

UNITED STATES PATENT AND TRADEMARK OFFICE

BEFORE THE PATENT TRIAL AND APPEAL BOARD

SANDVINE CORPORATION AND SANDVINE INCORPORATED ULC,
Petitioners

v.

PACKET INTELLIGENCE, LLC,
Patent Owner.

Case No. TBD
U.S. Patent No. 6,665,725

DECLARATION OF BILL LIN, PH.D.

I, Bill Lin, hereby declare the following:

I. BACKGROUND AND EDUCATION

1. My name is Bill Lin, and I am a Professor of Electrical and Computer Engineering and an Adjunct Professor of Computer Science and Engineering at the University of California, San Diego.

2. I received a Bachelor of Science in Electrical Engineering and Computer Sciences from University of California, Berkeley in May 1985; a Masters of Science in Electrical Engineering and Computer Sciences from the University of California, Berkeley in May 1988; and a Ph.D. in Electrical Engineering and Computer Sciences from the University of California, Berkeley in May 1991. Although I discuss my expert qualifications in more detail below, I also attach as **Appendix A** a recent and complete curriculum vitae, which details my educational and professional background and includes a selected listing of my relevant publications.

3. I have been involved in research and technology in all aspects of computer networking and computer design problems, including the design of data networks, high-performance switches and routers, high-performance packet monitoring and measurements, many-core processors, and systems-on-chip designs.

4. I am a named inventor on five patents in the field of computer engineering, including several patents in the field of computer networking, and I have published over 170 journal articles and conference papers in top-tier venues and publications.

5. I have also served or am currently serving as Associate Editor or Guest Editor for 3 ACM or IEEE journals, as General Chair on 4 ACM or IEEE conferences, on the Organizing or Steering Committees for 6 ACM or IEEE conferences, and on the Technical Program Committees of over 44 ACM or IEEE conferences.

6. In summary, I have over 25 years of experience in research and development in the areas of computer networking and computer design.

7. I have been retained by Erise IP, PA on behalf of Sandvine Corporation and Sandvine Incorporated ULC and am submitting this declaration to offer my independent expert opinion concerning certain issues raised in the Petition for *inter partes* Review (“Petition”). My compensation is not based on the substance of the opinions rendered here. As part of my work in connection with this matter, I have studied U.S. Patent No. 6,665,725 (“the ‘725 patent”), including the respective written descriptions, figures, claims, in addition to the original file history. Moreover, I have reviewed the two Petitions for *Inter Partes* Review of

the '725 patent and I have also carefully considered the following references discussed in the Petitions, in addition to all of the materials cited herein:

- U.S. Patent No. 6,115,393 to Engel et al. ("Engel"), entitled "Network Monitoring," filed July 21, 1995 and issued September 5, 2000 [EX1007], including Engel's Appendix VI (cited in Engel at 1:10-15, 6:1-3, 43:25-56) [EX1009]
- International Patent Pub. No. WO 97/23076 to Baker et al. ("Baker"), entitled "System and Method for General Purpose Network Analysis," filed December 13, 1996 and issued June 26, 1997 [EX1038]

II. LEGAL FRAMEWORK

8. I have been informed by counsel and understand that the first step in an unpatentability analysis involves construing the claims, as necessary, to determine their scope. And, second, the construed claim language is then compared to the disclosure of the prior art. In proceedings before the United States Patent and Trademark Office, I have been informed that the claims of an unexpired patent are to be given their broadest reasonable interpretation in light of the specification from the perspective of a person of ordinary skill in the art at the time of the invention. I have been informed that the '725 Patent is unexpired.

9. In comparing the claims of the '725 Patent to the prior art, I have carefully considered the '725 Patent and its prosecution history based upon my experience and knowledge in the relevant field. In my opinion, the broadest reasonable interpretation of the claim terms of the '725 Patent is generally consistent with the ordinary and customary meaning of those terms, as one skilled

in the relevant field would understand them at the time of the invention. For purposes of this proceeding, I have applied the claim constructions set forth in the claim construction section of the IPR Petition that this declaration accompanies when analyzing the prior art and the claims. For those terms that have not expressly been construed, I have applied the meaning of the claim terms of the ‘725 Patent that is generally consistent with the terms’ ordinary and customary meaning, as a person of ordinary skill in the art would have understood them at the time of the invention.

10. I have been informed by counsel that there are two ways in which a prior art patent or printed publication can be used to invalidate a patent. First, the prior art can be shown to “anticipate” the claim. Second, the prior art can be shown to “render obvious” the claim. My understanding of the two legal standards is set forth below.

11. I have been informed by counsel that, in general, for a patent claim to be invalid as “anticipated” by the prior art, each and every feature of the claim must be found, expressly or inherently, in a single prior art reference or product arranged as in the claim. Claim limitations that are not expressly found in a prior art reference are inherent if the prior art necessarily functions in accordance with, or includes, the claim limitations.

12. I have also been informed by counsel that an inventor is not entitled to a patent if his or her invention would have been obvious to a person of ordinary skill in the field of the invention at the time the invention was made. I understand that a patent claim is not patentable under 35 U.S.C. § 103 if the differences between the patent claim and the prior art are such that the claimed subject matter as a whole would have been obvious at the time the claimed invention was made to a person having ordinary skill in the art to which the subject matter pertains. Obviousness, as I understand it, is based on the scope and content of the prior art, the differences between the prior art and the claim, the level of ordinary skill in the art, and, to the extent that they exist and have an appropriate nexus to the claimed invention (as opposed to prior art features), secondary indicia of non-obviousness.

13. I have been informed that whether there are any relevant differences between the prior art and the claimed invention is to be analyzed from the view of a person of ordinary skill in the art at the time of the invention, in this case June 30, 1999. As such, my opinions below as to a person of ordinary skill in the art are as of the time of the invention, even if not expressly stated as such; for example, even if stated in the present tense.

14. In analyzing the relevance of the differences between the claimed invention and the prior art, I have been informed that I must consider the impact, if any, of such differences on the obviousness or non-obviousness of the invention as

a whole, not merely some portion of it. The person of ordinary skill faced with a problem is able to apply his or her experience and ability to solve the problem and also look to any available prior art to help solve the problem.

15. An invention is obvious if a person of ordinary skill in the art, facing the wide range of needs created by developments in the field, would have seen an obvious benefit to the solutions tried by the applicant. When there is a design need or market pressure to solve a problem and there are a finite number of identified, predictable solutions, it would be obvious to a person of ordinary skill to try the known options. If a technique has been used to improve one device, and a person of ordinary skill in the art would recognize that it would improve similar devices in the same way, using the technique would have been obvious.

16. It is my understanding that a precise teaching in the prior art directed to the subject matter of the claimed invention is not needed and that one may take into account the inferences and creative steps that a person of ordinary skill in the art would have employed in reviewing the prior art at the time of the invention. For example, if the claimed invention combined elements known in the prior art and the combination yielded results that were predictable to a person of ordinary skill in the art at the time of the invention, then this evidence would make it more likely that the claim was obvious. On the other hand, if the combination of known elements yielded unexpected or unpredictable results, or if the prior art teaches

away from combining the known elements, then this evidence would make it more likely that the claim that successfully combined those elements was not obvious.

17. I understand that hindsight must not be used when comparing the prior art to the invention for obviousness.

18. It is my understanding that obviousness may also be shown by demonstrating that it would have been obvious to modify what is taught in a single piece of prior art to create the subject matter of the patent claim. Obviousness may be shown by showing that it would have been obvious to combine the teachings of more than one item of prior art. In determining whether a piece of prior art could have been combined with other prior art or combined with or modified in view of other information within the knowledge of one of ordinary skill in the art, the following are examples of approaches and rationales that may be considered:

- Combining prior art elements according to known methods to yield predictable results;
- Simple substitution of one known element for another to obtain predictable results;
- Use of a known technique to improve similar devices (methods, or products) in the same way;
- Applying a known technique to a known device (method, or product) ready for improvement to yield predictable results;
- Applying a technique or approach that would have been "obvious to try" (choosing from a finite number of identified, predictable solutions, with a reasonable expectation of success);

- Known work in one field of endeavor may prompt variations of it for use in either the same field or a different one based on design incentives or other market forces if the variations would have been predictable to one of ordinary skill in the art; or
- Some teaching, suggestion, or motivation in the prior art that would have led one of ordinary skill to modify the prior art reference or to combine prior art reference teachings to arrive at the claimed invention.

19. I understand that the rationale for modifying a reference and/or combining references may come from sources such as explicit statements in the prior art, or the knowledge of one of ordinary skill in the art, including any need or problem known in the field at the time, even if different from the specific need or problem addressed by the inventor of the patent claim.

III. LEVEL OF A PERSON HAVING ORDINARY SKILL IN THE ART

20. I understand that the factors considered when determining the ordinary level of skill in the art include the complexity of the technology involved at the time of the alleged invention, the types of problems faced in the art at that time, prior art solutions to those problems, and the educational level of active workers in the field. As discussed throughout this declaration, I am familiar with each of these factors. *See, e.g.*, Section VI (discussing the state of the art and technology involved at the time of the alleged invention).

21. It is my opinion that a person having ordinary skill in the art at the time of the alleged invention of the '725 patent (June 30, 1999) would have had at least a bachelor's degree, or equivalent, in electrical engineering, computer

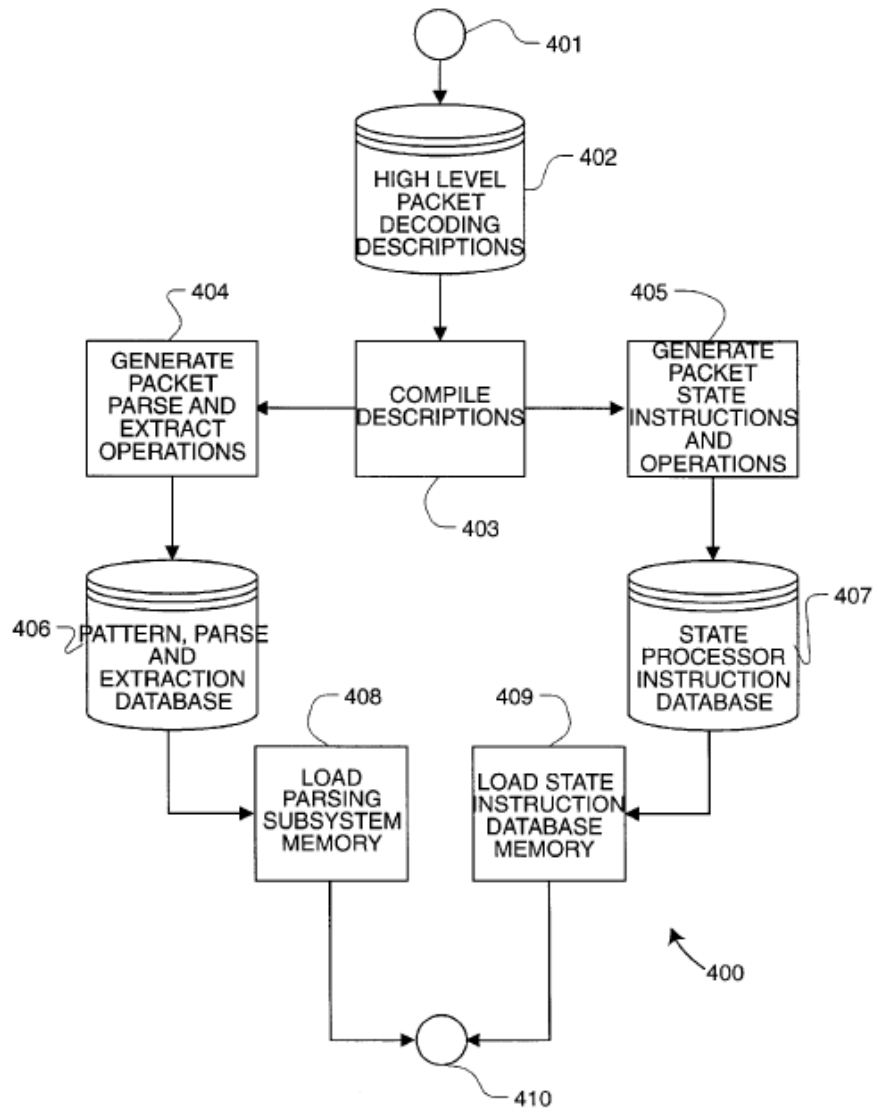
engineering, computer science, or a related field or an equivalent number of years of working experience, and one to two years of experience in networking environments, including at least some experience with network traffic monitors, analyzers, and/or firewalls or other intrusion detection systems.

22. As demonstrated by my experience above, I more than meet the definition of one of ordinary skill in the art. In addition, throughout the 1990s and 2000s, I worked with numerous people of ordinary skill in the field in question.

IV. OVERVIEW OF THE '725 PATENT

23. On its face, the '725 Patent is directed to a method of performing protocol specific operations on a packet. EX1033, '725 patent at Abstract. The method includes performing various functions such as receiving the packet and a set of protocol descriptions for protocols that may be used in the packet and performing protocol specific operations on the packet based on the protocol descriptions. *Id.* The protocol description includes child protocols, information related to the location of the child protocol, and any protocol specific operations to be performed on the packet for the particular protocol at a particular layer level. *Id.* The protocol specific operations may also include state processing operations. *Id.*

24. An exemplary implementation of the disclosed system is shown in Fig. 4:



V. OVERVIEW OF THE RELEVANT FILE HISTORIES

25. The '725 Patent was filed on June 30, 2000 with 18 claims. *See* EX1039, June 30, 2000 As-Filed '725 Application.

26. The Examiner rejected claims 1 and 16 under § 112 as indefinite, requesting that the term “none or more” be changed to “one or more.” The Examiner also rejected claims 1-3, 13, 14, 17, and 18 as anticipated by Bruell. EX1039, June 4, 2003 Rejection. The Examiner objected to claims 4-11 and 15 as

being dependent upon rejected base claims, but indicated that they would be allowable if rewritten in independent form noting that the prior art did not disclose a network monitor for analyzing different packets or frame formats for performing specific operations comprising a combination of, among other things:

- 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 to 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.
- Looking up a flow-entry database comprising one or more flow-entries, 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 existing flow-entry; 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 the parsing and extraction operations depend on the contents of one or more packet headers.

EX1039, June 4, 2003 Rejection.

27. In response, Applicants cancelled independent claim 1 and amended the remaining claims. EX1039, June 13, 2003 Response. Applicants asserted that the rejection of claims 13, 17, and 18 was erroneous, distinguishing Bruell from

the purported invention on numerous grounds, including that Bruell does not form a function of the selected portions for identifying the packet as belonging to conversational flow and does not perform processing that is a function of the state of the flow of the packet. *See* June 13, 2003 Response. Applicants subsequently filed a supplemental response amending claim 18. EX1039, June 27, 2003 Supplemental Response.

28. The Examiner issued a Notice of Allowance without any further explanation. EX1039, July 1, 2003 Notice of Allowance.

VI. BACKGROUND ON RELEVANT TECHNOLOGIES

29. As discussed above, the subject matter of the '725 patent generally relates to the field of network monitoring of computer networks. To better understand my opinions on the inventions claimed in the '725 patent and their relation to Engel and Baker, I provide below some background on computer networking, network monitoring, and content addressable memories.

A. Computer Networking and Protocol Layers

30. Computer networks based on layered protocol architectures have been well-known since at least the 1970s. The two most dominant models of protocol layers are the OSI model and the Internet model. Both models were defined in 1970s. The OSI model defines a seven-layer protocol stack whereas the Internet model defines a simpler five-layer protocol stack.

31. The Internet model is illustrated in Figure 1(a) below. The physical layer is concerned with the low-level details of transmitting raw bits over some physical medium (e.g., fiber optics, copper cables, or radio frequency transmissions).

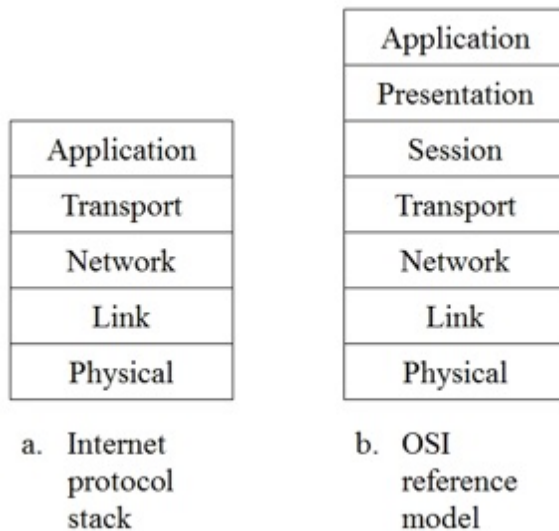


Figure 1: The Internet protocol stack (a) and OSI model (b).

32. The link layer is responsible for controlling access to an individual network by a host or transferring data between two hosts on the same network. Examples of link layer protocols include Ethernet and WiFi. Such networks are commonly referred to as *Layer 2* networks.

33. For reasons that are beyond the scope of this matter, there are practical limits on the size of each individual network, and the individual networks may use different addressing schemes or frame formats, or more broadly, different link layer protocols. Therefore, an *internetworking* protocol layer is needed so that

larger networks can be formed from individual networks that may span far geographical distances.

34. This is the role of the *network layer* in the Internet protocol stack, and the network-layer protocol for the Internet is called the IP protocol (Internet Protocol). Among other functions, the IP protocol provides a uniform addressing scheme across the Internet so that hosts on different networks can talk to each other. The IP protocol layer is commonly referred to as *Layer 3*, and the Layer 3 interconnection devices that are used to internetwork individual networks are commonly referred to as *routers*, which are responsible for forwarding (routing) packets across network boundaries.

35. While the IP protocol layer provides for Internet addressing and routing functions, the transport layer is concerned with the end-to-end transport of application-layer messages between application end-points. The two most widely used transport protocols are TCP and UDP, either of which can be used to transport application-layer messages. Whereas TCP provides a connection-oriented service to its applications that includes guaranteed delivery and congestion control, UDP provides a simpler connectionless service that does not provide for reliability or flow control.

36. Finally, the application layer is concerned with the application itself. Examples of application layer protocols include HTTP (for web browsing), SMTP (for emails), and FTP (for file transfers).

37. The seven-layer OSI reference model is illustrated in Figure 1(b) above. Five of the OSI model layers have roughly the same functions as their similarly named counterparts in the Internet model (namely, the physical, link, network, transport, and application layers). The OSI reference model has two additional layers, the session and presentation layers. The session layer controls the connections between computers, and the presentation layer establishes context between application-layer entities. These two layers are not present in the Internet model. The Internet model assumes that the services associated with these two layers are built into the application if they are needed.

B. Encapsulation

38. To transport application-layer messages across the Internet using TCP/IP, the application-layer message is first encapsulated into TCP segments. Each TCP segment adds a TCP header, which contains information that the receiver-side will use for connection and application identification, sequencing, and reliable transmission. The TCP segments are in turn encapsulated into IP (Layer 3) packets. Each IP packet adds an IP header, which contains, among other information, protocol identification information and the source and destination IP

addresses. Finally, the IP packets are in turn encapsulated into Layer 2 frames for forwarding within a Layer 2 network. Each Layer 2 frame adds a Layer 2 header, for example an Ethernet header if Ethernet is used. Among other information, the Ethernet header contains a Layer 2 address, also referred to as a MAC (media access control) address. This is illustrated in Figure 2 below. Similar encapsulations are performed when using another transport-layer protocol like UDP.

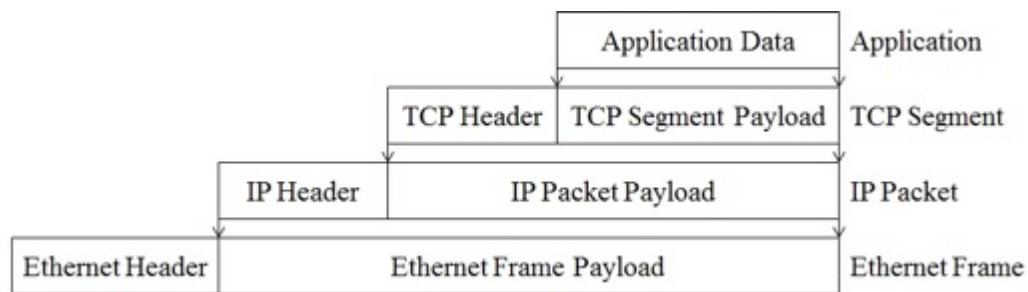


Figure 2: Encapsulation of TCP/IP.

39. The ideas of encapsulation and de-encapsulation are well-known since at least the 1970s when the Internet protocol stack was first defined. The various encapsulation steps are performed by the sending host as application messages go through the different protocols layers, with a layer-specific header added at each layer, before the final encapsulated packet leaves the host and enters the network. Similarly, on the receiving end, the host is responsible for de-encapsulating the received packet, extracting relevant header and payloads at each layer of the protocol stack, performing protocol specific operations, and passing the de-

encapsulated packet up the next layer for further processing. Since the de-encapsulation of packets requires the parsing and extraction of packet headers and payloads, techniques for packet parsing and extraction of specific information (e.g., protocol identification, port numbers, and address fields, etc) were also well-known since at least the 1970s.

C. Network Monitoring

40. The general goal of networking monitoring is to collect useful information from various parts of the network so that the network can be managed and controlled using the collected information. Most of the network devices are located in remote locations. Thus, network monitoring techniques have been developed to allow network management applications to check the states of remote network devices and collect information from them.

41. SNMP (Simple Network Management Protocol) is an Internet-standard protocol for collecting and organizing information about managed devices on IP networks and for modifying that information to change device behavior. It was initially specified in RFC 1067 in August 1988. (See EX1021, RFC 1067.) SNMP was further updated to version 2 in January 1996.

42. In network monitoring using SNMP, monitored information is seen as a set of managed objects. MIBs (Management Information Bases) define the sets of objects being monitored.

43. RMON (Remote Network Monitoring) is another important Internet standard that specifies how to monitor Internet traffic. RMON was originally standardized by RFC 1271 (see EX1022, RFC 1271) in November 1991, but it was later updated by RFC 1757 in February 1995. (See EX1023, RFC 1757.) The overall goal for RMON is to allow RMON-compliant network monitoring devices to be constructed. These devices are usually referred to as monitors or probes, which measure specific aspects of the network without interfering normal operations. These devices are often stand-alone devices and located in remote parts of the network. The RMON standard allows these devices to communicate over the network that they are monitoring.

44. Starting in the 1995-1996 timeframe, network monitoring products capable of monitoring performance at the application-level were becoming commercially available.

“CoroNet’s CoroNet Management System (CMS) is a Windows-based management platform that can also provide application-by-application fault and performance statistics across both LANs and WANs.” (See May 15, 1995, Network World, p. 17, available at: https://books.google.com/books?id=yBAEAAAAMBAJ&pg=PA37&dq=%22coronet+management+system%22&hl=en&sa=X&ved=0ahUKEwjDyuf_65TQAhXDKiYKHUJZDzEQ6AEIMTAF#v=onepage&q=%22coronet%20management%20system%22&f=false)

“The firm, which requested that its name not be mentioned, is using CoroNet Systems’s CoroNet Management System (CMS) to determine how much bandwidth a particular application is using.” (See May 1, 1995, Network World, p. 20, available at:

<https://books.google.com/books?id=thIEAAAAMBAJ&pg=PA20&dq=%22coronet+systems%22&hl=en&sa=X&ved=0ahUKEwj8z76K85TQAhWBKyYKHUVgBEgQ6AEIHZAB#v=onepage&q=%22coronet%20systems%22&f=false>)

“Compuware Corp. takes the mystery out of understanding and managing application-level performance in large complex enterprise networks with EcoNet 2.0... EcoNet’s differentiation (and strength) lies in its capability to effectively monitor network conversations at the application level.” (See September 30, 1996, InfoWorld, p. N/4, available at:

<https://books.google.com/books?id=LjoEAAAAMBAJ&pg=RA1-PA53-IA5&dq=%22Econet%22+compuware&hl=en&sa=X&ved=0ahUKEwjUhrvB5pTQAhWEbSYKHSIfBx0Q6AEIPzAJ#v=onepage&q=%22Econet%22%20compuware&f=false>)

“Compuware plans to announce Version 3.1 of EcoScope 3.1. The version has the capability to analyze application traffic at the protocol level, and also includes enhanced topology views and Web-based reporting.” (See April 20, 1998, InfoWorld, p. 10, available at: <https://books.google.com/books?id=vFAEAAAAMBAJ&pg=PA10&dq=infoworld+compuware+1998&hl=en&sa=X&ved=0ahUKEwj3u86Ln7PQAhWM0YMKHRcaAMkQ6AEIOzAI%23v=onepage&q&f=false>)

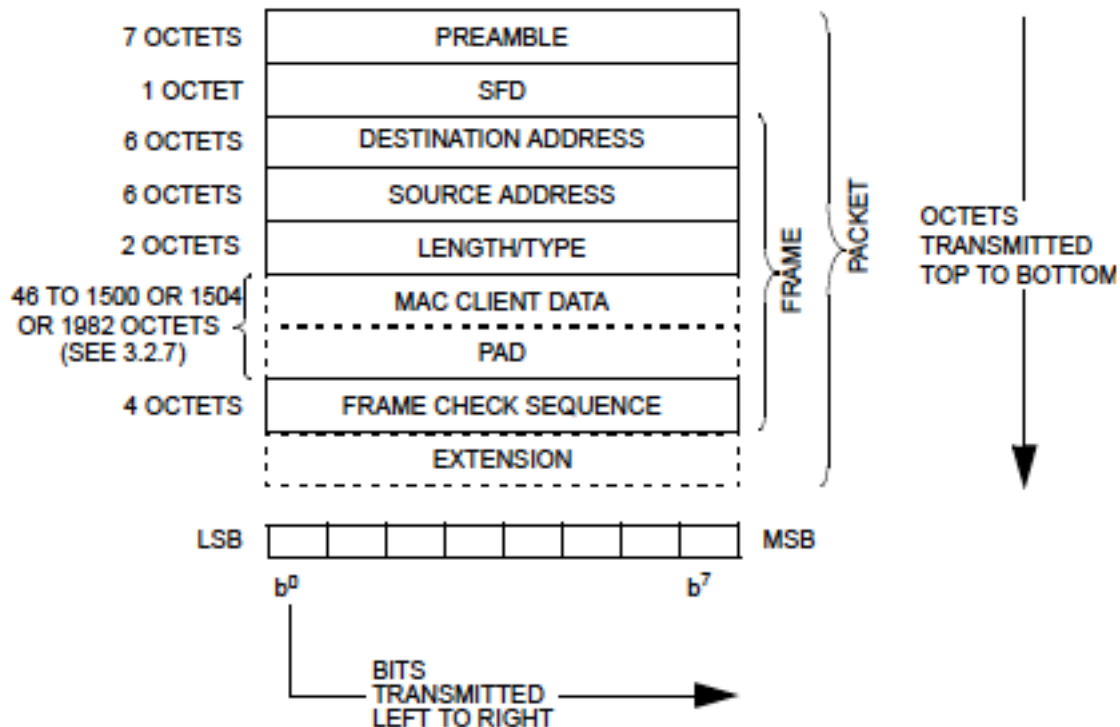
45. In response to the growing need for network monitoring at the application-level, the RMON standard was extended to add support for network- and application-layer monitoring, including classes of protocols at the network, transport, session, presentation, and application layers. This extended RMON standard, known as RMON2, was standardized by RFC 2012 in January 1997. (See EX1024, RFC 2012.) To cover network monitoring on higher layers of traffic, the RMON2 MIB added ten more groups, including groups that cover

Layer 3 traffic statistics and groups that cover traffic statistics by application protocols.

D. Packet Specifications

1. Ethernet Frame Format

46. The figure below shows the fields of an Ethernet packet:



EX1040, IEEE 802.3 Standard at p. 53; *see also* EX1041, IEEE 802.3 1997 Standard at p. 28. As depicted above and described in the IEEE 802.3 Standard (both in the current version of the standard (EX1040) and in the 1997 version of the Standard (EX1041)), the first field in an Ethernet frame is the Destination Address Field, which is a six-octet field indicating the station(s) for which the frame is intended. The second field in an Ethernet frame is the Source Address

Field, which is a six-octet field containing the physical address of the station sending the frame. IEEE 802.3 Standard at pp. 53, 54-55; IEEE 802.3 1997 Standard at pp. 28, 29.

47. The third field in an Ethernet frame is the Length/Type Field, which consists of a two-octet value used to determine whether the packet is an Ethernet II packet or an IEEE type packet. If the value of this field is greater than or equal to 1536 decimal (0600 hexadecimal), then the Length/Type field indicates that the packet is of type Ethernet II and the value of this field specifies the next layer protocol encapsulated in the Ethernet frame Data Field. IEEE 802.3 Standard at pp. 53, 55, 28; IEEE 802.3 1997 Standard at pp. 28, 30. I note that the '725 patent refers to this type of packet as "Ethernet Type/Version 2" and "Ethertype":

"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.

* * *

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 list 1712 shows the possible children for an Ethertype packet as indicated by what child recognition pattern is found offset 12."

‘725 Patent at 33:7-45, 13:66-14:21. If the Type Field of the Ethernet frame contains a value of hexadecimal 0800 (2048 decimal), then the next layer protocol is IP and the Ethernet frame Data Field will contain the IP header followed by the IP data. EX1042, RFC 894 at p. 1 (“IP datagrams are transmitted in standard Ethernet frames. The type field of the Ethernet frame must contain the value hexadecimal 0800. The data field contains the IP header followed immediately by the IP data.”); EX1043, RFC 1010 at p. 14; ‘725 Patent at Fig. 17A (“IP = 0x0800”), column 71 (etherType FIELD definition includes “ip(0x0800)”). I note that the IP protocol has also been referred to as “DOD IP” and that, even when referred to as “DOD IP,” the Type Field value assigned is hexadecimal 0800 (2048 decimal). RFC 1010 at p. 14. If the Type Field of the Ethernet frame contains a value of hexadecimal 0806 (2054 decimal), then the next layer protocol is ARP. ‘725 Patent at Fig. 17A (“ARP = 0x0806”), column 71 (etherType FIELD definition includes “arp(0x0806)”); RFC 1010 at p. 14.

48. If the value of the Type/Length Field of the Ethernet frame is less than or equal to 1500 decimal (05DC hexadecimal), then the Length/Type field indicates that the packet is of type IEEE, such as 802.2 or 802.3, and the value of this field specifies the number of data octets contained in the subsequent Data Field of the basic frame (i.e., it indicates a “Length”). IEEE 802.3 Standard at pp. 53,

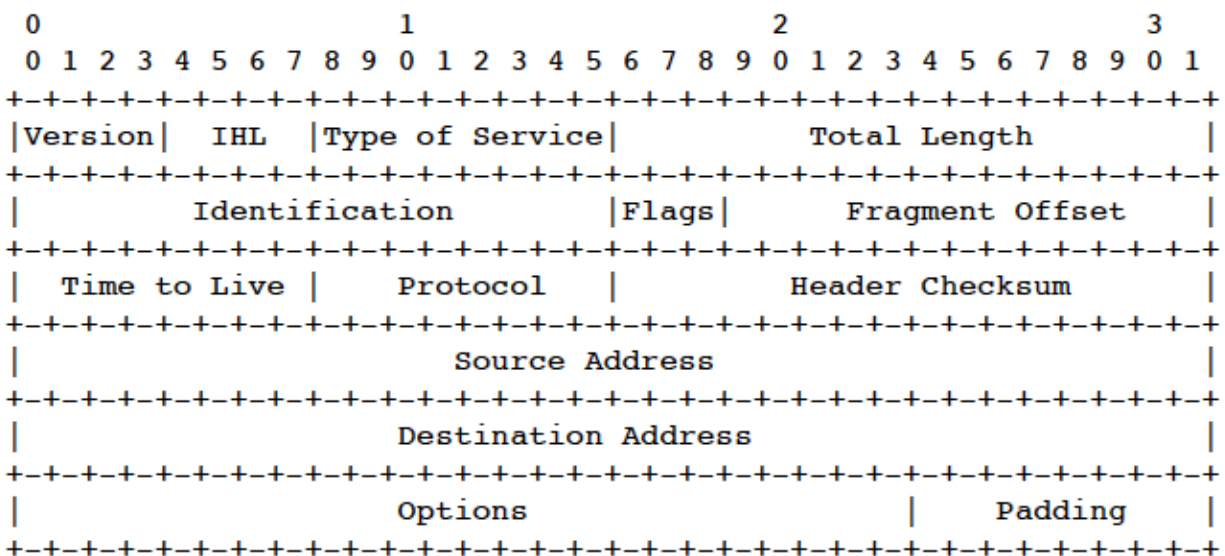
55; IEEE 802.3 1997 Standard at pp. 28, 30. The ‘725 patent refers to this type of packet as “IEEE type.”

“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.”

‘725 patent at 35:51-55; see also ‘725 Patent at 33:7-15 (reproduced above).

2. IP Datagram

49. The figure below shows the fields of an IP datagram header:



Example Internet Datagram Header

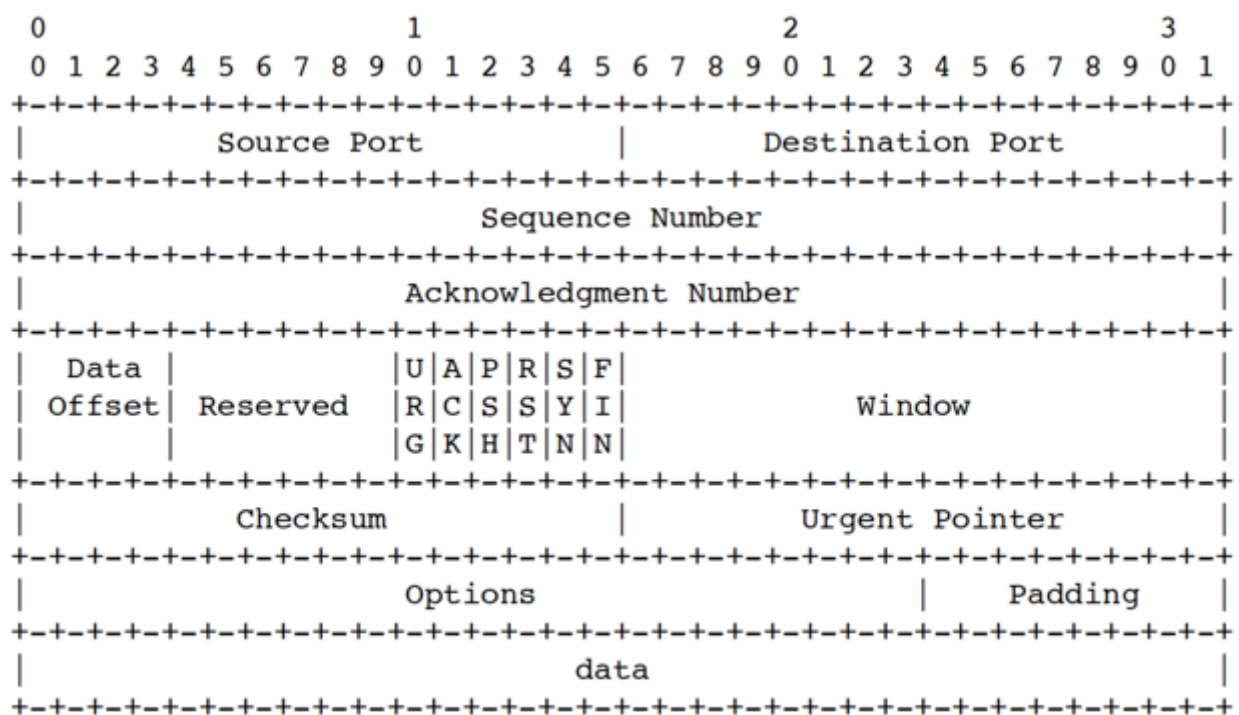
EX1044, RFC 791 at p. 11. The Protocol Field is an 8-bit field that “indicates the next level protocol used in the data portion of the [I]nternet datagram.” RFC 791 at p. 14. The values for the protocols are assigned. RFC 791 at p. 14; RFC 1010 at 4-5. For example, the assigned value for the TCP protocol is 6 decimal, the assigned value for the UDP protocol is 17 decimal, the assigned value for the

ICMP protocol is 1 decimal. RFC 1010 at p. 4; see also ‘725 patent at Fig. 17B item 1752 (ICMP=1, TCP=6, UCP=17).

50. The IHL Field is a 4-bit Internet Header Length Field that “is the length of the [I]nternet header in 32 bit words, and thus points to the beginning of the data.” RFC 791 at p. 11.

3. TCP Segment

51. The figure below shows the fields of a TCP segment:



TCP Header Format

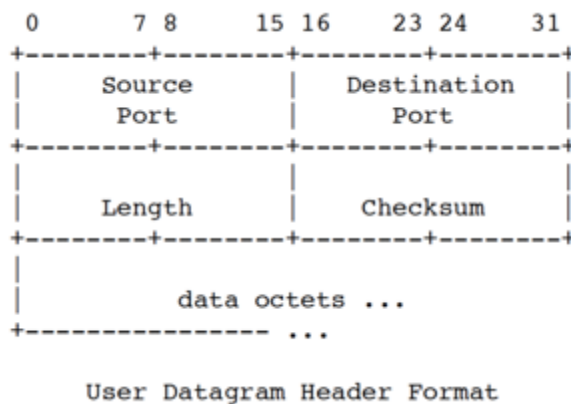
EX1045, RFC 793 at p. 15. The Source Port Field is a 16-bit field that includes the source port number. The Destination Port Field is a 16-bit field that includes the destination port number. RFC 793 at p. 15. Many application programs have

assigned or “well-known” port numbers. For example, the assigned port number for the FTP Data channel is 20 decimal, the assigned port number for the FTP Control channel is 21 decimal, the assigned port number for Telnet is 23 decimal, and the assigned port number for SMTP is 25 decimal. RFC 1010 at p. 6.

52. The Data Offset Field is a 4-bit field that is “[t]he number of 32 bit words in the TCP Header” and “indicates where the data begins.” RFC 793 at p. 16.

4. UDP Segment

53. The figure below shows the fields of a UDP segment:



EX1046, RFC 768 at p. 1. The Source Port Field indicates the port number of the sending process. The Destination Port Field indicates the destination port number. RFC 768 at pp. 1-2. The Length Field is “the length in octets of this user datagram including this header and the data.” RFC 768 at p. 2.

VII. OPERATION OF ENGEL

54. I was asked to review Engel and the source code found in Engel's Appendix VI and to provide an overview of the operation of the Engel packet monitor as disclosed in the Engel patent and the Appendix VI source code.

A. Packet Acquisition

55. Engel discloses a packet monitor as depicted in Figure 5 of the Engel patent, which includes network controller hardware described as follows:

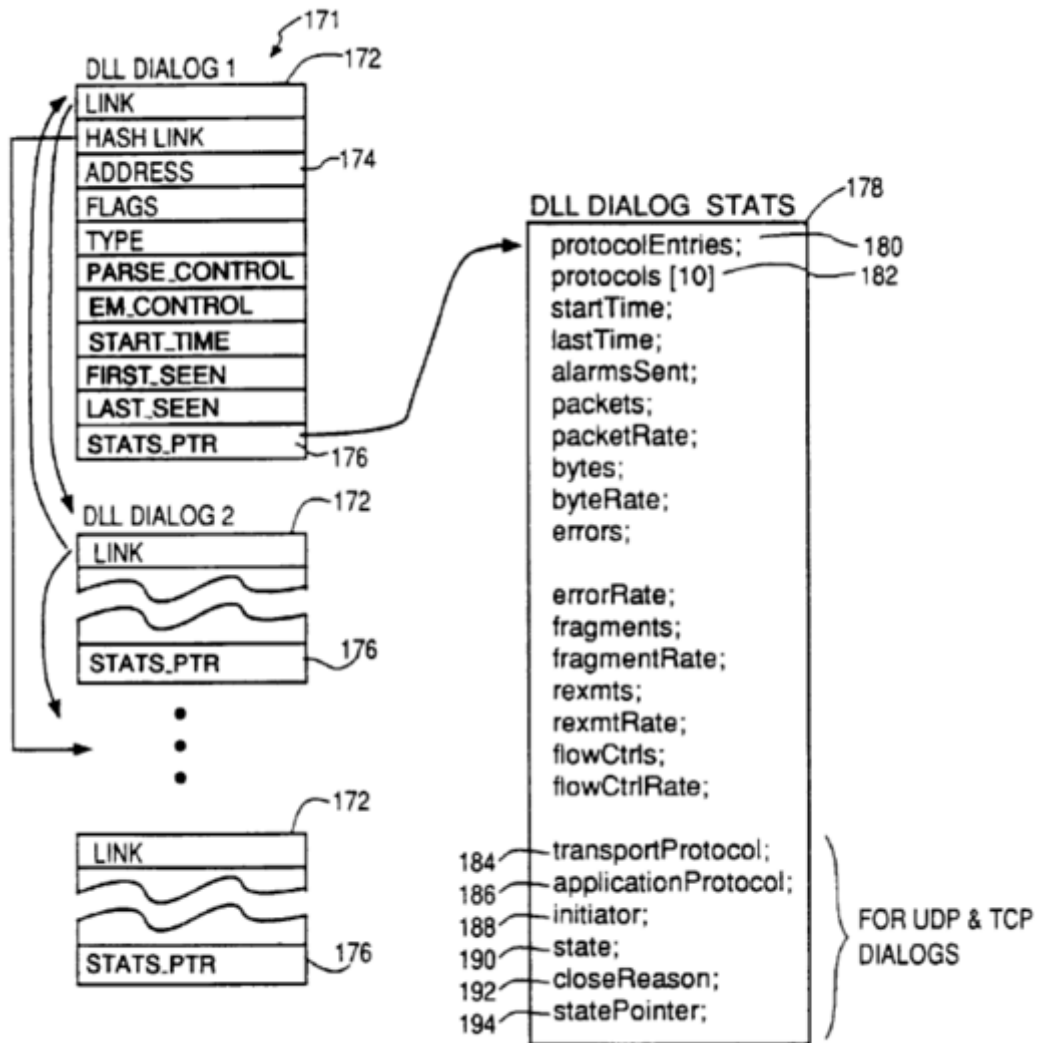
"Device Driver 24 manages the network controller hardware so that Monitor 10 is able to read and write packets from the network and it manages the serial interface. It does so both for the purposes of monitoring traffic (promiscuous mode) and for the purposes of communicating with the Management Workstation and other devices on the network. The communication occurs through the network controller hardware of the physical network (e.g. Ethernet). The drivers for the LAN controller and serial line interface are used by the boot load module and the MTM. They provide access to the chips and isolate higher layers from the hardware specifics."

EX1007, Engel at 10:45-56. Packet acquisition devices used to convert physical information on a network into packets for input into a monitor were well known and commonplace at the time of the alleged invention. The '725 patent specification confirms the same. EX1033, '725 patent at 6:11-15, 8:58-63. A skilled artisan would recognize that Engel's disclosed network controller hardware constitute a packet acquisition device.

B. Data Structures

1. Dialog Data Structures

56. Engel discloses that each dialog record includes the categories of information illustrated in Figure 7C:



EX1007, Engel at Fig. 7C; *see also* Engel at 18:63-19:23, 17:32-18:19. The Engel Appendix VI source code confirms that each dialog record includes a StatsAddrEntry data structure (corresponding to item 171 of Fig. 7C) linked via a

stats_ptr pointer (corresponding to item 176 of Fig. 7C) to an associated StatsDialogEntry data structure (corresponding to item 178 of Fig. 7C). *See* EX1009, Engel Appendix VI at p. 179 (defining StatsAddrEntry data structure in stats.h) and p. 181 (defining StatsDialogEntry data structure in stats.h). Dialog records are maintained for the data link layer, network layer, transport layer, and application layer protocols utilized in a packet. *See, e.g.*, Engel Appendix VI at p. 11 of rtp_dll_p.c (calling stats_dll_get_dialog); p. 51 of rtp_ip_p.c (calling stats_ip_get_dialog); pp. 126-127 of rtp_tcp_p.c (calling stats_tcp_get_dialog); p. 159 of rtp_udp_p.c (calling stats_udp_get_dialog); p. 73 of rtp_nfs_p.c (calling stats_nfs_get_dialog). Each dialog record is part of a doubly linked list, including forward and backward pointers to neighboring dialog records in an application-specific dialog hash table. Engel at 17:32-41, 18:63-67, Fig. 7C; Engel Appendix VI at p. 179 (defining StatsAddrEntry data structure in stats.h).

57. By way of example, the NFS application layer dialog records are stored as StatsAddrEntry records, with a pointer to a linked StatsNfsDialogEntry record. The StatsAddrEntry data structure is defined as follows:

```

*/
typedef struct stats_addr_entry_type {
    FBQentry_type          link;
    struct stats_addr_entry_type *hash_link;
    MibAddress             address;
    Uint32                 flags;
    MibDeviceType          type;
    Uint32                 parse_control;
    Uint32                 em_control;
    Uint32                 seconds_start_time;
    Uint32                 startTime;
    Uint32                 lastTime;
    Uint32                 *stats_ptr;
    Uint32                 *protocol_specific_ptr; /* Used by ip for
port mappings */
} StatsAddrEntry;

```

Engel appendix VI at p. 179 of stats.h. Each NFS dialog record is part of a doubly linked list, including forward and backward pointers to neighboring NFS dialog records in the nfs_dialog_hash_table (i.e., the “link” and “*hash_link” fields of the StatsAddrEntry dialog record). Engel Appendix VI at pp. 479, 484 of stats_nfs_p.c. The “address” field of the StatsAddrEntry data structure stores the source and destination IP addresses of the NFS dialog. Engel Appendix VI at pp. 480, 482 of stats_nfs_p.c.

58. I also note that the “FIRST_SEEN” field of Figs. 7B and 7C (corresponding to field 122 of Fig. 7A) is the startTime field of the StatsAddrEntry data structure and indicates “the time at which the Network Monitor first saw the communication” (i.e., the time the monitor first saw a packet for the associated dialog or application-specific server flow). Engel Appendix VI at p. 179 of stats.h; Engel at 18:4-6, Figs. 7A-7C. The startTime field is set upon initialization of a newly allocated StatsAddrEntry data structure (e.g., when a new dialog or

application-specific server record is created). Engel Appendix VI at pp. 477 and 482 of stats_nfs_p.c, p. 561 of stats_tcp_p.c.

59. The “LAST_SEEN” field of Figs. 7B and 7C (corresponding to field 124 of Fig. 7A) is the lastTime field of the StatsAddrEntry data structure and indicates “the time at which the last communication was seen” (i.e., the time the monitor last saw a packet associated with the dialog or application-specific server flow). Engel Appendix VI at p. 179 of stats.h; Engel at 18:6-7, Figs. 7A-7C. The lastTime field is set upon initialization of a newly allocated StatsAddrEntry data structure (e.g., when a new dialog or application-specific server record is created). Engel Appendix VI at pp. 477 and 482 of stats_nfs_p.c, p. 562 of stats_tcp_p.c. The lastTime field is also updated every time a packet for the associated flow is encountered. Engel Appendix VI at p. 75 of rtp_nfs_p.c, p. 127 of rtp_tcp_p.c, p. 187 of stats.h (calling and defining the stats_update_age_timers macro for setting the lastTime field to the current time).

60. The “START_TIME” field of Figs. 7B and 7C (corresponding to field 120 of Fig. 7A) is the seconds_start_time field of the StatsAddrEntry data structure and is a time used by the event manager to perform later rate calculations. Engel Appendix VI at p. 179 of stats.h; Engel at 18:2-4, Figs. 7A-7C. The “EM_CONTROL” field of Figs. 7B and 7C (corresponding to field 118 of Fig. 7A) is the em_control field of the StatsAddrEntry data structure and indicates that the

statistics for the associated record have changed since the event manager (EM) “last serviced the database to update rates and other variables” so that the EM will recalculate, for example, rates when the EM next services the database. Engel Appendix VI at p. 179 of stats.h; Engel at 17:62-67, Figs. 7A-7C. The seconds_start_time field is set to the current time and the em_control field is set when the first packet for the associated flow is encountered since the last time the EM serviced the database. Engel Appendix VI at p. 75 of rtp_nfs_p.c, p. 127 of rtp_tcp_p.c, pp. 188-189 of stats.h.

61. For dialog records, the “*stats_ptr” field of the StatsAddrEntry record is a pointer to the associated StatsNfsDialogEntry record. The StatsNfsDialogEntry data structure is defined as follows:


```

typedef struct stats_nfs_dialog_entry_type {
    StatsCount32      ops;
    StatsRatePerS     opRate;
    StatsCount32      readOps;
    StatsRatePerS     readOpRate;
    StatsCount32      writeOps;
    StatsRatePerS     writeOpRate;
    StatsCount32      writeCacheOps;
    StatsRatePerS     writeCacheOpRate;
    StatsCount32      readBytes;
    StatsRatePerS     readByteRate;
    StatsCount32      writeBytes;
    StatsRatePerS     writeByteRate;

    StatsCount32      rexmts;
    StatsRatePerS     rexmtRate;
    StatsCount32      flowCtrls;
    StatsRatePerS     flowCtrlRate;

    StatsCount32      rpcProgramFail; /* prog unavail or mismatch; */
    StatsRatePerS     rpcProgramFailRate; /* proc unavail or bad args */

    StatsCount32      rpcMismatch; /* wrong version number of rpc */
    StatsRatePerS     rpcMismatchRate;

    StatsCount32      rpcAuthFail; /* bad or rejected cred or verf; or auth too
weak */
    StatsRatePerS     rpcAuthFailRate;

    StatsCount32      nfsAccessErr; /* not owner, no access, read only, or stale
handle */
    StatsRatePerS     nfsAccessErrRate;

    StatsCount32      nfsHardErr; /* hard error, (e.g. disk error), */
    StatsRatePerS     nfsHardErrRate;

    StatsCount32      nfsResourceErr; /* file too big, no space, over quota,
*/
    StatsRatePerS     nfsResourceErrRate; /* or write cache flushed */

    StatsCount32      nfsUserErr; /* no entry or dir, file exists, no
device, */
    StatsRatePerS     nfsUserErrRate; /* not dir, is dir, name too long, dir
not empty */

    StatsNfsState      *state_ptr;
} StatsNfsDialogEntry;

```

Engel Appendix VI at pp. 249-250 of stats_nfs.h.

62. As shown above, each StatsNfsDialogEntry record stores various statistics, including the following statistical objects of type StatsCount32: ops, readOps, writeOps, writeCacheOps, readBytes, writeBytes, rexmts, flowCtrls, in addition to various error-related StatsCount32 statistical objects. Engel Appendix

VI at pp. 249-250 of stats_nfs.h. Each of these StatsCount32 data structures stores count, running_count, and high_thld variables:

```
typedef struct stats_count32_type
{
    Uint32      count;
    Uint32      running_count;
    Uint32      high_thld;
} StatsCount32;
```

Engel Appendix VI at p. 177 of stats.h. Engel describes these StatsCount32 statistical object classes as “counts”:

A count is a continuously incrementing variable which rolls around to 0 on overflow. It may be reset on command from the user (or from software). A threshold may be applied to the count and will cause an alarm when the threshold count is reached. The threshold count fires each time the counter increments past the threshold value. For example, if the threshold is set to 5, alarms are generated when the count is 5, 10, 15,

Engel at 15:47-54.

63. Each StatsNfsDialogEntry record also stores statistical objects of type StatsRatePerS (also called StatsRollingRate) including opRate, readOpRate, writeOpRate, writeCacheOpRate, readByteRate, writeByteRate, rexmtRate, flowCtrlRate, in addition to various error-related StatsRatePerS statistical objects. Engel Appendix VI at pp. 249-250 of stats_nfs.h. Each of these StatsRatePerS statistical object classes stores a current, high, and low StatsRateValue in addition to high_thld, count, and type variables:

```

typedef struct stats_rolling_rate_type
{
    StatsRateValue current;
    StatsRateValue high;
    StatsRateValue low;
    Uint32          high_thld;
    Uint32          count;
    Uint32          type;
} StatsRollingRate;

typedef StatsRollingRate StatsRatePerS;
typedef StatsRollingRate StatsRatePerH;

```

Engel Appendix VI at p. 178 of stats.h. Each of the current, high, and low StatsRateValue data structures stores count and time variables:

```

typedef struct stats_rate_value
{
    Uint32 count;
    Uint32 time;
} StatsRateValue;

```

Engel Appendix VI at p. 178 of stats.h. Engel describes these StatsRatePerS type statistical object classes as “rates”:

“A rate is essentially a first derivative of a count variable. The rate is calculated at a period appropriate to the variable. For each rate variable, a minimum, maximum and average value is maintained. Thresholds may be set on high values of the rate. The maximums and minimums may be reset on command. The threshold event is triggered each time the rate calculated is in the threshold region.”

Engel at 15:55-61.

64. Each StatsNfsDialogEntry record also stores a history of recorded events for the dialog in a StatsNfsState data structure, which includes a StatsNfsHistoryEntry array of for storing each even:

```

typedef struct nfs_history_entry
{
    Uint32      xid;
    Uint32      call_proc;
    Uint32      status;      /* NFS_ANSWERED or NFS_OUTSTANDING */
    Uint32      *ptr;        /* If unmount, ptr to StatsNfsFileSystemLink to unlink
on reply */
} StatsNfsHistoryEntry;

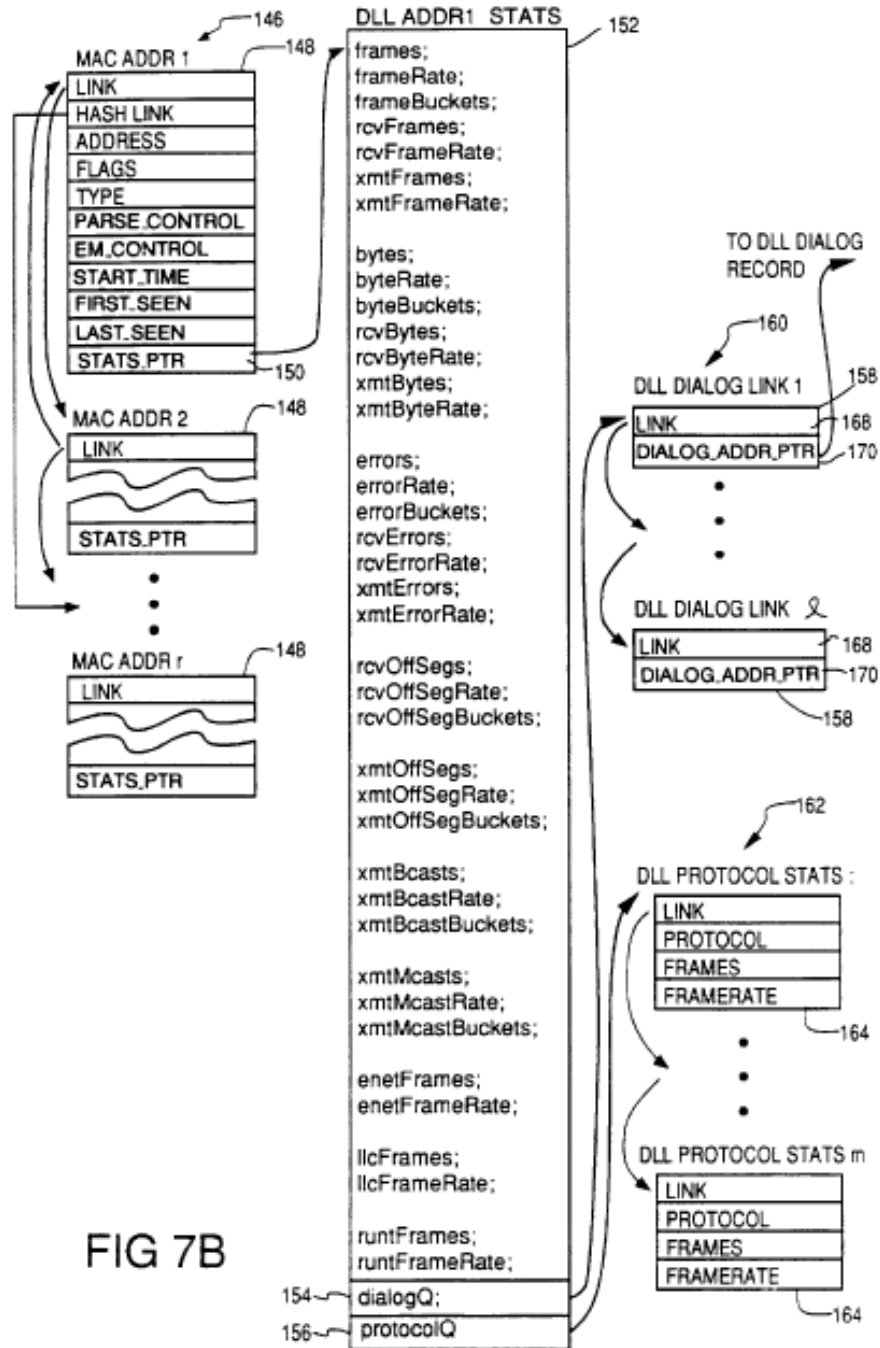
typedef struct stats_nfs_state
{
    Uint32      histx;
    StatsNfsHistoryEntry  history[NFS_HISTORY_ENTRIES];
} StatsNfsState;

```

Engel Appendix VI at p. 248 of stats_nfs.h. The size of the StatsNfsHistoryEntry array is set to store up to 8 events. Engel Appendix VI at p. 242 of stats_nfs.h (NFS_HISTORY_ENTRIES set to 8). Each event stored in the history array is identified by a unique transaction id (“xid”). *See also* Paragraph 116 below.

2. Client/Server Data Structures

65. Engel also discloses separately storing statistics for each client and each server participating in communications. Engel at 15:19, 18:41-62, Fig. 7B. The application-specific client and server data structures for storing such information are illustrated in Figure 7B:



Engel at Fig. 7B; *see also* Engel at 18:41-62, 17:32-18:40. Each client/server StatsAddrEntry record is part of a doubly linked list, including forward and backward pointers to neighboring records in an application-specific hash table.

Engel at 17:32-41, 18:41-45; Engel Appendix VI at p. 179 (defining StatsAddrEntry data structure in stats.h).

66. Additionally, each client/server record includes a pointer to a dialog link queue (corresponding to items 154 and 160 in Fig. 7B). Engel at 18:51-68, Fig. 7B. The dialog link queue includes a linked list of all dialogs in which the client/server has participated. Namely, when a new dialog is allocated for a given client/server pair, the new dialog record is added to the dialog link queue for each client and server record. Engel Appendix VI at pp. 483-484 of stats_nfs_p.c; *see also* Section VII.C.7 (paragraphs 108-110) below.

67. More specifically, Engel discloses separately storing statistics for each client and each server participating in communications for a particular application program (e.g., NFS, SMTP, FTP, Telnet, TFTP, SNMP, NSP (DNS), X Windows, LAT, NFT, RFA, VT, Netware). Engel at 15:24-28, Fig. 19, 29:24-28. In the NFS example, each NFS application-specific client and server record includes a StatsAddrEntry data structure (corresponding to item 146 of Fig. 7B) linked via a stats_ptr pointer (corresponding to item 150 of Fig. 7B) to an associated StatsNfsAddr data structure (corresponding to item 152 of Fig. 7B). *See* Engel Appendix VI at p. 179 (defining StatsAddrEntry data structure in stats.h) and pp. 247-248 (defining StatsNfsAddr data structure in stats_nfs.h). The StatsAddrEntry data structure is defined above in paragraph 57. Each NFS application-specific

client record is part of a doubly linked list, including forward and backward pointers to neighboring NFS client records in the `nfs_client_hash_table`. Engel Appendix VI at pp. 468, 474 of `stats_nfs_p.c`. Similarly, each NFS application-specific server record is part of a doubly linked list, including forward and backward pointers to neighboring NFS server records in the `nfs_server_hash_table`. Engel Appendix VI at pp. 475-476, 478 of `stats_nfs_p.c`. The “address” field of the `StatsAddrEntry` data structure stores the IP address of the NFS client/server participating in the communication. Engel Appendix VI at pp. 473-474, 477 of `stats_nfs_p.c`.

68. The “`*stats_ptr`” field of each NFS `StatsAddrEntry` record is a pointer to the associated `StatsNfsAddr` record. The `StatsNfsAddr` data structure is defined as follows:

```

typedef struct stats_nfs_addr_type
{
    UInt32                nfsLowVersion;
    UInt32                nfsHighVersion;
    UInt32                rpcLowVersion;
    UInt32                rpcHighVersion;
    UInt32                mountLowVersion;
    UInt32                mountHighVersion;

    StatsCount32          ops;
    StatsRatePerS         opRate;
    StatsBucketRate       opBuckets;
    StatsCount32          readOps;
    StatsRatePerS         readOpRate;
    StatsCount32          writeOps;
    StatsRatePerS         writeOpRate;
    StatsCount32          writeCacheOps;
    StatsRatePerS         writeCacheOpRate;

    StatsCount32          bytes;
    StatsRatePerS         byteRate;
    StatsBucketRate       byteBuckets;
    StatsCount32          readBytes;
    StatsRatePerS         readByteRate;
    StatsCount32          writeBytes;
    StatsRatePerS         writeByteRate;

    StatsCount32          errors;
    StatsRatePerS         errorRate;
    StatsBucketRate       errorBuckets;

    StatsCount32          rcvOffSegs;
    StatsRatePerS         rcvOffSegRate;
    StatsBucketRate       rcvOffSegBuckets;

    StatsCount32          xmtOffSegs;
    StatsRatePerS         xmtOffSegRate;
    StatsBucketRate       xmtOffSegBuckets;

    StatsCount32          flowCtrls;
    StatsRatePerS         flowCtrlRate;
    StatsBucketRate       flowCtrlBuckets;

    StatsCount32          rcvFrgmts;
    StatsRatePerS         rcvFrgmtRate;
    StatsCount32          xmtFrgmts;
    StatsRatePerS         xmtFrgmtRate;

    StatsCount32          rexmts;
    StatsRatePerS         rexmtRate;
    StatsBucketRate       rexmtBuckets;

    StatsRpc              rpcStats;

    StatsCount32          mountFailures;
    StatsRatePerS         mountFailureRate;

    StatsCount32          nfsErrPerm;
    StatsRatePerS         nfsErrPermRate;
    StatsCount32          nfsErrNoEnt;
    StatsRatePerS         nfsErrNoEntRate;
    StatsCount32          nfsErrIO;
    StatsRatePerS         nfsErrIORate;
    StatsCount32          nfsErrNXIO;
    StatsRatePerS         nfsErrNXIORate;
    StatsCount32          nfsErrAccess;
    StatsRatePerS         nfsErrAccessRate;
    StatsCount32          nfsErrExist;
    StatsRatePerS         nfsErrExistRate;
    StatsCount32          nfsErrNoDev;
    StatsRatePerS         nfsErrNoDevRate;
    StatsCount32          nfsErrNotDir;
    StatsRatePerS         nfsErrNotDirRate;
    StatsCount32          nfsErrIsDir;
    StatsRatePerS         nfsErrIsDirRate;
    StatsCount32          nfsErrFBig;
    StatsRatePerS         nfsErrFBigRate;
    StatsCount32          nfsErrNoSpace;
    StatsRatePerS         nfsErrNoSpaceRate;
    StatsCount32          nfsErrROFS;

```



```

StatsRatePerS      nfsErrROFSRate;
StatsCount32        nfsErrNameTooLong;
StatsRatePerS      nfsErrNameTooLongRate;
StatsCount32        nfsErrNotEmpty;
StatsRatePerS      nfsErrNotEmptyRate;
StatsCount32        nfsErrDQuota;
StatsRatePerS      nfsErrDQuotaRate;
StatsCount32        nfsErrStale;
StatsRatePerS      nfsErrStaleRate;
StatsCount32        nfsErrWFlush;
StatsRatePerS      nfsErrWFlushRate;

FBQhead_type        dialogQ;      /* Queue of StatsDialogLink types */
FBQhead_type        fileSystemQ;  /* Queue of StatsNfsFileSystemLink types */
} StatsNfsAddr;

```

Engel Appendix VI at pp. 245-248 of stats_nfs.h. As shown above, each NFS client/server record includes a pointer to a linked list of all NFS dialogs to which the client/server has participated (“dialogQ” field). Namely, when a new application-specific dialog (i.e., NFS dialog) is allocated for a given client/server pair, the new dialog record is added to the dialog link queue for each application-specific client and server record (i.e., each NFS client and server record). Engel Appendix VI at pp. 483-484 of stats_nfs_p.c; *see also* Section VII.C.7 (Paragraphs 108-110) below.

69. As shown above, each StatsNfsAddr record stores various statistics, including the following statistical objects of type StatsCount32: ops, readOps, writeOps, writeCacheOps, bytes, readBytes, writeBytes, errors, rexmts, flowCtrls, in addition to various error-related StatsCount32 statistics, and other statistics. Engel Appendix VI at pp. 245-248 of stats_nfs.h. Each of these StatsCount32 data structures stores count, running_count, and high_thld variables. *See* Paragraph 62 above.

70. Each StatsNfsAddr record also stores statistical objects of type StatsRatePerS (also called StatsRollingRate) including opRate, readOpRate, writeOpRate, writeCacheOpRate, byteRate, readByteRate, writeByteRate, errorRate, rexmtRate, flowCtrlRate, various error-related StatsRatePerS statistics, and other statistics. Engel Appendix VI at pp. 245-248 of stats_nfs.h. Each of these StatsRatePerS data structures stores a current, high, and low StatsRateValue in addition to high_thld, count, and type variables. *See* Paragraph 63 above. Each of the current, high, and low StatsRateValue data structures stores count and time variables. *See* Paragraph 63 above.

71. Each StatsNfsAddr record also stores statistical objects of type StatsBucketRate, including opBuckets, byteBuckets, errorBuckets, and rexmtBuckets, among others. Engel Appendix VI at pp. 245-248 of stats_nfs.h. The StatsBucketRate data structure is defined as follows:

```
typedef struct stats_bucket_rate_type
{
    Uint32      index;
    Uint32      rate[STATS_BUCKETS_PER_RATE]; /* 12 5-min count buckets */
    StatsRateValue max_rate[STATS_BUCKETS_PER_RATE]; /* 12 5-min max rate buckets */
    Uint32      count; /* partial count
over the bucket */
    Uint32      bucket_time; /* start time for the
count */
    StatsRateValue high; /* high rate over the current
bucket */
} StatsBucketRate;
```

Engel Appendix VI at pp. 178-179 of stats.h. Engel describes this statistics data structure as follows:

“The list of events shown in data structure 130 of FIG. 7a illustrates the type of data that is collected for this address when the parse control field bits are set to on. Some of the entries in DLL segment statistics data structure 130 are pointers to buckets for historical data. In the case where buckets are maintained, there are twelve buckets each of which represents a time period of five minutes duration and each of which generally contains two items of information, namely, a count for the corresponding five minute time period and a MAX rate for that time period. MAX rate records any spikes which have occurred during the period and which the user may not have observed because he was not viewing that particular statistic at the time.”

Engel at 18:20-32.

“The hourly rate is calculated from a sum of the last twelve 5 minute readings, as obtained from the buckets for the pertinent parameter. Each new reading replaces the oldest of the twelve values maintained. Maximum and minimum values are retained until reset by the user.”

Engel at 16:63-67. Accordingly, each StatsBucketRate metric stores a count and associated maximum for each of twelve contiguous 5-minute time intervals.

72. Each StatsNfsAddr record also stores an rpcStats statistical object of type StatsRpc. The StatsRpc data structure is defined as follows:

```

typedef struct stats_rpc_type {
    StatsCount32      versionMismatch;
    StatsRatePerS     versionMismatchRate;
    StatsCount32      authBadCred;
    StatsRatePerS     authBadCredRate;
    StatsCount32      authRejectedCred;
    StatsRatePerS     authRejectedCredRate;
    StatsCount32      authBadVerf;
    StatsRatePerS     authBadVerfRate;
    StatsCount32      authRejectedVerf;
    StatsRatePerS     authRejectedVerfRate;
    StatsCount32      authTooWeak;
    StatsRatePerS     authTooWeakRate;
    StatsCount32      authOther;
    StatsRatePerS     authOtherRate;
    StatsCount32      progUnavail;
    StatsRatePerS     progUnavailRate;
    StatsCount32      progVersionMismatch;
    StatsRatePerS     progVersionMismatchRate;
    StatsCount32      procUnavail;
    StatsRatePerS     procUnavailRate;
    StatsCount32      procGarbageArgs;
    StatsRatePerS     procGarbageArgsRate;
} StatsRpc;

```

Engel Appendix VI at 263 of stats.h. As discussed further in Paragraph 119, 125, and 127 below, these statistics are error related statistics of type StatsCount32 and StatsRatePerS.

C. Real Time Parse Routine

73. Upon receipt by the network monitor, a frame (i.e., packet) is first placed on a lan_rcv_frame_q buffer. When a frame is queued on the lan_rcv_frame_q buffer, the kernel main loop calls the real time parse (“rtp”) routine. Engel Appendix VI at p. 167-170 of rtp.c. For each frame to be parsed, the rtp routine calls the data link layer parse routine (rtp_dll_parse), passing as inputs to the routine a pointer to the frame and the length of the frame. Engel Appendix VI at p. 168 of rtp.c, p. 8 of rtp_dll_p.c.

1. Data Link Layer Parse Routine

74. As an initial matter, I note that the `rtp_dll_p.c` file handles both Ethernet and IEEE packets (as discussed above in Paragraphs 46-48) and the ‘725 Patent contemplates a single file for handling both types of packets. EX1033, ‘725 patent at 42:54-60 (“In an alternate embodiment, the Ethernet selection definition may be in the same Ethernet file 1911.”), 56:29 (“IEEE8032.pdl and IEEE8033.pdl (ethertype files)”). As provided in the `rtp_dll_p.c` source code file, the `rtp_dll_parse` routine initially extracts the MAC destination and source addresses and also the value Length/Type Field in the Ethernet header and stores them in the nested `mac_header` data structure of a local “`frame_ptr`” data structure. Engel Appendix VI at p. 8 (initializing `frame_ptr` data structure), p. 5 (defining `dll_framestr` data structure as including, in part, `mac_header` data structure of type `mac_headerstr`), p. 4 (defining `mac_headerstr` data structure as including `dst_addr` and `src_addr` variables each of type `MacAddress` and a length variable) of `rtp_dll_p.c`. The extracted source and destination MAC addresses from the packet header are also stored in globally accessible `MacAddress` data structures associated with the source and destination.

```
frame_ptr = (struct dll_framestr *)layer_ptr;
mac_src_addr = frame_ptr->mac_header.src_addr;
mac_dst_addr = frame_ptr->mac_header.dst_addr;
```

Engel Appendix VI at p. 8 of `rtp_dll_p.c`; see also p. 2.

75. After updating statistics objects related to the data link layer, the `rtp_dll_parse` routine determines whether the packet is Ethertype or 802.3/802.2 (i.e., IEEE) by checking the extracted value of the Length/Type Field (the “length” variable of the nested `mac_header` data structure of the `frame_ptr` data structure). As discussed in Paragraphs 46-48 above, the Length/Type Field is the two octets (also referred to as bytes) that follow the MAC destination and source addresses in the Ethernet frame header. The routine also determines that the packet is Ethertype if the extracted Length/Type Field value is greater than or equal to 1500 decimal (the value of `MAXDATA`):

```

/*
 * Check for ethernet or 802.3/802.2. Parse 802.2 if necessary.
 * Set the protocol type based on the ethernet type or the dsap.
 */
dll_length = MAC_HEADER_SIZE;
adjusted_data_length = data_length - MAC_HEADER_SIZE;

frame_ptr->mac_header.length = ntohs(frame_ptr->mac_header.length);

if ( (frame_ptr->mac_header.length >= MAXDATA)
    | ((frame_ptr->mac_header.length == XEROX_PUP)
    && (adjusted_data_length != XEROX_PUP) )
    | ((frame_ptr->mac_header.length == PUP_ADDR_TRANS)
    && (adjusted_data_length != PUP_ADDR_TRANS)))
{
    /* ethernet frame */
    dll_stats_enet;
    protocol = frame_ptr->mac_header.length;
}

```

Engel Appendix VI at pp. 14-15 of `rtp_dll_p.c`; also Engel Appendix VI at p. 2 (defining `MAXDATA` as 1500 decimal). As discussed in Paragraphs 46-48 above, this is consistent with the standard approach for determining whether a packet is Ethertype or IEEE. The routine also determines that the packet is Ethertype if the

value of the Length/Type Field is equal to the value assigned to the XEROX PUP protocol (512 decimal) and if the size of the Ethernet Data Field (determined by subtracting the MAC header length (MAC_HEADER_SIZE set to 14) from the total length of the Ethernet frame (data_length variable passed as input to rtp_dll_parse routine)) is not equal to the value assigned to the XEROX PUP protocol. The routine also determines that the packet is Ethertype if the value of the Length/Type Field is equal to the value assigned to the PUP Addr. Trans. protocol (513 decimal) and if the size of the Ethernet Data Field is not equal to the value assigned to the PUP Addr. Trans. protocol. Engel Appendix VI at pp. 14-15 of rtp_dll_p.c (above), p. 5 of rtp_dll_p.c (defining MAC_HEADER_SIZE as 14); EX1043, RFC 1010 at p. 14 (assigning XEROX PUP protocol value of 512 decimal and PUP Addr. Trans. protocol a value of 513 decimal); EX1007, Engel at 20:4-7 (citing RFC 1010 in discussing rtp_dll_parse routine).

76. If the packet is of Ethertype, a variable “protocol” is set to the value of the Ethernet Type Field (i.e., `protocol = frame_ptr->mac_header.length;`). Engel Appendix VI at pp. 14-15 of rtp_dll_p.c. In this way, the rtp_dll_p.c source code file includes the Type Field location in the packet (i.e., `frame_ptr->mac_header.length;`), where information is stored related to any child protocol of the Ethertype protocol. This “protocol” variable is used to determine which next protocol layer parse routine should be called.

77. Namely, after updating additional statistics objects related to the data link layer, the `rtp_dll_parse` routine calls the next layer's parse routine for the determined next layer protocol as set by the "protocol" variable:

```
/*
 * Parse next protocol
 */
switch (protocol)
{
    case DOD_IP:
        if (result == GOOD)
            result = rtp_ip_parse (layer_ptr + dll_length, data_length - dll_length);
        break;

    case ARP:
        if (result == GOOD)
        {
            ea = (struct ether_arp *) (layer_ptr + dll_length);
            ALIGN_LONG(ea->arp_tpa[0], dst_ip_addr.s_addr);
            if (dst_ip_addr.s_addr == myIpAddr)
                rtp_for_this_station = TRUE;
        }
        break;

    default:
        break;
}
```

Engel Appendix VI at p. 17 of `rtp_dll_p.c`. As discussed in Paragraph 47 above, the IP protocol is also referred to as "DOD IP," and is assigned a value of hexadecimal 0800 (2048 decimal). EX1043, RFC 1010 at p. 14. As set forth on p. 17 of the `rtp_dll_p.c` source code file, the `rtp_dll_parse` routine includes the IP (referred to as `DOD_IP`) and ARP child protocols, which are higher layer protocols of the Ethernet protocol for Ethertype or Ethernet II packets. I note that this is similar to the '725 patent's Ethernet.pdl file, which includes an "etherType" Field defined as including IP and ARP child protocols, among others. '725 patent at column 71 (etherType field includes "ip(0x0800)" and "arp(0x0806)"), column 72 (CHILDREN attribute for Ethertype packets defined as

“DESTINATION=EtherType”), 49:45-55 (CHILDREN attribute “describes how children protocols are determined”).

78. If the next layer protocol is DOD_IP (IP), the next layer’s parse routine is rtp_ip_parse and the rtp_dll_parse routine determines the start of the IP header by adding the dll_length variable to the pointer set at the start of the Ethernet frame. Engel Appendix VI at p. 17 of rtp_dll_p.c. In the case of an Ethertype packet, the dll_length variable is set to 14, which is the well-known length of the Ethernet II frame header, as discussed above in Paragraphs 46-48 and as disclosed in the ‘725 patent. Engel Appendix VI at p. 14 (dll_length = MAC_HEADER_SIZE), p. 5 (defining MAC_HEADER_SIZE as 14) of rtp_dll_p.c; ‘725 patent at column 73 (“the total length of the header for this ty[p]e of packet is fixed and at 14 bytes or octets in length”), 33:53-54 (“For the Ethertype packet, the start of the next layer header 14 byte from the start of the frame.”). In this way, the Engel rtp_dll_p.c file includes the Data Field location in the packet (`layer_ptr + dll_length`), where information is stored related to the IP child protocol of the Ethernet protocol. Accordingly, the inputs to the rtp_ip_parse routine are a pointer to the start of the IP datagram and the length of the IP datagram. Engel Appendix VI at p. 17 of rtp_dll_p.c, p. 48 of rtp_ip_p.c.

2. Internet Protocol Parse Routine

79. As provided in the `rtp_ip_p.c` source code file, the `rtp_ip_parse` routine initially extracts data from the IP datagram header, including the IHL (header length) Field (discussed in Paragraph 50 above), converts the header length (in 32-bit words) to the length in bytes, and stores it as a local variable called “`ip_hdr_length`” (`ip_hdr_length = datagram_ptr->ip_hl << 2;`). Engel Appendix VI at p. 48 of `rtp_ip_p.c`. In this way, the `rtp_ip_p.c` source code file includes the IHL Field location in the packet (i.e., `datagram_ptr->ip_hl`), where information is stored related to any child protocol of the IP protocol.

80. The `rtp_ip_parse` routine also extracts information from the IP datagram header, including the value of the Protocol Field (discussed in Paragraph 49 above) and stores it as a local variable called “`protocol`”

```
protocol = (UInt32)datagram_ptr->ip_p;
```

Engel Appendix VI at p. 49 of `rtp_ip_p.c`. In this way, the `rtp_ip_p.c` source code file includes the Protocol Field location in the packet (i.e., `(UInt32)datagram_ptr->ip_p;`), where information is stored related to any child protocol of the IP protocol. As discussed, below, this “`protocol`” variable is used to determine which next layer protocol parse routine should be called. The `rtp_ip_parse` routine also extracts information from the IP datagram header

including the source and destination IP addresses and stores them as global variables.

```
datagram_ptr = (struct ip *) layer_ptr;
ip_hdr_length = datagram_ptr->ip_hl << 2;
protocol = (Uint32)datagram_ptr->ip_p;

rtp_net_broadcast = FALSE;
rtp_net_multicast = FALSE;

/*
 * For alignment reasons, the ip addresses cannot be defined as long within
 * the frame structure. Since it's easier to handle them as long, convert them.
 */
ip_src_addr = (datagram_ptr->ip_src[0] << 24) + (datagram_ptr->ip_src[1] << 16) +
              (datagram_ptr->ip_src[2] << 8) + datagram_ptr->ip_src[3];
ip_dst_addr = (datagram_ptr->ip_dst[0] << 24) + (datagram_ptr->ip_dst[1] << 16) +
              (datagram_ptr->ip_dst[2] << 8) + datagram_ptr->ip_dst[3];
```

Engel Appendix VI at pp. 48-49 of rtp_ip_p.c; see also p. 44.

81. After updating various statistics objects related to the Internet protocol layer, the rtp_ip_parse routine calls the next layer's parse routine for the determined next layer protocol as set by the "protocol" variable:

```
/*
 * Parse the next protocol
 */
switch (protocol)
{
    case ICMP_PROTOCOL:
        result = rtp_icmp_parse (layer_ptr + ip_hdr_length,
                                datagram_ptr->ip_len - ip_hdr_length);
        break;

    case UDP_PROTOCOL:
        result = rtp_udp_parse (layer_ptr + ip_hdr_length,
                                datagram_ptr->ip_len - ip_hdr_length);
        break;

    case TCP_PROTOCOL:
        result = rtp_tcp_parse (layer_ptr + ip_hdr_length,
                                datagram_ptr->ip_len - ip_hdr_length);
        break;
```

Engel Appendix VI at pp. 56-57 of rtp_ip_p.c. As set forth on pp. 56-57 of the rtp_ip_p.c source code file, the rtp_ip_parse routine includes the ICMP, UDP, and

TCP child protocols, which are higher layer protocols of the Internet protocol. I note that the '725 patent's IP.pdl file similarly uses the Protocol Field to determine the next layer protocol and discloses the ICMP, UDP, and TCP protocols as being child protocols of the Internet protocol. '725 patent at column 79 (CHILDREN attribute defined as "DESTINATION=Protocol"), 49:45-55 (CHILDREN attribute "describes how children protocols are determined"), Fig. 17B item 1752 (ICMP=1, TCP=6, UDP=17), 34:2-3 ("The possible children of the IP protocol are shown in table 1752.").

82. If the next layer protocol is TCP_PROTOCOL, the next layer's parse routine is rtp_tcp_parse and the inputs to the routine are a pointer to the start of the TCP segment and the length of the TCP segment. Engel Appendix VI at pp. 56-57 of rtp_ip_p.c, p. 123 of rtp_tcp_p.c. If the next layer protocol is UDP_PROTOCOL, the next layer's parse routine is rtp_udp_parse and the inputs to the routine are a pointer to the start of the UDP segment and the length of the UDP segment. Engel Appendix VI at p. 56 of rtp_ip_p.c, p. 156 of rtp_udp_p.c. If the next layer protocol is ICMP_PROTOCOL, the next layer's parse routine is rtp_icmp_parse and the inputs to the routine are a pointer to the start of the ICMP message and the length of the ICMP message. Engel Appendix VI at p. 56 of rtp_ip_p.c, p. 32 of rtp_icmp_p.c.

83. For each of these protocols (TCP, UDP, and ICMP), the `rtp_ip_parse` routine determines the start of the next layer header by adding the `ip_hdr_length` variable (the length of the header in bytes) to the pointer set at the start of the IP datagram:

```
layer_ptr + ip_hdr_length,
```

Engel Appendix VI at pp. 56-57 of `rtp_ip_p.c`. The ‘725 patent discloses a similar procedure for determining the start of the next layer protocol. ‘725 patent at Fig. 17B (“L4 Offset = L3 + (IHL/4)”). In this way, the Engel `rtp_ip_p.c` file includes the Data Field location in the packet (`layer_ptr + ip_hdr_length,`), where information is stored related to the ICMP, UCP, and TCP child protocols of the Internet protocol.

3. Transmission Control Protocol Parse Routine

84. As provided in the `rtp_tcp_p.c` source code file, the `rtp_tcp_parse` routine initially extracts data from the TCP segment header, including the Data Offset Field (discussed in Paragraph 52 above), converts the header length (in 32-bit words) to the length in bytes, and stores it as a local variable called “`tcp_hdr_length`”:

```
tcp_ptr = (struct tcp_header *) rtp_ptr; /* Number of 32-bit words in hdr */  
tcp_hdr_length = tcp_ptr->th_off << 2;
```

Engel Appendix VI at p. 124 of `rtp_tcp_p.c`. The Data Offset Field indicates where the data begins, which is where the next layer/child protocol data begins. In

this way, the `rtp_tcp_p.c` source code file includes the Data Offset Field location in the packet (i.e., `tcp_ptr->th_off`), where information is stored related to any child protocol of the TCP protocol. The `rtp_tcp_parse` routine also extracts information from the TCP segment header including the source and destination ports, among other fields, and stores them as global variables.

```
tcp_ptr = (struct tcphdr *) layer_ptr;
tcp_hdr_length = tcp_ptr->th_off << 2;          /* Number of 32-bit words in hdr */
tcp_flags = tcp_ptr->th_flags;

/*
 * Convert fields to local representation.
 * Don't bother with checksum or urgent ptr.
 */
tcp_src_port = ntohs (tcp_ptr->th_sport);
tcp_dst_port = ntohs (tcp_ptr->th_dport);
tcp_window   = ntohs (tcp_ptr->th_win);

/*
 * For alignment reasons, the tcp sequence number and ack number
 * as well as the ip source and destination addresses cannot be
 * defined as longs within the frame structure. The ip addresses
 * have already been converted. Convert the seq and ack numbers now.
 */
tcp_seq = (tcp_ptr->th_seq[0] << 24) + (tcp_ptr->th_seq[1] << 16) +
          (tcp_ptr->th_seq[2] << 8)  + tcp_ptr->th_seq[3];

tcp_ack = (tcp_ptr->th_ack[0] << 24) + (tcp_ptr->th_ack[1] << 16) +
          (tcp_ptr->th_ack[2] << 8)  + tcp_ptr->th_ack[3];
```

Engel Appendix VI at p. 124 of `rtp_tcp_p.c`; see also pp. 111-112 (setting up “Global TCP data structures” including `tcp_src_port` and `tcp_dst_port`), Paragraphs 51-52 above.

85. The `rtp_tcp_parse` routine next determines whether there is a port to application program (also called a protocol) mapping for the port being used for the TCP communication by calling the `stats_port_to_protocol` routine and passing

as inputs a pointer to the source/destination IP address data structure and the source/destination port.

```
/*
 * Determine protocol.  If neither port is known, lump these stats in with the other
 * unknown port stats.
 */
tcp_protocol = stats_port_to_protocol (ip_dst_node_addr_ptr, tcp_ptr->th_dport,
TCP_PROTOCOL);
if (tcp_protocol == UNKNOWN_PORT)
    tcp_protocol = stats_port_to_protocol (ip_src_node_addr_ptr, tcp_ptr->th_sport,
TCP_PROTOCOL);
protocol = tcp_protocol;
```

Engel Appendix VI at p. 124 of rtp_tcp_p.c.

86. When called, the stats_port_to_protocol routine first determines if the port is well known by calling the stats_well_known_port routine and passing the port as an input.

```
/*
 *
 * Translate a port number to a protocol number
 */
Uint32 stats_port_to_protocol (ip_addr_ptr, port, transport_protocol)
    StatsAddrEntry      *ip_addr_ptr;
    Uint32               port, transport_protocol;
{
    register StatsPmapping *pmapping_ptr;

    if (stats_well_known_port (port) )
        return (port);

    pmapping_ptr = stats_pmap_lookup (ip_addr_ptr, port, transport_protocol);
    if (pmapping_ptr)
        return (pmapping_ptr->protocol);

    return (UNKNOWN_PORT);
}
```

Engel Appendix VI at pp. 632-633 of stats_p.c. If the port is in the statsWellKnownPorts port table, the port number is returned. Engel Appendix VI

at p. 644 of stats_p.c (defining stats_well_known_port routine as determining if the port in question is an entry in the statsWellKnownPorts data structure), pp. 632-633 of stats_p.c (stats_port_to_protocol routine returns the port if it is in the statsWellKnownPorts data structure), p. 183 of stats.h (defining statsWellKnownPorts data structure). If not, the stats_well_known_port routine calls the stats_pmap_lookup routine, passing as inputs a pointer to the source/destination IP address data structure, the source/destination port, and the transport_protocol (i.e., TCP). Engel Appendix VI at pp. 632-633 of stats_p.c.

87. The stats_pmap_lookup routine walks the linked list of StatsPmapping records for the source/destination IP address record to determine if the port used for the packet communication and the transport_protocol match an entry of the list. If there is a match, a pointer to the associated StatsPmapping record is returned to the stats_port_to_protocol routine.

```

/*****
 *
 * Given an ip address structure, walk the port map queue. If the port
 * is found, return the pointer to the structure, otherwise return NULL.
 */
StatsPmapping *stats_pmap_lookup (ip_addr_ptr, port, transport_protocol)

    StatsAddrEntry          *ip_addr_ptr;
    Uint32                  port, transport_protocol;

{
    if (ip_addr_ptr != NULL)
    {
        if (ip_addr_ptr->protocol_specific_ptr != NULL)
        {
            pmapQ_ptr = (FBQhead_type*) ip_addr_ptr->protocol_specific_ptr;
            pmap_ptr = (StatsPmapping *) pmapQ_ptr->pFlink;

```



```

        while (pmap_ptr != NULL)
        {
            if ( (pmap_ptr->port == port) &&
                (pmap_ptr->transport_protocol == transport_protocol) )
                return (pmap_ptr);
            pmap_ptr = (StatsPmapping *) pmap_ptr->link.pFlink;
        }
    }
    return (NULL);
}

```

Engel Appendix VI at pp. 507-508 of stats_pmap_p.c. Each StatsPmapping record defines a known port-to-application program association for the source/destination IP address in question.

```

/*
 * Format of structure used to keep a port to program mapping
 * These are placed in a queue associated with the ip address.
 * The queue is pointed to from protocol_specific_ptr in
 * the StatsAddrEntry type.
 */

typedef struct pmapping_type {
    FBQentry_type link;

    Uint32      port;
    Uint32      protocol;
    Uint32      program;
    Uint32      version;
    Uint32      transport_protocol;
} StatsPmapping;

```

Engel Appendix VI at pp. 260-261 of stats_pmap.h.

88. In this way, the stats_port_to_protocol routine determines whether there is a known port to application program association for the source/destination IP address in question. The stats_port_to_protocol routine returns the “protocol” identifier of the associated application program (e.g., “FTP_PORT,” “TELNET_PORT,” “SMTP_PORT,” “MOUNT_PORT,” “NFS_PORT,” “MAPPER_PORT”) if there is a match. If there isn’t a match,

“UNKNOWN_PORT” is returned. Engel Appendix VI at pp. 632-633 of stats_p.c, p. 132 of rtp_tcp_p.c (listing exemplary known application identifiers). The rtp_tcp_parse routine stores the returned protocol identifier/port as the variables “tcp_protocol” and “protocol.” Engel Appendix VI at p. 124 of rtp_tcp_p.c. As discussed, below, this “tcp_protocol” variable is used to determine which next layer protocol parse routine should be called.

89. Because the rtp_tcp_parse routine of the rtp_tcp_p.c source code file uses the extracted source/destination port to determine the next layer protocol’s parse routine, the rtp_tcp_p.c source code file includes the Source/Destination Port Field locations in the packet (i.e., `tcp_ptr->th_dport`, and `tcp_ptr->th_sport`), where information is stored related to any child protocol of the TCP protocol:

```
*/  
tcp_protocol = stats_port_to_protocol (ip_dst_node_addr_ptr, tcp_ptr->th_dport,  
TCP_PROTOCOL);  
if (tcp_protocol == UNKNOWN_PORT)  
    tcp_protocol = stats_port_to_protocol (ip_src_node_addr_ptr, tcp_ptr->th_sport,  
TCP_PROTOCOL);  
protocol = tcp_protocol;
```

Engel Appendix VI at p. 124 of rtp_tcp_p.c; see also p. 124.

90. After updating various statistics related to the TCP layer, the rtp_tcp_parse routine calls the respective next layer’s parse routine based on the program identified from the port to program map and set as the “tcp_protocol” variable:

```

/*
 * Parse the next layer
 */
switch (tcp_protocol)
{
    case FTP_PORT:
        /* result = rtp_ftp_parse (layer_ptr + tcp_hdr_length, data_length,
        TCP_PROTOCOL); */
        break;

    case TELNET_PORT:
        /* result = rtp_telnet_parse (layer_ptr + tcp_hdr_length, data_length,
        TCP_PROTOCOL); */
        break;

    case SMTP_PORT:
        /* result = rtp_smtp_parse (layer_ptr + tcp_hdr_length, data_length,
        TCP_PROTOCOL); */
        break;

    case MOUNT_PORT:
    case NFS_PORT:
    case MAPPER_PORT:
        result = rtp_rpc_parse (layer_ptr + tcp_hdr_length, data_length, TCP_PROTOCOL);
        break;

    default:
        break;
}

```

Engel Appendix VI at pp. 131-132 of rtp_tcp_p.c. As set forth on pp. 131-132 of the rtp_tcp_p.c source code file, the rtp_tcp_parse routine includes the parse routines for the FTP, Telnet, SMTP, RPC (PortMapper), NFS Mount, and NFS child protocols, which are higher layer protocols of the TCP protocol. I note that the ‘725 patent’s TCP.pdl file similarly uses the Source/Destination Port to determine the next layer protocol. ‘725 patent at column 81 (“tcpPort” field described as “the 16 bit field where the child protocol is located for the next layer beyond TCP”; defining SrcPort=tcpPort and DestPort=tcpPort; CHILDREN attribute defined as “DESTINATION=DestPort, SOURCE=Srcport”), 49:45-55 (CHILDREN attribute “describes how children protocols are determined”).

91. As set forth on p. 132 of rtp_tcp_p.c, if the program identified is FTP, the next layer's parse routine is rtp_ftp_parse. If the program identified is Telnet, the next layer's parse routine is rtp_telnet_parse. If the program identified is SMTP, the next layer's parse routine is rtp_smtp_parse. Finally, if the program identified is PortMapper, NFS, or NFS Mount, the next layer's parse routine is rtp_rpc_parse. Engel Appendix VI at p. 132 of rtp_tcp_p.c. The inputs to each of these parse routines is a pointer to the start of the application layer, the length of the data, and TCP_PROTOCOL. Engel Appendix VI at p. 132 of rtp_tcp_p.c; see also p. 130 (setting data_length = length – tcp_hdr_length).

92. For each of these protocols (FTP, Telnet, SMTP, and RPC), the rtp_tcp_parse routine determines the start of the next layer by adding the tcp_hdr_length variable (the length of the TCP header in bytes) to the pointer set at the start of the TCP segment:

```
(layer_ptr + tcp_hdr_length,
```

Engel Appendix VI at p. 132 of rtp_tcp_p.c. In this way, the Engel rtp_tcp_p.c file includes the Data Field location in the packet ((layer_ptr + tcp_hdr_length)), where information is stored related to the FTP, Telnet, SMTP, RPC (PortMapper), NFS Mount, and NFS child protocols of the TCP protocol.

4. User Datagram Protocol Parse Routine

93. As provided in the rtp_udp_p.c source code file, the rtp_udp_parse routine initially extracts data from the UDP segment header, including the source and destination ports, among other fields, and stores them as global variables. Engel Appendix VI at p. 156. The rtp_udp_parse routine also extracts information from the UDP segment header including the Length Field (discussed in Paragraph 53 above):

```
udp_ptr = (struct udphdr *) layer_ptr;

/*
 * Convert fields to local representation.
 * Don't bother with checksum or urgent ptr.
 */
udp_src_port = ntohs (udp_ptr->uh_sport);
udp_dst_port = ntohs (udp_ptr->uh_dport);
data_length = ntohs (udp_ptr->uh_ulen) - sizeof(struct udphdr);
```

Engel Appendix VI at p. 156 of rtp_udp_p.c; see also p. 152 (setting up “Global UDP data structures” including udp_src_port and udp_dst_port).

94. The rtp_udp_parse routine next determines whether there is a port to program mapping for the port being used for the UDP communication by calling the stats_port_to_protocol routine and passing as inputs the source or destination IP address and port, as discussed in Paragraphs 85-89 above.

```
* Determine protocol. If neither port is known, lump these stats in with the
* other unknown port stats.
*/
udp_protocol = stats_port_to_protocol (ip_dst_node_addr_ptr, udp_dst_port, UDP_PROTOCOL);

if (udp_protocol == UNKNOWN_PORT)
    udp_protocol = stats_port_to_protocol (ip_src_node_addr_ptr, udp_src_port,
UDP_PROTOCOL);
protocol = udp_protocol;
```

Engel Appendix VI at p. 157 of rtp_udp_p.c; see also pp. 260-261 of stats_pmap.h, pp. 632-633 of stats_p.c, pp. 507-508 of stats_pmap_p.c. Because the rtp_udp_parse routine of the rtp_udp_p.c source code file uses the extracted source/destination port to determine the next layer protocol's parse routine, the rtp_udp_p.c source code file includes the Source/Destination Field locations in the packet (i.e., (udp_ptr->uh_sport) and (udp_ptr->uh_dport)), where information is stored related to any child protocol of the UDP protocol. Engel Appendix VI at pp. 156-157 of rtp_udp_p.c.

95. After updating statistics objects related to the UDP layer, rtp_udp_parse calls the respective next layer's parse routine based on the program identified from the port to program map and set as the "udp_protocol" variable:

```
/*
 * Parse the next layer
 */
switch (udp_protocol)
{
    case MAPPER_PORT:
    case MOUNT_PORT:
    case NFS_PORT:
        result = rtp_rpc_parse (layer_ptr + sizeof(struct udphdr), data_length,
                                udp_protocol, UDP_PROTOCOL);
        break;
    default:
        break;
}
```

Engel Appendix VI at pp. 162-163 of rtp_udp_p.c. As set forth on pp. 162-163 of rtp_udp_p.c source code file, the rtp_udp_parse routine includes the parse routine (rtp_rpc_parse) for the RPC (PortMapper), NFS Mount, and NFS child protocols, which are higher layer protocols of the UDP protocol. I note that the '725 patent's

UDP.pdl file similarly uses the Source/Destination Port to determine the next layer protocol. ‘725 patent at column 83 (CHILDREN attribute defined as “DESTINATION=DestPort, SOURCE=Srcport”), 49:45-55 (CHILDREN attribute “describes how children protocols are determined”). As set forth on p. 163 of rtp_udp_p.c, if the program identified is PortMapper, NFS, or NFS Mount, the next layer’s parse routine is rtp_rpc_parse. Engel Appendix VI at p. 163 of rtp_tcp_p.c. The input to the rtp_rpc_parse routine is a pointer to the start of the application layer, the length of the data, the determined application program (i.e., PortMapper, NFS, or NFS Mount), and UDP_PROTOCOL. Engel Appendix VI at p. 163 of rtp_udp_p.c.

96. The rtp_udp_parse routine determines the start of the next layer by adding the size of the UDP header to the pointer set at the start of the UDP segment:

```
layer_ptr + sizeof(struct udphdr),  
UDP_PROTOCOL;
```

Engel Appendix VI at p. 163 of rtp_udp_p.c. In this way, the Engel rtp_udp_p.c file includes the Data Field location in the packet, where information is stored related to the child protocols of the UDP protocol.

5. Remote Procedure Call Parse Routine

97. The Remote Procedure Call parse routine (rtp_rpc_parse) extracts information from the packet, including unique transaction ID (“xid”) and the call

direction (i.e., whether it is a call or a reply). Engel Appendix VI at p. 104 of rtp_rpc_p.c. If the communication is a call, the rtp_rpc_parse routine additionally extracts the protocol, program, version, and transport_protocol values from the packet and stores the values as elements of the rpcInfo data structure based.

```
/*
 * Align all 32-bit integers to avoid a bus error
 */
rpc_info_ptr->xid      = ALIGN(rpc_frame_ptr->xid);
rpc_info_ptr->direction = ALIGN(rpc_frame_ptr->direction);

/*
 * Set up all field in the rpcInfo structure. The next level parse
 * routine will actually handle most of the fields.
 */
switch (rpc_info_ptr->direction)
{
    case RPC_CALL:
        rpc_info_ptr->call_version = ALIGN(rpc_frame_ptr->rpc.call.rpcvers);
        rpc_info_ptr->call_prog    = ALIGN(rpc_frame_ptr->rpc.call.prog);
        rpc_info_ptr->call_vers    = ALIGN(rpc_frame_ptr->rpc.call.vers);
        rpc_info_ptr->call_proc    = ALIGN(rpc_frame_ptr->rpc.call.proc);
        rpc_info_ptr->cred_length  = ALIGN(rpc_frame_ptr->rpc.call.cred.length);
        rpc_info_ptr->cred_ptr     = &rpc_frame_ptr->rpc.call.cred;
```

Engel Appendix VI at p. 104 of rtp_rpc_p.c. After assigning various values to other elements of the rpcInfo data structure based on whether the communication is a call or a reply and the contents of the packet, the rtp_rpc_parse routine hands off the packet to the next layer's parse routine, as determined by the transport layer parse routine (e.g., TCP or UDP).


```

/*
 * Hand off the message
 */
rpc_hand_off:
switch (protocol)
{
case MAPPER_PORT:
    result = rtp_pmap_parse (layer_ptr, length);
    break;

case NFS_PORT:
case MOUNT_PORT:
    result = rtp_nfs_parse (layer_ptr, length, protocol, transfer_protocol);
    break;

default:
    break;
}

```

Engel Appendix VI at p. 107 of rtp_rpc_p.c.

98. As set forth on p. 107 of rtp_rpc_p.c source code file, the rtp_rpc_parse routine includes the parse routine for the PortMapper and NFS child protocols, which are higher layer protocols of the RPC protocol. If the program identified is PortMapper, the next layer's parse routine is rtp_pmap_parse. Engel Appendix VI at p. 107 of rtp_rpc_p.c. If the program identified is NFS or NFS Mount, the next layer's parse routine is rtp_nfs_parse. Engel Appendix VI at p. 107 of rtp_rpc_p.c. The input to the rtp_pmap_parse routine is a pointer to the start of the application layer. Engel Appendix VI at p. 107 of rtp_rpc_p.c. The inputs to the rtp_nfs_parse routine is a pointer to the start of the application layer, the length of the data, the determined application program (i.e., PortMapper, NFS, or NFS Mount), and the transport layer protocol (e.g., TCP_PROTOCOL or UDP_PROTOCOL). Engel Appendix VI at p. 107 of rtp_rpc_p.c. Accordingly, the rtp_rpc_parse routine passes to the next layer parse routine, the start of the

application layer as determined by the transport layer parse routine (e.g., `rtp_tcp_parse` or `rtp_udp_parse`). Engel Appendix VI at p. 107 of `rtp_rpc_p.c`.

6. Portmap Parse Routine

99. The `rtp_pmap_parse` routine first determines the direction of the communication (i.e., whether it is a call or a reply). Engel Appendix VI at p. 95 of `rtp_pmap_p.c`. If the communication is a call, the `rtp_pmap_parse` routine saves the event in a nested history data structure of a `pmapData` data structure, including a “status” of “`PMAP_OUTSTANDING`”, the call procedure, source IP address, and `xid`. Engel Appendix VI at pp. 95-96 of `rtp_pmap_p.c`. If the call is for a “`GETPORT`”, “`SET`”, or “`UNSET`” request, `rtp_pmap_parse` further stores the “`program_number`”, “`version`”, and “`transport_protocol`” in the history data structure as they appear in the `pmap` protocol portion of the received packet. Engel Appendix VI at p. 96 of `rtp_pmap_p.c`.

100. If the communication is a reply, the `rtp_pmap_parse` routine walks back through the history entries stored in the nested history data structure of the `pmapData` data structure to determine if the destination IP address and the `xid` of the instant packet matches that of a call in the history data structure. If `rtp_pmap_parse` finds such an entry in the history data structure, the status is changed to “`PMAP_ANSWERED`”. Engel Appendix VI at pp. 96-97 of

rtp_pmap_p.c. Where the status is updated, the reply is processed for a respective “GET”, “SET”, or “UNSET” request as detailed below.

101. For a successful reply to a “GET” request, rtp_pmap_parse determines whether the program number corresponds to a program that is being monitored (stats_program routine). Engel Appendix VI at p. 98 of rtp_pmap_p.c. If so, rtp_pmap_parse calls the stats_pmap_lookup routine to determine whether the port and program have already been recorded for the specific source IP address and, if so, returns a pointer to the StatsPmapping data structure for the port/program. Engel Appendix VI at p. 98 of rtp_pmap_p.c, pp. 507-508 of stats_pmap_p.c, p. 260-261 of stats_pmap.h. If the port/program has not been recorded, rtp_pmap_parse calls the stats_pmap_get routine in order to attempt to allocate a StatsPmapping data structure for the new port/program pair. Engel Appendix VI at pp. 98-99 of rtp_pmap_p.c, pp. 508-509 of stats_pmap_p.c, p. 260-261 of stats_pmap.h. Once a new StatsPmapping data structure has been allocated, the rtp_pmap_parse routine stores the port, protocol, program, version, and transport_protocol values as found in the packet in the new data structure. Engel Appendix VI at 98-99 of rtp_pmap_p.c.

102. For a successful reply to a “SET” request, the same procedure is followed as with the “GET” request, except rtp_pmap_parse calls the stats_pmap_lookup routine to determine whether the port and program have

already been recorded for the specific destination IP address (i.e., the node that sourced the SET request) and, if so, returns a pointer to the StatsPmapping data structure for the port/program. Engel Appendix VI at pp. 99-100 of rtp_pmap_p.c.

103. For a successful reply to an “UNSET” request, rtp_pmap_parse removes the mapping of the specific destination IP address to the StatsPmapping data structure for the port/program and deallocates the associated StatsPmapping data structure. Engel Appendix VI at p. 100 of rtp_pmap_p.c, p. 509 of stats_pmap_p.c.

7. Network File System Parse Routine

104. The rtp_nfs_parse routine first finds or allocates the relevant statistics records for the client, server, and dialog. The rtp_nfs_parse routine calls the stats_nfs_get_client routine in order to find or allocate an NFS client StatsAddrEntry record given the source IP address. Engel Appendix VI at pp. 72-73 of rtp_nfs_p.c, p. 472 of stats_nfs_p.c. The stats_nfs_get_client routine, in turn calls the stats_nfs_lookup_client routine, which finds an NFS client StatsAddrEntry record given the source IP address or returns NULL, if a record for the NFS client is not found. Engel Appendix VI at p. 473 and pp. 467-473 of stats_nfs_p.c. In determining whether an NFS client record exists for the given IP address, the stats_nfs_lookup_client routine first computes a hash function on the IP address as follows:

```

/*
 * Compute ip address hash to get index into hash table
 */
nfs_client_hash = (*ip_addr & 0x0000ffff);
nfs_client_hash = ((nfs_client_hash + ((nfs_client_hash & 0xff00) >> 8)) &
(NFS_CLIENT_HASH_TABLE_SIZE - 1));

```

Engel Appendix VI at p. 468 of stats_nfs_p.c. The stats_nfs_lookup_client routine then uses the hash function as an index to the nfs_client_hash_table, thereby setting up a pointer to an indexed entry of the NFS client hash table. The stats_nfs_lookup_client routine then walks the linked list of NFS client records to determine whether the IP address matches an IP address of an entry in the database, based on a comparison of the extracted IP address with the IP address stored in each NFS client record of the NFS client hash table. Engel Appendix VI at p. 468 of stats_nfs_p.c.

105. Once the stats_nfs_lookup_client routine completes processing, processing is returned to the stats_nfs_get_client routine, which determines whether the NFS client is new or previously encountered:

```

nfs_addr_record_ptr = stats_nfs_lookup_client (ip_addr);

if (nfs_addr_record_ptr == NULL)

```

See Engel Appendix VI at p. 473 of stats_nfs_p.c. If an NFS client record doesn't already exist for the IP address, stats_nfs_get_client will attempt to allocate new StatsAddrEntry and StatsNfsAddr linked records for the client, and then will add the new client address structure to the nfs_client_hash_table at the indexed location

corresponding to the computed hash function for the IP address of the client participating in the communication. Engel Appendix VI at pp. 473-475, pp. 469-472 of stats_nfs_p.c. When the newly created StatsAddrEntry record for the NFS client is initialized, the address field is set to include the client IP address. Engel Appendix VI at pp. 473-474. The “type” variable of each of the newly created StatsNfsAddr rate statistics (e.g., opRate, rexmtRate) is set to STATS_RATE_10S, indicating a 10 second rate sample period. Engel Appendix VI at pp. 469-472 of stats_nfs_p.c; p. 75 of rtp_nfs_p.c. The stats_nfs_get_client routine will then add the new NFS client address structure to the nfs_client_hash_table at the indexed location corresponding to the computed hash function for the IP address of the client participating in the communication. Engel Appendix VI at p. 474 of stats_nfs_p.c. The nfs_client_hash_table includes an index that is the hash function computed as part of the lookup_client routine and also a pointer to the new NFS client address structure. Engel Appendix VI at p. 474 of stats_nfs_p.c.

106. The rtp_nfs_parse routine then calls the stats_nfs_get_server routine in order to find or allocate an NFS server StatsAddrEntry record given the destination IP address. Engel Appendix VI at pp. 72-73 of rtp_nfs_p.c, p. 476 of stats_nfs_p.c. The stats_nfs_get_server routine, in turn calls the stats_nfs_lookup_server routine, which finds an NFS server StatsAddrEntry record given the destination IP address or returns NULL, if a record for the NFS server is

not found. Engel Appendix VI at pp. 475-476 of stats_nfs_p.c. In determining whether an NFS server record exists for the given IP address, the stats_nfs_lookup_server routine first computes a hash function on the IP address as follows:

```
/*  
 * Compute ip address hash to get index into hash table  
 */  
nfs_server_hash = (*ip_addr & 0x0000ffff);  
nfs_server_hash = {(nfs_server_hash + {(nfs_server_hash & 0xff00} >> 8)} &  
                  (NFS_SERVER_HASH_TABLE_SIZE - 1)};
```

Engel Appendix VI at p. 475 of stats_nfs_p.c. The stats_nfs_lookup_server routine then uses the hash function as an index to the nfs_server_hash_table, thereby setting up a pointer to an indexed entry of the NFS server hash table. The stats_nfs_lookup_server routine then walks the linked list of NFS server records to determine whether the IP address matches an IP address of an entry in the database, based on a comparison of the extracted IP address with the IP address stored in each NFS server record of the NFS server hash table. Engel Appendix VI at pp. 475-476 of stats_nfs_p.c.

107. Once the stats_nfs_lookup_server routine completes processing, processing is returned to the stats_nfs_get_server routine, which determines whether the NFS server is new or previously encountered:

```
nfs_addr_ptr = stats_nfs_lookup_server (ip_addr);  
if (nfs_addr_ptr == NULL)
```

See Engel Appendix VI at p. 476 of stats_nfs_p.c. If an NFS server record doesn't already exist for the IP address, stats_nfs_get_server will attempt to allocate new StatsAddrEntry and StatsNfsAddr linked records for the server, will set the "startTime" and "lastTime" fields for the new server record to the current time, and then will add the new server address structure to the nfs_server_hash_table at the indexed location corresponding to the computed hash function for the IP address of the server participating in the communication. Engel Appendix VI at pp. 476-478, pp. 469-472 of stats_nfs_p.c. When the newly created StatsAddrEntry record for the NFS server is initialized, the address field is set to include the server IP address. Engel Appendix VI at p. 477. The "type" variable of each of the newly created StatsNfsAddr rate statistics (e.g., opRate, rexmtRate) is set to STATS_RATE_10S, indicating a 10 second rate sample period. Engel Appendix VI at pp. 469-472 of stats_nfs_p.c; p. 75 of rtp_nfs_p.c. The stats_nfs_get_server routine will then add the new NFS server address structure to the nfs_server_hash_table at the indexed location corresponding to the computed hash function for the IP address of the server participating in the communication. Engel Appendix VI at p. 478 of stats_nfs_p.c. The nfs_server_hash_table includes an index that is the hash function computed as part of the lookup_server routine and also a pointer to the new NFS server address structure. Engel Appendix VI at p. 478 of stats_nfs_p.c.

108. The `rtp_nfs_parse` routine then calls the `stats_nfs_get_dialog` routine in order to find or allocate an NFS dialog given the two IP address of the client/server participating in the communication. Engel Appendix VI at pp. 72-73 of `rtp_nfs_p.c`, p. 480 of `stats_nfs_p.c`. The `stats_nfs_get_dialog` routine, in turn calls the `stats_nfs_lookup_dialog` routine, which finds an NFS dialog given the two IP addresses or returns NULL, if a dialog is not found. Engel Appendix VI at p. 480 and pp. 478-480 of `stats_nfs_p.c`. In determining whether an NFS dialog exists given two IP addresses, the `stats_nfs_lookup_dialog` routine first computes a hash function on both IP addresses as follows:

```
/*
 * Compute hash function on both ip addresses
 */
xnfs_dialog_hash = (*server & 0xffff0000) + ((*server & 0x0000ffff) << 16) +
                  (*client & 0xffff0000) + ((*client & 0x0000ffff) << 16);
xnfs_dialog_hash = ((xnfs_dialog_hash + ((xnfs_dialog_hash & 0xff00) >> 8))
                  & (NFS_DIALOG_HASH_TABLE_SIZE - 1));
```

Engel Appendix VI at p. 479 of `stats_nfs_p.c`. The `stats_nfs_lookup_dialog` routine then uses the hash function as an index to the `nfs_dialog_hash_table`, thereby setting up a pointer to an indexed entry of the dialog hash table. The `stats_nfs_lookup_dialog` routine then walks the linked list of NFS dialog records to determine whether the IP address pair matches an IP address pair of an entry in the database, based on a comparison of the extracted IP addresses with the IP addresses stored in each dialog record of the NFS dialog hash table. Engel Appendix VI at pp. 478-480 of `stats_nfs_p.c`.

109. Once the stats_nfs_lookup_dialog routine completes processing, processing is returned to the stats_nfs_get_dialog routine, which determines whether the NFS dialog is new or previously encountered:

```
dialog_addr_ptr = stats_nfs_lookup_dialog (server, client);  
if (dialog_addr_ptr != NULL)  
    return (dialog_addr_ptr);
```

See Engel Appendix VI at p. 480 of stats_nfs_p.c. If an NFS dialog doesn't already exist for the IP address pair, stats_nfs_get_dialog will attempt to allocate new StatsAddrEntry and StatsNfsDialogEntry linked records for the dialog and will set the "startTime" and "lastTime" fields for the new dialog record to the current time. Engel Appendix VI at pp. 480-482 of stats_nfs_p.c. When the newly created StatsAddrEntry record for the dialog is initialized, the address field is set to include the source and destination IP addresses. Engel Appendix VI at p. 482 of stats_nfs_p.c. Additionally, the "type" variable of each of the newly created StatsNfsDialogEntry rate statistics (e.g., opRate, rexmtRate) is set to STATS_RATE_10S, indicating a 10 second rate sample period. Engel Appendix VI at pp. 482-483 of stats_nfs_p.c; p. 75 of rtp_nfs_p.c.

110. Next, the stats_nfs_get_dialog routine will "link the dialog statistics into the dialog queue for each nfs address stats." Engel Appendix VI at p. 483 of stats_nfs_p.c. Namely, when a new NFS dialog is allocated for a given client/server pair, the new dialog record is added to the dialog link queue for each

of the corresponding NFS client and server records. Engel Appendix VI at pp. 483-484 of stats_nfs_p.c. The stats_nfs_get_dialog routine will then add the new dialog address structure to the nfs_dialog_hash_table at the indexed location corresponding to the computed hash function for the IP address pair of the client/server participating in the dialog. Engel Appendix VI at p. 484 of stats_nfs_p.c. The nfs_dialog_hash_table includes an index that is the hash function computed as part of the lookup_dialog routine and also a pointer to the new dialog address structure. Engel Appendix VI at p. 484 of stats_nfs_p.c.

111. After finding or allocating an NFS dialog, the rtp_nfs_parse routine calls the stats_update_age_timers macro, which updates the “lastTime” field of the dialog record, the NFS client record, and the NFS server record:

```
/*
 * Macro to hit the age timer in each address structure
 */
#define stats_update_age_timers \
{ \
    if (this_seg_addr_ptr != NULL) \
        this_seg_addr_ptr->lastTime = stats_start_time.tv_sec; \
    if (src_seg_addr_ptr != NULL) \
        src_seg_addr_ptr->lastTime = stats_start_time.tv_sec; \
    if (dst_seg_addr_ptr != NULL) \
        dst_seg_addr_ptr->lastTime = stats_start_time.tv_sec; \
    if (src_node_addr_ptr != NULL) \
        src_node_addr_ptr->lastTime = stats_start_time.tv_sec; \
    if (dst_node_addr_ptr != NULL) \
        dst_node_addr_ptr->lastTime = stats_start_time.tv_sec; \
    if (dialog_addr_ptr != NULL) \
        dialog_addr_ptr->lastTime = stats_start_time.tv_sec; \
}
```

Engel Appendix VI at p. 187 of stats.h; *also* p. 75 of rtp_nfs_p.c (calling stats_update_age_timers macro). The lastTime field indicates the time of the last

seen packet for the associated flow. Engel Appendix VI at pp. 179 and 187 of stats.h; Engel at 18:6-8 (“a last_seen field 124 for the time at which the last communication was seen”), 18:63-67, Figs. 7A, 7C, 21.

112. The rtp_nfs_parse routine then calls the stats_set_rate_10s macro, which updates the em_control and seconds_start_time fields of the dialog record, the NFS client record, and the NFS server record.

```

/*
 * Macro for setting the 10 second sample rate indicator used by the event
 * manager. On seeing this set, the event manager calculates rates for
 * statistics in the structure.
 */
#define stats_set_rate_10s \
{ \
    if ( (src_node_addr_ptr != NULL) && (src_node_stats_ptr != NULL) ) \
    { \
        if ((src_node_addr_ptr->em_control & STATS_RATE_10S) == 0) \
        { \
            src_node_addr_ptr->em_control += STATS_RATE_10S; \
            src_node_addr_ptr->seconds_start_time = stats_start_time.tv_sec; \
        } \
    } \
    if ( (dst_node_addr_ptr != NULL) && (dst_node_stats_ptr != NULL) ) \
    { \
        if ((dst_node_addr_ptr->em_control & STATS_RATE_10S) == 0) \
        { \
            dst_node_addr_ptr->em_control += STATS_RATE_10S; \
            dst_node_addr_ptr->seconds_start_time = stats_start_time.tv_sec; \
        } \
    } \
    if ( (this == NFS_SERVER) && (this == NFS_SERVER) ) \
    { \
        if ((this->em_control & STATS_RATE_10S) == 0) \
        { \
            this->em_control += STATS_RATE_10S; \
            this->seconds_start_time = stats_start_time.tv_sec; \
        } \
    } \
}

...

if ( (dialog_addr_ptr != NULL) && (dialog_stats_ptr != NULL) ) \
{ \
    if ((dialog_addr_ptr->em_control & STATS_RATE_10S) == 0) \
    { \
        dialog_addr_ptr->em_control += STATS_RATE_10S; \
        dialog_addr_ptr->seconds_start_time = stats_start_time.tv_sec; \
    } \
} \
}

```

Engel Appendix VI at pp. 188-189 of stats.h; also p. 75 of rtp_nfs_p.c (calling stats_set_rate_10s macro). The em_control field (referred to in the Engel

specification as the EM_control field 118) indicates that the statistics for the associated record have changed since the event manager (EM) “last serviced the database to update rates and other variables.” Engel at 17:62-67; *see also* Engel Appendix VI at p. 75 of rtp_nfs_p.c, p. 179 of stats.h (defining StatsAddrEntry data structure). The seconds_start_time field (referred to in the Engel specification as the start_time field 120) is a time used by the event manager to perform later rate calculations. Engel Appendix VI at p. 75 of rtp_nfs_p.c; pp. 179, 188-189 of stats.h; Engel at 18:2-4, Figs. 7A, 7C, 21.

113. The rtp_nfs_parse routine next calls the nfs_stats_bytes macro, which updates the total byte count and byte rate of the NFS client record and the NFS server record. Engel Appendix VI at pp. 75 of rtp_nfs_p.c (calling nfs_stats_bytes routine), 65 of rtp_nfs_p.c (defining nfs_stats_bytes routine as calling stats_bytes_for_nodes macro); p. 193 of stats.h (defining stats_bytes_for_nodes macro as calling stats_count_bytes); pp. 628-629 of stats_p.c (defining stats_count_bytes routine). These counters effectively keep track of the total NFS bytes transferred to/from the associated node across all previously encountered dialogs and issue alarms if a certain threshold is exceeded. If the total byte count for the NFS client or server record for the time interval from the last issued alarm (i.e., measured by the running_count counter) exceeds a certain threshold, an alarm is issued to the Event Manager.

```

/*****
/*
 * Routine to update byte counters of type StatsCount32
 */
void stats_count_bytes (counter_ptr, rate_ptr, length, structure_ptr, alarm_type,
alarm_number,
user_ptr)

    StatsCount32      *counter_ptr;
    StatsRollingRate  *rate_ptr;
    Uint32             length;
    StatsAddrEntry     *structure_ptr;
    Uint32             alarm_type;
    Uint32             alarm_number;
    Byte               *user_ptr;
{
    if (structure_ptr == 0)
        return;

    counter_ptr->count = counter_ptr->count + length;
    counter_ptr->running_count = counter_ptr->running_count + length;

    if (rate_ptr != NULL)
        rate_ptr->count += length;

    /* If threshold comparison is enabled, signal */
    if (counter_ptr->high_thld != 0)
    {
        if (counter_ptr->running_count >= counter_ptr->high_thld)
        {
            stats_alarm_cntr (counter_ptr, structure_ptr,
                             alarm_type, alarm_number, (AlarmUserData *)user_ptr);
            counter_ptr->running_count = 0;
        }
    }

    return;
}

```

Engel Appendix VI at pp. 628-629 of stats_p.c. Namely, the stats_count_bytes routine increments the bytes count and running_count variables and also increments the byteRate count variable by adding the byte size of the current packet to the running totals for each NFS client and NFS server record. Engel Appendix VI at pp. 628-629 of stats_p.c. The stats_count_byte routine then tests, for each record, whether the bytes running_count variable has exceeded the high_thld variable. If so, an alarm is issued and the running_count variable is reset to 0. Engel Appendix VI at pp. 628-629 of stats_p.c.

114. The `rtp_nfs_parse` routine next updates the history of recorded events in the `StatsNfsState` data structure of the associated `StatsNfsDialogEntry` data structure. In particular, `rtp_nfs_parse` first determines the direction of the communication (i.e., whether it is a call or a reply) and if it is a call first checks to see if the packet is a retransmission and, if so, updates the `rexmts` statistics objects using the `stats_for_nodes` and `stats_for_dialog` macros. Engel Appendix VI at pp. 76-77 of `rtp_nfs_p.c` (calling `nfs_stats_rexmts`), p. 66 of `rtp_nfs_p.c` (defining `nfs_stats_rexmts` as calling `stats_for_nodes` and `stats_for_dialog` macros), pp. 198-199 of `stats.h` (defining `stats_for_dialog` macro, calling `stats_count` routine), pp. 191-192 of `stats.h` (defining `stats_for_node` macro, calling `stats_count` routine), pp. 626-627 of `stats_p.c` (defining `stats_count` routine). In particular, the `stats_for_nodes` and `stats_for_dialog` macros are used to update variables of the `rexmts` counter and `rexmtsRate` rate statistics objects for the associated NFS dialog, client, and server records. Namely, the `stats_count` routine increments the `rexmts` count and `running_count` variables and also increments the `rexmtsRate` count variable for each of the dialog, NFS client, and NFS server records.


```

/*****
 *
 * Routine to update counters of type StatsCount32
 */
void stats_count (counter_ptr, rate_ptr, structure_ptr, alarm_type, alarm_number,
user_ptr)

    StatsCount32      *counter_ptr;
    StatsRollingRate  *rate_ptr;
    StatsAddrEntry    *structure_ptr;
    Uint32            alarm_type;
    Uint32            alarm_number;
    Byte              *user_ptr;
{
    if (structure_ptr == 0)
        return;

    counter_ptr->count++;
    counter_ptr->running_count++;

    if (rate_ptr != NULL)
        rate_ptr->count++;

    /*
     * If the threshold has been exceeded, send a trap
     */
    if (counter_ptr->high_thld != 0)
    {
        if (counter_ptr->running_count >= counter_ptr->high_thld)
        {
            stats_alarm_cntr (counter_ptr, structure_ptr,
                             alarm_type, alarm_number, (AlarmUserData *)user_ptr);
            counter_ptr->running_count = 0;
        }
    }

    /*
     * Queue to Event Manager Task
     */

    return;
}

```

Engel Appendix VI at pp. 626-627 of stats_p.c. The stats_count routine then tests, for each record, whether the rexmts running_count variable has exceeded the high_thld variable. If so, an alarm is issued and the running_count variable is reset to 0. Engel Appendix VI at pp. 626-627 of stats_p.c. In this way, if the rexmts running_count of the NFS dialog, client, or server for the time interval from the last issued alarm (i.e., when the running_count variable was last set to 0) exceeds a

certain threshold, an alarm is issued to the Event Manager. Engel Appendix VI at pp. 626-627 of stats_p.c.

115. US Patent No. 6,839,751 (“the ‘751 patent,” EX1002), which is incorporated by reference into the ‘725 patent (EX1033, ‘725 patent at 1:22-27) discloses a CSConnectionRetransmissions metric. The rexmts statistics object, like the ‘751 Patent CSConnectionRetransmissions metric, stores information about the transport-level connection health for a given application (e.g., NFS) and both a specific Client-Server Pair (rexmts metric of the NFS dialog record) and a specific Server and all of its clients (rexmts metric of the NFS server record). Namely, like the ‘751 Patent CSConnectionRetransmission metric, the rexmts metric stores a count of the number retransmission call packets (i.e., the number of transport, data-bearing packet retransmissions from the Client(s) to the Server). *Compare* ‘751 Patent at 46:17-39 *with* discussion in Paragraphs 62, 69 and 114 above.

116. If the rtp_nfs_parse determines that the communication is a call and not a retransmission, the event is recorded in the history StatsNfsHistoryEntry array of the dialog and the status is set to “NFS_OUTSTANDING.” Engel Appendix VI at pp. 75, 76-77 of rtp_nfs_p.c; *see also* p. 248 of stats_nfs.h (defining StatsNfsState and StatsNfsHistoryEntry data structures). If the communication is determined to be a reply, rtp_nfs_parse searches the StatsNfsHistoryEntry data structure for the instant dialog to determine whether the

unique transaction id (“xid”) in the packet for this reply matches that of a call in the history and if the status for that call is “NFS_OUTSTANDING.” If both conditions are met, the status is changed to “NFS_ANSWERED.” Engel Appendix VI at p. 77 of rtp_nfs_p.c. As discussed in Paragraph 64 above, the history array size is set to 8 entries. When the number of call events, not including retransmission, exceeds 8, rtp_nfs_parse will begin overwriting previously recorded events. Engel Appendix VI at p. 77 of rtp_nfs_p.c.

117. If the communication is for a mount protocol (“MOUNT_PORT”) and the communication is a call, the rtp_nfs_parse routine updates the ops and opRate statistics objects using the stats_for_nodes and stats_for_dialog macros. Engel Appendix VI at p. 78 of rtp_nfs_p.c (calling nfs_stats_ops), p. 64 of rtp_nfs_p.c (defining nfs_stats_ops as calling stats_for_nodes and stats_for_dialog macros), pp. 198-199 of stats.h (defining stats_for_dialog macro, calling stats_count routine), pp. 191-192 of stats.h (defining stats_for_node macro, calling stats_count routine), pp. 626-627 of stats_p.c (defining stats_count routine). In particular, the stats_for_nodes and stats_for_dialog macros discussed above are used to update variables of the ops count and opRate rate statistics objects for the associated NFS dialog, client, and server records. Accordingly, if the ops running_count variable of the NFS dialog, client, or server for the time interval from the last issued alarm (i.e., when the running_count variable was last set to 0) exceeds a certain

threshold, an alarm is issued to the Event Manager. Engel Appendix VI at pp. 626-627 of stats_p.c.

118. If the request is to mount a file, rtp_nfs_parse next calls the stats_nfs_get_file_system routine to link the file to the client requesting to mount it and then stores a pointer to the file in the StatsNfsHistoryEntry data structure for the dialog. Engel Appendix VI at p. 78 of rtp_nfs_p.c; *see also* pp. 489-493 of stats_nfs_p.c (stats_nfs_get_file_system routine). If the request is to unmount a file, rtp_nfs_parse calls the stats_nfs_lookup_file_system routine to look up the file and then stores a pointer to the file in the StatsNfsHistoryEntry data structure for the dialog. Engel Appendix VI at p. 78 of rtp_nfs_p.c; *see also* pp. 485-487 of stats_nfs_p.c (stats_nfs_lookup_file_system routine).

119. If the communication is for a mount protocol reply message and if the reply indicates that an error occurred (e.g., the program is unavailable, the message was denied due to bad credentials, the mount failed), the rtp_nfs_parse routine updates various statistics objects of each of the NFS dialog, client, and server records relating to the error. Engel Appendix VI at pp. 80-82, 92 of rtp_nfs_p.c. The stats_for_nodes and stats_for_dialog macros discussed above are used for updating the various error related count statistics (i.e., rpcProgramFail, rpcMismatch, or rpcAuthFail count statistics of the dialog record, and similar count statistics (e.g., progUnavail, versionMismatch, authRejectedCred) of the rpcStats

data structure or errors and mountFailures count statistics of the NFS client and server records), and associated rate statistics (i.e., rpcProgramFailRate, rpcMismatchRate, or rpcAuthFailRate rate statistics of the dialog record, and similar rate statistics (e.g., progUnavailRate, versionMismatchRate, authRejectedCredRate) of the rpcStats data structure or errorRate and mountFailureRate rate statistics of the NFS client and server records). Engel Appendix VI at pp. 80-82, 92 of rtp_nfs_p.c, pp. 66-69 of rtp_nfs_p.c (relevant routines use stats_for_nodes and stats_for_dialog macros for updating statistics). Accordingly, if any error running_count variable of the NFS dialog, client, or server for the time interval from the last issued alarm (i.e., when the running_count variable was last set to 0) exceeds a certain threshold, an alarm is issued to the Event Manager.

120. If the reply did not otherwise result in an error and is an unmount response, rtp_nfs_parse removes the client's link to the file system(s) being unmounted. Engel Appendix VI at pp. 82-83 of rtp_nfs_p.c. If the reply is to a successful mount, the file system is marked as in use. Engel Appendix VI at pp. 83-84 of rtp_nfs_p.c. If the reply is to an unsuccessful mount, the client's link to the file system is removed, and, if necessary, deallocates the data structure of the file system. Engel Appendix VI at pp. 84-85 of rtp_nfs_p.c. The mountFailures and errors count statistics and mountFailureRate and errorRate statistics of the NFS

client record and NFS server record are also updated using the stats_for_nodes and stats_for_dialog macros discussed above. Engel Appendix VI at pp. 84-85, 92 of rtp_nfs_p.c; *see also* pp. 66, 68-69 of rtp_nfs_p.c (relevant routines use stats_for_nodes and stats_for_dialog macros for updating statistics). Accordingly, if the errors or mountFailures running_count variable of the NFS dialog, client, or server for the time interval from the last issued alarm (i.e., when the running_count variable was last set to 0) exceeds a certain threshold, an alarm is issued to the Event Manager.

121. If the communication is for the NFS protocol (“NFS_PORT”) and the communication is a call, the rtp_nfs_parse routine updates variables of the ops count and opRate rate statistics objects for the dialog record, the NFS client record, and the NFS server record using the stats_for_nodes and stats_for_dialog macros, as discussed above. Engel Appendix VI at p. 86 of rtp_nfs_p.c (calling nfs_stats_ops). Again, if the ops running_count variable of the NFS dialog, client, or server for the time interval from the last issued alarm (i.e., when the running_count variable was last set to 0) exceeds a certain threshold, an alarm is issued to the Event Manager.

122. The rtp_nfs_parse routine next updates various statistics objects of the dialog based on whether the request is a “NFS_READ”, “NFS_WRITECACHE”, or “NFS_WRITE”. Engel Appendix VI at pp. 86-87 of rtp_nfs_p.c. Specifically,

if the request is a “NFS_READ,” the readOps count and readOpRate rate statistics objects of the dialog record, NFS client record, and NFS server record are updated using the stats_for_nodes and stats_for_dialog macros, as discussed above. Engel Appendix VI at p. 86 of rtp_nfs_p.c (calling nfs_stats_read_ops), pp. 66-67 of rtp_nfs_p.c (defining nfs_stats_read_ops as calling stats_for_nodes and stats_for_dialog macros). If the readops running_count variable of the NFS dialog, client, or server for the time interval from the last issued alarm (i.e., when the running_count variable was last set to 0) exceeds a certain threshold, an alarm is issued to the Event Manager.

123. If the request is “NFS_WRITECACHE,” the writeCacheOps count and writeCacheOpRate rate statistics objects of the dialog record, NFS client record, and NFS server record are updated using the stats_for_nodes and stats_for_dialog macros. Engel Appendix VI at p. 86 of rtp_nfs_p.c (calling nfs_stats_write_cache_ops), p. 65 of rtp_nfs_p.c (defining nfs_stats_write_cache_ops as calling stats_for_nodes and stats_for_dialog macros). If the writeCacheOps running_count variable of the NFS dialog, client, or server for the time interval from the last issued alarm (i.e., when the running_count variable was last set to 0) exceeds a certain threshold, an alarm is issued to the Event Manager. Additionally, the writeBytes, writeByteRate statistics objects of the dialog record, NFS client record, and NFS server record are updated using the

stats_bytes_for_nodes and stats_bytes_for_dialogs macros. Engel Appendix VI at p. 86 of rtp_nfs_p.c (calling nfs_stats_write_bytes), pp. 65-66 of rtp_nfs_p.c (defining nfs_stats_write_bytes as calling stats_bytes_for_nodes and stats_bytes_for_dialog macros), p. 193 of stats.h (defining stats_bytes_for_nodes macro as calling stats_count_bytes); p. 199 of stats.h (defining stats_bytes_for_dialog macro as calling stats_count_bytes); pp. 628-629 of stats_p.c (defining stats_count_bytes routine). In particular, the stats_count_bytes routine sums the byte_count of the instant packet to the running total writeBytes count and running_count variables and also to the writeBytesRate count variable for each of the dialog, NFS client, and NFS server records. Engel Appendix VI at pp. 628-629 of stats_p.c. The stats_count_byte routine then tests, for each record, whether the byte running_count variable has exceeded the high_thld variable. If so, an alarm is issued and the running_count variable is reset to 0. Engel Appendix VI at pp. 628-629 of stats_p.c. In this way, if the writeBytes running_count variable of the NFS dialog, client, or server for the time interval from the last issued alarm (i.e., when the running_count variable was last set to 0) exceeds a certain threshold, an alarm is issued to the Event Manager. Engel Appendix VI at pp. 628-629 of stats_p.c.

124. If the request is a "NFS_WRITE," the writeOps count and writeOpRate rate statistics objects of the dialog record, NFS client record, and NFS

server record are updated using the `stats_for_nodes` and `stats_for_dialog` macros, as discussed above. Engel Appendix VI at p. 86 of `rtp_nfs_p.c` (calling `nfs_stats_write_ops`), p. 65 of `rtp_nfs_p.c` (defining `nfs_stats_write_ops` as calling `stats_for_nodes` and `stats_for_dialog` macros). If the `writeOps running_count` variable of the NFS dialog, client, or server for the time interval from the last issued alarm (i.e., when the `running_count` variable was last set to 0) exceeds a certain threshold, an alarm is issued to the Event Manager. Additionally, the `writeBytes`, `writeByteRate` statistics objects of the dialog record, NFS client record, and NFS server record are updated using the `stats_bytes_for_nodes` and `stats_bytes_for_dialogs` macros, as discussed above. Engel Appendix VI at p. 86 of `rtp_nfs_p.c` (calling `nfs_stats_write_bytes`), pp. 65-66 of `rtp_nfs_p.c` (defining `nfs_stats_write_bytes` as calling `stats_bytes_for_nodes` and `stats_bytes_for_dialog` macros). If the `writeBytes running_count` variable of the NFS dialog, client, or server for the time interval from the last issued alarm (i.e., when the `running_count` variable was last set to 0) exceeds a certain threshold, an alarm is issued to the Event Manager.

125. If the communication is for an NFS protocol reply and if the reply indicates that an error occurred (e.g., the program is unavailable, the message was denied due to bad credentials), the `rtp_nfs_parse` routine updates various statistics objects of each of the NFS dialog, client, and server records relating to the error.

Engel Appendix VI at pp. 87-89, 92 of rtp_nfs_p.c. Namely, the stats_for_nodes and stats_for_dialog macros are used for updating the various error related count statistics (i.e., rpcProgramFail, rpcMismatch, or rpcAuthFail count statistics of the dialog record, and similar count statistics (e.g., progUnavail, versionMismatch, authRejectedCred) of the rpcStats data structure and/or errors count statistic of the NFS client and server records), and associated rate statistics (i.e., rpcProgramFailRate, rpcMismatchRate, or rpcAuthFailRate rate statistics of the dialog record, and similar rate statistics (e.g., progUnavailRate, versionMismatchRate, authRejectedCredRate) of the rpcStats data structure and/or errorRate rate statistic of the NFS client and server records). Engel Appendix VI at pp. 87-89, 92 of rtp_nfs_p.c, pp. 66-68 of rtp_nfs_p.c (relevant routines use stats_for_nodes and stats_for_dialog macros for updating statistics). Accordingly, if any of the error running_count variables of the NFS dialog, client, or server for the time interval from the last issued alarm (i.e., when the running_count variable was last set to 0) exceeds a certain threshold, an alarm is issued to the Event Manager.

126. If the reply did not otherwise result in an error to this point, the rtp_nfs_parse routine updates various statistics objects of each of the NFS dialog, client, and server records relating to the status of the NFS reply. Engel Appendix VI at pp. 89-91 of rtp_nfs_p.c. Namely, if the reply is "NFS_OK," the readBytes,

readByteRate statistics objects of each of the dialog record, NFS client record, and NFS server record are updated using the stats_bytes_for_nodes and stats_bytes_for_dialogs macros. Engel Appendix VI at pp. 89-90 of rtp_nfs_p.c (calling nfs_stats_read_bytes), p. 65 of rtp_nfs_p.c (defining nfs_stats_read_bytes as calling stats_bytes_for_nodes and stats_bytes_for_dialog macros). If the readBytes running_count variable of the NFS dialog, client, or server for the time interval from the last issued alarm (i.e., when the running_count variable was last set to 0) exceeds a certain threshold, an alarm is issued to the Event Manager.

127. If the reply indicates an access/user/resource type of error (e.g., name too long, stale, file too big), the rtp_nfs_parse routine updates various statistics objects of each of the NFS dialog, client, and server records relating to the error. Engel Appendix VI at pp. 89-90 of rtp_nfs_p.c. Namely, the stats_for_nodes and stats_for_dialog macros are used for updating the various error related count statistics (e.g., nfsAccessError, nfsUserError, or nfsResourceError count statistics of the dialog record, and nfsErrPerm, nfsErrNoEnt, or nfsErrFBig count statistics of the NFS client and server records), and associated rate statistics (e.g., nfsAccessErrorRate, nfsUserErrorRate, or nfsResourceErrorRate rate statistics of the dialog record, and nfsErrPermRate, nfsErrNoEntRate, or nfsErrFBigRate rate statistics of the NFS client and server records). Engel Appendix VI at pp. 89-90 of rtp_nfs_p.c, pp. 69-72 of rtp_nfs_p.c (routines use stats_for_nodes and

stats_for_dialog macros for updating statistics). Accordingly, if any of the error running_count variables of the NFS dialog, client, or server for the time interval from the last issued alarm (i.e., when the running_count variable was last set to 0) exceeds a certain threshold, an alarm is issued to the Event Manager.

D. Age Out Timers

128. Engel discloses an aging mechanism for determining when dialog and address data structures should be discarded and the space recycled. According to Engel, the age out time is configurable:

“[A]ll statistical data collected by the Monitor is subject to age out. Thus, if no activity is seen for an address (or address pair) in the time period defined for age out, then the data is discarded and the space reclaimed so that it may be recycled. In this manner, the Monitor is able to use the memory for active elements rather than stale data. The user can select the age out times for the different components. The EM periodically kicks off the aging mechanism to perform this recycling of resources. STATS provides the routines which the EM calls, e.g. stats_age. . . .

The deallocate routine is called by the aging routines when a structure is to be recycled.”

Engel at 17:2-16.

129. Engel specifically discloses a statsNfsAgeDialog routine that determines whether the period of inactivity for each dialog address pair has exceeded a defined threshold (i.e., the nfsDialogAgeTimer variable of the monCtrl data structure). Engel Appendix VI at pp. 435-438 of stats_nfs_a.c. The period of inactivity is measured by subtracting the lastTime variable of the dialog

StatsAddrEntry record (i.e., the last time an NFS packet between the IP address pair of the dialog was seen) from the current time:

```
if (current_time.tv_sec - dialog_addr_ptr->lastTime < nonCtrl.nfsDialogAgeTimer)
    return;
```

Engel Appendix VI at p. 436; *see also* above discussion regarding lastTime variable. If the period of inactivity exceeds the nfsDialogAgeTimer value, the dialog is removed from the nfs_dialog_hash_table and from the dialogQ for each of the NFS client/server data structures, and the StatsAddrEntry, StatsNfsDialogEntry, and StatsNfsState data structures are deallocated. Engel Appendix VI at pp. 436-438 of stats_nfs_a.c.

E. Other Application Programs and Protocols Supported

130. The Engel Appendix VI code supports at least FTP, Telnet, SMTP, and NFS application programs. Engel Appendix VI at p. 132 of rtp_tcp_p.c. Engel further discloses supporting TFTP, SNMP, NSP (DNS), X Windows, LAT, NFT, RFA, VT, and Netware programs. Engel at Fig. 19, 15:35-39. The differences between the Engel Appendix VI get_dialog and lookup_dialog routines are insubstantial (e.g., the hash function is computed based on different extracted information such as IP addresses and/or ports). Given the standardized nature of these routines, one of skill in the art would recognize that the relevant routines for each of these programs would operate in a similar, if not identical, manner to those specifically disclosed for NFS. Further, Engel specifically discloses use of macros

for updating statistics (including application layer dialogs and server records). Engel at 16:30-33; Engel Appendix VI at pp. 187-207 of stats.h, pp. 626-629 of stats_p.c, p. 75 of rtp_nfs_p.c. Engel also discloses: “the Monitor can be readily extended to handle other protocol families.” Engel at 29:25-26. Upon reading the disclosures of Engel and the Engel Appendix VI source code, a skilled artisan would recognize that such macros would readily be utilized by other supported application layer parse routines.

131. Engel further discloses supporting additional network layer protocols (e.g., DecNet Routing, NCP) and transport layer protocols (e.g., UDP, NSP, NCP). Engel at 29:24-28, Fig. 19. Engel also discloses parsing different protocols at various layers in accordance with known standards. Engel at 20:1-28. Given the standardized nature of Engel’s source code files and parse routines, one of skill in the art would recognize that each supported protocol would be implemented through a separate source code file that includes information for parsing a packet in accordance with the known standard for the protocol.

F. Receiving Engel’s Protocol Descriptions

132. I was asked to consider whether Engel discloses “receiving a set of protocol descriptions for a plurality of protocols that conform to a layered model,” as required by claims 10 and 17.

133. I note that the ‘725 patent discloses “receiving” protocol descriptions only in the context of compiler and optimizer 310 that receives and “compiles” protocol “source-code information in the form of protocol description files.”

“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.”

‘725 patent at 14:63-15:3; see also Fig. 4 at steps 402, 403. I also note that, although the ‘725 patent discloses a compilation process that “generates,” “builds,” “fill[s],” or “load[s]” specific “internal data structures” during initialization (discussed in more detail in Paragraph 218 below), such data structures are not required by claims 10 or 17.

134. Engel disclose a compiler of the network monitor that receives and compiles source code files:

“The source code is written in C language. It may be compiled by a MIPS R3000 C compiler (Ver. 2.2) and run on a MIPS 3230 workstation which has a RISC-os operating system (MIPS products are available from MIPS Computer Systems, Inc. of Sunnyvale, Calif.).”

EX1007, Engel at 43:52-56, see also 10:5-7 (“Monitor 10 includes a MIPS R3000 general purpose microprocessor (from MIPS Computer Systems, Inc.)[.]”). After reviewing the source files found in Appendix VI of Engel, I confirm that the source

code is, in fact, written in the C programming language. Additionally, as discussed in detail above, each of the protocol-specific parsing source code files (i.e., `rtp_dll_p.c`, `rtp_ip_p.c`, `rtp_tcp_p.c`, `rtp_udp_p.c`, `rtp_rpc_p.c`, and `rtp_nfs_p.c`) provides commands that describe a protocol that may be used in packets encountered by the monitor, determines any protocols that may be encountered in the next layer of the packet (i.e., child protocol), and calls the appropriate parse routine for such protocols.

135. The programming languages C and C++ were first defined in 1972 and 1983, respectively. Around the same time that these programming languages were defined, compilers became available for compiling high-level source codes written in these programming languages into low-level executable machine instructions that can directly run on a computer processor (also referred to as a CPU or a microprocessor).

136. For example, Engel describes a network monitor that includes a MIPS R3000 microprocessor and provides source code in C for implementing various network monitoring functions. A skilled artisan would readily know that this source code in C when received by the disclosed compiler can be automatically compiled or translated to executable machine instructions that are stored in memory of the network monitor's microprocessor for subsequent execution. In the context of Engel's protocol-specific parsing source code files (i.e., `rtp_dll_p.c`,

rtp_ip_p.c, rtp_tcp_p.c, rtp_udp_p.c, rtp_rpc_p.c, and rtp_nfs_p.c), a skilled artisan would understand from Engel's disclosure that each of the commands described in the parsing files are translated into executable machine instructions and initialized in memory of the network monitor's microprocessor. Such executable machine instructions, in the context of the parsing commands and as discussed in detail above, instruct the processor on the specific parsing and extraction operations that are to be performed when a particular protocol is encountered by the monitor. Accordingly, Engel discloses a compiler of the network monitor that receives source code information in the form of protocol description files.

G. Engel's High-Level Protocol Description Language

137. I was asked to consider whether Engel discloses "protocol descriptions" that "are provided in a protocol description language" as required by claim 13.

138. I note that the '725 patent describes a "protocol description language" as follows:

"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.”

EX1033, ‘725 patent at 41:24-37. The ‘725 patent additionally states:

“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. ... This information is input into compiler and optimizer 310.”

‘725 patent at 9:29-40; see also Fig. 4. Thus, any high-level language that describes how to perform some of these functions would constitute a protocol description language.

139. Engel discloses source code written in a high-level language:

“The source code is written in C language. It may be compiled by a MIPS R3000 C compiler (Ver. 2.2) and run on a MIPS 3230 workstation which has a RISC-os operating system (MIPS products are available from MIPS Computer Systems, Inc. of Sunnyvale, Calif.).

EX1007, Engel at 43:52-56. After reviewing the source files found in Appendix VI of Engel, I confirm that the source code is, in fact, written in the C programming language.

140. A skilled artisan would understand that Engel’s protocol-specific parsing source code files (i.e., rtp_dll_p.c, rtp_ip_p.c, rtp_tcp_p.c, rtp_udp_p.c, and rtp_nfs_p.c), are protocol descriptions that are provided in a protocol description language within the meaning of the ‘725 patent. Namely, as discussed

in detail above, each of the protocol-specific parsing source code files provides commands that describe a protocol that may be used in packets encountered by the monitor, interprets header information, parses/extracts information based on the protocol described in the file, determines any protocols that may be encountered in the next layer of the packet (i.e., child protocol), and calls the appropriate parse routine for such protocols. *See* Paragraphs 74-127 above.

VIII. OPERATION OF BAKER

141. I was asked to review Baker and the source code found in Baker's disclosures and to provide an overview of the operation of the Baker network interfaces system as disclosed in Baker.

A. Overview

142. By way of background, Baker discloses a network interface system that supports the addition of new protocols to the system "without requiring substantial modification to the system or its control logic." EX1038, Baker at 3:1-8, 1:25-30. In particular, Baker discloses a network device including a data storage device 14 for storing protocol description files (PDFs) and a separate analysis control logic 16 for parsing received network frames based on information contained in the protocol description files. Baker at 10:10-35. Each Baker PDF includes a protocol control record that "define[s] the overall structure of a network protocol and reference[s] other information relating to the network protocol."

Baker at 12:28-32. The data present in each protocol control record is set forth in Table 1:

TABLE 1

PROTOCOL CONTROL RECORD		
Offset	Name	Description
0-3	name_length	length of protocol name in bytes including NULL terminator
4-7	protocol_name	name of protocol control record is describing
8-11	filename	name of file control record is stored in
12-15	numBits	total bit length of protocol header control record is describing
16-17	numFields	number of fields required to describe protocol header
18-19	curField	index of field currently referenced
20-23	outFlag	flag indicating template has been output to file
24-27	dbW	display bit width for protocol header display
28-31	fields	field records that describe protocol header
32-25	options	pointer to option control record to use if this protocol has optional fields
36-39	RT	pointer to protocol specific routing table

Baker at Table 1. Each protocol control record references field records (discussed in more detail below) at bytes 28-31, which describe the protocol header and include pointers to other records as discussed at Baker 12:25-19:10, Tables 1-11. As described in more detail below, each PDF includes information describing extraction operations that should be performed when it is determined that the protocol, which is the subject of the PDF, is in use in the packet. For example, each protocol description file includes a total bit length of the protocol header, a

number of fields required to describe the header, and field records, each describing a protocol header field, including a byte offset from the start of the protocol header and, if appropriate, an associated lookup structure for determining the next protocol control record to use. Baker at 19:11-20:36, Table 1 and 2.

B. Initialization

143. Baker discloses that upon initialization of the system, the protocol and associated control record information is extracted from all PDF files and a “ProtocolList is constructed consisting of a sorted vector of protocol records.” Baker at 20:35-21:7. “The PDF files are then read to memory” in the sequence described at 19:11-20:36. Baker at 21:7-9. Each protocol description control record and all associated records are shown in Tables 1-11 as they appear when resident in memory, upon system initialization. Baker at 19:3-6; *see also* Baker at 20:35-21:7, Fig. 3. Namely, Baker states:

“In the presently preferred embodiment, the initialization of the system includes a determination of the presence of PDF files and the extraction of the protocol and associated control record information from all of the PDF files found. The number of PDF files is determined, and a ProtocolList is constructed consisting of a sorted vector of protocol records at least the size of the number of PDF files found. The name of each protocol record found is inserted in the ProtocolList. The PDF files are then read to memory in the sequence described above for the PDF file writing. The lookup records that indicate a next protocol are associated with the appropriate entries in the ProtocolList.”

Baker at 20:35-21:11.

144. This initialization process is implemented by the Baker control logic (source code) through a `setup_protocols` function. Baker at 127:5-129:13. Namely, the `setup_protocols` function first determines the number of PDFs stored by the system by counting the number of PDF files:

```
// Determine number of protocol definitions that exist by counting files
//
if (_dos_findfirst("*.pdf", _A_NORMAL, &fileblock) == 0)
{
    do
    {
        ++number;
    } while(_dos_findnext(&fileblock) == 0);
}
```

Baker at 127:10-18. A `ProtocolList` vector is then setup to be twice the size of the current number of PDF files found.

```
// Setup a vector twice the size of the current number with number for the grow size
//
ProtocolList = new WCValsortedVector<protocols>(number*2, number);
```

Baker at 127:20-22.

145. Next, the `setup_protocols` function creates a new data structure of type “protocol” for each PDF (protocol description) and each protocol is inserted into the `ProtocolList` vector:

```

// Open each protocol definition file; create a protocol definition in memory; insert it into the vector
//
if (_dos_findfirst("*.pdf", _A_NORMAL, &fileblock) == 0)
{
do
{
FILE *fp = fopen(fileblock.name, "rb");
if ((fp != 0) && (fileblock.size > 56))
{
U32 tmp;
fread(&tmp, sizeof(tmp), 1, fp); // Get length of protocol name
//
S8 *pname = (S8 *)malloc(tmp);
fread(pname, tmp, 1, fp); // Get file name
fclose(fp);
protocol *proto = new protocol(pname, fileblock.name);
//
protocols p(pname, proto);
free(pname);
ProtocolList->insert(p);
}
} while(_dos_findnext(&fileblock) == 0);
}

```

Baker at 127:24-128:13. The ProtocolList vector is used as an index for searching the protocol descriptions supported by the system and includes a variable for storing a protocol name_length variable and a pointer to a protocol data structure. See e.g., Baker at 127:24-128:13 (reproduced above), 128:33-129:1 (reproduced below).

146. The protocol data structure includes the data elements depicted in the protocol control record (Table 1) of Baker:

```

class protocol
{
public:
//
// Constructors
protocol(): name_length(0), protocol_name(0), fname(0), num_fields(0),
            num_bits(0), out_flag(0), fs(0), dbW(32), opts(0), Rt(0) {}

            *      *      *

// Data fields
private:
    U32    name_length; // Length of protocol name in bytes
    S8     *protocol_name; // Protocol name in Ascii
    S8     *fname;       // File name in Ascii
    U32    num_bits;     // # of bits in fixed portion of protocol header
    U16    num_fields;   // # of fields used to describe protocol header
    U16    cur_field;    // Index of last field displayed
    U32    out_flag;     // Protocol has been output 'flag'
    U32    dbW;          // Display bit width (i.e. #bits per line)
    field  *fs;          // Fields for fixed portion of protocol header
    protocol *opts;      // Pointer to protocol for parsing options
    RouteTable *Rt;      // Pointer to Protocol Route Table
};

```

Baker at 149:12-154:8; see also Baker at Table 1.

147. The setup_protocols function then creates the required data structures for each PDF using the protocol::get_from_file function. Namely, for each PDF, the setup_protocols function opens the file, reads the name_length and protocol_name elements from the file, and searches the ProtocolList index for the associated protocol record stored in memory (ProtocolList->find(p, t)).

```

// Open each protocol definition file; create all required data structures
//
if (_dos_findfirst("*.pdf", _A_NORMAL, &fileblock) == 0)
{
do
{
FILE *fp = fopen(fileblock.name, "rb");
if ((fp != 0) && (fileblock.size > 56))
{
protocols p(fp), t; // Reads name_length and name
if (ProtocolList->find(p, t) != FALSE)
t.prot()->get_from_file(fp);
fclose(fp);
}
} while(_dos_findnext(&fileblock) == 0);
}

```

Baker at 128:24-129:5.

148. If a protocol record is found, the `protocol::get_from_file` function initializes the found protocol record (`t.prot()`) based on the protocol control record information in the associated PDF. The `get_from_file` function also creates an indexed array called “fs” of type “field” for storing each field sub-record of the PDF. Baker at 57:7-9. The size of the array is set according to the “num_fields” value read from the PDF, which indicates the number of fields required to describe the associated protocol header.


```

/*
 * Get protocol information from file
 */
void protocol::get_from_file(FILE *fp)
{
// What to do about file name ???
fread(&num_bits, sizeof(num_bits), 1, fp); // Read fixed header length in bits
fread(&num_fields, sizeof(num_fields), 1, fp); // Read number of fields in protocol description
fread(&cur_field, sizeof(cur_field), 1, fp); // Read index of last field displayed during configuration
cur_field = min(cur_field, (U16)(num_fields-1));
fread(&dbW, sizeof(dbW), 1, fp); // Read value of last requested display width in bits
out_flag = 0;
fs = new field(num_fields);

```

Baker at 56:30-57:11; see also Table 2.

149. The field data structure includes the data elements of the field sub-record depicted in Table 2 of Baker:

```

class field
{
public:

// Constructors
field(): fdwoff(0), fshk(0), fshr(0), ffmt(2),
         flflag(0), fname(0), fblen(8), reserved(0), fmult(1),
         fdspfield(1), fswap(1), fplen(0), ptr2stats(0), ptr2np(0),
         ptr2vary(0), ptr2csum(0), ptr2flt(0), ptr2rte(0) {}

        *      *      *

```

```

// Data Representation
private:
// General purpose field descriptors
U32    fplen;    // if not 0, value * field contents is FrameLen
U32    fblen;    // field length in bits
U32    fdwoff;   // field offset in bytes
U8     fshl;     // Number of bits to shift left
U8     fshr;     // Number of bits to shift right
U8     ffmt;     // Field output display format
U8     flflag;   // if TRUE this field contains length of option
U8     reserved; // not used...pad to align following fields
U8     fmult;    // multiply value by this amount before display
U8     fswap;    // Flag indicating the need to swap bytes and words
U8     fdspfield; // display this field on output
S8     *fname;   // user defined field name
stats  *ptr2stats; // Pointer to derived statistics class
lookup *ptr2np;   // Pointer to derived lookup class (next_protocol)
vary   *ptr2vary; // Pointer to Vary field value class
Checksum *ptr2csum; // Pointer to Checksum class for current protocol
criteria *ptr2flt;  // Pointer to Filter Criteria class for this field
RouteTable *ptr2rte; // Pointer to Route Determination class
};

```

Baker at 141:21-149:7; see also Baker at Table 2.

150. Returning to the `protocol::get_from_file` function, another `get_from_file` function is used to read field sub-record information from the associated PDF and store each element in the accompanying field element of the `fs` array at an indexed location, where the index sequentially increments from 0 to the value of `num_fields`.

```

fs = new field[num_fields];
for (U32 i=0; i < num_fields; i++)
    fs[i].get_from_file(fp);

```

Baker at 57:7-9. The `get_from_file` function for initializing the field elements of the `fs` array is reproduced below.

```
// Routine to set up existing field object from designated file
//
void get_from_file(FILE *fp)
{
    U32 tmp;
    fread(&fplen, sizeof(fplen), 1, fp);
    fread(&fblen, sizeof(fblen), 1, fp);
    fread(&fdwoff, (S8 *)&fname - (S8 *)&fdwoff, 1, fp);
//
    fread(&tmp, sizeof(tmp), 1, fp); // Read length of field name
    if (tmp != 0)
    {
        fname = new S8[tmp];
        fread(fname, tmp, 1, fp); // Read field name
    }
    else fname = 0;
    fread(&tmp, sizeof(tmp), 1, fp); // Read type of stats to be gathered
    ptr2stats = alloc_stat_structs(tmp, this);
    fread(&tmp, sizeof(tmp), 1, fp); // Read type of lookup structure
    if ((ptr2np = alloc_lookup_structs(tmp, this)) != 0)
        ptr2np->get_from_file(fp); // Read all lookup values for field
}
```

Baker at 147:28-148:13.

151. As depicted above, this `get_from_file` function uses an `alloc_lookup_structs` function to allocate a data structure called `ptr2np` for the next protocol lookup records (Baker at Table 4) referenced in the PDF based on the type of data structure needed.

```

lookup *alloc_lookup_structs(U32 type, field *f)
{
switch(type)
{
default:    cout << "LOOKUP Yipes!!!\n";
case NOLOOKUP: return(0);           // No p
case ONEVALUE: return(new lookup_valu());
case ARRAY:   return(new lookup_vect(1 << (32-f->shrbits())));
case TREE:    return(new lookup_tree());
}
}

```

Baker at 73:36-74:9.

152. Namely, if the current protocol has only one upper level protocol, alloc_lookup_structs (using a lookup_value constructor) allocates a new data structure of type “lookup_valu,” that includes a pointer to a data structure called “range” of type “verify” and also an “entrees” variable that indicates whether the next protocol entry has been setup.

```

////////////////////////////////////
// Derived Single Entry (Next Protocol) Lookup Class Definition
//   For use where the current protocol has only 1 upper level protocol
//
// Virtual member functions
// insert_protocol(protocol *p)   - Insert protocol pointer

```

```

// protocol *next_protocol(U32 value) - returns protocol pointer
////////////////////////////////////
class lookup_valu: public lookup
{
public:
//
// Constructor
lookup_valu()
{
    range = new verify;
    range->prot = 0;
    range->nxtidx = -1;
    entrees = 0;
}

*      *      *

// Data Representation
private:
    verify *range; // Pointer to protocol class
    U32    entrees; // Indicates whether the entry has been setup
};

```

Baker at 167:30-171:22.

153. If the current protocol has multiple upper level protocols, alloc_lookup_structs (using a lookup_vect constructor) allocates a new data structure of type “lookup_vect,” that includes: a “size” variable set to 256, indicating the array size in entries; an “entrees” variable that indicates the number of entries in the array; and a pointer to a data structure called “array” including 257 entries (size+1), each of type “verify” (i.e., each entry including the data elements defined in the verify data structure).

```

////////////////////////////////////
// Derived Multiple Entry (Next Protocol) Table Lookup Class Definition
//   For use where the values are small in size and number and closely
//   spaced. The IP protocol field is an excellent example of this
//   case. The numbers range from 0 to 255 so an array of pointers
//   requires 256 * sizeof(protocol *) or 1024 bytes.
//
//   Virtual member functions
//   insert_protocol(protocol *p)   - Insert protocol into lookup
//   protocol *next_protocol(U32 value) - Returns pointer to next protocol
////////////////////////////////////
class lookup_vect: public lookup
{
public:
//
// Constructor
    lookup_vect(U32 maxsz = 256)
    {
        size = maxsz;
        array = new verify[size + 1];
        memset(array, 0, (size + 1) * sizeof(verify));
    }

        *           *           *

// Data Representation
private:
    U32   size; // Array size in entries
    U32   entrees; // Number of entries in array
    verify *array; // Array of pointers to protocol classes
};

```

Baker at 171:23-177:33.

154. The verify data structure includes the data elements depicted in Table 4 of Baker:

```

protocol *prot;

```

```
S32  nxtidx;  
U32  minval;  
U32  maxval;  
U32  okbits;  
S8   *xlat;
```

Baker at 165:37-166:5; see also Table 4. As depicted above, the “prot” element of the verify data structure is a pointer to a protocol record.

155. Once the alloc_lookup_structs function allocates the appropriate ptr2np data structure for the lookup records, processing returns to the get_from_file function and ptr2np->get_from_file(fp) is called to read the lookup record values from the PDF and store them in the newly allocated ptr2np record.

```
if ((ptr2np = alloc_lookup_structs(tmp, this)) != 0)  
    ptr2np->get_from_file(fp); // Read all lookup values for field
```

Baker at 148:11-12.

156. For a multi-protocol ptr2np data structure, the number of valid array entries is first determined and then each lookup entry found in the PDF is stored as at least one indexed entry in the array data structure. Specifically, each lookup record is stored in the array at indexed locations including the minval and maxval variables of the lookup record (capped at maxval 256, the size of the array) and also at indexed locations including the even, odd or all values in between minval and maxval based on the okbits variable of the lookup record.

```

// Setup lookup from file
void get_from_file(FILE *fp)
{
    U32 num;
    fread(&num, sizeof(num), 1, fp);    // Write out number of valid array entries
    for (U32 j=0; j<num; j++)
    {
        verify v;
        v.get_from_file(fp);
        v.maxval = min(v.maxval, size);
        for (U32 i=v.minval; i<=v.maxval; i++)
        {
            if (((i & 1) && (v.okbits & 1)) || ((i & 1) == 0 && v.okbits > 1))
            {
                array[i].prot = v.prot;
                array[i].nxtidx = v.nxtidx;
                array[i].minval = i;
                array[i].maxval = i;
                array[i].okbits = v.okbits;
            }
        }
    }
}

```

Baker at 177:4-26. See also Baker at Table 4. In this way, the lookup structure is stored as an indexed array, and each entry is indexed by a valid value.

C. ParseFrame Control Logic

157. Baker describes ParseFrame control logic (depicted in Fig. 11) that “systematically parses through each network frame (at 104 to 108 and 128) until all known protocol headers have been parsed.” Baker at 36:32-34, Fig. 11. The source code includes a ParseFrame2 function that parses a network frame in accordance with Fig. 11.

158. Namely, the ParseFrame2 function receives as an input, a pointer to the initial protocol description record (called ProtoPtr in the source code and also referred to as the CurrentProtocol variable).

```

////////////////////////////////////
// ParseFrame(protocol *InitialProtocolPtr);
//   Parameters: p - pointer to initial protocol description object
//
//   Globals must be set as follows:
//       HwLen   - set to number of bits received from interface device
//       FramePtr - set to start of frame received from interface device
//       SrcIntf - set to indicate the interface number which received frame
////////////////////////////////////
void ParseFrame2(protocol *ProtPtr)
{
    RtePtr    = 0;    // Clear Routing Table Entry Pointer at start of frame
    ParseLen   = 0;    // Set # of Bits Parsed in Frame to 0 at start of frame
    ParseLevel = 0;    // ISO 7 layer position indication
    FrameLen   = HwLen; // Set Frame Length to Hardware Length as best guess
    ParsePtr   = FramePtr; // Set ParsePtr to point at start of Frame

```

Baker at 67:6-21; see also Table 14 (CurrentProtocol variable is “Pointer to protocol description control record in use”). I note that the ‘725 patent refers to this “initial protocol” as the “base protocol” used in a packet. ‘725 patent at 32:11-12 (“The packet type is the root of a tree (called base level).”), column 71 (setting virtualChildren FIELD as “ethernet(1)”; disclosing “now define the base for the children” and setting VirtualBase PROTOCOL as “{VirtualChildren=virtualChildren}”), 32:29-33 (“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.”), 33:16-20 (FIG. 16 shows the header 1600 (base level 1) of a complete Ethernet frame (i.e., packet”), 32:9-12, 35:56-65, 40:1-22, 42:3-11. The Baker source code supports initial/base protocols including Ethernet, FDDI, and Token Ring. Baker at 27:17-20 (Describing InfTypes variable as “Array of values defining the type of each interface in the network system”), 52:12 (“IntfTypes[256] - {ETHERNET, FDDI, TOKEN RING};”), 189:6.

159. As disclosed in Baker at 67:6-21 (reproduced above), the ParseFrame2 function sets locally defined variables, including the ParseLen (indicating the running total number of bits parsed in the frame) and ParseLevel (indicating the ISO layer position) variables to 0. And, the ParsePtr variable is set to point to the start of the frame. See also Baker at Fig. 11 (step 102).

160. The ParseFrame2 function then enters a while loop to call a ParseProtocol2 function, passing as an input, the pointer to the initial protocol description record “ProtPtr.”

```
while(ProtPtr != 0)
{
    ParseList.operator[] (ParseLevel++) = ParsePtr; // Save header pointer
//
    ProtPtr = ParseProtocol2(ProtPtr);
}
```

Baker at 67:27-32, Fig. 11 (steps 104, 106).

161. When processing is returned by the ParseProtocol2 function, the ParseFrame2 function determines whether the ProtPtr variable points to a valid next protocol description record, as determined and returned by the ParseProtocol2 and ParseFields2 functions discussed in Paragraphs 163-177 below. Baker at 67:27-32, Fig. 11 (steps 108, 128). If the ProtPtr variable points to a valid next protocol description record, the ParseFrame2 function again calls the ParseProtocol2 function, passing as an input, the pointer to the next protocol description record “ProtPtr.” Baker at 67:27-32, Fig. 11 (steps 108, 128).

162. The ParseFrame2 function iterates through the discoverable protocol layers in this manner (using the Protocol2 and ParseFields2 functions as discussed in detail below) until the ProtPtr variable returned is null. Baker at 67:27-32, Fig. 11 (step 108). When all known protocol headers have been parsed, “[a]ny remaining frame bits are parsed as application data (at 110, 112 and 130) and/or pad data (at 114, 116 and 132).” Baker at 36:34-36; see also Baker at 67:34-68:24. Where there is application data to be parsed, Baker discloses parsing such application data in the same manner as for other protocols (by calling the Protocol2 function, which calls the ParseFields2 function as discussed below), passing as an input to the Protocol2 function, a pointer to an application protocol record that defines how the application data should be parsed. Baker at 67:33-36.

D. ParseProtocol Control Logic

163. Baker describes ParseProtocol control logic (depicted in Fig. 12) that is responsible for parsing “all fixed and optional protocol fields.” Baker at 37:1-16, Fig. 12. The source code includes a ParseProtocol2 function that parses fixed and optional protocol fields in accordance with Fig. 12.

164. Namely, the ParseProtocol2 function receives as an input, the pointer to the initial/base protocol description record “p”. The ParseProtocol2 function then calls the ParseFields2 function, passing as inputs, the pointer to the initial/base protocol description record, and a HeaderLen variable.

```
////////////////////////////////////  
// Parse Protocol  
////////////////////////////////////  
protocol *ParseProtocol2(protocol *p)  
{  
//  
// Parse each field in the protocol header  
//  
protocol *LocalProto = ParseFields2(p, HeaderLen);
```

Baker at 66:12-20.

165. As discussed in detail below and depicted in Fig. 13, the ParseFields2 function parses fixed fields in the protocol header of a packet as defined by the current protocol description record, determining the next layer protocol as it progresses. Baker at 37:4-5, Fig. 12 (step 152), Fig. 13; see also Paragraphs 166-177 below. Once the fields that are fixed in location have been parsed using the

ParseFields2 function, the ParseProtocol2 function determines whether there are optional protocol fields as defined by the current protocol description record and, if so, parses those fields. Baker at 37:1-16, 66:21-34, Fig. 12. If all hardware bits end up being parsed, the ParseProtocol2 function returns “0.” Otherwise, when processing returns to the ParseFrame2 function, the ParseProtocol2 function returns the next protocol to use in parsing the frame. Baker at 37:13-16, 66:35-67:4.

E. ParseFields Control Logic

166. Baker describes ParseFields control logic (depicted in Fig. 13) that “parses the fields in each protocol header contained in a particular network frame by using field values obtained in accordance with information specified in associated protocol descriptions.” Baker at 30:10-15, Fig. 13. The source code includes a ParseFields2 function that parses the fields in a protocol header in accordance with the Fig. 13.

167. Namely, the ParseFields2 function receives as an input, a pointer to a particular protocol description record in the protocol data structure (protocol *p). The ParseFields2 function then sets a locally defined field variable (f) as a pointer to the protocol record’s field array (fs).

```

////////////////////////////////////
// Parse Fields
////////////////////////////////////
static protocol *ParseFields2(protocol *p, U32 &HeaderLen)
{
    protocol *LocalProto=0;
    field *f, *g;

    ProtoParseLen = 0;
    HeaderLen = p->numbits();
    f = g = p->fieldptr();

```

Baker at 64:1-10; see also Baker at 151:16 (field *fieldptr() const { return(fs); }).

168. The ParseFields2 function then enters a for loop to increment through the indexed fields of the protocol record's field array, starting at index "i=0."

```

for (U32 i = 0;
    i < p->numfields() && ProtoParseLen < HeaderLen && ParseLen < FrameLen;
    f = &g[i])
{

```

Baker at 64:11-14. The method for incrementing the index value will be discussed with reference to validating an extracted value below.

169. The ParseFields2 function then calls a get_value function (also referred to as the GetValue control logic) to locate and extract the associated field value from the packet, as defined by the currently indexed element of the protocol record's field array (fs[i]):

```

//
// Retrieve Current Field Value
//
    U32 val = f->get_value();

```

Baker at 64:18. Namely, the `get_value` function adds the value of the `fdwoff` variable (as defined in the currently indexed element of the protocol record's field array `fs[i]`) to the `ParsePtr` variable (which is set to the start of the frame being parsed). The `fdwoff` variable value is the byte offset of the associated field, measured from the start of the protocol header. Baker at Table 2, 148:26 (`fdwoff` variable of `fs` array described as "field offset in bytes"). Accordingly, when `fdwoff` variable for a particular field is added to the `ParsePtr` variable, the resulting value provides the location of the field in the packet. This enables the `get_value` function to search and locate the field value in the packet. The `get_value` function then sets a local variable "val" as the value of the located field from the packet, and modifies this value to, for example, eliminate any extraneous bits. The resulting extracted value is returned:

```

U32 get_value()
{
    U32 *ptr = (U32 *)(ParsePtr + fdwoff);
    U32 val = *ptr;
    if (fswap != 0)
    {
        val = wordswap(val);
        if (ptr2vary == 0)
            return((val << fshl) >> fshr);
        else // This field needs its value varied
            return(*ptr = wordswap(ptr2vary->vary_value(val)));
    }
    else
    {
        if (ptr2vary == 0)
            return((val << fshl) >> fshr);
        else // This field needs its value varied
            return(*ptr = ptr2vary->vary_value(val));
    }
}

```

Baker at 145:20-146:2. Baker similarly describes this process as follows:

“Referring now to Fig. 13, a value is extracted by the GetValue control logic (shown in Fig. 16) of the system (at 210) for each configured field that is relevant to the current network frame. As shown in Fig. 16, the fdwoff value is added to ParsePtr to locate and extract a 32-bit value (at 502) containing the field value, which is then byteswapped (at 510) if the fswap field is TRUE. If the ptr2vary field is NULL (at 512 or 518), indicating that the value does not need to be modified or varied, the field value is extracted (at 506) by shifting the 32-bit value left fshl bits and then right by fshr bits to eliminate any extraneous bits. If the ptr2vary field is non-NULL, the extracted 32-bit value is modified using the configured method (at 514 or 520) and the resulting 32-bit value overwrites the extracted 32-bit value (at 516 or 522). If the extracted value was byteswapped (at 510), the modified value is

swapped back (at 516) prior to overwriting the original value. The extracted value is returned (at 508) by the GetValue control logic.

Baker at 27:17-35; see also Baker at Fig. 13 (step 210), Fig. 16.

170. After processing is returned, the ParseFields2 function calls a value_ok function (also referred to as the ValidateValue control logic in the Baker), passing as inputs the extracted value “val,” the index of the associated field “i,” and a pointer to a local next protocol data structure:

```
// If the current field value is legal...  
//  
if (f->value_ok(val, i, LocalProto))
```

Baker at 64:25-38; see also Baker at 29:1-31, Fig. 13 (step 214), Fig. 14. The value_ok function determines whether the extracted field value is valid. Namely, the value_ok function first determines whether the ptr2np variable (as defined in the indexed element of the protocol record’s field array fs[i]) is set to “0,” indicating that a lookup structure is not configured for the particular field in question. If this is the case, the index “i” (used to determine which indexed entry of the field array should be parsed next) is incremented by one and “TRUE” is returned:

```

// Routine to check field value for validity
//
U32 value_ok(U32 value, U32 &i, protocol * &tp)
{
    verify *v;
    if (ptr2np == 0)
    {
        ++i;
        return(TRUE);
    }
}

```

Baker at 146:9-13; see also Baker at 29:7-11, Fig. 14 (steps 302, 318, 312).

171. Alternatively, it is determined (Baker at 146:9-13) that a lookup structure does exist for the particular field and the value_ok function tests whether the extracted value is found in that lookup structure.

```

if ((v = ptr2np->value_ok(value)) == 0)
{
    ++i;
    return(FALSE);
}

```

Baker at 146:14-18; see also Baker at 29:12-16, Fig. 14 (steps 304, 314, 316). If the value is not found in the lookup structure, the index “i” (used to determine which indexed entry of the field array should be parsed next) is incremented by one and “FALSE” is returned.

172. The specific function used to determine whether an extracted value is found in a particular multi-protocol lookup structure is as follows:

```

verify *value_ok(U32 value)
{

if (value > size)
    return(0);
else
    {
        verify *v = &array[value];
        if (v->okbits != 0)
            {
                if (value >= v->minval && value <= v->maxval)
                    {
                        if (value & 1) // It's an odd number
                            {
                                if (v->okbits & 1) // ODD and ALL numbers ok
                                    return(v);
                                }
                            else // It's an even number
                                {
                                    if (v->okbits > 1) // EVEN and ALL numbers ok
                                        return(v);
                                    }
                                }
                            }
                        return(0); // Value was not found
                    }
            }
}

```

Baker at 175:35-176:24. As detailed above, this function first tests to see if the extracted value is bigger than the size of the array (256). If not, the function sets up a pointer “v” to the array entry at an indexed location corresponding to the extracted value (array[value]). The function then tests whether the value is equal to or between the minval and maxval variables of the array entry at the value-indexed location. If so, and if the value is an odd number, the function verifies that

the okbits variable of the array entry at the value-indexed location indicates that odd or all values for the associated next protocol are ok. If the value is an even number, the function verifies that the okbits variable of the array entry at the value-indexed location indicates that even or all values for the associated next protocol are ok. If one of these last two checks is true, the function returns the pointer to the array entry at the value-indexed location. Otherwise, "0" is returned, indicating that a valid entry was not found.

173. The specific function used to determine whether an extracted value is found in a particular single protocol lookup structure is as follows:

```
verify *value_ok(U32 value)
{
    if (value >= range->minval && value <= range->maxval)
    {
        if (value & 1) // It's an odd number
        {
            if (range->okbits & 1)      // ODD and ALL numbers ok
                return(range);
        }
        else // It's an even number
        {
            if (range->okbits > 1) // EVEN and ALL numbers ok
                return(range);
        }
    }
    return(0);
}
```

Baker at 170:28-171:7. As detailed above, this function tests whether the value is equal to or between the minval and maxval variables of the verify record. If so,

and if the value is an odd number, the function verifies that the okbits variable of the record indicates that odd or all values for the associated next protocol are ok. If the value is an even number, the function verifies that the okbits variable of the record indicates that even or all values for the associated next protocol are ok. If one of these checks is true, the function returns a pointer to the verify record. Otherwise, “0” is returned, indicating that a valid entry was not found.

174. Returning to the main value_ok function, if it is determined that a lookup structure does exist for a particular field and that the extracted value is found in that lookup structure, the value_ok function determines whether the local next protocol variable “tp” (also referred to in the disclosure as NextProtocol) has been previously set and, if not, sets it to the “prot” variable of the lookup structure array entry at the value-indexed location corresponding to the extracted value. This “prot” variable is a pointer to the next protocol description data structure that will be used to parse the next layer of the packet. The index “i” (used to determine which indexed entry of the field array should be parsed next) is set to the nxtidx variable (also referred to in the disclosure as NextIndex) of the lookup structure array entry at the value-indexed location corresponding to the extracted value and “TRUE” is returned:

```

else
{
  if (tp == 0)
    tp = v->prot;
  i = v->nxtidx;
  return(TRUE);
}

```

Baker at 146:19-25; see also Baker at 29:17-31, Fig. 14 (steps 306, 308, 310, 312). According to Baker, and consistent with the source code (i.e., if (tp == 0)), the “first valid field parsed in a protocol that specifies the NextProtocol has precedence over all subsequent NextProtocol specifiers [sic].” Baker at 29:24-27.

175. Processing is then returned to the ParseFields2 function and processing continues as illustrated in Fig. 13 of Baker at step 217. Baker at 64:30-65:35. At step 222 of Fig. 13, the running number of bits parsed in the current frame (measured by the ParseLen variable) and the number of bits parsed in the current protocol header (measured by ProtoParseLen variable) are updated based on the “bitlen” value (also referred to as BitLength) of the field that was just parsed. Baker at 35:27-30, 65:20-24, Fig. 13 (step 222).

176. The ParseFields2 function then returns to the beginning of the for loop (discussed at Paragraph 168 above) and tests whether the now incremented index value “i” is: 1) less than the “numfields” variable of currently being parsed protocol description record (indicating that there is another field to parse); 2) whether the ProtoParseLen value is less than the HeaderLen value (indicating that

are additional bits in the current protocol header to be parsed); and 3) whether the ParseLen value is less than the FrameLen value (indicating that there are additional bits in the current frame to be parsed). Baker at 64:11-13, 36:13-17, Fig. 13 (steps 206, 208, 209). If all of these conditions are met, the ParseFields2 function begins parsing the next field, as determined by locating field entry of the field array at the incremented index location (f[i]). Baker at 64:11-13, Fig. 13.

177. The ParseFields2 function iterates through the indexed field elements in the manner discussed with respect to Paragraphs 168-176 above until the index value “i” is: 1) not less than the “numfields” variable of currently being parsed protocol description record (indicating that there all fields have been parsed); 2) the ProtoParseLen value is not less than the HeaderLen value (indicating that every bit in the current protocol header has been parsed); and 3) whether the ParseLen value is less than the FrameLen value (indicating that every bit in the current frame has been parsed). Baker at 64:11-13, 36:13-17, Fig. 13 (steps 206, 208, 209). At this point, processing returns to the ParseProtocol2 function for determining whether there are optional protocol fields for the current protocol description. See Paragraph 165 above.

F. Protocol Descriptions

178. I was asked to consider whether the Baker protocol descriptions describe protocols that “conform to a layered model” as required by claim 1 of the ‘725 patent.

179. Baker specifically discloses supporting the Ethernet protocol (Baker at 21:32-22:5), and also mentions supporting protocols “for any possible protocol specification”:

“It will be appreciated by those skilled in the art that any possible organization of fields for any possible protocol specification is contemplated for the network interface system 10 of the present invention.”

Baker at 12:21-24. Additionally, Baker specifically mentions configuring “IP, TCP, UDP and IPX checksum capabilities” as part of the application control logic, and provides source code to support such capabilities for these protocols. Baker at 34:16-21, 135:28-141:8, 213:24-35. Further, Baker discloses a “ParseLvl” variable that is the “Zero based protocol level in ISO reference model of protocol being parsed (current protocol).” Baker at Table 14 (at 25:8-10). A skilled artisan would understand that the “ISO reference model” referenced in Baker refers to the OSI model I discuss in Paragraphs 30-37 above, which is a layered model. The source code confirms the same, describing the “ParseLevel” variable as the “ISO 7 layer position indication.” Baker at 67:19. Baker further discloses that PDFs may include a lookup structure record “for determining the next protocol control record

to use,” and accordingly describe how to determine the protocol used in a packet at a next upper-layer level. Baker at 15:10-17, Table 4. Upon reading these disclosures, a person having skill in the art at the time of the alleged invention would understand that Baker discloses supporting protocols that conform to a layered model, such as the ISO/OSI model.

G. Storing a Database in a Memory

180. I was asked to consider whether Baker discloses “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” as required by claim 1 of the ‘725 patent.

181. For ease of review, I will discuss the individual requirements of this claim phrase separately.

1. *“storing a database in 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”*

182. As an initial matter, I note that in one embodiment, the ‘725 patent discloses a database including a Protocol Table containing information on the possible protocols and a series of one or more Look Up Tables associated with a protocol in the Protocol Table. The Look Up Tables are used to identify known protocols and their children:

“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.”

‘725 patent at 14:39-43.

“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.”

‘725 patent at 39:22-54.

183. As discussed in Section VIII.B above, Baker discloses an initialization process for reading the contents of PDF files into data structures stored in memory. In particular, Baker discloses that, upon initialization and when PDF files are read into memory as described above, the resulting protocol description data structures include a data structure of protocol records (each corresponding to a PDF) and further provides a ProtocolList sorted vector that is used as an index for searching the protocol descriptions supported by the system. Baker at 127:20-128:13, 128:33-129:1; see also Paragraphs 143-147 above. The Baker protocol data structure like the Protocol Table of the '725 patent, contains information on the possible protocols known by the system. *Compare* '725 patent at 39:22-23 ("A table 1850, called the protocol table (PT) has an entry for each protocol known by the monitor 300[.]"), Fig. 18B *with* Baker's protocol data structure and ProtocolList as described above. Also, like the Protocol Table of the '725 patent, the Baker data structure of protocol records includes an indication of the header length. *Compare* '725 patent at 39:22-30 (describing protocol table as including "flags to indicate the header length") *with* Baker at 154:2 (each protocol record includes an "num_bits" variable describing the "# of bits in fixed portion of protocol header"), Table 1 ("numBits" indicates the total bit length of the protocol header).

184. Additionally, and as discussed in detail Paragraphs 148-150 above, each protocol record includes an indexed array of field sub-records and each indexed field entry in the array specifies, for example, the location of the associated field in the packet header that may relate to the next layer (child) protocol and also the bit length of the field. *Compare* ‘725 patent at 39:22-30 (describing protocol table as including “a description of where the field that specifies next protocol [] can be found in the header, the length of the next protocol field”) *with* Baker at 145:20-146:2, 27:17-35, Fig. 13 (step 210), Fig. 16 (fdwoff variable used to determine the location of a field in the packet that may relate to next protocol), 148:25-26 (each field entry in indexed field array includes an “fdwoff” variable describing the “field offset in bytes” and an “fblen” variable describing the “field length in bits”), Table 2 (field sub-records include “fdwoff” indicating the byte offset from the start of the protocol header and “fblen” indicating the “length of field in bits”), Fig. 4A (Ethernet Type field (at indexed location 4) includes Bit Offset and Bit Length), Fig. 5A (Frame Type field (at indexed location 3) and Source/Destination Socket fields (at indexed locations 7 and 8) includes Bit Offset and Bit Length); see also Paragraphs 148-150 above.

185. Further, like the ‘725 patent’s Fig. 18B embodiment, the Baker protocol description data structures are organized for locating the child related information for any protocol. Namely, like the Protocol Table and associated

series of one or more Look Up Tables (LUTs) of the ‘725 patent, each of Baker’s protocol records is associated with one or more next protocol lookup structures. *Compare* ‘725 patent at 39:31-32 (“For any protocol, there also are one or more lookup tables (LUTs)”), 14:39-43 (“a series of Look Up Tables 1870 (LUT’s) that are used to identify known protocols and their children”) *with* Baker at Table 4, 15:10-17 (“Lookup structures can be used for determining the next protocol control record to use ...”), Figs. 5A, 5C-E. And, like the LUTs of the ‘725 patent, Baker discloses that, when there are multiple upper level protocols, each next protocol lookup structure has 256 entries and is indexed by a value extracted from the next protocol field in the packet. *Compare* ‘725 patent at 39:33-35 (“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.”) *with* Baker at 171:23-177:33, 165:37-166:5, and 177:4-26 as discussed in detail in Paragraphs 151-156 above (multi-protocol next protocol lookup structures are stored as an indexed array including 256 entries, and each entry is indexed by a valid extracted value), Table 4.

186. Accordingly, the protocol data structure, like the ‘725 patent Protocol Table, is used to extract a field that specifies the next protocol, which is then used as an address into the next protocol lookup structure for determining the next/child protocol. *Compare* ‘725 patent at 40:1-7 (“When using the data structure of FIG. 18B ... [t]he 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).”) *with* Baker at 145:20-146:2, 64:18, 148:26, 27:17-35, Fig. 13 (step 210), Fig. 16, Table 2 discussed in detail in Paragraph 169 (*get_value* function determines location of field in packet and extracts value from it) *and* 64:25-38, 175:35-176:24, 29:1-31, Fig. 13 (step 214), Fig. 14 discussed in detail in Paragraphs 170-174 (*value_ok* function uses extracted value as index to lookup structure to determine if value is for a valid protocol). When the next protocol is identified using the lookup structure/LUT, it is used as an index into the protocol table. *Compare* ‘725 patent at 40:27-31 (“If a LUT lookup results in a node code indicating a protocol node ... the “next protocol” from LUT lookup is used as an index into the protocol table 1850.”) *with* Baker at 146:19-25, 29:17-31, 67:27-32, Fig. 11 (steps 108, 128), Fig. 14 (steps 306, 308, 310, 312), Table 4 discussed in Paragraphs 161 and 174 (if the extracted value is found in a lookup structure, set pointer to next protocol’s data structure).

187. Further, and as discussed in more detail in Paragraphs 211-219 below, Baker discloses a compilation process consistent with that disclosed in the ‘725 patent for producing searchable and indexed data structures, which are a database similar to the ‘725 Fig. 18B database, as discussed in detail above. See also, e.g., EX1028, 1996 Newton’s Telecom Dictionary at p. 169 (defining “database” as “A collection of data structured and organized in a disciplined fashion so that access is

possible quickly to information of interest. . . . Databases are stored on computers in different ways. Some are comma delineated. They differentiate between their fields with commas[.]”).

188. For at least these reasons, it is my opinion that a person having ordinary skill in the art at the time of the invention would understand that Baker discloses storing a database in a memory, the database generated from the set of protocol description files and including a protocol data structure containing information on the possible protocols and organized for locating the child protocol related information for any protocol, as required by claim 1.

2. *“the data structure contents indexed by a set of one or more indices”*

189. In the Fig. 18B embodiment of the ‘725 patent, each LUT is “indexed by one byte of the child recognition pattern that is extracted from the next protocol field in the packet.” ‘725 patent at 39:33-35. Additionally, when a lookup results in a valid next protocol, “the “next protocol” from LUT lookup is used as an index into the protocol table 1850.” ‘725 patent at 40:27-31.

190. Like the ‘725 patent, Baker discloses that, for multi-protocol lookup structures, each next protocol lookup structure is indexed by a value extracted from the next protocol field in the packet. Baker at 171:23-177:33, 165:37-166:5, and 177:4-26 as discussed in detail in Paragraphs 151-156 above (multi-protocol next protocol lookup structures are stored as an indexed array that is indexed by a valid

extracted value), Table 4; *also* 64:25-38, 175:35-176:24, 29:1-31, Fig. 13 (step 214), Fig. 14 discussed in detail in Paragraphs 170-174 (value_ok function uses extracted value as index to lookup structure to determine if value is for a valid protocol). Additionally, when the next protocol is identified using the lookup structure, it is used as an index into the protocol table. Baker at 146:19-25, 29:17-31, 67:27-32, Fig. 11 (steps 108, 128), Fig. 14 (steps 306, 308, 310, 312), Table 4 discussed in Paragraph 161 and 174 (if the extracted value is found in a lookup structure, set pointer to next protocol's data structure).

191. Moreover, and as discussed in detail above, Baker discloses a ProtocolList sorted vector that is used as an index for searching the protocol descriptions supported by the system. Baker at 127:20-128:13, 128:33-129:1; see also Paragraphs 143-147 above. Baker further discloses that each protocol record includes an indexed array of field sub-records. Baker at 56:30-57:11, 147:28-148:13, 141:21-149:7, and Table 2 discussed in detail in Paragraphs 148-150 above; *also* Fig. 4A, Fig. 5A. The index of each field sub-record is used during parsing “to determine where parsing of the current protocol will continue after the current field.” Baker at 16:47-53; *also* Baker at 64:11-13, 36:13-17, and Fig. 13 (steps 206, 208, 209) discussed in Paragraphs 176-177 above, Fig. 4D, 5C-5E (Next Index), 16:60-67, Table 4.

192. Accordingly, upon reading the disclosures of Baker, a person having ordinary skill in the art at the time of the invention would understand that Baker discloses a data structure with contents indexed by a set of one or more indices as described in the ‘725 patent.

3. *“the database entry indexed by a particular set of index values including an indication of validity”*

193. With respect to this element, I note that the ‘725 patent discloses:

“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.”

‘725 patent at 39:39-50; see also 35:4-11.

“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.”

‘725 patent at 14:57-61.

194. As discussed above, Baker discloses that, for multi-protocol lookup structures, each next protocol lookup entry is indexed by a valid protocol identification value. Baker at 171:23-177:33, 165:37-166:5, and 177:4-26 as

discussed in detail in Paragraphs 151-156 above (multi-protocol next protocol lookup structures are stored as an indexed array that is indexed by a valid extracted value), Table 4; *also* 64:25-38, 175:35-176:24, 29:1-31, Fig. 13 (step 214), Fig. 14 discussed in detail in Paragraphs 170-174 (value_ok function uses extracted value as index to lookup structure to determine if value is for a valid protocol).

195. Moreover, each entry in a multi-protocol lookup structure array includes a “prot” variable (“Protocol” of Table 4 and Figs. 4D, 5C-5E) that is an indication of validity consistent with the disclosed embodiments of the ‘725 patent. Namely, similar to the ‘725 patent “node code,” the “prot” variable of a lookup structure has one of two values: (1) a pointer to a protocol record, indicating the protocol that has been recognized as the next (child) protocol; or (2) a “null” code indicating there is no valid entry (referred to in Baker as an “unknown” or “illegal” entry).

Frame Type Next Protocol Structure					
Protocol	Next Index	Minimum	Maximum	Mask	Translation
[None]	4	0x00	0x00	ALL	"Illegal Protocol"
GP1	4	0x01	0x01	ALL	"GP1"
GP2	5	0x02	0x02	ALL	"GP2"
[None]	4	0x03	0xFF	ALL	"Illegal Protocol"

Fig. 5C

Baker at Fig. 5C.

Source Socket Next Protocol Structure					
Protocol	Next Index	Minimum	Maximum	Mask	Translation
[None]	8	0x0000	0x0142	ALL	"Unknown Protocol"
GP3	8	0x0143	0x018F	ODD	"GP3"
GP4	8	0x0143	0x018F	EVEN	"GP4"
[None]	8	0x0190	0xFFFF	ALL	"Illegal Protocol"

Fig. 5D

Baker at Fig. 5D; see also Fig. 5E, Table 4 (describing "Protocol" variable as "pointer to protocol description structure"); see also Paragraphs 151-156 (describing initialization process for lookup structures, including "prot" variable of each lookup structure array entry).

196. Accordingly, a skilled artisan would recognize that the "prot" variable of each entry of the next protocol lookup array is "an indication of validity" as described in the '725 patent and Baker discloses a database entry indexed by a particular set of index values including an indication of validity as required by claim 1.

4. *"wherein the child protocol related information includes a child recognition pattern"*

197. With respect to this element of claim 1, the '725 patent discloses:

"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."

'725 patent at 14:54-56.

“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.”

‘725 patent at 32:22-27.

“For each of the IDs in the list, a list of the child recognition patterns that need to be compared. For example, 64:0800₁₆ 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). 83:8137₁₆ 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.”

‘725 patent at 35:23-30.

198. Baker discloses a “Generic Protocol” (GP) (described in Baker at 21:32-22:5, Table 13) that defines three fields of a packet that may lead to identification of the child protocol (Frame Type, Source Socket, Destination Socket):

Index	Field Name	Byte Offset	Bit Length	Left Shift	Right Shift	Checksum	Frame Length	Header Length	Statistics	Lookup Structure	Filter	Format
0	Version No.	0	4	0	28	0	0	0	Cnt/Index&Cnt	(None)	(None)	decimal
1	HeaderLen	0	4	4	28	0	0	32	Sum/Index&Cnt	Fig 5b	(None)	decimal
2	Frame Length	0	16	8	16	0	8	0	Sum	(None)	(None)	decimal
3	Frame Type	0	8	24	24	0	0	0	Index&Cnt	Fig 5c	Fig 10,Idx 2	hex
4	Checksum	4	16	0	16	ptr	0	0	(None)	(None)	(None)	hex
5	Control	4	8	16	24	0	0	0	(None)	(None)	(None)	bitfield
6	Hop Count	4	8	24	24	0	0	0	(None)	(None)	(None)	decimal
7	Source Socket	8	16	0	16	0	0	0	(None)	Fig 5d	(None)	hex
8	Destination Socket	8	16	16	16	0	0	0	(None)	Fig 5e	(None)	hex
9	Source Address	12	32	0	0	0	0	0	(None)	(None)	(None)	hex
10	Destination Address	16	32	0	0	0	0	0	(None)	(None)	(None)	hex

Figure 5a

EX1038 at Fig. 5A. When stored in the Baker disclosed data structures and as discussed in detail in Paragraphs 148-156 above, each of the GP fields illustrated

in Fig. 5a is stored as an indexed field sub-record in the field array of the GP protocol record. The ptr2np variable of each record will be a pointer to the associated next protocol lookup structure – Fig. 5c (for the Frame Type field), Fig. 5d (for the Source Socket field), and 5e (for the Destination Socket field).

199. As depicted in Figs. 5c-5e, each of these lookup structures includes multiple protocol entries over a range of corresponding valid values for the associated protocol:

Frame Type Next Protocol Structure					
Protocol	Next Index	Minimum	Maximum	Mask	Translation
[None]	4	0x00	0x00	ALL	"Illegal Protocol"
GP1	4	0x01	0x01	ALL	"GP1"
GP2	5	0x02	0x02	ALL	"GP2"
[None]	4	0x03	0xFF	ALL	"Illegal Protocol"

Figure 5c

Baker at Fig. 5c.

Source Socket Next Protocol Structure					
Protocol	Next Index	Minimum	Maximum	Mask	Translation
[None]	8	0x0000	0x0142	ALL	"Unknown Protocol"
GP3	8	0x0143	0x018F	ODD	"GP3"
GP4	8	0x0143	0x018F	EVEN	"GP4"
[None]	8	0x0190	0xFFFF	ALL	"Illegal Protocol"

Figure 5d

Baker at 5d.

Destination Socket Next Protocol Structure					
Protocol	Next Index	Minimum	Maximum	Mask	Translation
[None]	9	0x0000	0x0142	ALL	"Unknown Protocol"
GP3	9	0x0143	0x018F	ODD	"GP3"
GP4	9	0x0143	0x018F	EVEN	"GP4"
[None]	9	0x0190	0xFFFF	ALL	"Illegal Protocol"

Figure 5e

Baker at Fig. 5e. When stored in the Baker disclosed data structures and as discussed in detail in Paragraphs 151-156 above, each of these lookup structures is stored as an indexed array, and indexed by a valid protocol identification value.

200. Like the example provided in the '725 patent at 35:23-30 (reproduced above), Baker discloses that each possible child protocol includes a list of the child recognition patterns that need to be compared. In the case of Fig. 5C, the value to look for is 0x01 (hex) for the child to be of type GP1 while the value to look for is 0x02 (hex) for the child to be of type GP2. Baker at Fig. 5C. Similarly, Baker discloses that the values to look for are odd values between 0x0143 (hex) and 0x018F (hex) for the child to be of type GP3 while the values to look for are even values between 0x0143 (hex) and 0x018F (hex) for the child to be of type GP4. Baker at Fig. 5D, 5E. In this way, Baker discloses that the child protocol, e.g., GP1, of a particular protocol, e.g., GP, is indicated by a child recognition pattern, e.g., 0x01 (hex) at a location within the header of the particular protocol.

201. As discussed previously and like the ‘725 patent disclosure at 14:54-56 (reproduced above), during the parsing process, a value is extracted from a field of a packet and used as an index to search the next protocol lookup structure for an entry in the array that matches the extracted value. Baker at 145:20-146:2, 64:18, 148:26, 27:17-35, Fig. 13 (step 210), Fig. 16, Table 2 discussed in detail in Paragraphs 169 (get_value function determines location of field in packet and extracts value from it) *and* 64:25-38, 175:35-176:24, 29:1-31, Fig. 13 (step 214), Fig. 14 discussed in detail in Paragraphs 170-174 (value_ok function uses extracted value as index to lookup structure to determine if value is for a valid protocol).

202. Upon reading the Baker disclosures, an ordinarily skilled artisan would recognize that the Baker data structures are organized for locating child protocol related information for a protocol and such protocol related information includes “a child recognition pattern” as described in the ‘725 patent.

H. Searching the Packet

203. I was asked to consider whether Baker discloses “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” as required by claim 1 of the ‘725 patent.

204. As discussed in detail above, the Baker control logic parses through each network frame, starting with the “initial” or “base” protocol of the frame, such as Ethernet. See Paragraphs 157-177 above. For each layer, including the base layer, Baker discloses parsing the fields of the packet as defined by the associated protocol description record’s indexed field array. Namely, for each field that is to be parsed in accordance with the protocol description and starting at index (i=0) of the field array, the ParseFields2 function initiates the get_value function to determine the location of the associated field in the packet and extract a value from the field. For any field parsed, if the field does not have a ptr2np variable set (i.e., the value of the element is null), it cannot be used to identify the child protocol (i.e., via a next protocol lookup array) and the field array index is incremented. If the field does have a ptr2np variable set (i.e., the value of the element is not null), the field’s extracted value is a “child recognition pattern” in the terms of the ‘725 patent. See Paragraphs 197-202 above. In this way, the search for a field that includes a child recognition pattern continues by searching the packet for the field indicated by the indexed field array entry and incrementing the field array index until the field being searched contains a valid ptr2np variable.

205. I note that this implementation of Baker is similar to ‘725 patent’s disclosure at 36:9-39 (describing incrementally searching an indexed array for field that includes child recognition pattern, indicating that the field may relate to the

child protocol). Accordingly, searching the packet for the field indicated by the indexed field array entry and incrementing the field array index until the field being searched contains a valid ptr2np variable constitutes “searching including indexing the data structure until a valid entry is found” as described in the ‘725 patent.

206. Additionally, Baker discloses searching the packet for a valid child recognition pattern. Namely, Baker uses the extracted child recognition pattern to determine if the value is for a valid protocol. For multi-protocol lookups, the value_ok function uses the extracted child recognition pattern (value) as an index to the associated lookup array. See Paragraphs 170-172 and 190 above. In this way, the search for a field that includes a valid child recognition pattern continues by searching the packet for the field indicated by the indexed field array entry and incrementing the field array index until the field being searched contains a child recognition pattern that itself is valid, as determined through (in the case of multi-protocol lookups) an indexed search of an associated multi-protocol lookup array.

207. As such, it is my opinion that a person having skill in the art at the time of the alleged invention would understand that Baker discloses 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 as described in the ‘725 patent.

I. Rapid Searches

208. I was also asked to consider whether Baker discloses “whereby the data structure is configured for rapid searches using the index set” as required by claim 1 of the ‘725 patent.

209. I note that the ‘725 patent discloses that the data structure used in Fig. 18B and discussed at length above “permits rapid searches to be performed.” Namely, the ‘725 patent states:

“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.”

‘725 patent at 14:33-62; see also 39:13-40:47, 34:19-25. As further disclosed in the ‘725 patent, performing address link computations requires performing arithmetic computations to determine pointers to data nodes. ‘725 patent at 34:20-26. A

skilled artisan would recognize that the data structure of the ‘725 patent Fig. 18B embodiment is configured for “rapid searches” by using a child recognition pattern/code to index the Lookup Tables, such that searching the Lookup Tables is performed by indexing locations in memory rather than performing address link computations.

210. For all the reasons discussed in Paragraphs 182-202, the Baker protocol description data structure and associated next protocol lookup structures are similar to the Fig. 18A Protocol Table and associated series of one or more Lookup Tables. Further, the Baker data structure, like the ‘725 patent Fig. 18B embodiment, uses a child recognition pattern to index multi-protocol next protocol lookup arrays. Baker at 171:23-177:33, 165:37-166:5, and 177:4-26 as discussed in detail in Paragraphs 151-156 above (multi-protocol next protocol lookup structures are stored as an indexed array that is indexed by a valid extracted value), Table 4; *also* 64:25-38, 175:35-176:24, 29:1-31, Fig. 13 (step 214), Fig. 14 discussed in detail in Paragraphs 170-174 (`value_ok` function uses extracted value as index to lookup structure to determine if value is for a valid protocol). See also Paragraphs 197-202. Searching the Baker multi-protocol lookup arrays is performed in the same way as searching the ‘725 patent Lookup Tables (LUTs) – by indexing locations in memory rather than performing address link

computations, and is therefore a “rapid search” using the index set discussed in Paragraph 170-172 above as described in the ‘725 patent.

J. Compiling Baker’s Protocol Description Files

211. I was also asked to consider whether Baker’s protocol description files “are provided in a protocol description language” and whether Baker discloses “compiling the PDL descriptions to produce the database,” as required by claim 2 of the ‘725 patent.

212. I note that the ‘725 patent describes a “protocol description language” as follows:

“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.”

EX1033, ‘725 patent at 41:24-37. The ‘725 patent additionally states:

“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. ... This information is input into compiler and optimizer 310.”

‘725 patent at 9:29-40; see also Fig. 4. For at least the reasons discussed below, Baker discloses commands written in a protocol description language as required by claim 2.

213. The ‘725 patent discloses examples of PDL files, which are written in a protocol description language. EX1033, ‘725 patent at columns 44-94. Each file describes, for example, commands for particular protocol. ‘725 patent at columns 71-94 (including PDL files for several protocols). Similarly, Baker discloses that each PDF (similar to the PDL of the ‘725 patent) describes commands for a particular protocol. Baker at 11:22-25 (“[E]ach of these protocol description records with its associated field, statistics, lookup, and filter record information is also written to a protocol specific protocol description file (PDF).”).

214. According to the ‘725 patent, an exemplary command in a high-level protocol description language is, for example, a “HEADER” attribute that “describe[s] the length of the protocol header.” ‘725 patent at 48:41-50; *see also* ‘725 patent at column 73 (“HEADER { LENGTH=14 }”). Baker similarly discloses a “numBits” attribute that describes “the total bit length of the protocol header.” Baker at Table 1 (“numBits” attribute), 11:32; *also*

```
fread(&num_bits, sizeof(num_bits), 1, fp); // Read fixed header length in bits
```

Baker at 57:1 (discussed in detail in Paragraphs 143-156 with respect to initialization).

215. Another exemplary command in a high-level protocol description language is, for example, a “PROTOCOL” definition used to “define the order of the FIELDS and GROUPs within the protocol header.” ‘725 patent at 47:34-48:18; *see also* ‘725 patent at column 79 (PROTOCOL section provides “Detailed packet layout for the IP datagram. This includes all fields and format. All offsets are relative to the beginning of the header.”). Similarly, Baker discloses a “fields” attribute that references the associated “field records that describe the protocol header” where each field record includes, for example, a “fblen” attribute describing “the length of the field in bits” and a “fdwoff” attribute describing “the byte offset from the start of protocol header,” among other attributes. Baker at Table 1 (“fields” attribute), Table 2 (“fblen” and “fdwoff” attributes), 11:36-40, 11:49-50; *also*

```
fread(&fblen, sizeof(fblen), 1, fp);  
fread(&fdwoff, (S8 *)&fname - (S8 *)&fdwoff, 1, fp);
```

Baker at 147:35-36 (discussed in detail in Paragraphs 143-156 with respect to initialization).

216. Yet another exemplary command in a high-level protocol description language is, for example, a “CHILDREN” attribute “used to describe how children protocols are determined.” ‘725 patent at 49:45-55; *see also* ‘725 patent at column 79 (“CHILDREN { DESTINATION=Protocol }”). Similarly, Baker discloses a “ptr2np” attribute of each field record that includes a “pointer to lookup

structure/class . . . next protocol definition to use (0 = none)” and the “next protocol lookup records” are described with reference to Table 4 as including a “Protocol” attribute describing the “pointer to protocol description record,” among other attributes. Baker at Table 2 (“ptr2np” attribute), Table 4 (“Protocol” attribute), 8:13-15; *also*

```
fread(&tmp, sizeof(tmp), 1, fp); // Read type of lookup structure
if ((ptr2np = alloc_lookup_structs(tmp, this)) != 0)
    ptr2np->get_from_file(fp); // Read all lookup values for field
```

Baker at 148:10-12; *also* 165:37-166:5, 177:4-26 (discussed in detail in Paragraphs 143-156 with respect to initialization).

217. Upon reading the Baker disclosure, one of ordinary skill in the art would recognize that Baker’s PDFs are written in a high-level programming language, because, upon initialization, the system is able to “extract[] the protocol and associated control record information” from the file, construct a ProtocolList, and read the PDF file into memory in the sequence described at 11:26-12:6; *see also* Paragraphs 143-156 above discussing initialization process. Accordingly, Baker’s PDFs include commands in a high-level language that describe protocols that may be encountered by the monitor and, for example, how to interpret header information and how to determine from the packet header information the protocol at the next layer, and therefore discloses protocol description files that are provided in a protocol description language, as required by claim 2.

218. I also note that the ‘725 patent discloses that when compiler and optimizer 310 “executes” it performs a compilation process that “generates,” “builds,” “fill[s],” or “load[s]” specific “internal data structures” during initialization. ‘725 patent at 9:42-44 (“When compiler and optimizer 310 executes, it generates two sets of internal data structures.”), 9:53-54 (“The other internal data structure that is built by compiler 310 is the set of state patterns and processes 326.”), 9:22-27 (“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.”), 22:7-10 (“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.”), 13:53-57 (“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.”), 14:63-15:17, Fig. 4.

219. Baker discloses a compilation process consistent with that disclosed in the ‘725 patent. Namely, Baker discloses implementation of the system on a network device capable of storing programmably configurable protocol description

files and programmed to run an initialization/compilation process that includes receiving PDF files and, as discussed above, “extract[ing] the protocol and associated control record information” from the file, constructing a ProtocolList, and reading the contents of the PDF file into memory in the sequence described at 11:26-12:6.

“[T]he initialization of the system includes a determination of the presence of PDF files and the extraction of the protocol and associated control record information from all of the PDF files found. The number of PDF files is determined, and a ProtocolList is constructed consisting of a sorted vector of protocol records at least the size of the number of PDF files found. The name of each protocol record found is inserted in the ProtocolList. The PDF files are then read to memory in the sequence described above for the PDF file writing. The lookup records that indicate a next protocol are associated with the appropriate entries in the ProtocolList.”

Baker at 12:7-18.

“In one preferred form, the system of the present invention may employ a CPU or other hardware implementable method for analyzing data from a network in response to selectively programmed parsing, filtering, statistics gathering, and display requests. Moreover, the system of the present invention may be incorporated in a network device, such as a network analyzer, bridge, router, or traffic generator, including a CPU and a plurality of input devices, storage devices, and output devices, wherein frames of network data may be received from an associated network, stored in the storage devices, and processed by the CPU based upon one or more programmably configurable protocol descriptions also stored in the storage devices.”

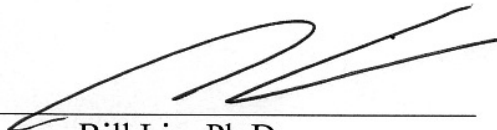
Baker at 2:63-3:7; see also Baker at 6:15-32, 6:41-47. I discuss the Baker initialization/compilation process in detail in Paragraphs 143-156 above. Additionally, as discussed in Paragraphs 143-177 above, the

initialization/compilation process produces a searchable and indexed database. Accordingly, a person having skill in the art at the time of the invention would understand Baker to disclose compiling the protocol description language descriptions to produce the database, as required by claim 2.

IX. CONCLUSION

220. I declare that all statements made herein on my own knowledge are true to the best of my knowledge and recollection, and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under section 1001 of Title 18 of the United States Code.

Date: 2/9/17

By: 
Bill Lin, Ph.D.

APPENDIX A

BILL LIN
1005 Valleyside Lane
Encinitas, CA 92024
Tel: 858.531.2441
Email: billmbox@gmail.com

EDUCATION

- **Ph.D.**, Electrical Engineering and Computer Sciences
University of California, Berkeley, May 1991
- **Masters of Science**, Electrical Engineering and Computer Sciences
University of California, Berkeley, May 1988
- **Bachelor of Science**, Electrical Engineering and Computer Sciences
University of California, Berkeley, May 1985

PROFESSIONAL EXPERIENCE

1/1997 – present

University of California, San Diego
Professor, Electrical and Computer Engineering
Adjunct Professor, Computer Science and Engineering

2/1992 – 12/1996

IMEC, Leuven, Belgium
Group Head, Systems Control and Communications Group

AREAS OF EXPERTISE

- All aspects of computer networks and computer architecture problems, including the design of data networks, high-performance switches and routers, high-performance packet monitoring and measurements, wireless communications and mobile computing, multimedia and graphics systems, many-core processors, and systems-on-chip designs. Source code analysis for communications and computing applications.

PROFESSIONAL ACTIVITIES

- Served or serving as Editor on 3 ACM or IEEE journals, as General Chair on 4 ACM or IEEE conferences, on the Organizing or Steering Committees for 5 ACM or IEEE conferences, and on the Technical Program Committees of 44 ACM or IEEE conferences.

SELECTED PROFESSIONAL ACTIVITIES

- General Chair, ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2010
- Chair of Steering Committee, ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2010-present (Chair since 2012)
- Guest Editor, IEEE Transactions on Network and Service Management (TNSM), 2012-2013
- General Co-Chair, ACM SIGMETRICS, 2015
- General Chair, ACM/IEEE International Symposium on Quality-of-Service (IWQoS), 2011
- Steering Committee, ACM/IEEE International Symposium on Quality-of-Service (IWQoS), 2011-present
- Associate Editor, ACM Transactions on Design Automation and Electronics Systems (TODAES), 2010-2015
- Editorial Board, International Journal of Embedded Systems, 2003-present
- General Chair, ACM/IEEE Symposium on Networks-on-Chips (NOCS), 2009

PUBLICATIONS

- Published over 170 journal articles and conference papers in top-tier venues and publications.

SELECTED PUBLICATIONS

- Bill Lin, Isaac Keslassy, "Frame-Aggregated Concurrent Matching Switch," ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), Orlando, FL, December 3-4, 2007, pp. 107-116.
- Jerry Chou, Bill Lin, "Coarse Circuit Switching by Default, Re-Routing Over Circuits for Adaptation," Journal of Optical Networking, vol. 8, no. 1, pp. 33-50, January 2009.
- Jerry C. Chou, Bill Lin, Subhabrata Sen, Oliver Spatscheck, "Proactive Surge Protection: A Defense Mechanism for Bandwidth-Based Attacks," IEEE/ACM Transactions on Networking, 17(6): 1711-1723 (2009).
- Bill Lin, Isaac Keslassy, "The Interleaved Matching Switch Architecture," IEEE Transactions on Communications, vol. 57, no. 12, December 2009.
- Bill Lin, Isaac Keslassy, "The Concurrent Matching Switch Architecture," IEEE/ACM Transactions on Networking, vol. 18, no. 4, August 2010, pp. 1330-1343.
- Chia-Wei Chang, Seungjoon Lee, Bill Lin, Jia Wang, "The Taming of the Shrew: Mitigating Low-Rate TCP Targeted Attack," IEEE Transactions on Network and Service Management, 7(1): 1-13 (2010).
- Nan Hua, Bill Lin, Jun Xu, Haiquan Zhao, "BRICK: A Novel Exact Active Statistics Counter Architecture," IEEE/ACM Transactions on Networking, vol. 19, no. 3, June 2011, pp. 670-682.
- Guanyao Huang, Chia-Wei Chang, Chen-Nee Chuah and Bill Lin, "Measurement-Aware Monitor Placement and Routing: A Joint Optimization Approach for Network-Wide Measurements," IEEE Transactions on Network and Service Management, vol. 9, no. 1, March 2012, pp. 48-59.
- Hao Wang, Haiquan Zhao, Bill Lin, and Jun Xu, "DRAM-based statistics counter array architecture with performance guarantee." IEEE/ACM Transactions on Networking, vol. 20, no. 4 (2012): 1040-1053.
- Hao Wang, Haiquan (Chuck) Zhao, Bill Lin, and Jun (Jim) Xu, "Robust Pipelined Memory System with Worst Case Performance Guarantee," IEEE Transactions on Computers, October 2012.
- Chia-Wei Chang, Han Liu, Guanyao Huang, Bill Lin and Chen-Nee Chuah, "Distributed Measurement-Aware Routing: Striking a Balance between Measurement and Traffic Engineering," IEEE INFOCOM, Orlando, FL, March 25-30, 2012.
- Xiaodong Yu, Bill Lin, Michela Becchi, "Revisiting State Blow-Up: Automatically Building Augmented-FA While Preserving Functional Equivalence," IEEE Journal on Selected Areas in Communications, 32(10): 1822-1833 (2014).
- Hao Wang, Bill Lin, "Reservation-Based Packet Buffers with Deterministic Packet Departures," IEEE Transactions Parallel Distributed Systems, 25(5): 1297-1305 (2014).
- Weijun Ding, Jun (Jim) Xu, Jim Dai, Yang Song, Bill Lin, "Sprinklers: A Randomized Variable-Size Striping Approach to Reordering-Free Load-Balanced Switching," ACM CoNEXT 2014.

PATENTS

- Mitigating Low-Rate Denial-of-Service Attacks in Packet-Switched Networks. United States Patent 8,443,444. Issued in May 14, 2013.
- Systems and Methods for Proactive Surge Protection. United States Patent 7,860,004. Issued in December 28, 2010.
- Method and Apparatus for Scan Block Caching. United States Patent 7,672,005. Issued in March 2, 2010.
- Design Environment and a Design Method for Hardware/Software Co-Design. United States Patent 5,870,588. Issued in February 9, 1999.
- System and Method for Generating a Hazard-Free Asynchronous Circuit. United States Patent 5,748,487. Issued in May 5, 1998.