required for the initial state of the new flow and storing a new flow-entry for the new flow in the flow-entry database, including storing one or more statistical measures kept in the flow-entry,

wherein every packet passing though the connection point is received by the packet acquisition device, and

wherein at least one step of the set consisting of of step (a) and step (b) includes identifying the protocol being used in the packet from a plurality of protocols at a plurality of protocol layer levels,

such that the flow-entry database is to store flow entries for a plurality of conversational flows using a plurality of protocols, at a plurality of layer levels, including levels above the network layer.

2. A method according to claim **1**, wherein step (b) includes

extracting identifying portions from the packet,

wherein the extracting at any layer level is a function of the protocol being used at the layer level, and

wherein the looking up uses a function of the identifying portions.

3. A method according to claim 1, wherein the steps are carried out in real time on each packet passing through the connection point.

4. A method according to claim 1, wherein the one or more statistical measures include measures selected from the set consisting of the total packet count for the flow, the time, and a differential time from the last entered time to the present time.

5. A method according to claim 1, further including reporting one or more metrics related to the flow of a flow-entry from one or more of the statistical measures in the flow-entry.

6. A method according to claim 1, wherein the metrics include one or more quality of service (QOS) metrics.

7. A method according to claim 5, wherein the reporting is carried out from time to time, and wherein the one or more metrics are base metrics related to the time interval from the last reporting time.

8. A method according to claim 7, further comprising calculating one or more quality of service (QOS) metrics from the base metrics.

9. A method according to claim 7, wherein the one or more metrics are selected to be scalable such that metrics from contiguous time intervals may be combined to determine respective metrics for the combined interval.

10. A method according to claim 1, wherein step (c) includes if the packet is of an existing flow, identifying the last encountered state of the flow and performing any state operations specified for the state of the flow starting from the last encountered state of the flow; and wherein step (d) includes if the packet is of a new flow, performing any state operations required for the initial state of the new flow.

11. A method according to claim 10, further including reporting one or more metrics related to the flow of a flow-entry from one or more of the statistical measures in the flow-entry.

12. A method according to claim 11, wherein the reporting is carried out from time to time, and wherein the one or more metrics are base metrics related to the time interval from the last reporting time.

13. A method according to claim 12, wherein the reporting is part of the state operations for the state of the flow.

14. A method according to claim 10, wherein the state operations include updating the flow-entry, including storing identifying information for future packets to be identified with the flow-entry.

15. A method according to claim 14, further including receiving further packets, wherein the state processing of each received packet of a flow furthers the identifying of the application program of the flow.

16. A method according to claim 15, wherein one or more metrics related to the state of the flow are determined as part of the state operations specified for the state of the flow.

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

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

(b) a memory for storing a database for containing one or more flow-entries for previously encountered conversational flows to which a received packet may belong, a conversational flow including an exchange of a sequence of one or more packets in any direction between two network entities as a result of a particular activity using a particular layered set of one or more network protocols, a conversational flow further having a set of one or more states, including an initial state; and

(c) an analyzer subsystem coupled to the packet acquisition device configured to lookup for each received packet whether a received packet belongs to a flow-entry in the flow-entry database, to update the flow-entry of the existing flow including storing one or more statistical measures kept in the flow-entry in the case that the packet is of an existing flow, and to store a new flow-entry for the new flow in the flow-entry database, including storing one or more statistical measures kept in the flow-entry if the packet is of a new flow,

wherein the analyzer subsystem is further configured to identify the protocol being used in the packet from a plurality of protocols at a plurality of protocol layer levels, and

wherein the database is to store flow entries for a plurality of conversational flows using a plurality of protocols, at a plurality of layer levels, including levels above the network layer.

18. A packet monitor according to claim 17, further comprising:

a parser subsystem coupled to the packet acquisition device and to the analyzer subsystem configured to extract identifying information from a received packet,

wherein each flow-entry is identified by identifying information stored in the flow-entry, and wherein the cache lookup uses a function of the extracted identifying information.

19. A packet monitor according to claim 17, wherein the one or more statistical measures include measures selected from the set consisting of the total packet count for the flow, the time, and a differential time from the last entered time to the present time.

20. A packet monitor according to claim 17, further including a statistical processor configured to determine one or more metrics related to a flow from one or more of the statistical measures in the flow-entry of the flow.

21. A packet monitor according to claim 20, wherein the statistical processor determine and reports the one or more metrics from time to time.

* * * * *

PATENT APPLICATION FEE DETERMINATION RECORD

Effective October 1, 2000

Application or Docket Number: 09/608126

CLAIMS AS FILED - PART I

FOR	(Column 1) NUMBER FILED	(Column 2) NUMBER EXTRA
BASIC FEE	[redacted]	
TOTAL CLAIMS	21 minus 20=	*
INDEPENDENT CLAIMS	3 minus 3 =	*
MULTIPLE DEPENDENT CLAIM PRESENT		

* If the difference in column 1 is less than zero, enter "0" in column 2

SMALL ENTITY TYPE ☐ RATE	FEE	OR	OTHER THAN SMALL ENTITY RATE	FEE
[redacted]	$385	OR	[redacted]	$770
X$ 9=		OR	X$18=	
X43		OR	X86 =	
+145		OR	290=	
TOTAL		OR	TOTAL	770

CLAIMS AS AMENDED - PART II

AMENDMENT A	(Column 1) [redacted]	CLAIMS REMAINING AFTER AMENDMENT	(Column 2) [redacted]	HIGHEST NUMBER PREVIOUSLY PAID FOR	(Column 3) PRESENT EXTRA
	Total	* 21	Minus	** 21	= —
	Independent	* 2	Minus	*** 3	= —
	FIRST PRESENTATION OF MULTIPLE DEPENDENT CLAIM				

SMALL ENTITY RATE	ADDITIONAL FEE	OR	OTHER THAN SMALL ENTITY RATE	ADDITIONAL FEE
X$ 9=		OR	X$18=	
X43=		OR	X86 =	
+145=		OR	290	
TOTAL ADDIT. FEE		OR	TOTAL ADDIT. FEE	

AMENDMENT B	(Column 1) [redacted]	CLAIMS REMAINING AFTER AMENDMENT	(Column 2) [redacted]	HIGHEST NUMBER PREVIOUSLY PAID FOR	(Column 3) PRESENT EXTRA
	Total	*	Minus	**	=
	Independent	*	Minus	***	=
	FIRST PRESENTATION OF MULTIPLE DEPENDENT CLAIM				

SMALL ENTITY RATE	ADDITIONAL FEE	OR	OTHER THAN SMALL ENTITY RATE	ADDITIONAL FEE
X$ 9=		OR	X$18=	
X43		OR	X86	
+145		OR	290=	
TOTAL ADDIT. FEE		OR	TOTAL ADDIT. FEE	

AMENDMENT C	(Column 1) [redacted]	CLAIMS REMAINING AFTER AMENDMENT	(Column 2) [redacted]	HIGHEST NUMBER PREVIOUSLY PAID FOR	(Column 3) PRESENT EXTRA
	Total	*	Minus	**	=
	Independent	*	Minus	***	=
	FIRST PRESENTATION OF MULTIPLE DEPENDENT CLAIM				

SMALL ENTITY RATE	ADDITIONAL FEE	OR	OTHER THAN SMALL ENTITY RATE	ADDITIONAL FEE
X$ 9=		OR	X$18=	
X43		OR	86	
+145		OR	+290	
TOTAL ADDIT. FEE		OR	TOTAL ADDIT. FEE	

* If the entry in column 1 is less than the entry in column 2, write "0" in column 3.
** If the "Highest Number Previously Paid For" IN THIS SPACE is less than 20, enter "20."
*** If the "Highest Number Previously Paid For" IN THIS SPACE is less than 3, enter "3."
The "Highest Number Previously Paid For" (Total or Independent) is the highest number found in the appropriate box in column 1.

PATENT APPLICATION FEE DETERMINATION RECORD
Effective December 29, 1999

Application or Docket Number: 09/608126

CLAIMS AS FILED - PART I

	(Column 1) NUMBER FILED	(Column 2) NUMBER EXTRA
FOR		
BASIC FEE		
TOTAL CLAIMS	21 minus 20=	* 1
INDEPENDENT CLAIMS	2 minus 3 =	*
MULTIPLE DEPENDENT CLAIM PRESENT		

SMALL ENTITY TYPE ☐		OR	OTHER THAN SMALL ENTITY	
RATE	FEE		RATE	FEE
	345.00	OR		690.00
X$ 9=		OR	X$18=	18
X39=		OR	X78=	
+130=		OR	+260=	
TOTAL		OR	TOTAL	708

* If the difference in column 1 is less than zero, enter "0" in column 2

CLAIMS AS AMENDED - PART II

AMENDMENT A

	(Column 1) CLAIMS REMAINING AFTER AMENDMENT		(Column 2) HIGHEST NUMBER PREVIOUSLY PAID FOR	(Column 3) PRESENT EXTRA
Total	* 21	Minus	** 21	= —
Independent	* 2	Minus	*** 3	= —
FIRST PRESENTATION OF MULTIPLE DEPENDENT CLAIM				

SMALL ENTITY		OR	OTHER THAN SMALL ENTITY	
RATE	ADDITIONAL FEE		RATE	ADDITIONAL FEE
X$ 9=		OR	X$18=	
X39=		OR	X78=	
+130=		OR	+260=	
TOTAL ADDIT. FEE		OR	TOTAL ADDIT. FEE	

AMENDMENT B

	(Column 1) CLAIMS REMAINING AFTER AMENDMENT		(Column 2) HIGHEST NUMBER PREVIOUSLY PAID FOR	(Column 3) PRESENT EXTRA
Total	*	Minus	**	=
Independent	*	Minus	***	=
FIRST PRESENTATION OF MULTIPLE DEPENDENT CLAIM				

RATE	ADDITIONAL FEE		RATE	ADDITIONAL FEE
X$ 9=		OR	X$18=	
X39=		OR	X78=	
+130=		OR	+260=	
TOTAL ADDIT. FEE		OR	TOTAL ADDIT. FEE	

AMENDMENT C

	(Column 1) CLAIMS REMAINING AFTER AMENDMENT		(Column 2) HIGHEST NUMBER PREVIOUSLY PAID FOR	(Column 3) PRESENT EXTRA
Total	*	Minus	**	=
Independent	*	Minus	***	=
FIRST PRESENTATION OF MULTIPLE DEPENDENT CLAIM				

RATE	ADDITIONAL FEE		RATE	ADDITIONAL FEE
X$ 9=		OR	X$18=	
X39=		OR	X78=	
+130=		OR	+260=	
TOTAL ADDIT. FEE		OR	TOTAL ADDIT. FEE	

* If the entry in column 1 is less than the entry in column 2, write "0" in column 3.
** If the "Highest Number Previously Paid For" IN THIS SPACE is less than 20, enter "20."
***If the "Highest Number Previously Paid For" IN THIS SPACE is less than 3, enter "3."
The "Highest Number Previously Paid For" (Total or Independent) is the highest number found in the appropriate box in column 1.

C_ _ EET FOR CON_ _ _ ING DA

Line	Code	Serial No.	Filing Date	Status	Document No.	Issue Date
104	68	60/141,903	6-30-99			
105						
106						
107						
108						
109						
110						
111						
112						
113						
114						
115						
116						
117						

Condition and Status Codes for Continuing Data

CONDITION CODE:

71	Continuation of application No.
81	which is a continuation of application No.
91	and a continuation of application No.
72	Continuation-in-part of application No.
82	which is a continuation-in-part of application No.
75	and a continuation-in-part of application No.
74	Division of application No.
84	which is a division of application No.
76	and a division of application No.
86	, said application No.
89	Application No.
90	and application No.
92	each
65	filed as application No.
66	substitute for application No.
68	Provisional application No.

STATUS CODE

01	Patent No.
03	abandoned
04	SIR No.

NOTE I: When the code 86 and 92 are used, they must be followed by 81, 82 or 84 – condition beginning with "which is"

NOTE II: Codes 71, 72 and 74 may be used only on the first line; one of them must be used on the first line in regular continuing data. 66 or 68 may be used on the first line in Substitute or Provisional cases. Remember, however, that if there is a Provisional and other continuing data, the Provisional is always listed last

SEARCHED

Class	Sub.	Date	Exmr.
709	223, 224 231, 232 252, 231 32 43	6/19/03	TVU
370 379 704		12/15/03	TVU
709 714 340	220 39 825	5/18/04	TVU

INTERFERENCE SEARCHED

Class	Sub.	Date	Exmr.
709	224 223	5/18/04	TVU

SEARCH NOTES
(INCLUDING SEARCH STRATEGY)

	Date	Exmr.
EAST search (US, EPO, JPO, IBM) IEEE NPL	6/19/03	TVU
updated	12/15/03	TVU
updated	5/18/04	TVU

(RIGHT OUTSIDE)

ISSUE SLIP STAPLE AREA (for additional cross references)

POSITION	INITIALS	ID NO.	DATE
FEE DETERMINATION	J.L.		7/24/00
O.I.P.E. CLASSIFIER		49	7/[illegible]/00
FORMALITY REVIEW	NDu	8531	08/21/00
RESPONSE FORMALITY REVIEW	LH	60185	12-14-00

INDEX OF CLAIMS

✓ ..... Rejected — N ..... Non-elected
= ..... Allowed — I ..... Interference
(Through numeral) ... Canceled — A ..... Appeal
— ..... Restricted — O ..... Objected

Claim Final	Claim Original	Date										
		6/20/03	9/5/03	4/18/04								
1	1	✓	✓	=								
2	2			=								
3	3			=								
4	4			=								
5	5			=								
6	6			=								
7	7			=								
8	8			=								
9	9			=								
10	10			=								
11	11			=								
12	12			=								
13	13			=								
14	14			=								
15	15			=								
16	16			=								
17	17			=								
18	18			=								
19	19			=								
20	20			=								
21	21	✓	✓	=								
	22											
	23											
	24											
	25											
	26											
	27											
	28											
	29											
	30											
	31											
	32											
	33											
	34											
	35											
	36											
	37											
	38											
	39											
	40											
	41											
	42											
	43											
	44											
	45											
	46											
	47											
	48											
	49											
	50											

Claim Final	Claim Original	Date
	51	
	52	
	53	
	54	
	55	
	56	
	57	
	58	
	59	
	60	
	61	
	62	
	63	
	64	
	65	
	66	
	67	
	68	
	69	
	70	
	71	
	72	
	73	
	74	
	75	
	76	
	77	
	78	
	79	
	80	
	81	
	82	
	83	
	84	
	85	
	86	
	87	
	88	
	89	
	90	
	91	
	92	
	93	
	94	
	95	
	96	
	97	
	98	
	99	
	100	

Claim Final	Claim Original	Date
	101	
	102	
	103	
	104	
	105	
	106	
	107	
	108	
	109	
	110	
	111	
	112	
	113	
	114	
	115	
	116	
	117	
	118	
	119	
	120	
	121	
	122	
	123	
	124	
	125	
	126	
	127	
	128	
	129	
	130	
	131	
	132	
	133	
	134	
	135	
	136	
	137	
	138	
	139	
	140	
	141	
	142	
	143	
	144	
	145	
	146	
	147	
	148	
	149	
	150	

If more than 150 claims or 10 actions staple additional sheet here

(LEFT INSIDE)

PATENT APPLICATION

09608126

CONTENTS

	Date Received (Incl. C. of M.) or Date Mailed		Date Received (Incl. C. of M.) or Date Mailed
1. Application 18 Pgs papers.		42.	
2. Ltr. Req. Det. fee	10/02/00	43.	
3. Decl. & Serial Oath	11-6-00	44.	
4. I.D.S. w/ References	4-12-02	45.	
5. Restriction	7-10-03	46.	
6. Ext. of Time 1 mth	11/3/03	47.	
7. Amdt A	11/3/03	48.	
8. Final Restriction	12/23/03	49.	
9. Change of Address	4/1/04	50.	
10. Interview Summary	4/16/04	51.	
11. Reply RCE/EOT 1 mth	4/19/04	52.	
12. Amdt B	4/19/04	53.	
13. Terminal Disclaimer	4-19-04	54.	
14. Notice of Allowance	06/4/04	55.	
15.	1/18/05	56.	
16.		57.	
17.		58.	
18.		59.	
19.		60.	
20.		61.	
21.		62.	
22.		63.	
23.		64.	
24.		65.	
25.		66.	
		67.	
		68.	
		69.	
		70.	
		71.	
		72.	
		73.	
		74.	
33.		75.	
34.		76.	
35.		77.	
36.		78.	
37.		79.	
38.		80.	
39.		81.	
		82.	

(LEFT OUTSIDE)