(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
26 April 2007 (26.04.2007)

PCT

(10) International Publication Number
WO 2007/045839 A2

(51) International Patent Classification:
H04L 29/06 (2006.01) ***H04L 12/56*** (2006.01)

(21) International Application Number: PCT/GB2006/003828

(22) International Filing Date: 13 October 2006 (13.10.2006)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
11/255,124 20 October 2005 (20.10.2005) US

(71) Applicant *(for all designated States except US)*: **LEVEL 5 NETWORKS, INC.** [US/US]; 840 West California Avenue, Suite 240, Sunnyvale, CA 94086 (US).

(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: **POPE, Steve, L.** [GB/GB]; 25 Greville Road, Cambridge CB1 3QJ (GB). **ROBERTS, Derek** [GB/GB]; 25 Metcalfe Road, Cambridge CB4 2DB (GB). **RIDDOCH, David, J.** [GB/GB]; 68 Tenison Road, Cambridge CB1 2DW (GB). **YU, Ching** [US/US]; 2952 Gala Court, Santa Clara, CA 95051 (US). **CHIANG, John, Mingyung** [US/US]; 1975 Nutmeg Court, San Jose, CA 95131 (US). **CHU, Der-Ren** [US/US]; 6237 Empress Court, San Jose, CA 95129 (US).

(74) Agents: SLINGSBY, Philip, Roy et al.; Page White & Farrer, Bedford House, John Street, London WC1N 2BF (GB).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LV, LY, MA, MD, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RS, RU, SC, SD, SE, SG, SK, SL, SM, SV, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IS, IT, LT, LU, LV, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: HASHING ALGORITHM FOR NETWORK RECEIVE FILTERING

(57) Abstract: Roughly described, a network interface device is assigned a maximum extent-of-search. A hash function is applied to the header information of each incoming packet, to generate a hash code for the packet. The hash code designates a particular subset of the table within which the particular header information should be found, and an iterative search is made within that subset. If the search locates a matching entry before the search limit is exceeded, then the incoming data packet is delivered to the receive queue identified in the matching entry. But if the search reaches the search limit before a matching entry is located, then device delivers the packet to a default queue, such as a kernel queue, in the host computer system. The kernel is then responsible for delivering the packet to the correct endpoint.

HASHING ALGORITHM FOR NETWORK RECEIVE FILTERING

BACKGROUND

1. Field of the Invention

[0001] The invention relates to network interfaces, and more particularly to queue-based network receive mechanisms supporting a plurality of receive queues in a host computer system.

2. Description of Related Art

[0002] When data is to be transferred between two devices over a data channel, such as a network, each of the devices must have a suitable network interface to allow it to communicate across the channel. Often the network is based on Ethernet technology. Devices that are to communicate over a network are equipped with network interfaces that are capable of supporting the physical and logical requirements of the network protocol. The physical hardware component of network interfaces are referred to as network interface cards (NICs), although they need not be in the form of cards: for instance they could be in the form of integrated circuits (ICs) and connectors fitted directly onto a motherboard, or in the form of macrocells fabricated on a single integrated circuit chip with other components of the computer system.

[0003] Most computer systems include an operating system (OS) through which user level applications communicate with the network. A portion of the operating system, known as the kernel, includes protocol stacks for translating commands and data between the applications and a device driver specific to the NIC, and the device drivers for directly controlling the NIC. By providing these functions in the operating system kernel, the complexities of and differences among NICs can be hidden from the user level application. In addition, the network hardware and other system resources (such as memory) can be safely shared by many applications and the system can be secured against faulty or malicious applications.

[0004] It is desirable for the network interface device to be capable of supporting standard transport level protocols such as TCP, UDP, RDMA and ISCSI at user level: i.e. in such a way that they can be made accessible to an application program running on the computer. TCP is defined in RFC:0793 Transmission Control Protocol. J. Postel. Sep-01-1981, and UDP is defined in RFC:0768 User Datagram Protocol. J. Postel. Aug-28-1980, both incorporated by reference herein. Support of transport protocols at user level enables data transfers which require use of standard protocols to be made without requiring data to traverse the kernel stack. Involving the kernel stack requires context switches, which can significantly degrade performance of the computer system overall. To avoid this, standard transport protocols can be implemented both within transport libraries accessible only to the operating system of the computer as well as within transport libraries accessible to user level applications.

[0005] Fig. 1A illustrates one implementation of this. In this architecture the TCP (and other) protocols are implemented twice: as denoted TCP1 and TCP2 in Fig. 2. In a typical operating system TCP2 will be the standard implementation of the TCP protocol that is built into the operating system of the computer. In order to control and/or communicate with the network interface device an application running on the computer may issue API (application programming interface) calls. Some API calls may be handled by the transport libraries that have been provided to support the network interface device. API calls which cannot be serviced by the transport libraries that are available directly to the application can typically be passed on through the interface between the application and the operating system to be handled by the libraries that are available to the operating system or modules within the operating system. For implementation with many operating systems it is convenient for the transport libraries to use existing Ethernet/IP based control-plane structures: e.g. SNMP and ARP protocols via the OS interface.

[0006] There are a number of difficulties in implementing transport protocols at user level. Most implementations to date have been based on porting pre-existing kernel code bases to user level. Examples of these are Arsenic and Jet-stream. These have demonstrated the potential of user-level transports, but have not addressed a number of the problems required to achieve a complete, robust, high-performance commercially viable implementation.

[0007] Fig. 1B shows an architecture employing a standard kernel TCP transport (TCPk). In operation, on packet reception from the network interface hardware (NIC), the NIC transfers data into pre-allocated data buffer (a) and invokes the OS interrupt handler by means of the interrupt line. (Step I). The interrupt handler manages the hardware interface e.g. posts new receive buffers and passes the received (in this case Ethernet) packet looking for protocol information. If a packet is identified as destined for a valid protocol e.g. TCP/IP it is passed (not copied) to the appropriate receive protocol processing block. (Step ii).

[0008] TCP receive-side processing takes place and the destination port is identified from the packet. If the packet contains valid data for the port then the packet is engaged on the port's data queue (step iii) and that port marked (which may involve the scheduler and the awakening of blocked process) as holding valid data.

[0009] The TCP receive processing may require other packets to be transmitted (step iv), for example in the cases that previously transmitted data should be retransmitted or that previously enqueued data (perhaps because the TCP window has opened) can now be transmitted. In this case packets are enqueued with the OS "NDIS" driver for transmission.

[0010] In order for an application to retrieve a data buffer it must invoke the OS API (step v), for example by means of a call such as recv(), select() or poll(). This has the effect of informing the application that data has been received and (in the case of a recv() call) copying the data from the kernel buffer to the application's buffer. The copy enables the kernel (OS) to reuse its network buffers, which have special attributes such as being DMA accessible and means that the application does not necessarily have to handle data in units provided by the network, or that the application needs to know a priori the final destination of the data, or that the application must pre-allocate buffers which can then be used for data reception.

[0011] It should be noted that on the receive side there are at least two distinct threads of control which interact asynchronously: the up-call from the interrupt and the system call from the application. Many operating systems will also split the up-call to avoid executing too much code at interrupt priority, for example by means of "soft interrupt" or "deferred procedure call" techniques.

[0012] The send process behaves similarly except that there is usually one path of execution. The application calls the operating system API (e.g. using a send() call) with data to be transmitted (Step vi). This call copies data into a kernel data buffer and invokes TCP send processing. Here protocol is applied and fully formed TCP/IP packets are enqueued with the interface driver for transmission.

[0013] If successful, the system call returns with an indication of the data scheduled (by the hardware) for transmission. However there are a number of circumstances where data does not become enqueued by the network interface device. For example the transport protocol may queue pending acknowledgments or window updates, and the device driver may queue in software pending data transmission requests to the hardware.

[0014] A third flow of control through the system is generated by actions which must be performed on the passing of time. One example is the triggering of retransmission algorithms. Generally the operating system provides all OS modules with time and scheduling services (driven by the hardware clock interrupt), which enable the TCP stack to implement timers on a per-connection basis.

[0015] If a standard kernel stack were implemented at user-level then the structure might be generally as shown in Fig. 1C. The application is linked with the transport library, rather than directly with the OS interface. The structure is very similar to the kernel stack implementation with services such as timer support provided by user level packages, and the device driver interface replaced with user-level virtual interface module. However in order to provide the model of a asynchronous processing required by the TCP implementation there must be a number of active threads of execution within the transport library:

[0016] (I) System API calls provided by the application

[0017] (ii) Timer generated calls into protocol code

[0018] (iii) Management of the virtual network interface and resultant upcalls into protocol code. (ii and iii can be combined for some architectures)

[0019] However, this arrangement introduces a number of problems:

[0020] (a) The overheads of context switching between these threads and implementing locking to protect shared-data structures can be significant, costing a significant amount of processing time.

[0021] (b) The user level timer code generally operates by using operating system provided timer/time support. Large overheads caused by system calls from the timer module result in the system failing to satisfy the aim of preventing interaction between the operating system and the data path.

[0022] (c) There may be a number of independent applications each of which manages a sub-set of the network connections; some via their own transport libraries and some by existing kernel stack transport libraries. The NIC must be able to efficiently parse packets and deliver them to the appropriate virtual interface (or the OS) based on protocol information such as IP port and host address bits.

[0023] (d) It is possible for an application to pass control of a particular network connection to another application for example during a fork() system call on a Unix operating system. This requires that a completely different transport library instance would be required to access connection state. Worse, a number of applications may share a network connection which would mean transport libraries sharing ownership via (inter process communication) techniques. Existing transports at user level do not attempt to support this.

[0024] (e) It is common for transport protocols to mandate that a network connection outlives the application to which it is tethered. For example using the TCP protocol, the transport must endeavor to deliver sent, but unacknowledged data and gracefully close a connection when a sending application exits or crashes. This is not a problem with a kernel stack implementation that is able to provide the "timer" input to the protocol stack no matter what the state (or existence) of the application, but is an issue for a transport library which will disappear (possibly ungracefully) if the application exits, crashes, or stopped in a debugger.

[0025] In addition, in order that incoming data packets be delivered to the data port of the correct application, a mechanism is required which examines the header information of the packet and looks up the associated destination queue. Such filtering is commonly performed on a number of fields of the header, including source and destination ports and addresses. In order to maximize efficiency, it is preferable that the filtering be accomplished in hardware on the network interface device, rather than in software within the kernel or user level drivers.

[0026] One means of filtering packets in the network interface card is by presenting the packet header information to a content addressable memory (CAM) which associates each combination of header bit information with a specified receive port. But in a TCP/IP packet, for example, the header information may be as wide as 96 bits: 32 source IP address bits, 32 destination IP address bits, 16 source port bits, and 16 destination port bits. A CAM could be provided that is wide enough to accommodate the full width of all the fields against which filtering is to be performed, but such a CAM becomes prohibitively large and expensive if it is desired that the number of potential destination ports is large, and especially if it is to be implemented as only part of a custom integrated circuit. Modern servers in certain environments may require huge numbers of network endpoints, such that filtering through a CAM is not practical.

[0027] In situations where CAM lookups are too expensive, one might consider instead performing iterative lookups through a table, comparing the input data (the header bit information, in the case of an incoming data packet filter) with various entries in the table successively until a match is found. But iterative lookups can take a long time, often more time than is available before the next incoming packet arrives. Again, this problem worsens as the maximum number of network endpoints supported by the system increases.

[0028] In situations where CAM lookups are too expensive and iterative lookups will take too long, one might consider a hashed lookup, which is a form of hybrid between the parallel approach of a CAM and the sequential approach of an iterative lookup. In a hashed lookup, the table is subdivided into a number of subsets of entries. The input data is passed through a consistent mathematical "hashing function", which converts it to an index (called a hash code) pointing to the particular list (or "bucket") within which the data belongs. When new data is to be added to the table, it is simply inserted into an available free entry of the list pointed to by its hash code. When input data is to be located in the table, an iterative search is performed within the list pointed to by the hash code.

[0029] Many hashing algorithms are known, and some examples may be found in Knuth, "Art of Computer Programming", Volume 3: Sorting and Searching (2nd Edition), incorporated herein by reference. In one variation, the buckets pointed to by the hash codes are contiguous and of fixed length, and when a bucket is filled, additional entries are placed into a common overflow area. The buckets may even be limited to only one entry. In another variation, the buckets pointed to by the hash codes are linked lists of entries, which can have variable length.

All the linked lists may be interspersed together within a single combined region of memory. In yet another variation, known as open hashing, each bucket has a fixed length (for example a length of one), and if the bucket is filled, a secondary hash function is used to hop to a different next entry of the table. If that entry is filled, then the secondary hash function is used to hop again to yet another entry, and so on, until an available entry is found. Multiple level hashing algorithms are also available. In the open hashing case the subset of entries pointed to by the primary hash function is considered herein to include all the entries that will be reached in multiple iterations of the secondary hash function, so that the subset may be discontiguous. Whatever algorithm is used for placing entries into the table, the same algorithm is used for searching the table.

[0030] One problem that arises in most hashing algorithms is the problem of clustering: because the designer of the hashing algorithm does not know in advance what the input data will be that is to be stored in the table, there is always some probability that the hash function will distribute the data unevenly among the various possible hash codes. Uneven data distribution means that while the average length of a list (and therefore the average search time to find a matching entry) may be small, some hash codes will point to lists that are very long (and therefore take a long time to search). In implementations having a maximum list length, uneven data distributions increase the probability that some lists will overflow, forcing the search algorithm into its back-up mechanism, such as searching a common overflow area.

[0031] Different hashing algorithms handle the clustering problem differently, but most suffer from the problem that the iterative search time required to search the list pointed to by a heavily used hash code can become exorbitant. In the context of network receive packet filtering, such a long search time means the network interface device may not be ready to handle a future packet when it arrives, thereby potentially dropping the packet. Merely applying a hashing solution to the problem of filtering incoming data packets to identify the correct destination queue therefore may not be sufficient.

[0032] In order to address this issue, roughly described, a network interface device is assigned a maximum extent-of-search. A hash function is applied to the header information of each incoming packet, to generate a hash code for the packet. The hash code designates a particular subset of the table within which the particular header information should be found, and an iterative search is made within that subset. If the search locates a matching entry before the search limit is exceeded, then the incoming data packet is delivered to the receive queue identified in the matching entry. But if the search reaches the search limit before a matching entry is located, then device delivers the packet to a default queue, preferably a kernel queue, in the host computer system. The kernel is then responsible for delivering the packet to the correct endpoint.

BRIEF DESCRIPTION OF THE DRAWINGS

[0033] The invention will be described with respect to specific embodiments thereof, and reference will be made to the drawings, in which:

[0034] Fig. 1A illustrates an implementation of a transport library architecture.

[0035] Fig. 1B shows an architecture employing a standard kernel TCP transport with a user level TCP transport.

[0036] Fig. 1C illustrates an architecture in which a standard kernel stack is implemented at user-level.

[0037] Fig. 2 is a simplified block diagram of a typical computer system that can incorporate the invention.

[0038] Figs. 3A, 3B and 4 are software functional views of the computer system of Fig. 2.

[0039] Fig. 5 is a block diagram of various data structures used by the system of Fig 2.

[0040] Figs. 6-9 are flowcharts illustrating functions that take place for receiving data using the structures of Fig. 5.

[0041] Fig. 10 is a flow chart of pertinent steps that are performed by the operating system kernel of Fig. 2 on system startup.

[0042] Fig. 11 is a flow chart of pertinent steps performed by the kernel driver initialization routine called in Fig. 10.

[0043] Fig. 12 is an example broad outline of steps that an application process might follow in the system of Fig. 2 for the reception of IP packets from the network.

[0044] Fig. 13 is a flowchart detail of steps that might be performed during the library initialization step of Fig. 12.

[0045] Fig. 14 is a flowchart illustrating significant steps performed by the kernel resource allocation routine in Fig. 13.

[0046] Fig. 15 is a flowchart of the user level routine in the transport library of Fig. 3A for establishing a socket.

[0047] Fig. 16 is a flowchart of the user level routine transport library of Fig. 3A for binding a port number to a socket.

[0048] Fig. 17 is a flowchart of pertinent steps of the user level recvFrom() routine called in step Fig. 12.

[0049] Fig. 18 is a flowchart of pertinent steps of the user level listen() routine called in step Fig. 12.

[0050] Fig. 19 is a flowchart of pertinent steps of the kernel routinely for creating a new filter table entry.

[0051] Fig. 20 is a flowchart of pertinent steps of a routine in the user level driver called for removing a filter entry.

[0052] Fig. 21 is a detail of the filter table and logic unit of Fig. 5.

[0053] Fig. 22 is a flow chart illustrating the overall functions performed by filter table search logic of Fig. 21 in order to search for a match.

[0054] Fig. 23 is a flow chart detail of the step in Fig. 22 of performing a TCP_full search.

[0055] Fig. 24 is a flow chart detail of the step in Fig. 22 of performing a TCP_wildcard search.

[0056] Fig. 25 is a flow chart detail of the step in Fig. 22 of performing a UDP_full search.

[0057] Fig. 26 is a flow chart detail of the step in Fig. 22 of performing a UDP_wildcard search.

[0058] Fig. 27 is a flow chart showing pertinent steps that the kernel driver performs upon receipt of a data packet.

DETAILED DESCRIPTION

[0059] The following description is presented to enable any person skilled in the art to make and use the invention, and is provided in the context of a particular application and its requirements. Various modifications to the disclosed embodiments will be readily apparent to those skilled in the art, and the general principles defined herein may be applied to other embodiments and applications without departing from the spirit and scope of the present invention. Thus, the present invention is not intended to be limited to the embodiments shown, but is to be accorded the widest scope consistent with the principles and features disclosed herein.

[0060] Fig. 2 is a simplified block diagram of a typical computer system 210 which can communicate via a network 212 with other computer systems such as 230, 232 and 234. Computer system 210 includes a network interface card (NIC) 216 communicating via a communication channel 218 with a host subsystem 214. The host subsystem 224 includes a processor subsystem 220 which includes at least one processor, a host memory subsystem 222, and a core logic subsystem 224. The core logic subsystem 224 provides bridges among the processor subsystem 220, the host memory subsystem 222 and the communication channel 218. The host subsystem 214 may also include other devices 226 in communication with the communication channel 218. As used herein, the host subsystem 214 is considered to be one type of "computing device". Other types of computing devices include non-reconfigurable processing devices for which network connectivity is desired.

[0061] The network interface card 216 provides an interface to outside networks, including an interface to the network 212, and is coupled via network 212 to corresponding interface devices in other computer systems. Network 218 may comprise many interconnected computer systems and communication links. These communication links may be wireline links, optical links, wireless links or any other mechanism for communication of information. While in one embodiment network 218 is the Internet, in other embodiments, network 218 may be any suitable computer network or combination of networks. In and embodiment described herein, network 218 supports an Ethernet protocol.

[0062] Host memory subsystem 222 typically includes a number of memories including a main random access memory (RAM) for storage of instructions and data during program execution, and a read only memory (ROM) in which fixed instructions and data are stored. One or more levels of cache memory may also be included in the host memory subsystem 222. For simplicity of discussion, the host memory subsystem 222 is sometimes referred to herein simply as "host memory". As used herein, virtual memory is considered part of the host memory subsystem even though part of it may be stored physically at various times on a peripheral device.

[0063] The communication channel 218 provides a mechanism for allowing the various components and subsystems of computer system 210 to communicate with each other. In one embodiment the communication channel 218 comprises a PCI Express bus. Other embodiments may include other buses, and may also include multiple buses. The PCI bus and its progeny, including the version known as PCI Express, support burst transfer protocols such as that described above. PCI express is described in PCI Special Interest Group, "PCI Express Base Specification 1.0a", April 15, 2003, incorporated herein by reference.

[0064] Computer system 210 itself can be of varying types including a personal computer, a portable computer, a workstation, a computer terminal, a network computer, a television, a mainframe, a server or any other data processing system or user devices. Due to the ever-changing nature of computers and networks, the description of computer system 210 depicted in Fig. 2 is intended only as a specific example for purposes of illustrating an embodiment of the present invention. Many other configurations of computer system 210 are possible having more or less components, and configured similarly or differently than, the computer system depicted in Fig. 2.

[0065] Fig. 3A is another view of the computer system 210, presented to better illustrate certain software and functional features. The computer runs an operating system 321 which is capable of supporting application processes 322 also running on the computer. As used herein, a "process" is a program, for example an application program, in execution. A process may run multiple "threads", which can also be thought of as lightweight processes. A library 323 of instructions is stored by the computer and available to the applications. The part of the library usable for communications with the NIC 216 is termed a transport library 324. Included in the operating system kernel is a driver component 325: a dedicated set of instructions which allow for data transfer with the NIC 216. Each application would normally take the form of a software program running on the computer, but it could be embedded in firmware. Some operating systems include applications in addition to fundamental operating system code.

[0066] The NIC 216 supports resources of a number of types: i.e. resources having capabilities of different natures. Examples include DMA queues, event queues, timers and support resources for remote apertures of the type described in WO2004/025477, incorporated by reference herein. Each type of resource (331-334) is provided from a dedicated hardware resource pool which can support numerous instances of resources of the respective type. In order for such an instance to be made operational it is configured by means of instructions from the computing device 214, as described in more detail below.

[0067] The NIC 216 communicates with the computing device 214 over the bus 218. In this example the bus is a PCI bus, but the invention is not limited to such a bus. Data transmitted over the PCI bus 218 is associated with a destination address and is received by whichever entity that is connected to the bus has had that address allocated to it. In a typical PC implementation the addresses are allocated in pages of 4 or 8 kB. One or more of these pages may be allocated to the NIC 216. Blocks 351 and 352 represent allocated pages on the PCI bus 218.

[0068] The NIC 216 has a bus interface controller (BIC) 335, a resource configuration unit (RCU) 336 and a bus mapping table 337. The resource configuration unit processes communications received from the computer that provide instructions on the allocation, re-allocation and de-allocation of resources on the NIC 216, and configures the resources in accordance with such instructions. The kernel driver 325 stores a record of which resources on the NIC 216 are allocated. When a resource is to be allocated the driver 325 identifies a suitable free resource of the required type on the NIC 216 and transmits an allocation instruction to the NIC 216. The instruction identifies the resource and specifies the details of how it is to be allocated, including details of the internal configuration of the resource (e.g. in the case of a timer the amount of time it is to run for). That instruction is passed to the resource configuration unit. The resource configuration unit then loads the specified configuration into the identified resource. The instruction also includes an ownership string, which may be an identification of which application or process on the computer is using the resource. The resource configuration unit stores these in a row of the bus mapping table. When a resource is to be re-allocated the relevant entries in the resource's own configuration store and in the bus mapping table are altered as necessary. When a resource is to be de-allocated it is disabled and any rows of the bus mapping table that relate to it are deleted.

[0069] During setup of the system one or more pages (351, 352) on the bus 218 are allocated to the NIC 216. Part of this address space (page 351) can be used by the kernel driver 325 to send instructions to the NIC 216. Other pages (e.g. page 352) can be used for communication between application processes such as application 322 and the resources 331-334. The resource configuration unit 336 stores a record of the pages that are allocated to the NIC 216 for use by resources. Note that in some embodiments, some or all of the functions of the resource configuration unit 336 may alternatively be provided by the kernel driver 325 itself.

[0070] When an application 322 wishes to open a data connection over the network it calls a routine in the user level transport library 324 to cause the NIC resources that are required for the connection to be allocated. Standard types of network connection require standard sets of resources; for example: an event queue, transmit and receive DMA command queues, and a set of DMA'able memory buffers. For example a typical set may contain one TX command queue, one RX command queue, two timers, and on the order of 100-1000 DMA memory buffers.

[0071] The user level transport library 324 includes routines that can be called directly by the application process 322 and that initiate the allocation of such standard sets of resources, including set numbers of resources of different types. The transport library also includes routines that allow a resource of each type to be allocated, re-allocated or de-allocated individually. The presence of both these types of instruction means that standard connections can be set up efficiently, and yet non-standard groups of resources can be created, and existing connections can be reconfigured on a resource-by-resource basis. As used herein, a "user level stack" is any protocol processing software that runs in unprotected mode. A "protocol stack" is the set of data structures and logical entities associated with the networking interfaces. This includes sockets, protocol drivers, and the media device drivers.

[0072] The routines for allocation, re-allocation and de-allocation of resources require access to restricted memory mapped addresses, such as page 351 for sending configuration instructions to the NIC 216. Since the user level transport library 324 lacks the necessary privilege level to perform these accesses, these routines in the user level transport library 324 make calls to the kernel driver 325. In a Unix environment, for example, such calls might take the form of IOCtl() system calls. These calls cause an initial context switch to a kernel level process, which in turn communicate the instructions to the NIC 216 for the allocation of the resources as specified in the routines. Those instructions specify the identity of the application or process with which the resources are to be associated, and the nature of the resources. The instructions are processed by the resource configuration unit 336 of the NIC 216.

[0073] The space on the bus 218 that is allocated to the NIC 216 can be split dynamically between the resources on the bus 218. Once one or more pages 352 have been allocated to the NIC 216 for use by resources those resources can be allocated one or more individual sub-page addresses within that page, corresponding to locations as illustrated at 353, 354. Thus each resource can have a part of the total space allocated to it. A record of which part of the total space is allocated to which resource is stored in the bus mapping table 337. The effect is that a single page of the bus can be used for communication to resources of multiple types and/or resources that relate to multiple connections and/or resources that are associated with multiple applications or processes on the computer 214. As a result, the total bus space can be used relatively efficiently.

[0074] The usage of the allocated bus space 352 is managed by the kernel driver 325. When a resource is to be allocated the RCU identifies using a data store whose content it manages an unused block in the space on the bus that has already been allocated for use by resources of the NIC 216, the space being of the size required for the resource. It then stores in that data store the identity of the resource ("resource ID"), the address of the block within the allocated space ("sub-page ID"), and the identity of the application or process that is to use the resource ("process tag"); and sends a message to the resource configuration unit 336 to cause it to store corresponding data in the bus mapping table 337. If the RCU finds that table 337 indicates the address to be already occupied then it returns an error code to the driver. The sub-page address may need to be supplemented with the address of the page in which the sub-page lies if that cannot be inferred as a result of only a single page having been allocated for use by the resources. If the total space allocated for use by resources is insufficient then the kernel driver allocates it more space. Having allocated the resources, the RCU returns a success message to the kernel driver. The allocated page and sub-page addresses are returned to and mapped into the virtual address space of the user level process that requested the resources in order that it can access them by means of that data. Another context switch then takes place back to the user level calling process.

[0075] An application that has had resources allocated to it can access them by sending data (e.g. by means of load/store cycles through a virtual memory mapping) to the relevant bus page, at the sub-page address corresponding to the respective resource. Since these addresses are part of the application's virtual address space, no context switch to any kernel level processes are required in order to perform these accesses. Any data sent to pages allocated to resources is picked off the bus 218 by the bus interface controller 335. It directs that data to the appropriate one of the resources 331-334 by performing a look-up in the table 337 to identify the identity of the resource to which the sub-page address has been allocated. An application can also access a resource by means other than a bus write: for example by means of direct memory access

(DMA). In those instances, the NIC 216 checks that the identity of the application/process from which the access has been received matches the identity indicated in the table 337 for the resource. If it does not match, the data is ignored. If it matches, it is passed to the relevant resource. This adds to security and helps to prevent corruption of the resources by other applications.

[0076] The set of resources allocated to an application or process may be considered to constitute a virtual network interface (VNIC).

[0077] Once a virtual interface has been composed, it may be reconfigured dynamically. As one example of dynamic reconfiguration, a resource that is no longer required may be freed-up. To achieve this the application using the resource calls a de-allocation routine in the user level transport library 323. The de-allocation routine calls the kernel driver 325, which instructs the RCU to de-allocate the resource by disabling it, clearing its status and deleting its row in the table 337.

[0078] As another example of dynamic reconfiguration, additional resources may be added to the VNIC. The process is analogous to that described above for initial composition of the VNIC.

[0079] As yet another example of dynamic reconfiguration, resources may be passed from one application or process to another. This is most useful in the situation where a single application has multiple processes and wants to pass control of a resource from one process to another, for example if data from the network is to be received into and processed by a new process. To achieve this the application using the resource calls a re-allocation routine in the transport library 323. The re-allocation routine calls the kernel driver 325, which instructs the

RCU to re-allocate the resource modifying its row in the table 337 to specify the identity of the application or process that is taking over its control.

[0080] In some instances it may be desirable for resources of one type to communicate with resources of another type. For example, data received from the network 240 may be being passed to an application 322 for processing. The application has a queue 326 in a memory 327 connected to the bus 218. The queue is managed in part by the transport library 323, which provides a DMA queue resource 331 on the NIC 216 with an up-to-date pointer to the next available location on the queue 326. This is updated as the application reads data from the queue 326. When data is received from the network it is passed to an event queue resource 332, which writes it to the location identified by the pointer and also triggers an event such as an interrupt on the computing device 214 to indicate that data is available on the queue. In order for this to happen the event queue resource 332 must learn the pointer details from the DMA queue resource 331. This requires data to be passed from the DMA queue resource to the event queue resource.

[0081] To achieve this the "process tag" column of the table 337 can be treated more generally as an ownership tag, and can link the DMA queue to the related event queue. To achieve this the ownership tag of the event queue can be set to the identity of the related DMA queue. When the DMA queue needs to pass data to the related event queue it can identify the event queue from the table 337 by performing a look-up on its own identity in the ownership tag column.

[0082] Data intended to be passed from one resource to another can be checked by the bus interface controller 335 to ensure that it is compatible with the settings in the table 337. Specifically, when data is to be sent from one resource to another the bus controller checks that there is a row in the table 337 that has the identity of the resource that is the source of the data in the ownership tag field, and the identity of the resource that is the intended destination of the data in the resource ID field. If there is no match then the data is prevented from reaching its destination. This provides additional security and protection against corruption. Alternatively, or in addition, it may be permitted for one resource to transmit data to another if both are in common ownership: in this example if their resource ID fields indicate that they are owned by the same process, application or other resource.

[0083] The identities of resources linked in this way can also be reconfigured dynamically by means of the re-configuration routines in the transport library.

[0084] Fig. 3B shows yet another software functional view of the system of Fig. 2. Some of the features illustrated in Fig.3B are as follows:

[0085] (I) TCP code which performs protocol processing on behalf of a network connection is located both in the transport library, and in the OS kernel. The fact that this code performs protocol processing is especially significant.

[0086] (ii) Connection state and data buffers are held in kernel memory and memory mapped into the transport library's address space

[0087] (iii) Both kernel and transport library code may access the virtual hardware interface for and on behalf of a particular network connection

[0088] (iv) Timers may be managed through the virtual hardware interface, (these correspond to real timers on the network interface device) without requiring system calls to set and clear them. The NIC generates timer events which are received by the network interface device driver and passed up to the TCP support code for the device.

[0089] It should be noted that the TCP support code for the network interface device is in addition to the generic OS TCP implementation. This is suitably able to co-exist with the stack of the network interface device.

[0090] As a result of the above mechanisms, the operating system and many application programs can each maintain multiple TX, RX and Event DMA command queues. Fig. 4 illustrates this feature. As can be seen, the operating system 321 maintains, via kernel driver 325, TX, RX and Event data queues. Each such queue has an associated DMA command queue, not shown in Fig. 4 but maintained in the host memory 222 by the kernel driver 325. Multiple applications 322 can also be running in the computer system, each with its own instance of user level driver 324. Each such application 322 maintains, via its respective user level driver instance 324, TX, RX and Event data queues. As with the kernel queues, each such TX, RX and Event data queue has an associated DMA command queue, not shown in Fig. 4 but maintained in the host memory 222 by the respective user level driver 324. Note that the kernel driver 325 is also able to communicate data packets received by the kernel, to the user level driver of individual target applications. This communication occurs by standard interprocess communication mechanisms of the operating system.

[0091] Fig. 5 is a block diagram of various data structures used by the system of Fig 2 in order to support separate transmit and receive queues for each of the VNICs. The diagram indicates which structures exist in host memory 222 and which exist on the NIC 216. The transmit and receive data buffers, the transmit and receive DMA descriptor queues, as well as one or more event queues, are all resident in host memory 222 and made up of generalized buffers which can be discontiguous and interspersed with each other in host memory 222. In Fig. 5, the buffers being used as receive data buffers are identified as "RX DATA BUF #n", and the buffers being used for a RX DMA command queue (or more simply, a receive queue) are identified as "RX QUEUE BUF #n". The buffers being used for the event queue are identified as "EV QUEUE BUF #n". Additional buffers in host memory 222, not shown explicitly in Fig. 5, are used for transmit data buffers and for a TX DMA command queue (also called simply a transmit queue). One process may have any number of transmit, receive and event queues, and all of them share the pool of generalized buffers that have been mapped into that process's virtual address space. Other processes use a different pool of generalized buffers mapped into *their* respective virtual address spaces.

[0092] Individual buffers may be either 4k or 8k bytes long in one embodiment, and they are chained together into *logically* contiguous sequences by means of *physically* contiguous descriptors in a buffer descriptor table 510 stored in the NIC 216. For example, one receive queue might occupy buffers 512, 514 and 516 in host memory 222, which are discontiguous and possibly out-of-order regions of memory. They are chained together into a single *logically* contiguous space by the *physically* contiguous entries 518, 520 and 522 in the buffer descriptor table 510. The entries 518, 520 and 522 are written and managed by the host 214 kernel driver and are viewed as a wrap-around ring. So for example, if the host wishes to define a receive queue having 64k entries for receive data buffer descriptors, and each buffer is 4k in size, then the host will allocate a physically contiguous sequence of 16 entries in buffer descriptor table 510 for this receive queue. Similarly, one event queue might occupy buffers 526, 528 and 530 in host memory 222. These buffers are discontiguous and possibly out-of-order in host memory, but are chained together into a single *logically* contiguous wrap-around space by the *physically* contiguous entries 532, 534 and 536 in the buffer descriptor table 510. The buffer descriptor table 510 is indexed by "buffer ID", and each of its entries identifies, among other things, the base address of the corresponding buffer in host memory 222.

[0093] In order to keep track of the state of each of the transmit, receive and event queues for the many user-level applications that might be in communication with NIC 216 at the same time, the NIC 216 includes a receive queue descriptor table 540, a transmit queue descriptor table 541, and an event queue descriptor table 542. The transmit, receive and event queue descriptor tables are shown in Fig. 5 as separate tables, each containing the entire table, but it will be appreciated that in different embodiments the three tables can be implemented as a single unified table, or one of the tables can be implemented as separate sub-tables (divided by columns or by rows or by both), or some combination of these variations might be implemented. Each receive queue has a corresponding receive queue ID, which is used as an index into the receive queue descriptor table 540. The designated entry in the receive queue descriptor table 540 is the starting point for describing the state and other characteristics of that particular receive queue, as viewed by the NIC 216. Each such entry identifies, among other things:

[0094] * whether the queue is a kernel queue, user queue or another kind of queue;

[0095] * the size of the queue (number of receive data buffer descriptors it can contain)

[0096] * the ID of the event queue associated with this receive queue;

[0097] * buffer ID of base buffer for this receive queue;

[0098] * "device centric" read and write pointers into this receive queue.

[0099] The host 214 maintains "host centric" versions of the read and write pointers as well, and when it has added additional receive buffers to the queue, it so notifies the NIC 216 by writing its updated host centric receive queue write pointer into the address on the NIC of the device centric receive queue write pointer for the particular receive queue.

[0100] As shown in Fig. 5, the NIC 216 also includes a filter table and logic block 550. Because the NIC 216 can support multiple simultaneous connections between user-level applications and remote agents on LAN 212, and because the NIC 216 supports these using multiple transmit and receive queues, one function performed by the NIC 216 is to direct each incoming data packet to the correct receive queue. The mechanisms used by NIC 216 to make this determination are described in detail hereinafter, but generally the filter table and logic 550 maintains a correspondence between packet header information and destination receive queue ID. The filter table and logic 550 thus uses the header information from the incoming packet to determine the ID of the proper destination receive queue and uses that receive queue ID to index into the receive queue descriptor table 540. The receive queue ID is the starting point for the NIC 216 to obtain all required information about the destination receive queue for proper forwarding of the packet data.

[0101] Thus logically described, in order to deliver a received data packet to the destination receive queue in host memory 222, the NIC 216 first uses the header information of the data packet to look up, in the filter table 550, the appropriate destination receive queue ID. It then uses the ID of the particular receive queue to look up, in the receive queue descriptor table 540, the buffer ID of the base buffer containing the receive descriptor queue. The NIC 216 also obtains from the same place, the current device centric read pointer into that receive descriptor queue. It then uses the base buffer ID as a base, and the device centric read pointer high order bits as an offset, into the buffer descriptor table 510, to obtain the base address in host memory 222 of the buffer that contains the particular receive queue. The NIC then uses that base address as a base, and as an offset the device centric receive queue read pointer low order bits times the number of bytes taken up per descriptor, as a starting host memory address for retrieving entries from the particular receive descriptor queue. The NIC does not allocate separate space for maintaining a write pointer into any local cache into which these entries will be written.

[0102] Each entry in the particular receive descriptor queue contains, among other things:

[0103] * the buffer physical address of a receive data buffer; and

[0104] * a 2-byte aligned offset into the receive data buffer.

[0105] Entries for kernel receive descriptor queues can identify the buffer physical address itself, rather than a buffer ID, because the kernel is trusted to write the correct physical address whereas a user level queue is not.

[0106] The NIC 216 then uses the buffer ID of the current receive data buffer as another index into buffer descriptor table 510 to retrieve the buffer descriptor for the buffer into which the current receive data is to be written. Note this buffer descriptor is an individual entry in buffer descriptor table 510; unlike the descriptors for buffers containing receive queues or event queues, this buffer descriptor is not part of a ring. The NIC 216 obtains the physical address in host memory 222 of the current receive data buffer, and then using that physical address as a base, and the 2-byte aligned offset from the receive descriptor queue entry as an offset, it determines the physical starting address in host memory 222 into which the data transfer should begin. The NIC 216 then transfers the received data packet into host memory 222 beginning at that address.

[0107] The system handles transmit queues in a similar manner.

[0108] The receive queue descriptor table 540 entry designated by the receive queue ID, as previously mentioned, also contains the ID of the receive event queue associated with the particular receive queue. Similarly, the transmit queue descriptor table 541 entry designated by the transmit queue ID contains the ID of the event queue associated with the particular transmit queue. All of the event queues for all the applications 322 are described by respective entries in the event queue descriptor table 542. The entry in the event queue descriptor table 542 identified by a queue ID from the receive or transmit queue descriptor table 540 or 541 is the starting point for describing the state and other characteristics of that particular event queue, as viewed by the NIC 216.

[0109] Note that as illustrated in Fig. 5, whereas each slot (e.g. 532, 534, 518) shown in the buffer descriptor table 510 represents a single descriptor, each slot (e.g. 526, 528, 514) in the host memory 222 represents a memory "page" of information. A page might be 4k or 8k bytes long for example, so if a receive data buffer descriptor in a receive queue occupies either 4 or 8 bytes, then each slot 512, 514 or 516 as shown in Fig. 5 might hold 512, 1k or 2k receive data buffer descriptors.

[0110] Figs. 6-9 are flowcharts illustrating functions that take place for receiving data using the structures of Fig. 5. In Fig. 6, a host receive event management module in the user level transport library for a given application process receives an indication from other software to expect data from the network (step 611). The module is also activated in response to receipt by the host of a receive queue empty event as described hereinafter (step 610). The module also may be activated periodically on expiration of a polling loop or timer (step 612). The host subsystem 214 will push receive buffers onto this receive queue in response to these triggers, but limited so as to guarantee that the corresponding event queue will not overflow. The host subsystem 214 therefore will not queue more data buffers for receive data than can be accommodated in the receive event queue by the number of receive completion events that would be generated.

[0111] After determining the amount of space currently available in the receive event queue, in step 616 the host subsystem 214 determines a number 'M', being the lesser of the number of data buffers available for queuing of receive data, and the minimum number of receive data buffers that can be represented by receive completion events in the space available in the receive event queue as determined in step 614.

[0112] In step 618, it is determined whether M is greater than or equal to some minimum threshold. Preferably the minimum threshold is 1, but in other embodiments a larger number may be chosen for the threshold. If M is less than the threshold, then the host receive event queue management module 600 simply goes inactive to await the next activation event (step 630).

[0113] If M is greater than or equal to the minimum threshold, then in step 622, the host subsystem 214 updates (modulo-increments) its host centric receive queue write pointer by M entries. In step 624 the host subsystem 214 writes M available receive data buffer descriptors into the receive queue beginning at the entry previously (before step 622) designated by the host centric receive queue write pointer. In step 626 the host subsystem 214 notifies the NIC 216 of the updated write pointer, and in step 628, the NIC 216 updates its own device centric receive queue write pointer for the specified receive queue. In one embodiment, steps 626 and 628 are combined into a single step in which the host subsystem 214 writes the updated write pointer into a memory mapped location of the device centric receive queue write pointer. In step 630, the host receive event queue management module goes inactive to await the next activation event.

[0114] Fig. 7 is a flowchart illustrating functions initiated on the NIC 216 upon receipt of data from the network 212 which it has determined is destined for a particular receive queue.

The incoming data is placed into a RX FIFO, and when a high water mark is reached, the NIC receive data module 700 is activated (step 710). In step 712, the NIC 216 retrieves the descriptor of the next receive data buffer and offset from the entry of the receive queue designated by device centric receive queue read pointer for the particular destination queue. In step 714, the NIC 216 updates (modulo-increments) its device centric receive queue read pointer for that queue. The NIC does not at this time notify the host 214 of the new read pointer since that notification will occur implicitly or explicitly through event delivery as described below.

[0115] In step 716, the NIC 216 writes data from the incoming packet into the receive data buffer designated by the retrieved descriptor, beginning at the specified offset. Writing continues by DMA until either the end of the current data buffer is reached or the end of the incoming data packet is reached, or both.

[0116] The NIC 216 detects and reports a queue empty alert when it believes it has retrieved and used the last buffer descriptor in the particular receive queue. This alert is combined into a single event descriptor with the receive completion event. In particular, the NIC 216 determines in step 718 whether it believes it has used the last receive buffer identified by a descriptor in the receive queue. The NIC can determine this by comparing its device centric receive queue read pointer to its device centric receive queue write pointer for the particular receive queue. If not, that is the NIC knows there are more receive buffer descriptors in the receive queue, then no alert is necessary and in step 720, the NIC determines whether end-of-packet has been reached. If not, then the NIC receive data module 700 returns to step 712 to retrieve the descriptor for the next receive data buffer. No event is asserted to indicate Receive Data Buffer Full in this embodiment. The host 214 will become aware of which receive data buffers are full based on the receive data buffers identified consecutively in the receive queue, beginning at the host-centric RX queue read pointer.

[0117] If step 720 determines that end-of-packet was reached, then in step 722 the NIC 216 asserts a receive completion event to cover all the receive data buffers that contain data from the packet. The receive completion event descriptor format includes a receive descriptor queue empty flag ('rx_desc_q_empty'), but in the receive completion event written in step 722, this flag is not set because the NIC 216 has determined (in step 718) that additional receive buffer descriptors remain in the receive queue. Note that in this embodiment, only one receive completion event will be asserted even if the packet data spans multiple buffers in receive data buffers. Multiple buffers are chained together by consecutive entries in the receive queue. Note also that if end-of-packet does not coincide with the end of a receive buffer, then the remaining space in the buffer is left unused.

[0118] Returning to step 718, if the NIC 216 believes that the last receive data buffer identified by a descriptor in the receive queue has been retrieved in step 712, then the NIC does not wait until end-of-packet before reporting the receive completion event. Instead, in step 724 the NIC asserts a receive completion event to cover all the receive data buffers that contain data from the packet. In this receive completion event the rx_desc_q_empty flag is set. If packet data remains in the NIC's RX FIFO when this occurs, it is lost.

[0119] In an embodiment in which the NIC supports more than one network port, the NIC 216 does not batch receive completion events. Receive completion events do not indicate completion of more than one receive data buffer. This embodiment supports both standard size data packets, in which data packets have a relatively small maximum length and the receive data buffers are at least as large as the maximum data packet length, and "jumbo" data packets, in which a data packet can be longer and can span more than one data buffer. A given receive queue is either in standard mode or jumbo mode. If the queue is in standard mode, then absent an error, every receive data buffer filled will contain an end-of-packet, so no receive completion event will indicate completion of more than one data buffer and the problem will not arise. If the queue is in jumbo mode then it is still the case that no receive completion event will indicate completion of more than one data buffer, since the NIC writes a receive completion event for each data buffer it fills. The receive completion event format includes a "RX_Jumbo_Cont" bit which the NIC sets in order to notify the host subsystem that the subject data buffer does not contain an end-of-packet (i.e. there will be a continuation buffer). This embodiment therefore does not batch receive completion events. The receive completion event still includes a copy of the NIC's updated device centric receive queue read pointer, which now points to the specific descriptor from the receive queue for whose data buffer the event indicates completion. The receive completion event format also indicates the NIC port number from which the packet was received.

[0120] Returning to the Fig. 7 embodiment, after both steps 724 and 722, once the NIC has asserted a receive completion event, the NIC receive data module 700 then returns to an inactive state (step 726).

[0121] In both steps 724 and 722, the NIC asserts a receive completion event containing certain information. Fig. 8 is a flowchart detail of this step. In step 810, the NIC 216 writes the receive completion event into the corresponding receive event queue beginning at the entry identified by the device centric receive event queue write pointer for that event queue. In step 812, NIC 216 correspondingly updates its own receive event queue write pointer. In step 814, if enabled, the NIC 216 generates a wake-up event for this event queue and writes it into an event queue associated with a char driver in the kernel. In step 816, again if enabled, the NIC 216 generates an interrupt to activate the host char driver event queue handler, then disables interrupts. In step 818, the host char driver event queue handler, upon reaching the wake-up event, activates the receive event handler in the process that owns the specified receive queue.

[0122] Fig. 9 is a flow chart illustrating pertinent functions of a host receive event handler module 900 of the user level transport library of the process that owns the particular destination receive queue. Module 900 is activated either by the host char driver in response to a wakeup event (step 818, Fig. 8), or on selection periodically by a polling loop or expiration of a timer (step 912).

[0123] In step 920, the host subsystem 214 retrieves the event descriptor at the location in the event queue designated by the receive event queue read pointer. If this new event is not in the cleared state (step 922), then the receive event queue contains events for handling at this time. In step 926, it is determined whether the new event is a receive completion event. In one embodiment receive event queue cannot contain any events other than receive completion events, but in another embodiment it can. Thus if the current event is something other than a receive completion event, such as a management event, then it is handled in step 928.

[0124] If the current event is a receive completion event, then in step 940 the host 214 determines whether the Receive Queue Empty flag is set. If so, then the module (in step 942) triggers the host receive event queue management module 600 in order to replenish the receive queue with additional receive data buffers. In step 944, the host 214 determines further whether any of a variety of error types are indicated by the receive completion event descriptor. If so, then in step 946 the host 214 handles the error. Note that some of the error types included in step 944 may actually be detected before or after the receive queue empty test of step 940, some may bypass the replenishment (triggered by step 942) of receive buffer descriptors in receive queue for the time being, and some may bypass processing of the data packet in step 948. The details of such error handling are not important for an understanding of the invention.

[0125] In step 948, assuming no serious error has been detected, the host 214 processes the newly received packet data, including protocol processing. This may require chaining together several receive data buffers in sequence as designated by consecutive receive queue entries. The host 214 knows the starting buffer and offset of the packet from the buffer descriptor in the receive queue pointed to by the host centric receive queue read pointer, and knows the end of the packet either from the receive packet byte count identified in the receive completion event or from the copy of the device centric receive queue read pointer that might be included in the receive completion event. After processing the packet data in these buffers, the host may release the buffers back into a pool for eventually re-writing into the receive queue for re-use by different incoming packet data.

[0126] In step 950, if the higher level software is so designed, the host subsystem 214 may reprogram the receive queue entry pointed to by the host centric receive queue read pointer with a descriptor for a new available receive data buffer, and may do the same with respect to all consecutively subsequent receive queue entries up to but not including the receive queue entry pointing to the beginning of data for the next receive packet. In step 930, the host subsystem 214 modulo-increments the host centric receive queue read pointer for the receive queue by the number of buffers represented in the current receive completion event. In step 932, the host subsystem 214 clears the event descriptor at the location in receive event queue identified by the current receive event queue read pointer, and in step 934 the host subsystem 214 modulo-increments the receive event queue read pointer. The module then loops back to step 920 to retrieve the next event descriptor, and so on until a cleared entry is retrieved and the module goes inactive (step 924).

[0127] If in step 922 it is determined that the retrieved next event descriptor is cleared, then the receive event queue contains no more events for handling at this time. In one embodiment, the host receive event handler 900 would then simply go inactive to await the next activation trigger (step 924). In another embodiment, in step 923, if the host centric receive event queue read pointer has changed, then the host writes the updated pointer value into the NIC's device centric receive event queue read pointer. The host receive event handler 900 then goes inactive in step 924.

[0128] Fig. 10 is a brief flow chart of pertinent steps that are performed by the operating system kernel 325 on system startup. In a step 1010, the system calls a kernel driver initialization routine, and when that returns, in step 1020, the kernel blocks and awaits calls or other interrupts from the various hardware and software entities in the system. Many other steps of course are performed during kernel startup, but they are not shown in Fig. 10 since they are not pertinent to the present description.

[0129] Fig. 11 is a flow chart of pertinent steps performed by the kernel driver initialization routine 1010. In step 1110, the routine initializes the receive filter table in the NIC 216 such that each entry in the table contains a value which will be recognized by the NIC as being empty. This can be done by setting an empty flag bit for each entry, or preferably, by writing a unique known value into each entry. In the present embodiment the value indicating an empty entry is zero. The kernel driver 325 maintains its own host-centric copy of the receive filter table in host memory 222, and in step 1112, the kernel driver initializes the host receive filter table to all zeros as well.

[0130] In step 1114, the kernel driver programs initial search limits for four kinds of filter table hash searches into the NIC 216. Before explaining this step, it will be useful to understand the organization of the receive filter table. There are many ways to organize the receive filter table, but in the present embodiment, all types of entries are intermixed in a single table address space. Four types of entries are supported: TCP_full, TCP_wildcard, UDP_full and UDP_wildcard. Table 1 below illustrates the format for an entry in the receive filter table.

Entry Type	**bit[108]**	**bits[107:96]**	**bits[95:64]**	**bits[63:48]**	**bits[47:16]**	**bits[15:0]**
TCP_Full	0	Q_ID[11:0]	destIP[31:0]	destPORT[15:0]	srcIP[31:0]	srcPORT[15:0]
TCP_Wildcard	0	Q_ID[11:0]	destIP[31:0]	destPORT[15:0]	0	0
UDP_Full	1	Q_ID[11:0]	destIP[31:0]	destPORT[15:0]	srcIP[31:0]	srcPORT[15:0]
UDP_Wildcard	1	Q_ID[11:0]	destIP[31:0]	0	0	destPORT[15:0]

[0131] It can be seen that each entry contains up to five fields for identifying a particular TCP or UDP endpoint in the host subsystem 214 (protocol (TCP or UDP), source IP address, source port number, destination IP address, destination port number), plus one for the associated receive queue ID. The queue ID field points to the entry in the receive queue descriptor table 541 (Fig. 5), into which an incoming packet should be delivered when the endpoint data specified in the entry matches that of the header of the incoming packet.

[0132] The four fields, source IP address, source port number, destination IP address, and destination port number are referred to herein as the endpoint data portion of the entry. For a TCP_full entry, all four fields of endpoint data are filled. An incoming data packet will not match such an entry unless it is a TCP packet and all four endpoint data fields match corresponding fields in the packet header. For a TCP_wildcard entry, the destination IP address and destination port number are filled in, and the remaining fields contain zeros. An incoming data packet will match a TCP_wildcard entry if the incoming packet is a TCP packet and the two filled-in endpoint fields of the entry match corresponding fields in the packet header, regardless of the source IP address and port. For a UDP_full entry, all four fields of endpoint data are filled. An incoming data packet will not match such an entry unless it is a UDP packet and all four endpoint data fields match corresponding fields in the packet header. For a UDP_wildcard entry, like the TCP_wildcard entry, only the destination IP address and ports are filled and the remaining fields contain zeros. The UDP_wildcard entry format differs from that of the TCP_wildcard entry format since the destination port number is located in the field in which, for the TCP_full entry, contains the source port number. An incoming data packet will match a UDP_wildcard entry if the incoming packet is a UDP packet and the two filled-in endpoint data fields of the entry match corresponding fields in the packet header, again regardless of the source IP address and port.

[0133] It will be appreciated that another implementation might support different table entry formats, different protocols, and/or different entry types. As one example, entries may contain other kinds of numeric range indications rather than a wildcard for the entire field. As another example, particular bits of an address or port field may be wildcarded out (such as the low-order bits of an IPv6 address). As can be seen, as used herein, therefore, the "matching" of an entry to particular header information does not necessarily require complete equality. It is sufficient that it satisfies all conditions specified for a match.

[0134] All invalid entries in the table are written with all zeros. Obsolete entries are removed ("tombstoned"), at least in the first instance, by marking them with all zeros as well.

[0135] Although all four entry types co-exist in the same filter table, separate searches are performed through the table for each type. If the incoming data packet is a UDP packet, then a search is performed first for a matching UDP_full entry, and if that fails, then for a matching

UDP_wildcard entry. If the incoming data packet is a TCP packet, then a search is performed first for a matching TCP_full entry, and if that fails, then for a matching TCP_wildcard entry. All four kinds of searches are hashed searches, described in detail below. It will be seen that these searches will proceed through only a limited number of iterations before aborting. In one embodiment, the search limits for all four types of entries are the same, but in the present embodiment they can be different. In particular, the UDP_wildcard search limit is kept relatively small because UDP packets are relatively rare. It is not likely that the receive filter table will develop very long search chains of valid UDP entries, since not many software processes will need to program very many UDP filters. Similarly, the TCP_wildcard search limit is typically kept relatively small because again, few software processes are expected to require more than a few TCP_wildcard filter entries. The only one of these four types of entries that might require a high search limit is the TCP_full entry type, since a separate filter table entry might be required for example for each TCP connect() call made by an application or kernel process. In any event, if these assumptions are incorrect in a particular environment or at a particular point in time, the search limits can be adjusted dynamically as described hereinafter.

[0136] Accordingly, in step 1114, the kernel driver initialization routine programs into the NIC 216 the initial search limits for each of the four types of filter table entries. In step 1116, the driver routine returns to the kernel.

[0137] Fig. 12 is an example broad outline of steps that an application process might follow in the system of Fig. 2 for the reception of IP packets from the network 212. This flowchart covers only the TCP and UDP transport level protocols; other embodiments can support other protocols, including but not limited to SCTP, RTP, ICMP and IGMP.

[0138] In a step 1210, when the application first starts up, its libraries are initialized. This includes the user level transport library 324, which is initialized into the application's virtual address space.

[0139] Step 1212 begins an example sequence of steps in which the application process uses a UDP transport protocol. In step 1212 the application makes a call to the socket() routine of the user level transport library 324, specifying that it would like a UDP socket. In step 1214, the application process calls the bind() routine of the user level transport library 324 in order to bind the socket to a port. In step 1216, the application process makes a call to the recvfrom() routine of the user level transport library 324, specifying the socket handle, the receive buffer, the buffer length, and the source IP address and port from which an incoming packet is expected. As described below, the recvfrom() routine of the User Level Transport Library, among other things, writes an appropriate UDP_full-type tuple into the receive filter table on the NIC 216 and in the host 214. If the application wishes to receive packets from any IP address and Port, these latter two fields may contain zeros. In this case the recvfrom() routine of the User Level Transport Library will write an appropriate UDP_wildcard-type tuple into the receive filter table on the NIC 216 and in the host 214.

[0140] After the recvfrom() call, the application blocks until a packet satisfying the specified criteria comes up from the NIC 216 or from the kernel driver 325. At that time, in step 1218, the application processes the received data and returns, in this example, to step 1216 to await the next incoming data packet.

[0141] Steps 1216 and 1218 are repeated many times, most likely interspersed with many other functions performed by the application process. When the application has finished with the socket that it had created in step 1212, then in step 1220, the application makes a call to the close() routine of the user level transport library 324 in order to close the socket.

[0142] Alternatively to the UDP sequence beginning with step 1212, step 1222 begins an example sequence of steps in which the application process uses a TCP transport protocol. In step 1222, instead of calling the socket routine of the user level transport library 324 to specify the UDP protocol, it calls the socket routine to specify the TCP protocol. In step 1224 the application process calls the bind() routine similarly to step 1214, in order to bind the socket to a port. In step 1226, since the transport protocol is now TCP, the application process calls the listen() routine of the user level transport library 324, in order to listen for an incoming TCP connection request. Alternatively in step 1226 the application process may call the accept() routine of the user level transport library 324. Both the listen() and accept() routines invoke the kernel to write into the receive filter table on the NIC 216 and in the host 214 a new IP tuple of type TCP_wildcard, so that any incoming TCP connection request (SYN) having the appropriate destination IP address and port number in its header, will be sent up to the present application process for handling.

[0143] In step 1230 the application process makes a call to the recv() function of the user level transport library 324, specifying a socket handle, the buffer into which data should be delivered, and the buffer length. At this point the application blocks until a packet satisfying the specified criteria comes up from the NIC 216 or from the kernel driver 325. At that time, in step 1230, new incoming data arrives in the buffer and the application processes it. The application then may return to the recv() step 1228 to await the next incoming data packet. Again, many other steps may take place in between those illustrated in the flow chart. In step 1232, the application terminates, or the connection may be terminated, at which time the application makes a call to the close() routine of the user level transport library 324 in order to close the socket.

[0144] Note that for TCP connections initiated by the present application process, instead of calling the listen() routine in step 1226, typically the application will make a call to the connect() routine of the User Level Transport Library, specifying the socket handle and the destination IP address and port number. The connect() routine of the User Level Transport Library will, among other things, invoke the kernel driver 325 to write into the receive filter table on the NIC 216 and the host 214 a new IP tuple of type TCP_full, so that only those incoming TCP packets having four matching endpoint data fields will be sent up into the present application process.

[0145] Fig. 13 is a flowchart detail of steps that might be performed during the library initialization step 1210 of Fig. 12. In step 1310, the user level transport library 324 is loaded into the application's virtual address space. In a conventional system, the transport library loaded in step 1310 might be a very thin layer (or incorporated within another general purpose library such as glibc), which does little more than make calls to the kernel driver and return any results to the application. In the system of Fig. 2, however, the transport library loaded in step 1310 is more sophisticated in that it includes the user level routines described herein.

[0146] In step 1312, as part of the initialization of the user level transport library, a resource allocation routine in the kernel driver 325 is invoked. The kernel level routine is required for allocating resources in the network interface card and the host memory subsystem 222, since these resources are outside the virtual address space of the application, or involve direct hardware accesses that advisedly are restricted to kernel processes. After resource allocation, the user level driver initialization routine 1210 may perform a number of other steps before it returns to the application in step 1314.

[0147] Fig. 14 is a flowchart illustrating significant steps performed by the kernel resource allocation routine 1312. In step 1410, the kernel begins by allocating memory for the generalized buffers that will be used to hold the transmit, receive and event queues. It then maps the buffers into the application's virtual address space so that the application can read and write to them directly. (See previous description with respect to Fig. 3A). In step 1412, the kernel routine installs descriptors for these buffers in the buffer descriptor table 510.

[0148] In step 1414, the kernel routine allocates a minimum set of the buffers for each of the transmit, receive and event queues requested, and programs their buffer IDs into the transmit, receive and event queue descriptor tables 540, 541 and 542. In step 1416, the kernel routine determines the "doorbell" address in the NIC 216 for each of the transmit and receive queues, and maps them as well into the application's virtual address space. The doorbell address is the address to which the user level transport library will write a value, in order to notify the NIC either that a transmit buffer is ready or that a receive buffer can be released. For transmit queues, the doorbell address is the address of the device centric transmit queue read pointer in the transmit queue descriptor table 540 entry for the particular transmit queue. For receive queues, the doorbell address is the address of the device centric receive queue write pointer in the receive queue descriptor table 541 entry for the particular receive queue.

[0149] In step 1420, the resource allocation routine returns to the application with handles for the resources allocated, with the base virtual addresses of the transmit, receive and event queues, and virtual memory addresses corresponding to the doorbells allocated in the transmit and receive queue descriptor tables 541 and 540.

[0150] Fig. 15 is a flowchart of the user level routine in the transport library 324 for establishing a socket. The socket routine is called in both steps 1212 and 1222 of Fig. 12. In step 1510, the routine first determines whether the user level transport library 324 contains the software code necessary to support the specified protocol. If not, then in step 1512, makes a system call to pass the request on to the kernel to handle. If it does support the specified protocol, then in step 1514 it allocates its internal state for the new socket. In step 1516 the routine returns to the application program with a handle for the new socket.

[0151] Fig. 16 is a flowchart of the user level routine transport library 324 for binding a port number to a socket. The bind routine is called in both steps 1214 and 1224 of Fig. 12. In step 1610, the routine first determines whether the caller provided a port number. If not, then in step 1612, a port number is assigned. If the caller did specify a port number, then in step 1614, the routine determines whether the port number is legal for the current user process. At least four different mechanisms might be used in the same or different embodiments to ensure that different processes do not interfere with each other's use of particular IP address/port number combinations, and that user processes do not improperly operate through physical or logical port numbers or other resources that should be reserved for the kernel. In one mechanism, a system-wide policy exists which allocates all port numbers within a particular range to the user stack only. The user level bind() routine can be designed to immediately accept only those requests from a user level process to bind to one of such port numbers, or to immediately pass such requests on to the kernel to handle.

[0152] In a second mechanism, during the resource allocation step 1312, performed during initialization of a particular instance of the user level driver, the kernel allocates a unique IP address for that instance to use as it wishes. If each instance of the user level driver has its own exclusively assigned IP address, then the instance can manage the available port numbers for use with that IP address without risking interference with any other process. This mechanism is useful only if there are sufficient numbers of IP addresses available to the computer system to allocate to the various requesting processes. Again, this mechanism can be used in conjunction with the first, to reject or pass on to the kernel all user level requests to bind to a kernel-only port number, regardless of the exclusivity of an assigned IP address.

[0153] In a third mechanism, again during initialization of a particular instance of the user level driver, the initialization routine makes a number of anticipatory bind() calls to the kernel in order to form a pool of port numbers that the user level driver instance can later allocate to the application program upon receipt of bind() calls to the user level driver. This mechanism can succeed with far fewer IP addresses available to the computer system, but also undesirably involves a context switch (during library initialization) for each port number to be added to the pool.

[0154] In yet a fourth mechanism, no IP address/port number combinations are pre-allocated to the particular instance of the user level driver. Instead, the user level bind() routine invokes the kernel bind() routine for each user level bind() call received. This mechanism utilizes IP address/port number combinations most conservatively, but may require more context switches than any of the first, second and third mechanisms. In an embodiment, this fourth mechanism is used only as a backup, for example if the user level process requires more port numbers than were made available using the anticipatory bind() calls in the third mechanism.

[0155] If in step 1614 the user level bind() routine determines that the requested port number is not available to the current instance of the user level driver, or otherwise cannot determine whether is available, then in step 1616, the routine makes a call to the kernel bind()

routine to pass the request on to the kernel to handle. If the fourth mechanism above is the only way that the particular embodiment avoids conflicting or illegal allocation of address/port number combinations, then step 1616 will be taken during every user level call to the bind() routine 1600. Otherwise, step 1616 will be taken only as a backup if pre-allocated port numbers have been exhausted, or if the routine otherwise cannot determine that the requested port number is available.

[0156] If the specified port number is legal, or if a port number was assigned by the routine in step 1612, then in step 1618 the routine updates the application's state internally, to bind the port number with the specified socket. The routine returns to the caller in step 1620.

[0157] Fig. 17 is a flowchart of pertinent steps of the user level recvFrom() routine called in step 1216 (Fig. 12). This routine is called with the socket handle, the address of a buffer that the application wishes to use, the length of the buffer, and optionally the source IP address and port from which the UDP packet is expected. The source IP address and port may be left as zeros if the application does not know them. The application does not necessarily know the destination IP address and port to which the packet will be directed, but in step 1710, the recvFrom() routine obtains this information from its own records of the IP address and port associated with the specified socket. In step 1712, the recvFrom() routine pushes one or more RX descriptors onto the RX queue if needed. It then updates the RX queue write pointer and writes the updated write pointer to the NIC doorbell address previously assigned. (See step 611, Fig. 6). In step 1714 the routine invokes the kernel routine to create a new filter entry, specifying the UDP protocol, the source and destination IP addresses and port numbers, and the handle previously assigned for the desired destination receive queue. In step 1716, the routine blocks, and awakens on the receipt of a receive completion event as described with respect to step 910 (Fig. 9). In step 1718 (corresponding to step 948 in Fig. 9) the routine then performs TCP/IP receive processing on received packet, and copies the payload to the buffer that the application had specified in the arguments in step 1216. In another embodiment the payload copy can be avoided, for example by delivering headers and payloads separately, and delivering data in-order with respect to the sequence numbers in the headers. In step 1720 the routine frees the receive buffer corresponding to the RX descriptor just used, and in step 1722 the routine exits and returns to the application.

[0158] Note that the user level recv() routine 1230 (Fig. 12) is in pertinent part the same as the recvFrom() routine of Fig. 17, except that since the filter has already been applied, steps 1710 and 1714 are omitted.

[0159] Fig. 18 is a flowchart of pertinent steps of the user level listen() routine called in step 1226 (Fig. 12). This routine is called with only the socket handle, since the source IP address and port from which a SYN packet might arrive are not yet known. In step 1810, the listen() routine obtains the destination IP address and port from the user level driver's records associated with the specified socket handle. In step 1812, the routine pushes one or more RX descriptors onto the RX queue if needed, anticipating the receipt of a TCP connection request packet (SYN packet). It then updates the RX queue write pointer and writes the updated write pointer to the NIC doorbell address previously assigned. In step 1814, the routine invokes the kernel routine to create a new filter entry, specifying the TCP protocol, the destination IP address and port number, and the handle previously assigned for the desired destination receive queue. The routine sets the ip_src and src_port arguments to 0. In step 1816, the routine blocks, and awakens on the receipt of a receive completion event. When one is received, in step 1818 the routine frees the receive buffer corresponding to the RX descriptor just used, and in step 1820 it is determined whether the received packet is in fact a SYN. If not, then the packet is dropped and in step 1822 another RX descriptor is pushed onto the RX queue if needed (similarly to step 1812). The routine then loops back again to block in anticipation of the next RX completion event (step 1816). If the received packet was in fact a SYN, and the newly created stream cannot be processed in the context of the accepting application address space, then in step 1824 the routine programs a new TCP_full entry with the IP source address and port from the SYN packet. In step 1826 the routine exits and returns to the application with the source IP address and port from the SYN packet, the new TCP connection having been established.

[0160] The user level accept() routine is in pertinent part the same as the listen() routine of Fig. 18, except that the routine will more likely program the new TCP_full entry into the filter table because the application is more likely to follow up with a fork() into a new address space.

[0161] Note that the User Level Transport Library routines that invoke the kernel to set up a new filter, also maintain a local copy of the filters that they already had set up. In this way they can avoid the context switch to the kernel to duplicate work that has already been performed.

[0162] Fig. 19 is a flowchart of pertinent steps of the kernel routinely for creating a new filter table entry. This routine is invoked at the request of the user level transport library during a listen(), accept(), recvfrom() or connect() call, or it can also be invoked by the kernel itself. When called, the routine of Fig. 19 is provided with the protocol name or number, the source and destination IP address and ports for matching with incoming data packets, and the handle for the receive queue as it was provided to by the kernel during a resource allocation step 1312. In step 1910, the routine looks up the queue ID from the receive queue handle. The queue ID is the index into the receive queue descriptor table 541 (Fig. 5). In step 1912, the routine determines whether the protocol for which the filter entry is to be placed, is either a TCP or a UDP protocol.

If it is neither, then in step 1914, the routine returns with an error because the filter table in the present embodiment does not support any other protocols. Incoming data packets using a different protocol will automatically default to a kernel queue for handling or delivery to the proper user level queue.

[0163] If the protocol is TCP or UDP, then in step 1916, the routine performs a hashed search in the host receive filter table for an empty slot. For UDP filters, a search of the UDP_wildcard entries is performed if the ip_src and src_port arguments are null. If either the ip_src or src_port argument contains a value, then a search of the UDP_full entries is performed. Alternatively, the API can include a separate parameter to indicate whether a full or wildcard search should be performed. For TCP filters, a search of the TCP_wildcard entries is performed if the ip_src and src_port arguments are null. If either the ip_src or src_port argument contains a value, then a search of the TCP_full entries is performed. In each case, the search is bounded by the appropriate search limit for the TCP_full, TCP_wildcard, UDP_full or UDP_wildcard protocol as programmed in step 1114 (Fig. 11). The search algorithm itself is described in detail with respect to Figs 22-26, and is the same as that used by the NIC against the NIC copy of the filter table upon receipt of a packet from the network.

[0164] In step 1918, if an empty slot was found before the appropriate search limit was reached, then the routine programs both the NIC and the host receive filter tables at the identified entry with the queue ID and the endpoint data as provided in step 1900. The kernel routine then returns to the caller in the user level transport library.

[0165] If the search limit was reached before an empty slot was found, then the routine makes a decision whether to increase the search limit for the particular kind of entry (step 1920). If not, then in step 1922 the routine simply sets up a software redirect for this set of endpoint data. The redirect is a data structure that the kernel driver consults on receipt of every packet to see whether the packet should be delivered to the kernel stack or a user transport managed endpoint. In one embodiment it is a separate table of filter entries, structured similarly to the host receive filter table.

[0166] If the routine does decide to increase the search limit, then in step 1924, the routine simply continues the hashed search of step 1916 until an empty slot is found. When one is found, then in step 1926, the routine programs the NIC and host receive filter tables at the identified entry with the queue ID and endpoint data from step 1900. Then in step 1928, the routine programs the new search limit for the particular type of entry, into the NIC and then in the kernel. It will be appreciated that steps 1920 and 1924 may be reversed in sequence or combined, so that the number of additional hops required to find an available entry can be taken into account in the decision of whether to increase the search limit.

[0167] In the close() function of the user level driver, called both at steps 1220 and 1232 of Fig. 12, one of the functions that must be performed is the removal of the socket's entries from the IP filter table. The routine in the user level driver accomplishes this by making a request to the RemoveFilterEntry() routine 2000 of the kernel, illustrated in Fig. 20. In step 2010, the routine looks up the filter table entry from the data associated with the socket handle specified. In step 2012, the routine determines whether the entry is located in the software redirect table or in the host filter table. If it is in the software redirect table, then in step 2014, the routine removes the entry from the software redirect table and returns (step 2026). If the entry is in the host filter table, when the entry is identified in step 2014, then in one embodiment, the routine can simply remove the identified entry in both the NIC receive filter table and the host receive filter table (in that order) (step 2018). As mentioned, removal involves simply writing a tombstone value into the entry, for example all zeros. However, it would be useful at this time to also perform certain garbage collection functions in order to keep the search chain lengths reasonable. These garbage collection functions can be performed at other times in the same or different embodiments, but the present embodiment conveniently performs them at least whenever a filter entry is to be removed. Accordingly, before the step of tombstoning the identified entry (step 2018), the routine first determines in step 2020 whether removal of the identified entry, and all tombstones previously written in the present chain, would reduce the overall maximum chain length. If not, then only the single entry removal is effected (step 2018). Garbage collection is typically an expensive process, so step 2020 limits the number of times that garbage collection is performed to only those times when a significant performance improvement can be achieved, such as through a reduction in the overall maximum chain length. Hysteresis can be applied here to avoid flapping, and a weight can be applied so that the chain length is reduced more aggressively if it has become longer than the maximum allowed for full line rate performance.

[0168] If the maximum chain length would be reduced, then in step 2022, the routine removes the identified entry from the table, as well as all tombstones in the table (or only those in the chain, if the chain is very long compared to others in the table), reprogramming the table to bring up into the previously tombstone the locations, entries and that were previously father down in the search chain. Finally, in step 2024, the routine programs the new (smaller) search limit for the particular kind of entry into the NIC 216 and then makes it effective in the kernel as well. In an embodiment, the routine may skip step 2024 in the event that the new search limit would be smaller than some predetermined minimum chain length for the particular kind of entry. Many other optimizations will be apparent.

[0169] As mentioned, when packets arrive, the filter table and logic unit 550 (Fig. 5) first determines the queue ID of the receive queue into which the packet should be delivered. Fig. 21 is a detail of the filter table and logic unit 550. The filter table itself is implemented as two sub-tables 2110-0 and 2110-1 (collectively 2110). The software is unaware of this implementation detail, and instead sees only a single table. The hardware in the NIC 216 decodes the write address from the kernel driver software and places even numbered entries in filter sub-table 2110-0 and odd numbered entries in filter sub-table 2110-1. Thus filter sub-table 2110-0 contains entries 0, 2, 4, 6 and so on, whereas filter sub-table 2110-1 contains entries 1, 3, 5 and so on. The implementation of the filter table as two sub-tables enables parallel entry lookup per cycle to reduce the total lookup time. It will be appreciated that other implementations can use a single sub-table, or more than two sub-tables.

[0170] Both of the filter sub-tables 2110-0 and 2110-1 are addressed by a 13-bit index provided by filter table search logic 2112. A13-bit index can address up to 8192 entries, which for two sub-tables, comes to 16,384 entries numbered 0 through 16,383. Four index values are reserved for the kernel NET driver queues, so only 16,380 entries are represented in the table. The filter table search logic 2112 is described hereinafter, but basically, it receives the header data of the incoming data packet and uses it to derive a hash key, then uses the hash key to derive a hash code which is the first 13-bit index to be searched in the filter table 2110. The filter table search logic also calculates subsequent entries to search if the first entry in neither of the two filter sub-tables matches that of the incoming header data, and also forces the selection of a default kernel queue for delivery of the current data packet if the search limit is reached before a match is found. The filter table search logic also determines a match type (TCP_full, TCP_wildcard, UDP_full or UDP_wildcard) in dependence upon the header data and the state of the search algorithm.

[0171] The various formats for an entry in the filter table 2110 are set forth in the table above. As shown in Fig. 21, the endpoint data from the selected entry of filter sub-table 2110-0 is provided to one input of match logic 2114-0, and the endpoint data from the selected entry of filter sub-table 2110-1 is provided to the corresponding input of match logic 2114-1. The other input of each of the match logic units 2114-0 and 2114-1 (collectively 2114) receives the header data of the incoming data packet. The match type is provided from the filter table search logic 2112 to both match logic units, each of which then outputs a "match" signal to a hit logic unit 2116. If the match type is TCP_full, then match logic units 2114-0 at 2114-1 will indicate a match only if the incoming packet type is TCP and all four fields of endpoint data match the corresponding fields of the incoming header data. If the match type is TCP_wildcard, then the match logic units 2114 will indicate a match if the incoming packet type is TCP and bits 95:48 of the endpoint data in the table contains the same destination IP address and destination port as the incoming header data. The source IP address and source port as indicated in the header data are ignored. If the match type is UDP_full, then match logic units 2114-0 at 2114-1 will indicate a match only if the incoming packet type is UDP and all four fields of endpoint data match the corresponding fields of the incoming header data. If the match type is UDP_wildcard, then match logic units 2114 will indicate a match if the incoming packet type is UDP and bits 95:64 of the filter endpoint data contain the same destination IP address, and bits 15:0 of the endpoint data contain the same destination port number, as indicated in the header data of the incoming packet.

[0172] If either match logic unit 2114-0 or 2114-1 indicates a match, then hit logic 2116 so notifies the filter table search logic 2112. The Q_ID fields of the currently selected entries in both filter sub-tables are provided to two of three inputs of a multiplexer 2118, and hit logic 2116 provides a select input so that the multiplexer 2118 will select the queue ID from the currently selected entry of the correct filter sub-table 2110-0 or 2110-1. As mentioned, if no matching entry has been found after the search limit has been reached, then the filter table search logic 2112 provides a signal to the multiplexer 2118 to select to the default queue ID provided on a third input of the multiplexer 2118. The default queue ID in one embodiment is queue 0, which is defined to be a kernel queue. In other embodiments the default queue ID can be programmable. In any event, whether or not a match has been found, the multiplexer 2118 outputs the queue ID indicating the particular receive queue to which the NIC 216 should deliver the incoming data packet.

[0173] Note that in a multiple CPU system, there can be multiple kernel queues. In such a system it is advantageous to distribute failed-search traffic as evenly as possible among the different kernel queues so as not to overload any particular one of them. It is also desirable that all failed-search traffic involving a particular connection always be delivered to a single one of the kernel queues. In order to accomplish both of these goals, the low order bit or bits of the initial hash key code already calculated is used to select the default kernel queue for delivery of failed-search packets. In particular, the kernel driver 325 first programs a register in the NIC 216 (not shown) to indicate the number of kernel queues in use. If a search fails, the NIC 216 then uses the low order 1 or 2 bits (depending on whether there are 2 or 4 processors) of the initial hash code in order to select the particular CPU whose kernel queue will receive the incoming data packet. Other methods may be used instead to select the kernel queue to which a failed-search packet will be delivered. As one alternative, a different hash function can be used, different from the one described herein for searching the filter table. For example, the Toeplitz hash function described in Microsoft, Scalable Networking with RSS, Win HEC 2005 Update (April 19, 2005), incorporated by reference herein, can be used.

[0174] If a data packet has arrived without an IP header, or containing a mal-formed IP header, (such as with an IP header checksum error), then the NIC 216 delivers this packet to an overall default kernel queue, which is queue 0.

[0175] Fig. 22 is a flow chart illustrating the overall functions performed by filter table search logic 2112 in order to conduct a hashed search for a match. These functions can be implemented in software or firmware for and embedded processor, or can be controlled by a finite state machine, or can be implemented in another way all as known in the art or yet to be developed. In step 2210, the logic first determines from the protocol ID field in the header of the incoming data packet, which transport protocol is being used. Only TCP and UDP packets are filtered using the filter table, so if the transport protocol ID field indicates a different transport protocol, then in step 2212, the search logic immediately terminates the search and causes the multiplexer 2118 to select the default queue ID. It will then be up to the kernel to handle the particular incoming packet, including determining whether it needs to be forwarded to one of the user level transport libraries 324.

[0176] If the protocol of the incoming data packet is TCP, then in step 2218, the filter table search logic 2112 performs a hashed TCP_full search as described in more detail hereinafter. If this search finds a match before the TCP_full search limit is reached, then in step 2216, the filter table search logic 2112 cause the multiplexer 2118 to select the queue ID from the matching filter sub-table 2110-0 or 2110-1. If the TCP_full search limit is reached before a match is found in step 2218, then in step 2220, a TCP_wildcard search is performed. By performing a TCP_full search before a TCP_wildcard search, the more specific filter is given precedence over the more general filter. If the TCP_wildcard search finds a match before the TCP_wildcard search limit is reached, then in step 2216, the filter table search logic 2112 causes the multiplexer 2118 to select the queue ID from the matching filter sub-table 2110-0 or 2110-1. Otherwise, in step 2212 the filter table search logic 2112 causes the multiplexer 2118 to select the default queue ID.

[0177] If the protocol of the incoming data packet is UDP, then in step 2213, the filter table search logic 2112 performs a hashed UDP_full search as described in more detail hereinafter. If this search finds a match before the UDP_full search limit is reached, then in step 2216, the filter table search logic 2112 cause the multiplexer 2118 to select the queue ID from the matching filter sub-table 2110-0 or 2110-1. If the UDP_full search limit is reached before a match is found in step 2213, then in step 2214, a UDP_wildcard search is performed. If the UDP_wildcard search finds a match before the UDP_wildcard search limit is reached, then in step 2216, the filter table search logic 2112 causes the multiplexer 2118 to select the queue ID from the matching filter sub-table 2110-0 or 2110-1. Otherwise, in step 2212 the filter table search logic 2112 causes the multiplexer 2118 to select the default queue ID.

[0178] As with all flow charts herein, it will be appreciated that many of the steps in the flow chart of Fig. 22 can be combined, performed in parallel or performed in a different sequence without affecting the functions achieved.

[0179] Fig. 23 is a flow chart detail of the step 2218 of performing a TCP_full search (Fig. 22). Initially, in a step 2310, an iteration count k is set to 0. The initial hash code is then calculated by first determining a hash key, and then passing the key through a first hash function h1(key). In particular, in step 2312, the hash key is determined by a fold_by_3 function performed on the destination IP address, the destination port number, the source IP address and the source port number, respectively, of the incoming data packet, as follows:

key = destIP [31:0] XOR{destPORT[15:0], srcIP[31:16]} XOR {srcIP[15:0], srcPORT[15:0]}

[0180] In step 2314, the key is passed through the first hash function h1(key) to determine the initial hash code which is then to be used as an index into the filter table 2110 as previously described. The first hash function is implemented through a linear feedback shifting register (LFSR) sequence with characteristic polynomial X^16+X^3+1 on the 32 bit key. This value is taken modulo 2^13 (by taking only the low order 13 bits of the LFSR output) to form the hash code. This function randomizes the key sufficiently so that even packets sharing the same IP address or port number are not likely to be mapped into the same initial code. The modulus 2^13 is the number of entries that can be held in one of the filter sub-tables 2110, so the truncation to only the low order 13 bits effectively keeps the hash code within the total index range of the table. In an implementation, the hash function can be performed in a single cycle by flattening the serial shift register into parallel XOR functions of the 32 bit key. Other first hash functions can be used instead of this one, but this one is preferred.

[0181] Now that the initial index has been calculated in step 2314, in step 2316 the match logic units 2114 and hit logic unit 2116 indicate to the filter table search logic 2112 whether the header data matches either the endpoint data in the currently selected entry of filter sub-table 2110-0 or that of the currently selected entry of filter sub-table 2110-1. The filter table search logic 2112 indicates TCP_full as the match type provided to match logic units 2114 and 2114. If the current entry of one of the two filter sub-tables matches, then a match has been found (step 2318) and the filter table search logic 2112 cause the multiplexer 2118 to select the appropriate queue ID (step 2216, Fig. 22). If not, then in step 2320, the filter table search logic increments the iteration count k and determines whether the TCP_full search limit has been reached. If so, then the routine exits (step 2322) such that a TCP_wildcard search can be performed (step 2220 in Fig. 22). If the search limit has not been reached, then in step 2324, the index is incremented by a second hash function h2 of the key and truncated again to 13 bits. The second hash function need not be as expensive as the first, but it is advantageously a function of the key in order to provide sufficient randomness to minimize overlapping hop sequences for different incoming header data. In an embodiment, the second hash function is simply

$$h2(key) = 13\ LSB\ of\ (key*2-1).$$

[0182] Combining the two has functions, it can be seen that the filter table index for the k'th search is given by

$$index(k) = (h1(key) + k*h2(key))\ mod\ 2^13.$$

If the filter table 2110 had not been divided into sub-tables, then the modulus would be 2^14 instead of 2^13. In general, if the overall filter table can hold N entries, and they are divided into P parallel tables (where P is an integer greater than or equal to 1), then the modulus would be M=N/P. Note that while the embodiment described herein maintains a current value for k, it uses this value only for determining when the search limit is reached. The different index values are maintained and calculated separately, without using the current value of k. Another embodiment could instead use the current value of k to calculate each index value, for example by using the combined mathematical function above for index(k).

[0183] Once the next index has been calculated in step 2324, the TCP_full search logic returns to step 2316 to determine whether the header data matches the endpoint data in either filter table entry now pointed to by the new index. Eventually either a match will be found (step

2318) or the TCP_full search limit will be reached (2322) and the TCP_full search step 2218 will terminate.

[0184] Fig. 24 is a flow chart detail of the step 2220 of performing a TCP_wildcard search (Fig. 22). It is very similar to the TCP_full search of Fig. 23, except that the fold-by-three function operates on different parts of the header, the match type is TCP_wildcard instead of TCP_full, and the TCP_wildcard search limit is different and the TCP_full search limit. In particular, in a step 2410, an iteration count k is initialized to 0. In step 2412, the hash key is determined by the following fold_by_3 function performed on the destination IP address and port number:

key = destIP[31:0] XOR {destPORT[15:0], 16'h0000} XOR 32'h0

[0185] In step 2414, the key is passed through the first hash function h1(key) to determine the initial hash code which is then to be used as an index into the filter table 2110 as previously described. The same first hash function is used as is set forth above for the TCP_full search, although in another embodiment a different first hash function might be used.

[0186] The initial index is applied to the filter sub-tables 2110, and in step 2416 the match logic units 2114 and hit logic unit 2116 indicate to the filter table search logic 2112 whether the header data matches either the endpoint data in the currently selected entry of filter sub-table 2110-0 or that of the currently selected entry of filter sub-table 2110-1. The filter table search logic 2112 indicates TCP_wildcard as the match type provided to match logic units 2114 and 2114. If the current entry of one of the two filter sub-tables matches, then a match has been found (step 2418) and the filter table search logic 2112 cause the multiplexer 2118 to select the appropriate queue ID (step 2216, Fig. 22). If not, then in step 2420, the filter table search logic increments the iteration count k and determines whether the TCP_wildcard search limit has been reached. If so, then the routine exits (step 2422) such that the default queue ID can be selected (step 2212 of Fig. 22). If the search limit has not been reached, then in step 2424, the index is incremented by the second hash function h2 of the key and truncated again to 13 bits. The second hash function is again the same as for the TCP_full search, but it need not be in a different embodiment. Once the next index has been calculated, the TCP_wildcard search logic returns to step 2416 to determine whether the header data matches the endpoint data in either filter table entry now pointed to by the new index. Eventually either a match will be found (step 2418) or the TCP_wildcard search limit will be reached (2422) and the TCP_wildcard search step 2220 will terminate.

[0187] Fig. 25 is a flow chart detail of the step 2213 of performing the UDP_full search (Fig. 22). Again it is very similar to the TCP_full search of Fig. 23, except that the fold-by-three function operates on different bits of the header, the match type is UDP_full, and the UDP_full search limit is different. In particular, in a step 2510, the iteration count k is initialized to 0. In step 2512, the hash key is determined by the following fold_by_3 function performed on the destination IP address and port number:

key = {destIP[31:1], ~destIP[0]} XOR {destPORT[15:0], srcIP[31:16]} XOR {srcIP[15:0], srcPORT[15:0]}

[0188] In step 2514, the key is passed through the first hash function h1(key) to determine the initial hash code which is then to be used as an index into the filter table 2110 as previously described. The same first hash function is used as is set forth above for the TCP_full and TCP_wildcard searches, although again, in another embodiment, a different first hash function might be used for UDP_full searches.

[0189] The initial index is applied to the filter sub-tables 2110, and in step 2516 the match logic units 2114 and hit logic unit 2116 indicate to the filter table search logic 2112 whether the header data matches either the endpoint data in the currently selected entry of filter sub-table 2110-0 or that of the currently selected entry of filter sub-table 2110-1. The filter table search logic 2112 indicates UDP_full as the match type provided to match logic units 2114 and 2114. If the current entry of one of the two filter sub-tables matches, then a match has been found (step 2518) and the filter table search logic 2112 cause the multiplexer 2118 to select the appropriate queue ID (step 2216, Fig. 22). If not, then in step 2520, the filter table search logic increments the iteration count k and determines whether the UDP_full search limit has been reached. If so, then the routine exits (step 2522) such that the default queue ID can be selected (step 2212 of Fig. 22). If the search limit has not been reached, then in step 2524, the index is incremented by the second hash function h2 of the key and truncated again to 13 bits. The second hash function is again the same as for the TCP_full and TCP_wildcard searches, but it need not be in a different embodiment. Once the next index has been calculated, the UDP_full search logic returns to step 2516 to determine whether the header data matches the endpoint data in either filter table entry now pointed to by the new index. Eventually either a match will be found (step 2518) or the UDP_full search limit will be reached (2522) and the UDP_full search step 2214 will terminate.

[0190] Fig. 26 is a flow chart detail of the step 2214 of performing the UDP_wildcard search (Fig. 22). Again it is very similar to the TCP_full search of Fig. 23 and the TCP_wildcard search of Fig. 24, except that the fold-by-three function operates on different parts of the header, the match type is UDP_wildcard, and the UDP_wildcard search limit is different. In particular, in a step 2610, the iteration count k is initialized to 0. In step 2612, the hash key is determined by the following fold_by_3 function performed on the destination IP address and port number:

key = destIP[31:0] XOR 32'h0 XOR {16'h0, destPORT[15:0]}

It can be seen that in a compact hardware implementation of the search algorithm, all three fold-by-three functions can be performed with common XOR hardware by preceding it with a multiplexer to select to the XOR logic the particular fields of the incoming packet header data that are required for the particular search type.

[0191] In step 2614, the key is passed through the first hash function h1(key) to determine the initial hash code which is then to be used as an index into the filter table 2110 as previously described. The same first hash function is used as is set forth above for the TCP_full and TCP_wildcard searches, although again, in another embodiment, a different first hash function might be used for UDP_wildcard searches.

[0192] The initial index is applied to the filter sub-tables 2110, and in step 2616 the match logic units 2114 and hit logic unit 2116 indicate to the filter table search logic 2112 whether the header data matches either the endpoint data in the currently selected entry of filter sub-table 2110-0 or that of the currently selected entry of filter sub-table 2110-1. The filter table search logic 2112 indicates UDP_wildcard as the match type provided to match logic units 2114 and 2114. If the current entry of one of the two filter sub-tables matches, then a match has been found (step 2618) and the filter table search logic 2112 cause the multiplexer 2118 to select the appropriate queue ID (step 2216, Fig. 22). If not, then in step 2620, the filter table search logic increments the iteration count k and determines whether the UDP_wildcard search limit has been reached. If so, then the routine exits (step 2622) such that the default queue ID can be selected (step 2212 of Fig. 22). If the search limit has not been reached, then in step 2624, the index is incremented by the second hash function h2 of the key and truncated again to 13 bits. The second hash function is again the same as for the TCP_full and TCP_wildcard searches, but it need not be in a different embodiment. Once the next index has been calculated, the

UDP_wildcard search logic returns to step 2616 to determine whether the header data matches the endpoint data in either filter table entry now pointed to by the new index. Eventually either a match will be found (step 2618) or the UDP_wildcard search limit will be reached (2622) and the UDP_wildcard search step 2214 will terminate.

[0193] As previously mentioned, if the incoming data packet is mal-formed or uses a protocol that is not supported in the filter table 2110, or if it uses the supported protocol but a match was not found in the filter table 2110 before the appropriate search limit(s) was (were) reached, then the NIC 216 will deliver the incoming data packet to a receive queue of the kernel driver 325. Fig. 27 is a flow chart showing pertinent steps that the kernel driver performs upon receipt of such a data packet. Initially, in step 2710, the kernel routine determines whether the incoming data packet uses the TCP or UDP protocol. If not, then in step 2712, the kernel driver processes the packet in whatever manner is appropriate for the particular packet format. If the incoming data packet does use TCP or UDP, then in step 2714, the kernel driver 325 performs a hashed search, with no search limit imposed, of the software redirect table. In step 2716, if no match was found, then the kernel driver 325 simply delivers the packet to a normal kernel network stack (step 2718). If a match was found then in step 2720 the kernel driver 325 delivers the packet to the proper user level receive process. In order to avoid contention with the NIC 216 attempting to deliver its own data packets to receive queues in an application's transport library 324, the delivery of the packet from the kernel driver 325 to the user level transport library 324 occurs by some communication channel other than through the use of the receive queue. Typical standard operating system mechanisms can be used to notify the user level driver of the availability of this packet.

[0194] As used herein, the "identification" of an item of information does not necessarily require the direct specification of that item of information. Information can be "identified" in a field by simply *referring* to the actual information through one or more layers of indirection, or by identifying one or more items of different information which are together sufficient to determine the actual item of information. In addition, the term "indicate" is used herein to mean the same as "identify".

[0195] Additionally, as used herein, a given signal, event or value is "responsive" to a predecessor signal, event or value if the predecessor signal, event or value influenced the given signal, event or value. If there is an intervening processing element, step or time period, the given signal, event or value can still be "responsive" to the predecessor signal, event or value. If the intervening processing element or step combines more than one signal, event or value, the signal output of the processing element or step is considered "responsive" to *each* of the signal, event or value inputs. If the given signal, event or value is the same as the predecessor signal, event or value, this is merely a degenerate case in which the given signal, event or value is still considered to be "responsive" to the predecessor signal, event or value. "Dependency" of a given signal, event or value upon another signal, event or value is defined similarly.

[0196] The foregoing description of preferred embodiments of the present invention has been provided for the purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise forms disclosed. Obviously, many modifications and variations will be apparent to practitioners skilled in this art. As an example, whereas in the embodiments described herein it is the header fields of an incoming packet which are compared to fields in the filter table to detect a matching filter table entry, in another embodiment other aspects of the content of the incoming packet can be compared instead. As another example, whereas the filter table in the NIC in the embodiments described herein have a tabular format, it will be appreciated that a table is only one possible format for what is more generally known as a database. Another embodiment might implement a filter database having a different structure that need not be tabular. The embodiments were chosen and described in order to best explain the principles of the invention and its practical application, thereby enabling others skilled in the art to understand the invention for various embodiments and with various modifications as are suited to the particular use contemplated. It is intended that the scope of the invention be defined by the following claims and their equivalents.

CLAIMS

1. A method for processing data packets received from a network, comprising the steps of:

receiving from a computing device an indication of a first maximum extent-of-search;

iteratively searching a filter table for a matching entry having first comparison criteria that are satisfied by the content of a first incoming data packet received from the network; and

where a matching entry is identified in the step of searching after no more than a number of iterations indicated by the first maximum extent-of-search, delivering the packet to a receive queue in the computing device that is indicated by the matching entry.

2. A method according to claim 1, wherein the first maximum extent-of-search applies to a first incoming data packet type, and a second maximum extent-of-search applies to a second incoming data packet type, further comprising the steps of:

receiving from the computing device an indication of the second maximum extent-of-search;

iteratively searching a filter table a second time for an entry having comparison criteria that are satisfied by the content of a second incoming data packet received from the network;

if a matching entry is identified in the step of searching a second time, after no more than a number of iterations indicated by the second maximum extent-of-search, delivering the packet to a receive queue in the computing device that is indicated by the matching entry; and

if no matching entry is identified in the step of searching after the number of iterations indicated by the second maximum extent-of-search, delivering the packet to a default receive queue in the host.

3. A method according to claim 2, wherein the first and second incoming data packet types are distinguished at least by protocol type.

4. A method according to claim 1, wherein the first comparison criteria comprise first values for a first set of comparison fields,

and wherein the step of iteratively searching comprises the step of iteratively searching the filter table for an entry having values in the first set of comparison fields that match corresponding fields in a header in the incoming packet.

5. A method according to claim 4, wherein the first set of comparison fields includes source and destination IP address fields and source and destination port fields.

6. A method according to claim 4, wherein the first set of comparison fields includes a destination IP address field and a destination port field, but excludes any source IP address fields and any source port fields.

7. A method according to claim 4, wherein no matching entry is identified in the step of searching after the number of iterations indicated by the first maximum extent-of-search, further comprising the steps of:

iteratively searching the filter table a second time for an entry having a second set of comparison fields that all match corresponding fields in the header of the first incoming data packet received from the network, the second set of comparison fields excluding at least one field included in the first set of comparison fields;

if a matching entry is identified in the step of searching a second time, after no more than a number of iterations indicated by a second maximum extent-of-search, delivering the packet to a receive queue in the computing device that is indicated by the matching entry.

8. A method according to claim 7, wherein the first set of comparison fields includes source and destination IP address fields and source and destination port fields,

and wherein the second set of comparison fields includes the destination IP address field and the destination port field and excludes the source IP address field and the source port field.

9. A method according to claim 1, wherein no matching entry is identified in the step of searching after the number of iterations indicated by the first maximum extent-of-search, further comprising the steps of:

iteratively searching the filter table a second time for an entry having second comparison criteria that are satisfied by the content of the first incoming data packet received from the network, the second comparison criteria being less restrictive than the first comparison criteria;

if a matching entry is identified in the step of searching a second time, after no more than a number of iterations indicated by a second maximum extent-of-search, delivering the packet to a receive queue in the computing device that is indicated by the matching entry.

10. A method according to claim 9, further comprising the step of, if no matching entry is identified in the step of searching a second time, after the number of iterations indicated by the second maximum extent-of-search, delivering the packet to a default receive queue in the host.

11. A method according to claim 9, further comprising the step of receiving from the computing device an indication of the second maximum extent-of-search.

12. A method according to claim 9, wherein the number of iterations indicated by the second maximum extent-of-search differs from the number of iterations indicated by first maximum extent-of-search.

13. A method according to claim 1, further comprising the steps of:

after the step of iteratively searching the filter table, receiving from the computing device an indication of a second maximum extent-of-search, the second maximum extent-of-search indicating a number of iterations different from that indicated by the first maximum extent-of-search;

after the step of receiving from the computing device an indication of a second maximum extent-of-search, iteratively searching the filter table a second time for a matching entry having first comparison criteria that are satisfied by the content of a second incoming data packet received from the network; and

only if a matching entry is identified in the step of searching a second time, after no more than a number of iterations indicated by the second maximum extent-of-search, delivering the packet to a receive queue in the computing device that is indicated by the matching entry.

14. A method according to claim 1, further comprising the steps of:

receiving from the computing device a plurality of entries for the filter table; and

programming the plurality of entries into the filter table.

15. A method according to claim 1, wherein the first comparison criteria comprise first values for a first set of comparison fields,

and wherein the step of iteratively searching comprises the step of comparing the values in the first set of comparison fields of each i'th entry in the filter table to the corresponding fields in the header of the first incoming data packet, for sequentially incrementing values of k from 0 up to one less than the number of iterations indicated by the first maximum extent-of-search, or until the matching entry is identified, whichever occurs earlier, where:

i(k) = (h1(key) + k*h2(key)) modulo M,

key is a value dependent upon header information in the header of the first incoming data packet,

h1 and h2 are mathematical functions, and

M is the total index range of the filter table.

16. A method according to claim 1, wherein the first comparison criteria comprise first values for a first set of comparison fields,

and wherein the filter table comprises P sub-tables each of which can hold M entries, P being an integer greater than or equal to 1,

and wherein the step of iteratively searching comprises the step of comparing the values in the first set of comparison fields of each i(k)'th entry in each of the filter sub-tables to the corresponding fields in the header of the first incoming data packet, for sequential values of k incrementing from 0 up to one less than the number of iterations indicated by the first maximum extent-of-search, or until the matching entry is identified, whichever occurs earlier, where:

i(k) = (h1(key) + k*h2(key)) modulo M,

key is a value dependent upon header information in the header of the first incoming data packet, and

h1 and h2 are mathematical functions.

17. A method according to claim 16, wherein the key is a first function of the header information if the first incoming data packet has a first data packet type, and is a second function of the header information different from the first function, if the incoming data packet has a second data packet type.

18. A method according to claim 16, wherein the function h2(key) is given by:

h2(key) = (key*2-1) modulo M.

19. A method according to claim 1, further comprising the step of, if no matching entry is identified in the step of searching after the number of iterations indicated by the first maximum extent-of-search, delivering the packet to a first default receive queue in the host.

20. A method according to claim 19, wherein the step of delivering the packet to a receive queue in the computing device that is indicated by a matching entry comprises the step of writing data from the packet into memory in the computing device that is accessible to a user level process without requiring a context switch to a process having greater privilege than the user level process,

and wherein the step of delivering the packet to a first default receive queue in the host comprises the step of writing data from the packet into memory in the computing device that is accessible only to kernel level processes.

21. A method for processing data packets received from a network, comprising the steps of:

receiving from a computing device an indication of a first maximum extent-of-search;

iteratively searching a filter table for a matching entry having first comparison criteria that are satisfied by the content of a first incoming data packet received from the network; and

where no matching entry is identified in the step of searching after the number of iterations indicated by the first maximum extent-of-search, delivering the packet to a first default receive queue in the host.

22. A method according to claim 21, further comprising the steps of:

iteratively searching the filter table a second time for an entry having second comparison criteria that are satisfied by the content of a second incoming data packet received from the network; and

where no matching entry is identified in the step of searching a second time, after the number of iterations indicated by the first maximum extent-of-search, delivering the packet to a second default receive queue in the host, the second default receive queue being different from the first default receive queue.

23. A method according to claim 21, for use where the computing device includes a plurality of default receive queues,

further comprising the step of selecting the first default receive queue from among the plurality of default receive queues in dependence upon the content of the first incoming data packet.

24. A method according to claim 21, wherein the first comparison criteria comprise first values for a first set of comparison fields,

and wherein the filter table comprises P sub-tables each of which can hold M entries, P being an integer greater than or equal to 1,

and wherein the step of iteratively searching comprises the step of comparing the comparison fields of each i(k)'th entry in each of the filter sub-tables to the corresponding fields in the header of the first incoming data packet, for sequential values of k incrementing from 0 up to one less than the number of iterations indicated by the first maximum extent-of-search, or until the matching entry is identified, whichever occurs earlier, where:

i(k) = (h1(key) + k*h2(key)) modulo M,

key is a value dependent upon header information in the header of the first incoming data packet, and

h1 and h2 are mathematical functions,

the method further comprising the step of selecting the first default receive queue from among the plurality of receive queues in dependence upon the key.

25. A method according to claim 21, wherein the step of delivering the packet to a first default receive queue in the host comprises the step of writing data from the packet into memory in the computing device that is accessible only to kernel level processes.

26. A method according to claim 21, further comprising the step of, in response to receipt of the packet into the first default receive queue, the computing device forwarding data from the packet toward a target process selected in dependence upon header information in the header of the packet.

27. A method according to claim 26, wherein the step of delivering the packet to a first default receive queue in the host comprises the step of writing data from the packet into memory in the computing device that is accessible only to kernel level processes,

and wherein the target process is a user level process.

28. A method according to claim 21, wherein the first comparison criteria comprise first values for a first set of comparison fields, further comprising, prior to the step of iteratively searching, the step of iteratively searching the filter table preliminarily for an entry having a preliminary set of comparison fields that all match corresponding fields in the header of the first incoming data packet received from the network,

wherein no matching entry is identified in the step of searching the filter table preliminarily, after the number of iterations indicated by a preliminary maximum extent-of-search,

and wherein the first set of comparison fields excludes at least one field included in the preliminary set of comparison fields.

29. A method according to claim 28, further comprising the step of receiving from the computing device an indication of the preliminary maximum extent-of-search.

30. A method according to claim 28, wherein the number of iterations indicated by the preliminary maximum extent-of-search differs from the number of iterations indicated by first maximum extent-of-search.

31. A method according to claim 28, wherein the preliminary set of comparison fields includes source and destination IP address fields and source and destination port fields,

and wherein the first set of comparison fields includes the destination IP address field and the destination port field and excludes the source IP address field and the source port field.

32. A method for controlling a network interface device, for use with a host computing device having a plurality of network data receive queues, comprising the steps of:

programming a receive filter table accessible to the network interface device, each entry in the receive filter table identifying a receive queue to which incoming packets are to be delivered if their content satisfies criteria indicated by the entry; and

programming the network interface device with an indication of a first maximum extent-of-search to which the network interface device should search in the filter table before delivering an incoming packet to a default receive queue.

33. A method according to claim 32, wherein the step of programming a receive filter table accessible to the network interface device comprises the step of programming, into a first entry in the receive filter table, an indication of a first receive queue and an indication of first criteria for testing against incoming packets,

further comprising the step of receiving first data into the first queue in response to receipt by the network interface device of an incoming packet having content satisfying the first criteria.

34. A method according to claim 33, further comprising the step of receiving second data into the default receive queue in response to receipt by the network interface device of a second packet whose content does not satisfy the first criteria.

35. A method according to claim 34, wherein the content of the second packet satisfies none of the criteria indicated in the table.

36. A method according to claim 34, wherein the default queue is not identified anywhere in the table.

37. A method according to claim 34, further comprising the steps of:

identifying a particular destination user level process in dependence upon the content of the second data packet; and

delivering the second data from the default queue to the particular destination user level process.

38. A method according to claim 34, wherein the step of receiving the first data into the first queue comprises the step of receiving the first data into memory in the computing device that is accessible to a user level process without requiring a context switch to a process having greater privilege than the user level process,

and wherein the step of receiving second data into the default receive queue comprises the step of receiving the second data into memory in the computing device that is accessible only to kernel level processes.

39. A method according to claim 32, further comprising the step of receiving second data into the default receive queue in response to receipt by the network interface device of a second packet whose content does satisfy the first criteria.

40. A method according to claim 32, wherein the step of programming a receive filter table comprises the step of programming a plurality of entries in the receive filter table with:

values for a respective set of comparison fields, and

an identification of a destination receive queue to which incoming packets are to be delivered if the respective set of comparison fields match corresponding fields in a header in the incoming packet.

41. A method according to claim 40, wherein the first maximum extent-of-search indicates a first maximum number of search iterations that the network interface device should

perform while searching for an entry having a first set of comparison fields that all match corresponding fields in the header of a first incoming data packet received from the network,

further comprising the step of programming the network interface device with an indication of a second maximum extent-of-search to which the network interface device should search in the filter table after failing to find an entry having a first set of comparison fields that all match corresponding fields in the header of the first incoming data packet and before delivering an incoming packet to a default receive queue, the second maximum extent-of-search indicating a second maximum number of search iterations that the network interface device should perform while searching for an entry having a second set of comparison fields that all match corresponding fields in the header of a first incoming data packet, the second set of comparison fields excluding at least one field included in the first set of comparison fields.

42. A method according to claim 41, wherein the first set of comparison fields includes source and destination IP address fields and source and destination port fields,

and wherein the second set of comparison fields includes the destination IP address field and the destination port field and excludes the source IP address field and the source port field.

43. A method according to claim 32, further comprising the steps of:

receiving first data into a first queue from the network interface device in response to receipt by the network interface device of an incoming packet; and

reprogramming the first maximum extent-of-search indication in the network interface device after the step of receiving.

44. A method according to claim 32, wherein the first maximum extent-of-search applies to a first incoming data packet type, and wherein the step of programming a receive filter table accessible to the network interface device further comprises the step of programming, into

a second entry in the receive filter table, an indication of a second receive queue and an indication of second criteria for testing against incoming packets, further comprising the steps of:

programming the network interface device with an indication of a second maximum extent-of-search to which the network interface device should search in the filter table for incoming packets of a second incoming data packet type, before delivering an incoming packet to a default receive queue, the second criteria including an indication of the second incoming data packet type; and

receiving second data into the second queue in response to receipt by the network interface device of an incoming packet of the second incoming data packet type and having content satisfying the second criteria.

45. A method according to claim 32, wherein the step of programming a receive filter table, for first criteria to be associated with a predetermined first receive queue, comprises the step of iteratively searching through an accessible copy of the filter table, at successive entries i(k), for sequentially incrementing values of k where:

i(k) = (h1(key) + k*h2(key)) modulo M,

key is a value dependent upon the first criteria,

h1 and h2 are mathematical functions, and

M is the total index range of the filter table.

46.A method according to claim 45, wherein the key is a first function of the first criteria if the first criteria requires that a matching incoming packet has a first data packet type, and is a second function of the first criteria different from the first function, if the first criteria requires that a matching incoming packet has a second data packet type.

47. A method for processing data packets received from a network, for use with a plurality of processors each having a respective kernel receive queue, comprising the steps of:

searching a filter table for an entry matching the content of a first incoming data packet received from the network; and

where no matching entry is identified in the step of searching:

selecting a selected one of the kernel receive queues in dependence upon the content of the first incoming data packet; and

delivering the packet to the selected one of the kernel receive queues.

48. A method according to claim 47, wherein the step of selecting a selected one of the kernel receive queues comprises the step of performing a hash function on at least one header field of the first incoming packet.

49. A method according to claim 47, wherein the step of selecting a selected one of the kernel receive queues comprises the step of performing the Toeplitz function on at least one header field of the first incoming packet.

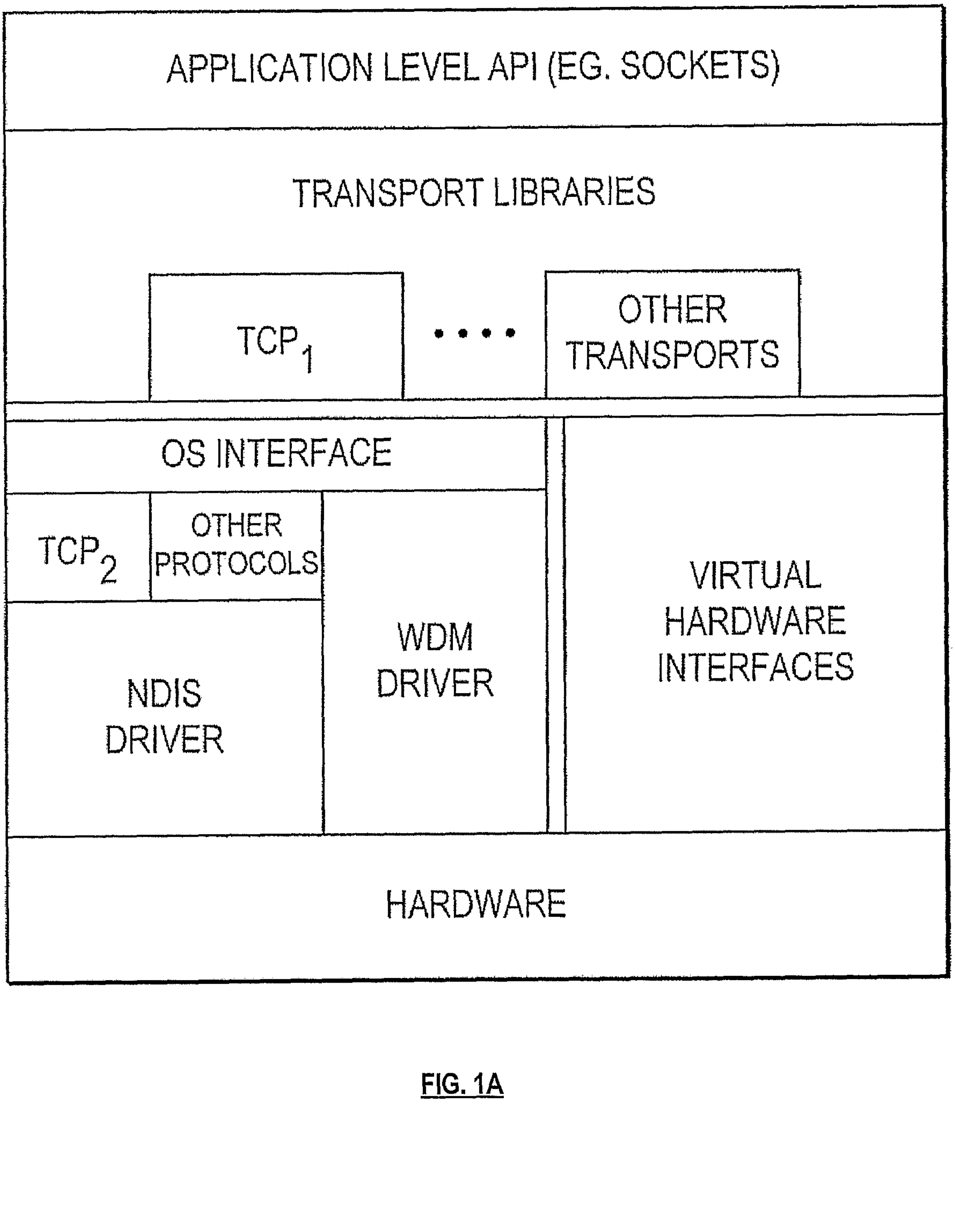
APPLICATION LEVEL API (EG. SOCKETS)
TRANSPORT LIBRARIES
TCP_1
• • • •
OTHER TRANSPORTS
OS INTERFACE
TCP_2
OTHER PROTOCOLS
NDIS DRIVER
WDM DRIVER
VIRTUAL HARDWARE INTERFACES
HARDWARE

FIG. 1A

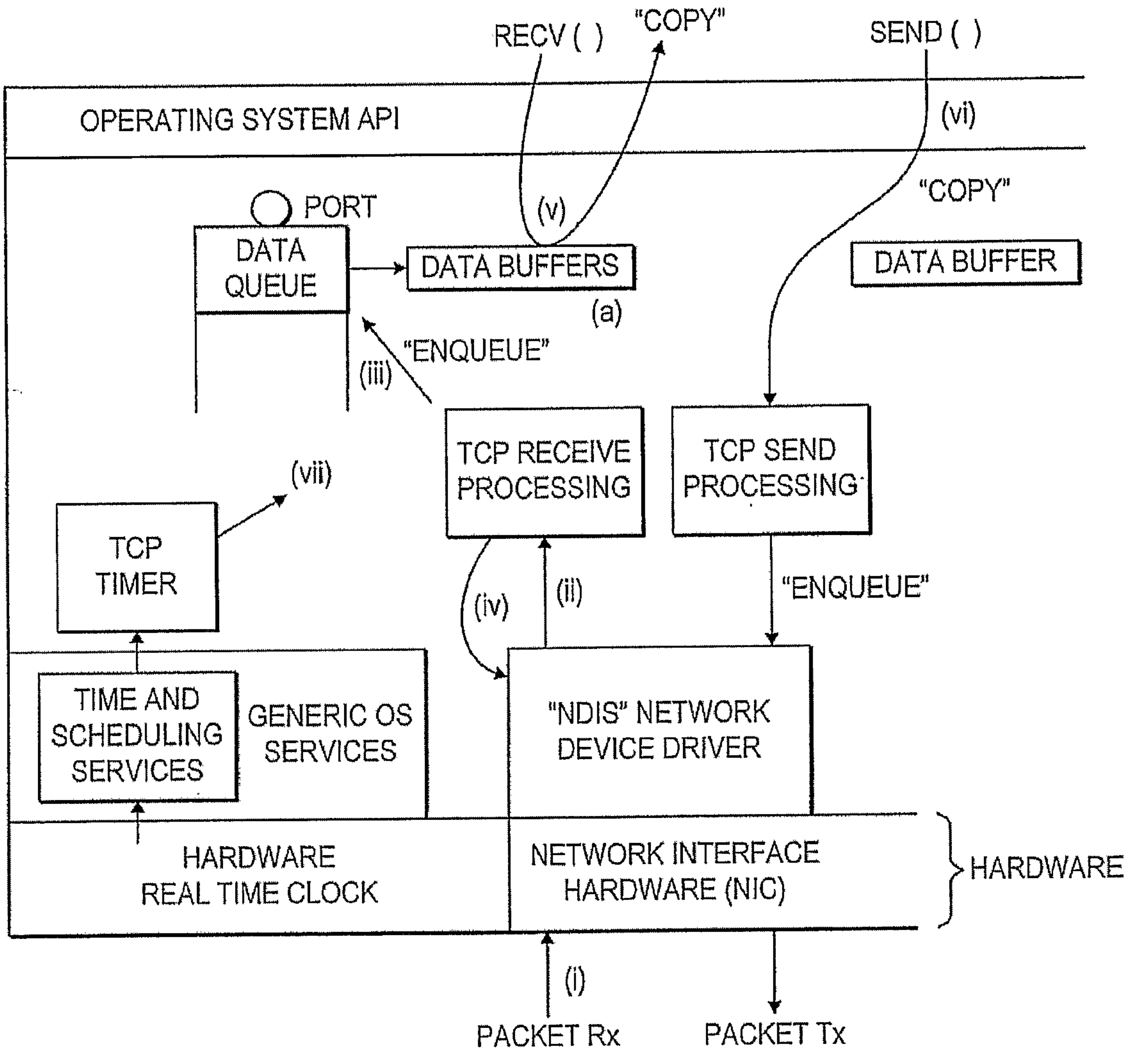

FIG. 1B
(PRIOR ART)

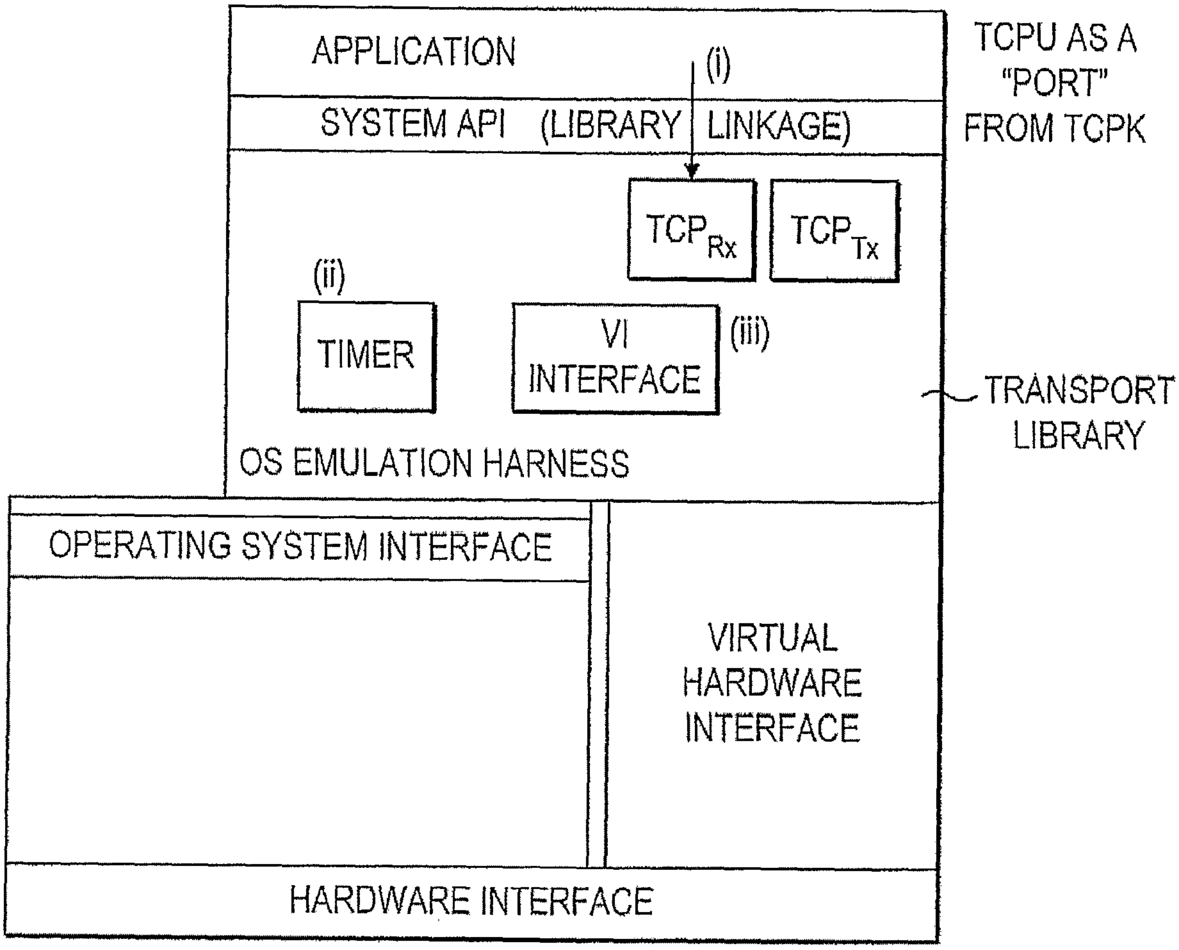
APPLICATION
(i)
SYSTEM API (LIBRARY LINKAGE)
TCPU AS A "PORT" FROM TCPK
TCP$_{Rx}$
TCP$_{Tx}$
(ii)
TIMER
VI INTERFACE
(iii)
TRANSPORT LIBRARY
OS EMULATION HARNESS
OPERATING SYSTEM INTERFACE
VIRTUAL HARDWARE INTERFACE
HARDWARE INTERFACE

FIG. 1C

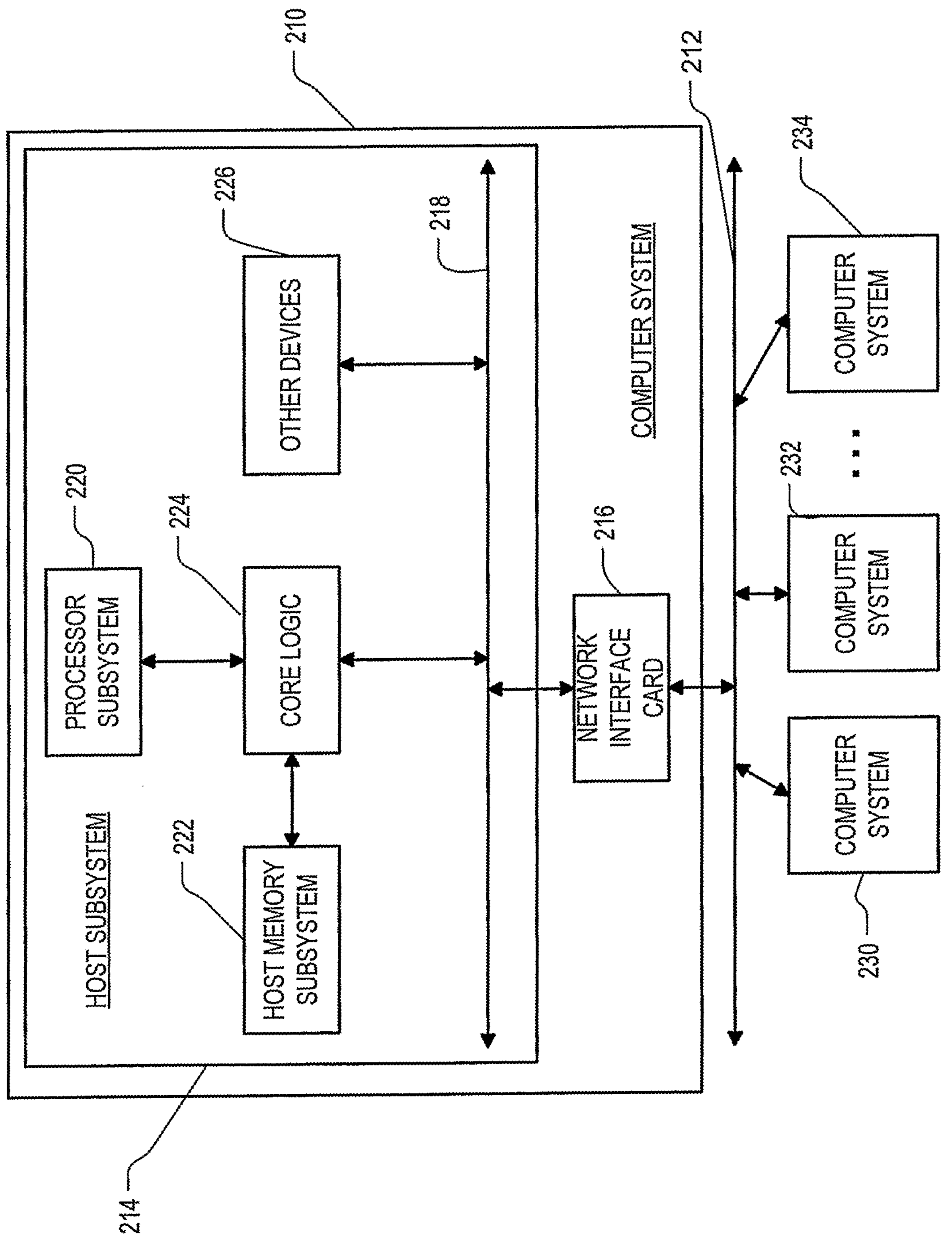
210
212
214
HOST SUBSYSTEM
PROCESSOR SUBSYSTEM
220
CORE LOGIC
224
HOST MEMORY SUBSYSTEM
222
OTHER DEVICES
226
218
COMPUTER SYSTEM
NETWORK INTERFACE CARD
216
COMPUTER SYSTEM
234
COMPUTER SYSTEM
232
COMPUTER SYSTEM
230
. . .

FIG. 2

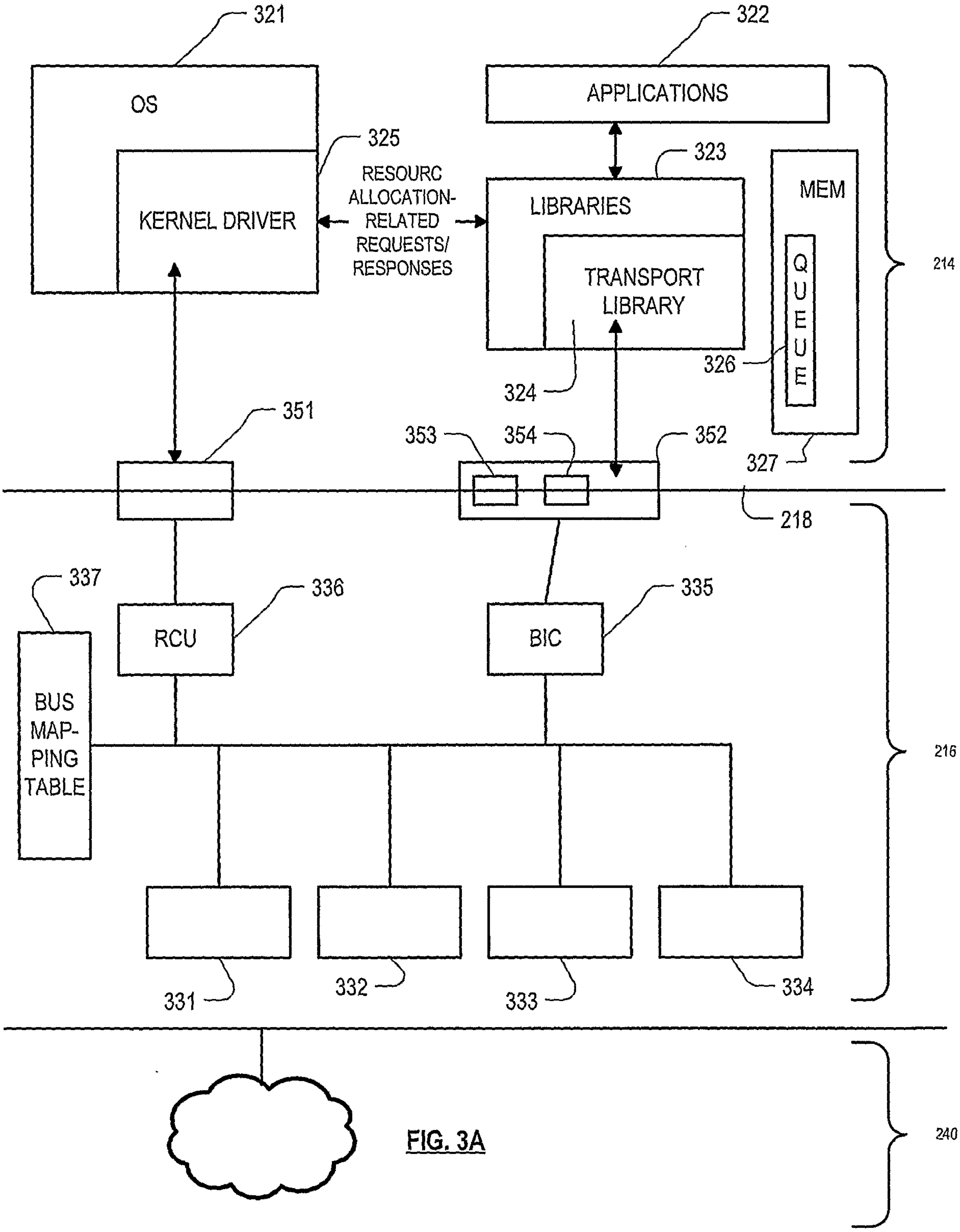
321
OS
KERNEL DRIVER
325
RESOURC ALLOCATION-RELATED REQUESTS/ RESPONSES
322
APPLICATIONS
323
LIBRARIES
TRANSPORT LIBRARY
324
MEM
QUEUE
326
327
214
351
353
354
352
218
337
BUS MAPPING TABLE
RCU
336
BIC
335
216
331
332
333
334
240

FIG. 3A

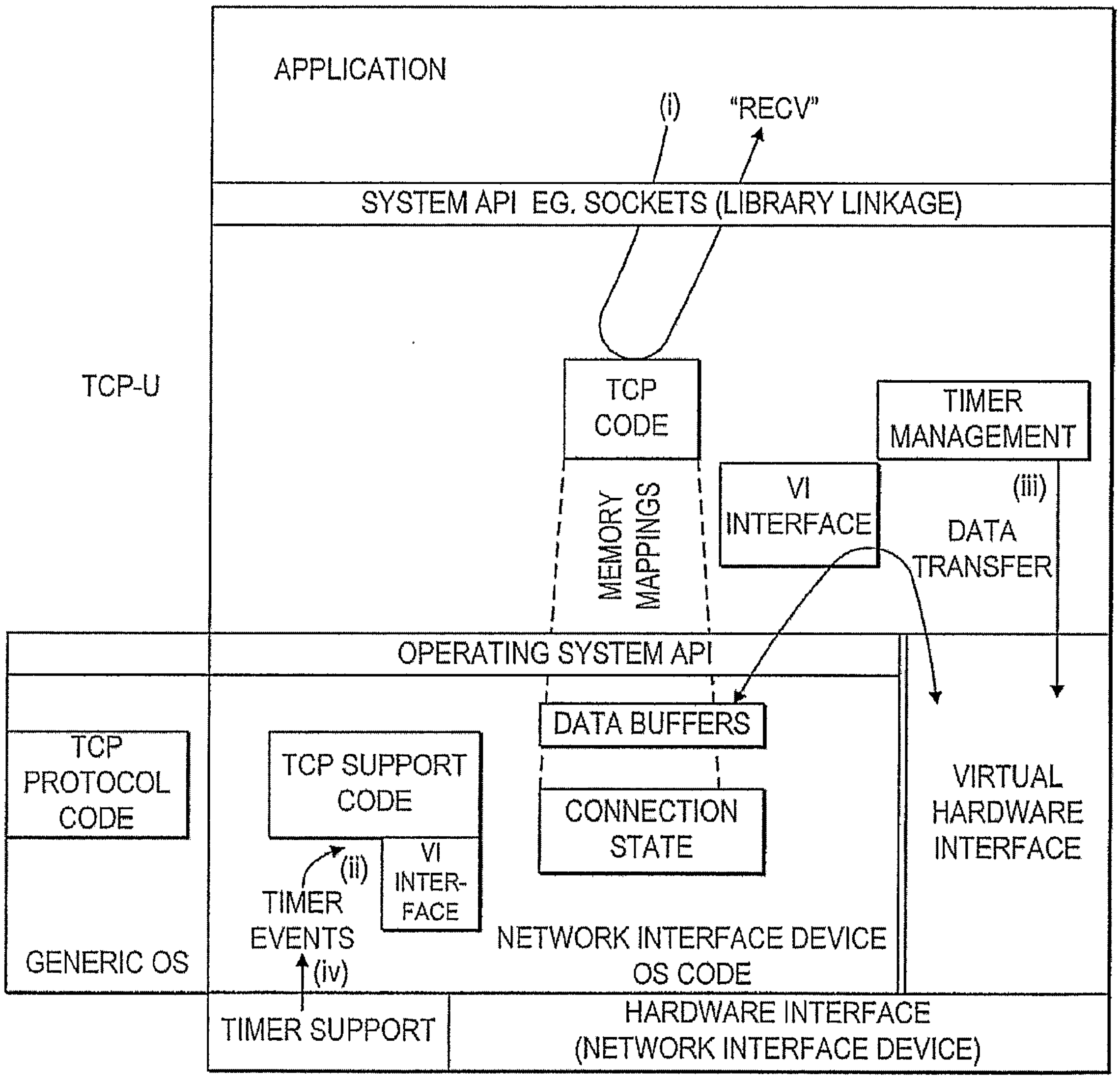
APPLICATION
(i)
"RECV"
SYSTEM API EG. SOCKETS (LIBRARY LINKAGE)
TCP-U
TCP CODE
TIMER MANAGEMENT
VI INTERFACE
(iii)
DATA TRANSFER
MEMORY MAPPINGS
OPERATING SYSTEM API
DATA BUFFERS
TCP PROTOCOL CODE
TCP SUPPORT CODE
CONNECTION STATE
VIRTUAL HARDWARE INTERFACE
(ii)
VI INTER-FACE
TIMER EVENTS
GENERIC OS
NETWORK INTERFACE DEVICE OS CODE
(iv)
TIMER SUPPORT
HARDWARE INTERFACE (NETWORK INTERFACE DEVICE)

FIG. 3B

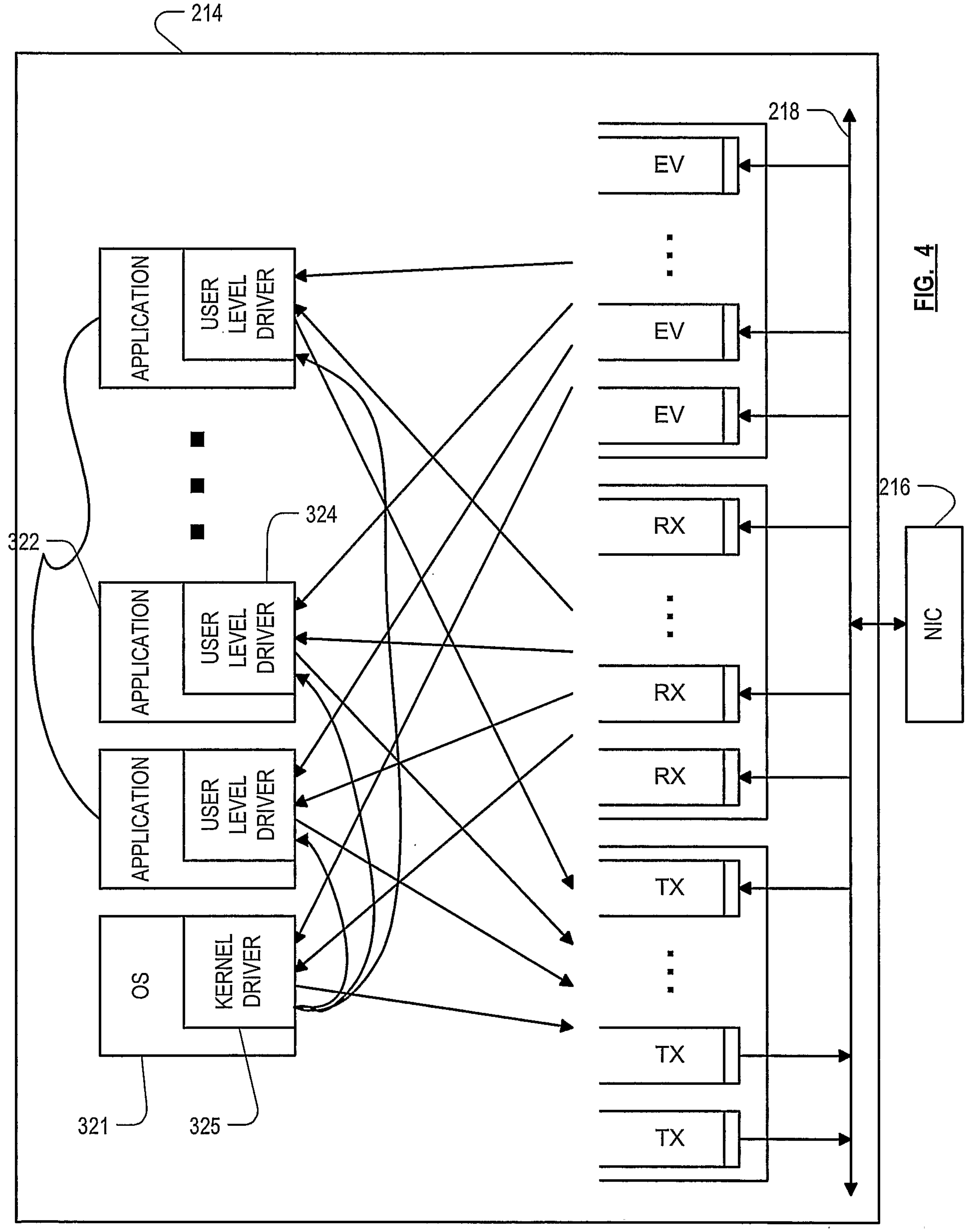
214
218
EV
EV
EV
RX
RX
RX
TX
TX
TX
216
NIC
APPLICATION
USER LEVEL DRIVER
APPLICATION
USER LEVEL DRIVER
324
322
APPLICATION
USER LEVEL DRIVER
OS
KERNEL DRIVER
321
325

FIG. 4

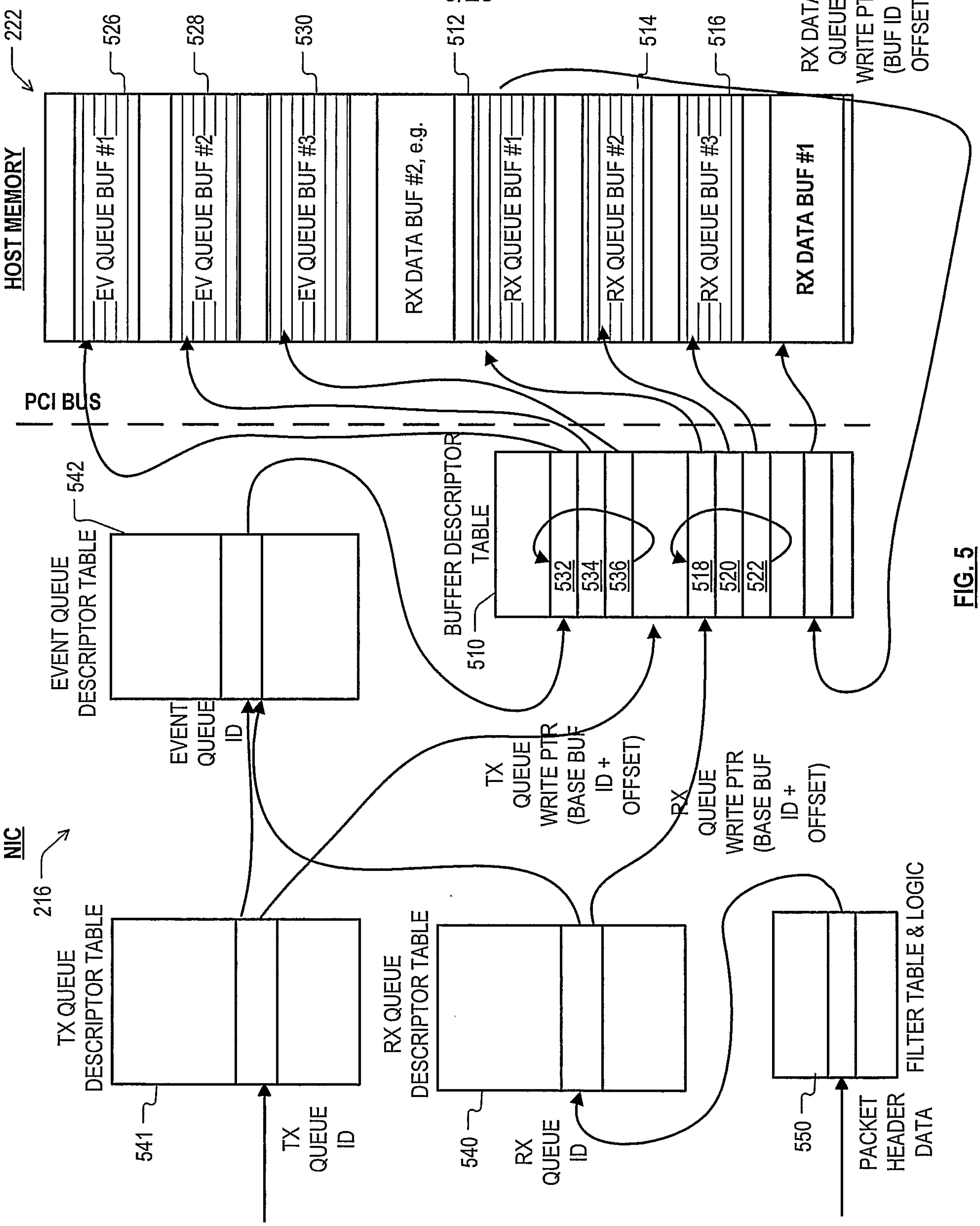
HOST MEMORY
222
EV QUEUE BUF #1
526
EV QUEUE BUF #2
528
EV QUEUE BUF #3
530
RX DATA BUF #2, e.g.
512
RX QUEUE BUF #1
514
RX QUEUE BUF #2
516
RX QUEUE BUF #3
RX DATA BUF #1
RX DATA QUEUE WRITE PTR (BUF ID + OFFSET)
PCI BUS
NIC
216
TX QUEUE DESCRIPTOR TABLE
541
TX QUEUE ID
EVENT QUEUE DESCRIPTOR TABLE
542
EVENT QUEUE ID
BUFFER DESCRIPTOR TABLE
510
532
534
536
518
520
522
TX QUEUE WRITE PTR (BASE BUF ID + OFFSET)
RX QUEUE WRITE PTR (BASE BUF ID + OFFSET)
RX QUEUE DESCRIPTOR TABLE
540
RX QUEUE ID
FILTER TABLE & LOGIC
550
PACKET HEADER DATA

FIG. 5

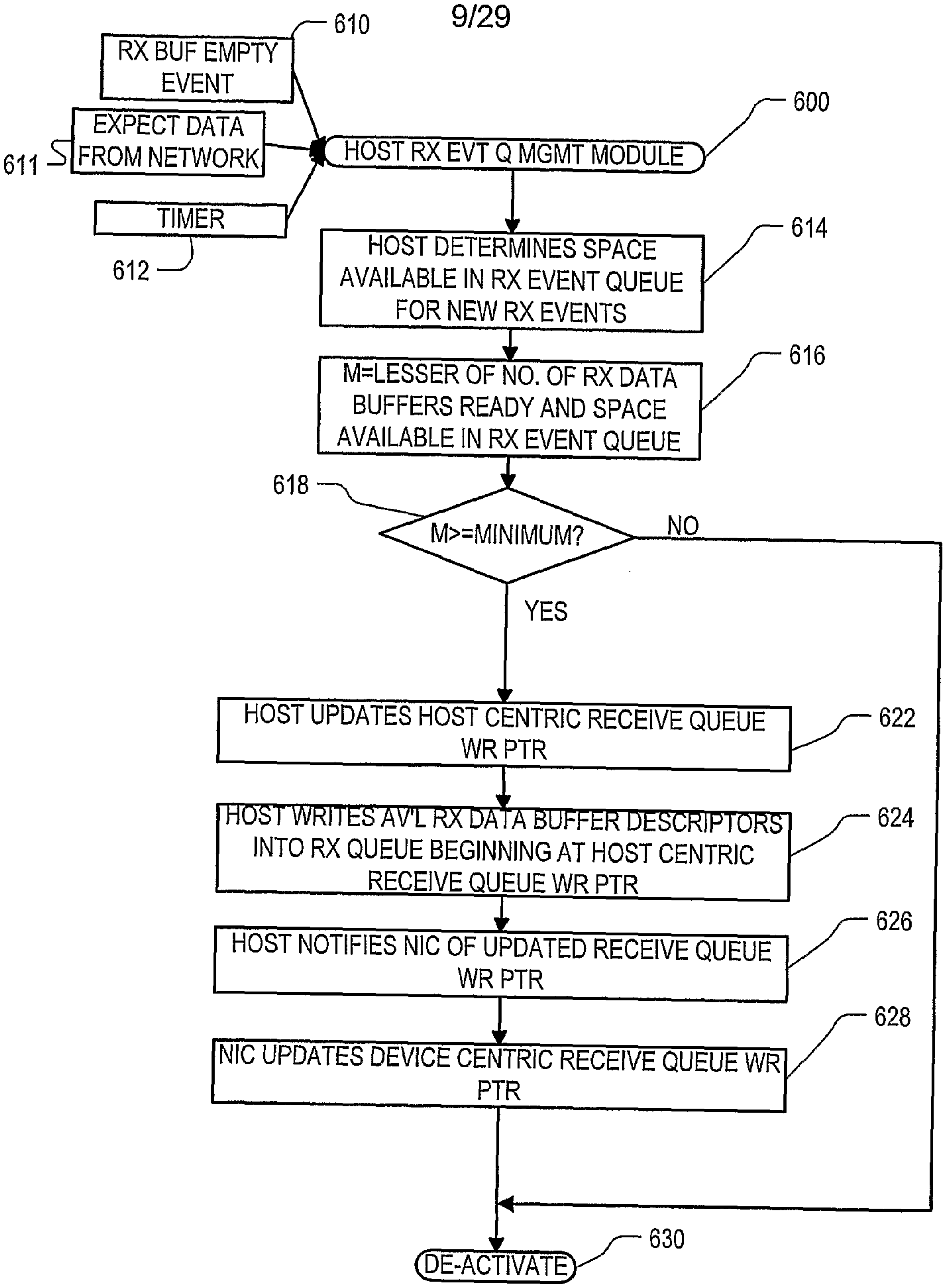
RX BUF EMPTY EVENT
610
EXPECT DATA FROM NETWORK
611
TIMER
612
HOST RX EVT Q MGMT MODULE
600
HOST DETERMINES SPACE AVAILABLE IN RX EVENT QUEUE FOR NEW RX EVENTS
614
M=LESSER OF NO. OF RX DATA BUFFERS READY AND SPACE AVAILABLE IN RX EVENT QUEUE
616
618
M>=MINIMUM?
NO
YES
HOST UPDATES HOST CENTRIC RECEIVE QUEUE WR PTR
622
HOST WRITES AV'L RX DATA BUFFER DESCRIPTORS INTO RX QUEUE BEGINNING AT HOST CENTRIC RECEIVE QUEUE WR PTR
624
HOST NOTIFIES NIC OF UPDATED RECEIVE QUEUE WR PTR
626
NIC UPDATES DEVICE CENTRIC RECEIVE QUEUE WR PTR
628
DE-ACTIVATE
630

FIG. 6

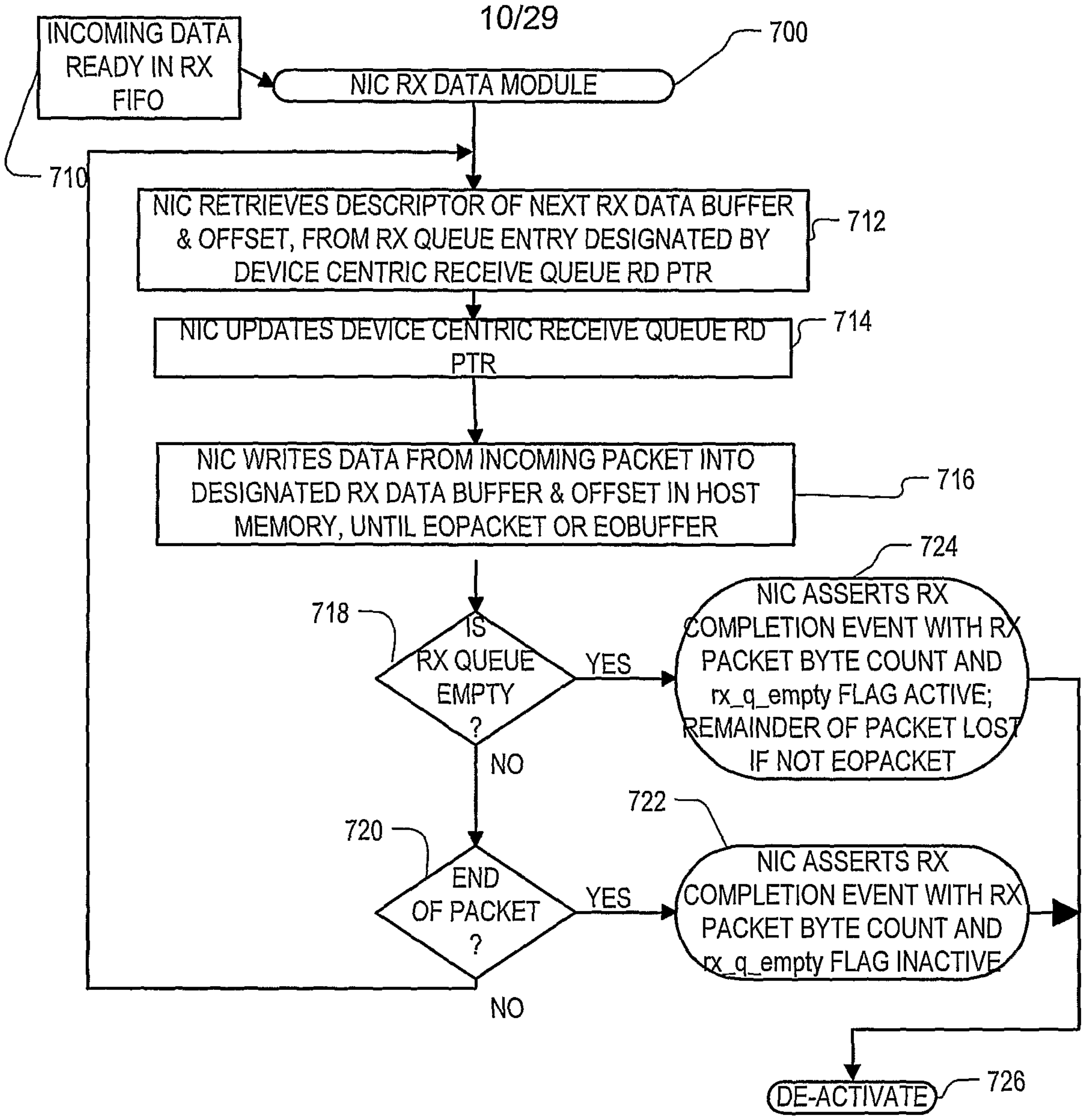
INCOMING DATA READY IN RX FIFO
710
NIC RX DATA MODULE
700
NIC RETRIEVES DESCRIPTOR OF NEXT RX DATA BUFFER & OFFSET, FROM RX QUEUE ENTRY DESIGNATED BY DEVICE CENTRIC RECEIVE QUEUE RD PTR
712
NIC UPDATES DEVICE CENTRIC RECEIVE QUEUE RD PTR
714
NIC WRITES DATA FROM INCOMING PACKET INTO DESIGNATED RX DATA BUFFER & OFFSET IN HOST MEMORY, UNTIL EOPACKET OR EOBUFFER
716
718
IS RX QUEUE EMPTY ?
YES
NO
724
NIC ASSERTS RX COMPLETION EVENT WITH RX PACKET BYTE COUNT AND rx_q_empty FLAG ACTIVE; REMAINDER OF PACKET LOST IF NOT EOPACKET
720
END OF PACKET ?
YES
NO
722
NIC ASSERTS RX COMPLETION EVENT WITH RX PACKET BYTE COUNT AND rx_q_empty FLAG INACTIVE
DE-ACTIVATE
726

FIG. 7

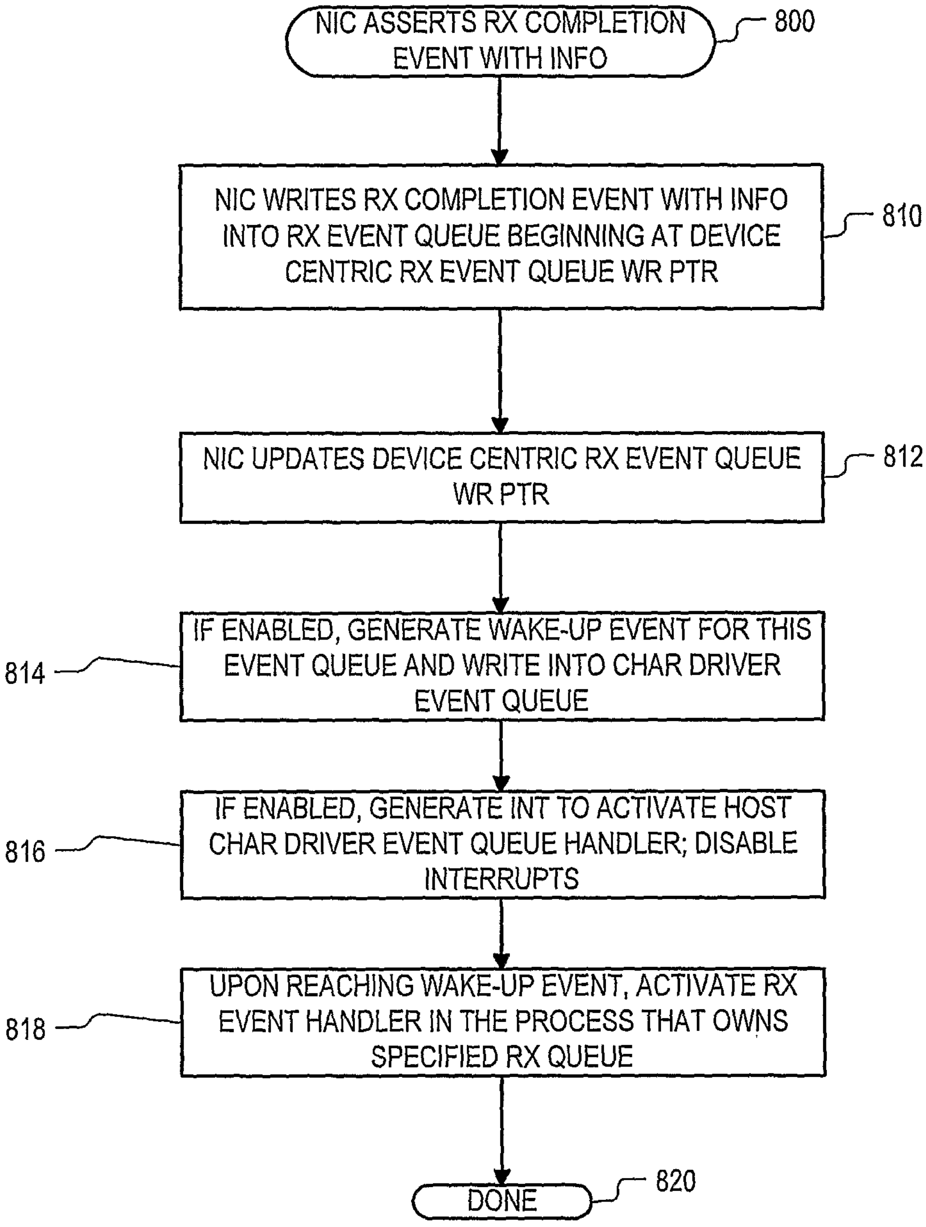
NIC ASSERTS RX COMPLETION EVENT WITH INFO
800
NIC WRITES RX COMPLETION EVENT WITH INFO INTO RX EVENT QUEUE BEGINNING AT DEVICE CENTRIC RX EVENT QUEUE WR PTR
810
NIC UPDATES DEVICE CENTRIC RX EVENT QUEUE WR PTR
812
814
IF ENABLED, GENERATE WAKE-UP EVENT FOR THIS EVENT QUEUE AND WRITE INTO CHAR DRIVER EVENT QUEUE
816
IF ENABLED, GENERATE INT TO ACTIVATE HOST CHAR DRIVER EVENT QUEUE HANDLER; DISABLE INTERRUPTS
818
UPON REACHING WAKE-UP EVENT, ACTIVATE RX EVENT HANDLER IN THE PROCESS THAT OWNS SPECIFIED RX QUEUE
DONE
820

FIG. 8

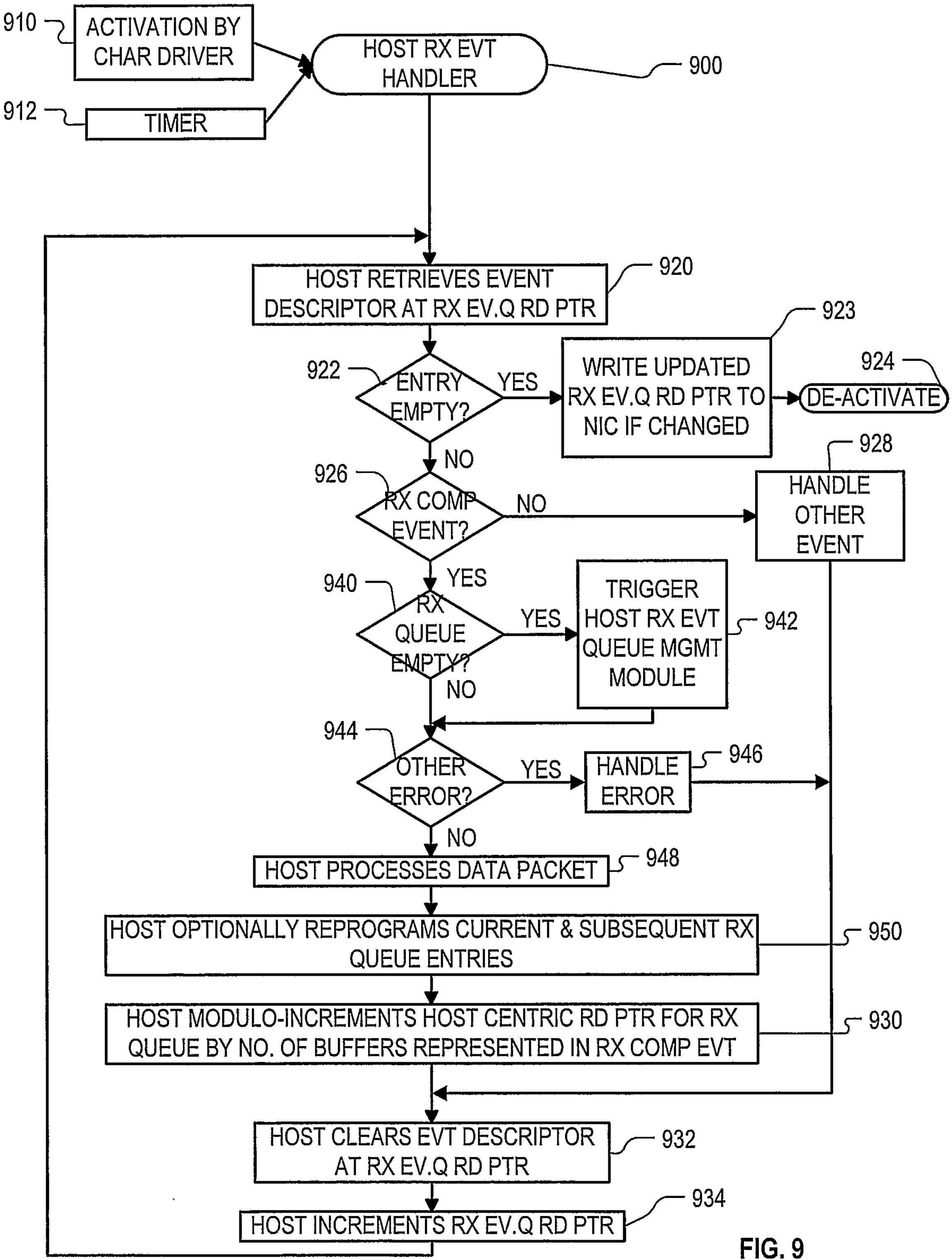
910
ACTIVATION BY CHAR DRIVER
912
TIMER
HOST RX EVT HANDLER
900
HOST RETRIEVES EVENT DESCRIPTOR AT RX EV.Q RD PTR
920
922
ENTRY EMPTY?
YES
WRITE UPDATED RX EV.Q RD PTR TO NIC IF CHANGED
923
924
DE-ACTIVATE
NO
926
RX COMP EVENT?
NO
HANDLE OTHER EVENT
928
YES
940
RX QUEUE EMPTY?
YES
TRIGGER HOST RX EVT QUEUE MGMT MODULE
942
NO
944
OTHER ERROR?
YES
HANDLE ERROR
946
NO
HOST PROCESSES DATA PACKET
948
HOST OPTIONALLY REPROGRAMS CURRENT & SUBSEQUENT RX QUEUE ENTRIES
950
HOST MODULO-INCREMENTS HOST CENTRIC RD PTR FOR RX QUEUE BY NO. OF BUFFERS REPRESENTED IN RX COMP EVT
930
HOST CLEARS EVT DESCRIPTOR AT RX EV.Q RD PTR
932
HOST INCREMENTS RX EV.Q RD PTR
934

FIG. 9

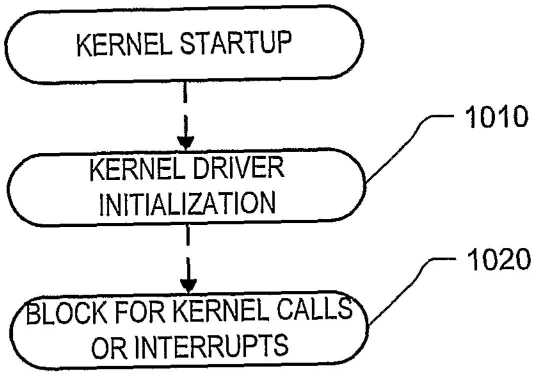
KERNEL STARTUP
KERNEL DRIVER INITIALIZATION
1010
BLOCK FOR KERNEL CALLS OR INTERRUPTS
1020

FIG. 10

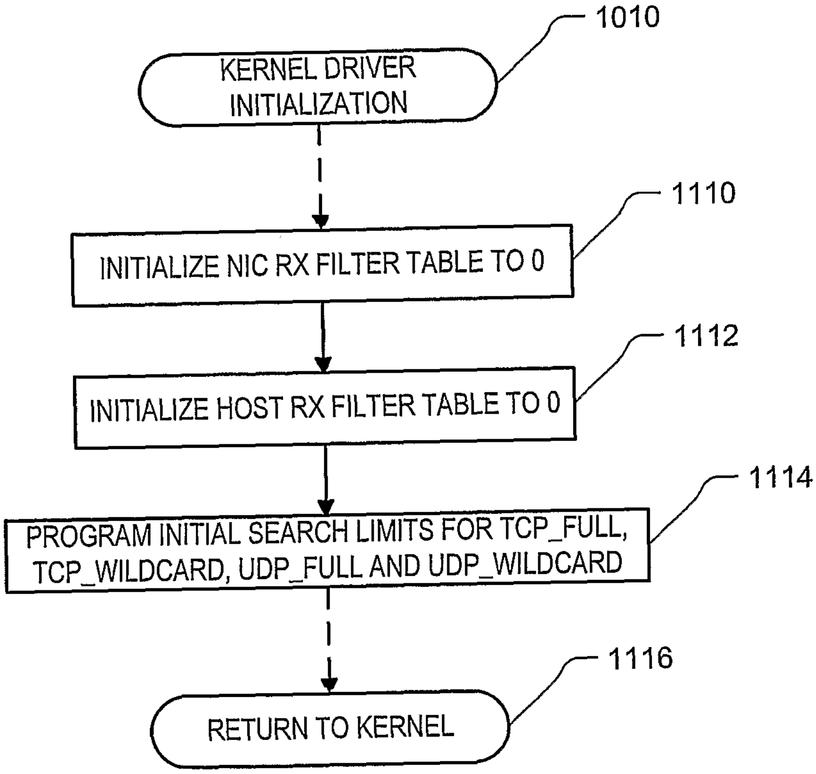
KERNEL DRIVER INITIALIZATION
1010
INITIALIZE NIC RX FILTER TABLE TO 0
1110
INITIALIZE HOST RX FILTER TABLE TO 0
1112
PROGRAM INITIAL SEARCH LIMITS FOR TCP_FULL, TCP_WILDCARD, UDP_FULL AND UDP_WILDCARD
1114
RETURN TO KERNEL
1116

FIG. 11

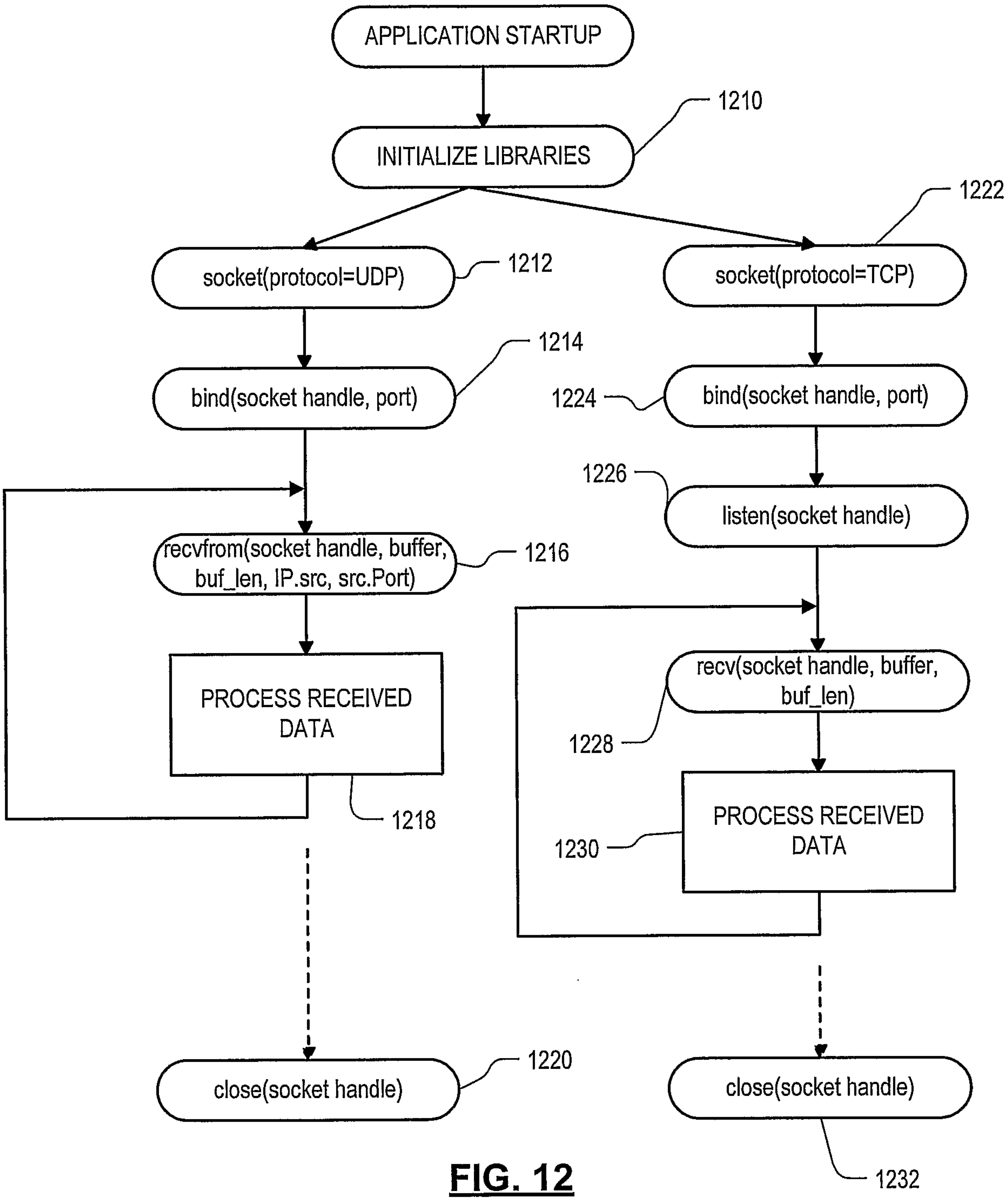
APPLICATION STARTUP
INITIALIZE LIBRARIES
1210
socket(protocol=UDP)
1212
socket(protocol=TCP)
1222
bind(socket handle, port)
1214
bind(socket handle, port)
1224
1226
listen(socket handle)
recvfrom(socket handle, buffer, buf_len, IP.src, src.Port)
1216
PROCESS RECEIVED DATA
1218
recv(socket handle, buffer, buf_len)
1228
PROCESS RECEIVED DATA
1230
close(socket handle)
1220
close(socket handle)
1232

FIG. 12

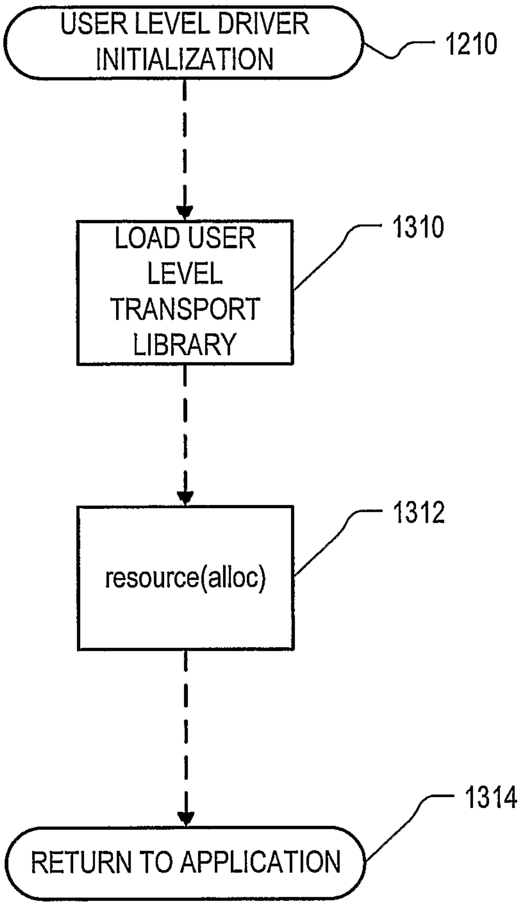
USER LEVEL DRIVER INITIALIZATION
1210
LOAD USER LEVEL TRANSPORT LIBRARY
1310
resource(alloc)
1312
RETURN TO APPLICATION
1314

FIG. 13

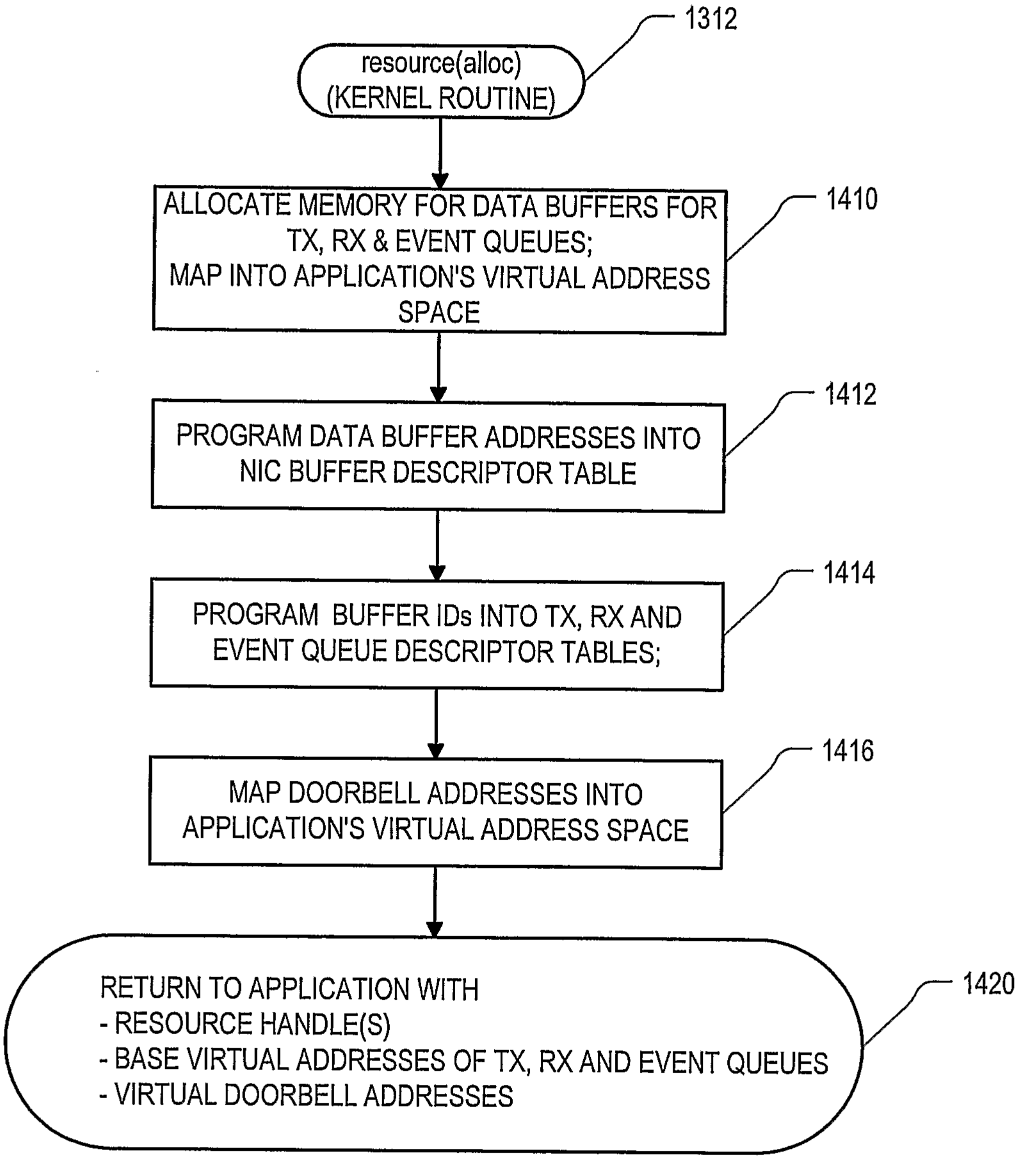
1312
resource(alloc)
(KERNEL ROUTINE)
1410
ALLOCATE MEMORY FOR DATA BUFFERS FOR TX, RX & EVENT QUEUES;
MAP INTO APPLICATION'S VIRTUAL ADDRESS SPACE
1412
PROGRAM DATA BUFFER ADDRESSES INTO NIC BUFFER DESCRIPTOR TABLE
1414
PROGRAM BUFFER IDs INTO TX, RX AND EVENT QUEUE DESCRIPTOR TABLES;
1416
MAP DOORBELL ADDRESSES INTO APPLICATION'S VIRTUAL ADDRESS SPACE
1420
RETURN TO APPLICATION WITH
- RESOURCE HANDLE(S)
- BASE VIRTUAL ADDRESSES OF TX, RX AND EVENT QUEUES
- VIRTUAL DOORBELL ADDRESSES

FIG. 14

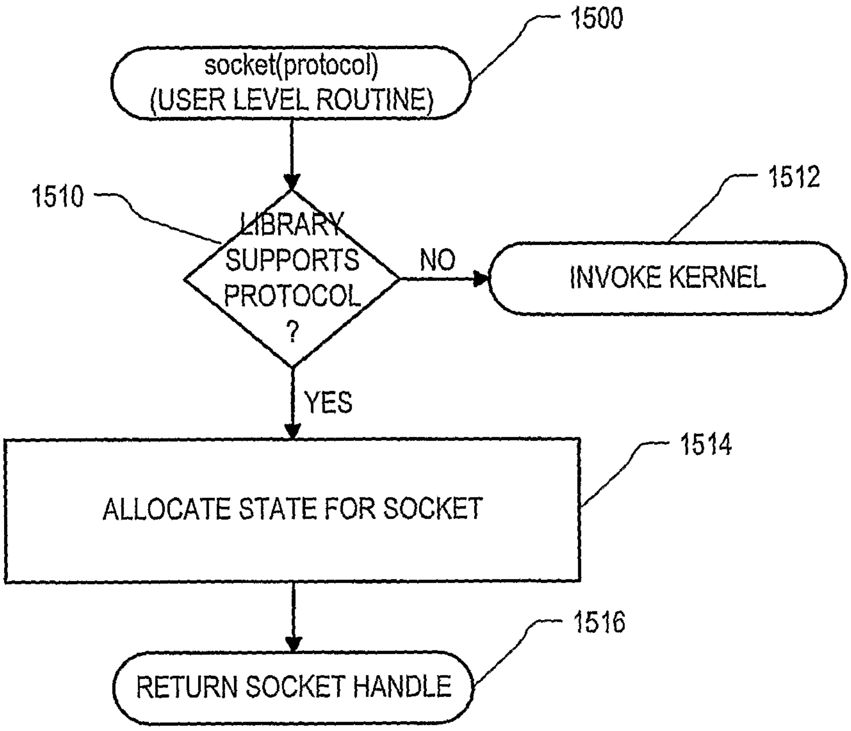
1500
socket(protocol)
(USER LEVEL ROUTINE)
1510
LIBRARY SUPPORTS PROTOCOL ?
NO
1512
INVOKE KERNEL
YES
1514
ALLOCATE STATE FOR SOCKET
1516
RETURN SOCKET HANDLE

FIG. 15

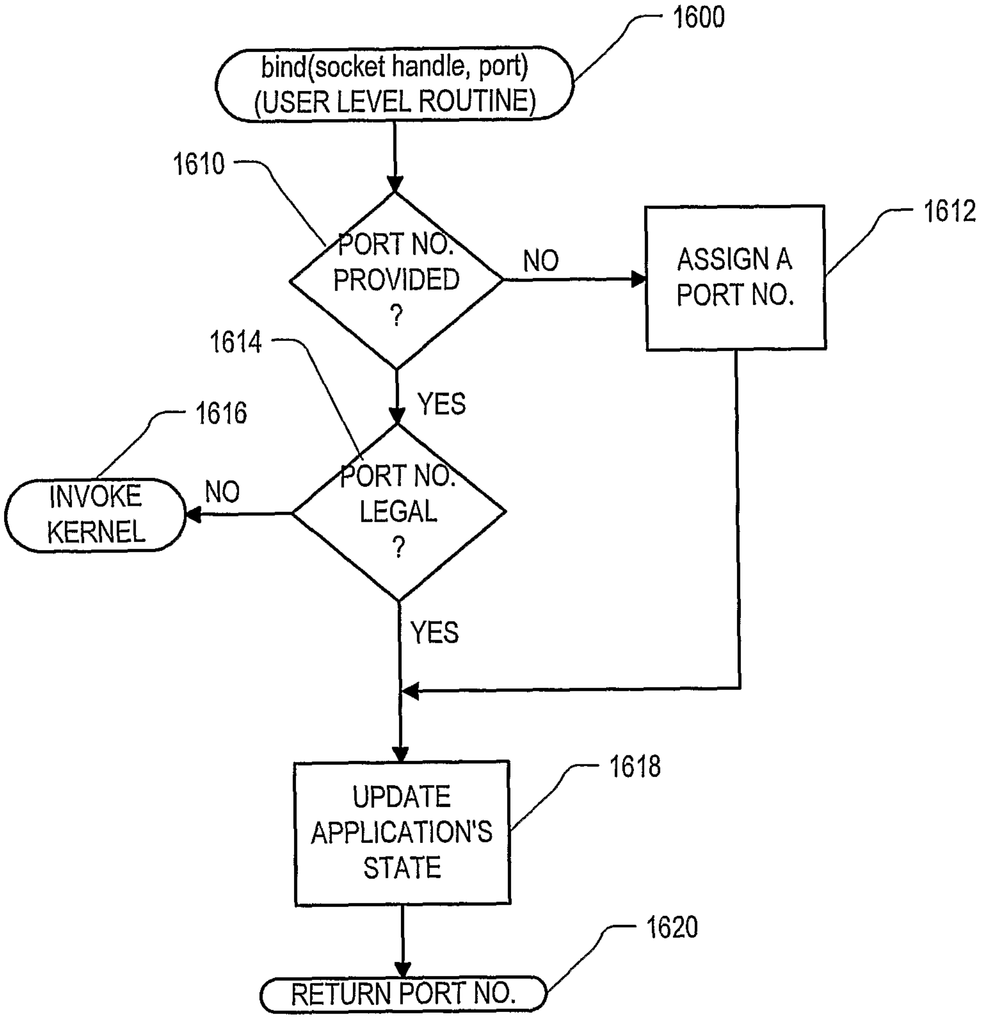
bind(socket handle, port)
(USER LEVEL ROUTINE)
1600
1610
PORT NO. PROVIDED ?
NO
ASSIGN A PORT NO.
1612
YES
1614
PORT NO. LEGAL ?
NO
INVOKE KERNEL
1616
YES
UPDATE APPLICATION'S STATE
1618
RETURN PORT NO.
1620

FIG. 16

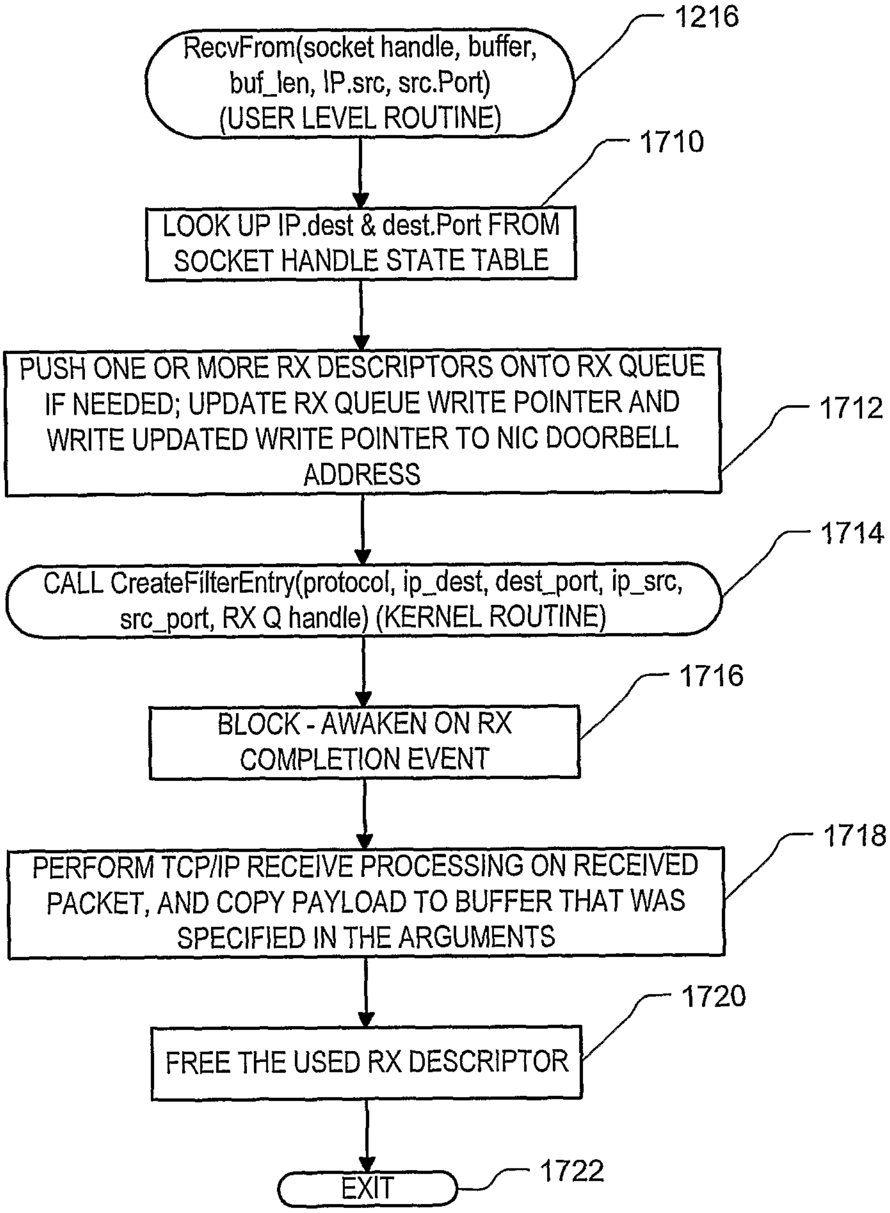
RecvFrom(socket handle, buffer, buf_len, IP.src, src.Port) (USER LEVEL ROUTINE)
1216
LOOK UP IP.dest & dest.Port FROM SOCKET HANDLE STATE TABLE
1710
PUSH ONE OR MORE RX DESCRIPTORS ONTO RX QUEUE IF NEEDED; UPDATE RX QUEUE WRITE POINTER AND WRITE UPDATED WRITE POINTER TO NIC DOORBELL ADDRESS
1712
CALL CreateFilterEntry(protocol, ip_dest, dest_port, ip_src, src_port, RX Q handle) (KERNEL ROUTINE)
1714
BLOCK - AWAKEN ON RX COMPLETION EVENT
1716
PERFORM TCP/IP RECEIVE PROCESSING ON RECEIVED PACKET, AND COPY PAYLOAD TO BUFFER THAT WAS SPECIFIED IN THE ARGUMENTS
1718
FREE THE USED RX DESCRIPTOR
1720
EXIT
1722

FIG. 17

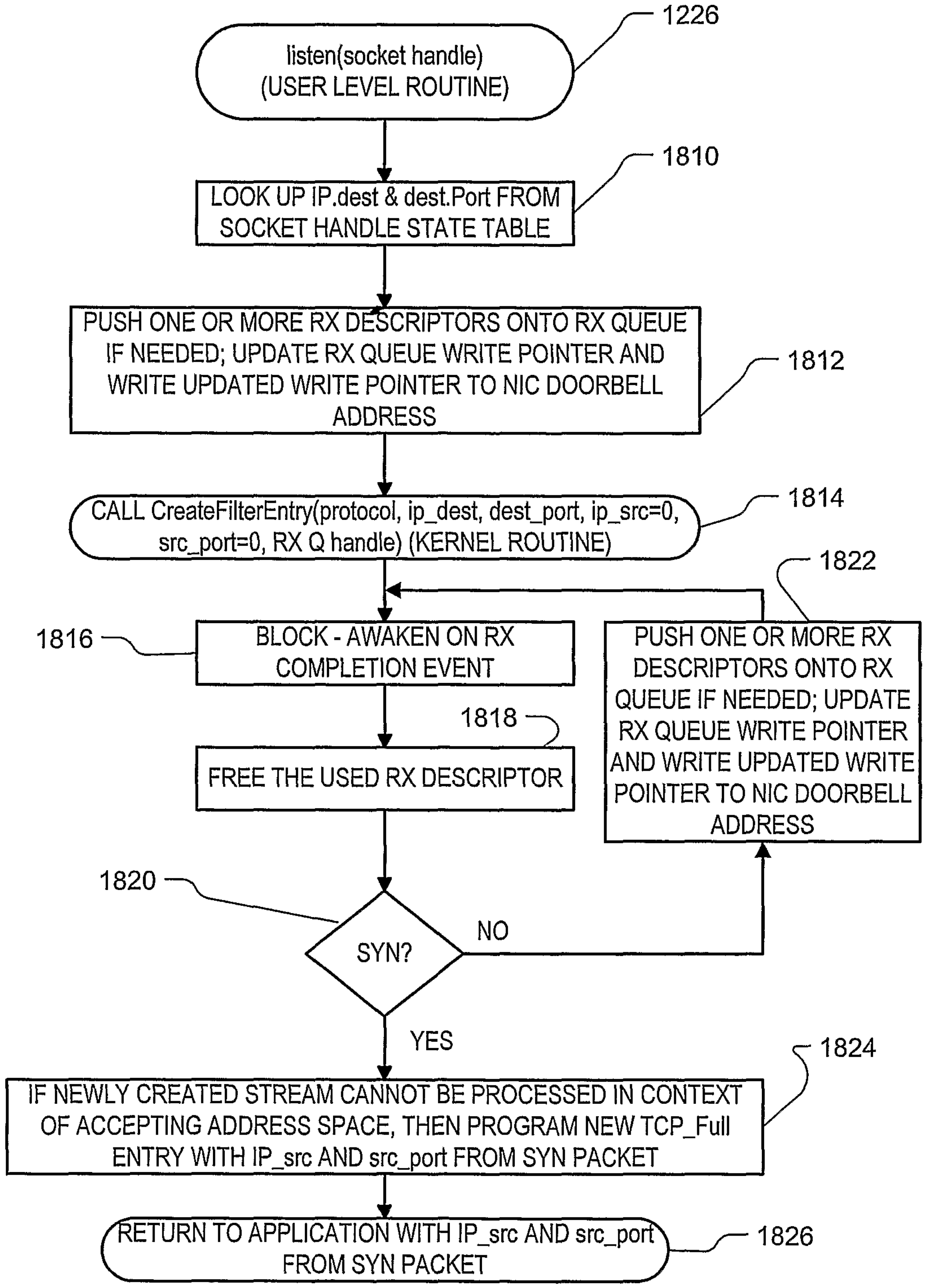
1226
listen(socket handle)
(USER LEVEL ROUTINE)
1810
LOOK UP IP.dest & dest.Port FROM SOCKET HANDLE STATE TABLE
PUSH ONE OR MORE RX DESCRIPTORS ONTO RX QUEUE IF NEEDED; UPDATE RX QUEUE WRITE POINTER AND WRITE UPDATED WRITE POINTER TO NIC DOORBELL ADDRESS
1812
CALL CreateFilterEntry(protocol, ip_dest, dest_port, ip_src=0, src_port=0, RX Q handle) (KERNEL ROUTINE)
1814
1822
1816
BLOCK - AWAKEN ON RX COMPLETION EVENT
PUSH ONE OR MORE RX DESCRIPTORS ONTO RX QUEUE IF NEEDED; UPDATE RX QUEUE WRITE POINTER AND WRITE UPDATED WRITE POINTER TO NIC DOORBELL ADDRESS
1818
FREE THE USED RX DESCRIPTOR
1820
SYN?
NO
YES
1824
IF NEWLY CREATED STREAM CANNOT BE PROCESSED IN CONTEXT OF ACCEPTING ADDRESS SPACE, THEN PROGRAM NEW TCP_Full ENTRY WITH IP_src AND src_port FROM SYN PACKET
RETURN TO APPLICATION WITH IP_src AND src_port FROM SYN PACKET
1826

FIG. 18

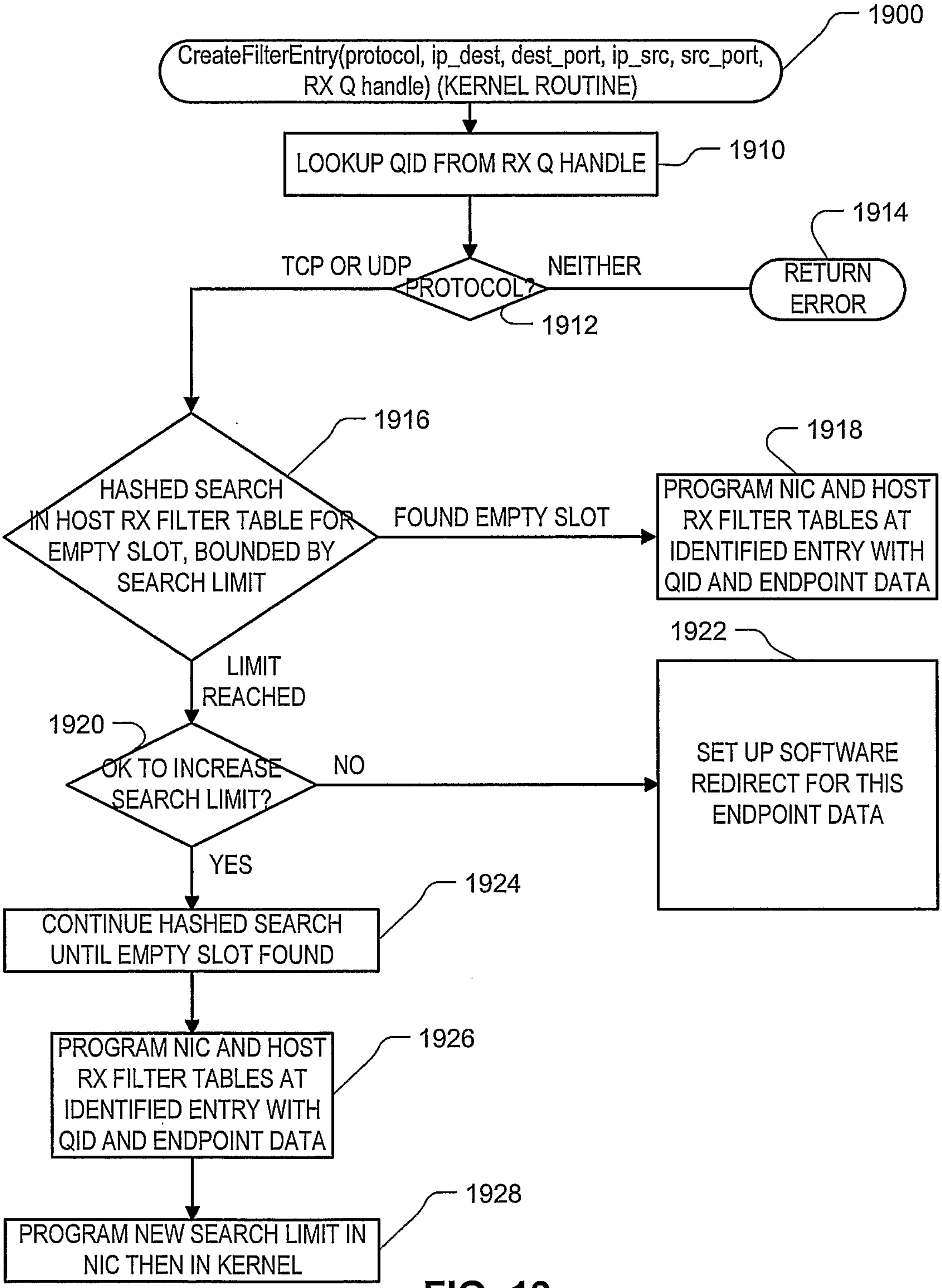
1900
CreateFilterEntry(protocol, ip_dest, dest_port, ip_src, src_port, RX Q handle) (KERNEL ROUTINE)
LOOKUP QID FROM RX Q HANDLE
1910
TCP OR UDP
PROTOCOL?
NEITHER
1912
1914
RETURN ERROR
1916
HASHED SEARCH IN HOST RX FILTER TABLE FOR EMPTY SLOT, BOUNDED BY SEARCH LIMIT
FOUND EMPTY SLOT
1918
PROGRAM NIC AND HOST RX FILTER TABLES AT IDENTIFIED ENTRY WITH QID AND ENDPOINT DATA
LIMIT REACHED
1920
OK TO INCREASE SEARCH LIMIT?
NO
1922
SET UP SOFTWARE REDIRECT FOR THIS ENDPOINT DATA
YES
CONTINUE HASHED SEARCH UNTIL EMPTY SLOT FOUND
1924
PROGRAM NIC AND HOST RX FILTER TABLES AT IDENTIFIED ENTRY WITH QID AND ENDPOINT DATA
1926
PROGRAM NEW SEARCH LIMIT IN NIC THEN IN KERNEL
1928

FIG. 19

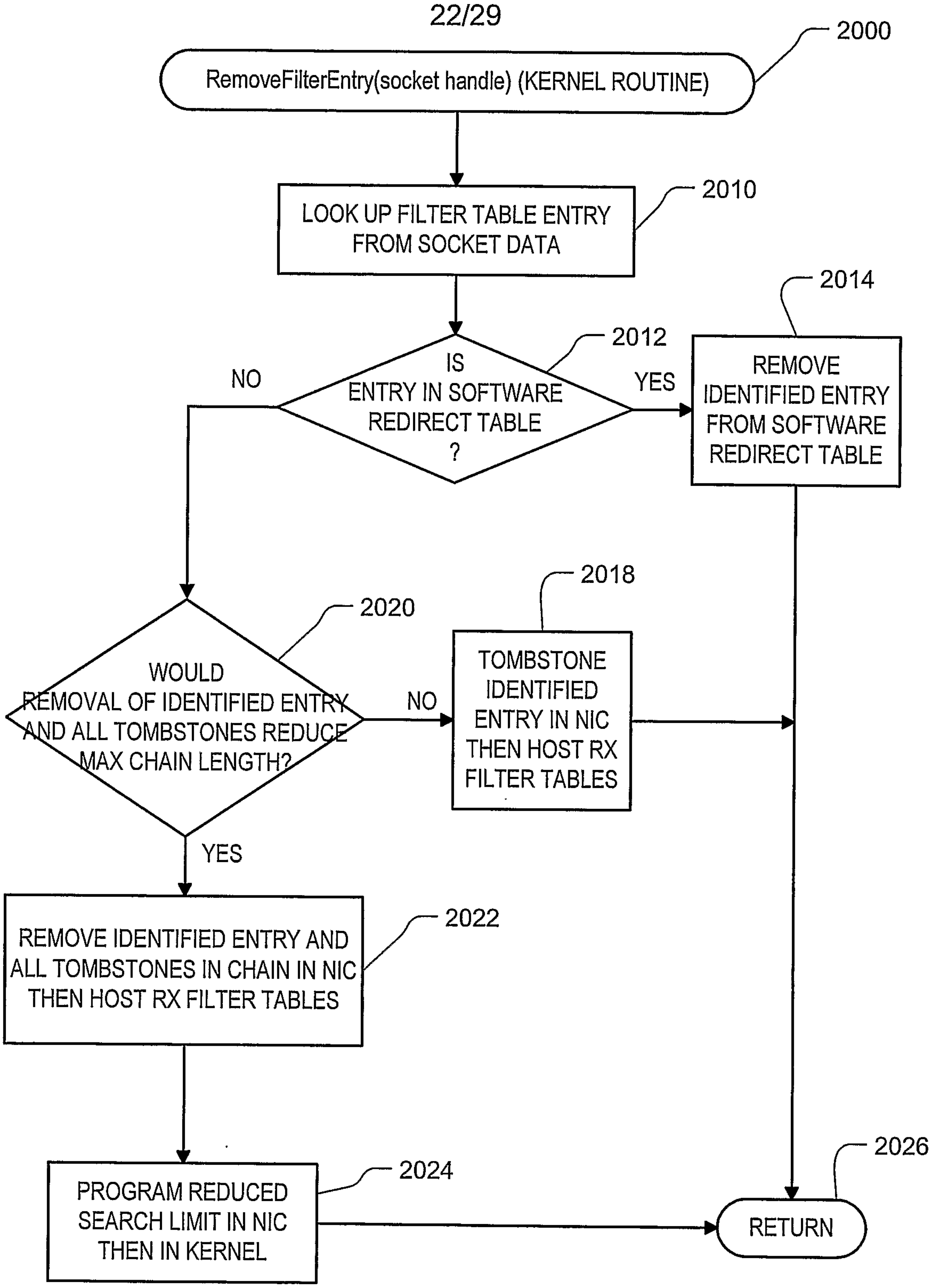
2000
RemoveFilterEntry(socket handle) (KERNEL ROUTINE)
2010
LOOK UP FILTER TABLE ENTRY FROM SOCKET DATA
2012
IS ENTRY IN SOFTWARE REDIRECT TABLE ?
NO
YES
2014
REMOVE IDENTIFIED ENTRY FROM SOFTWARE REDIRECT TABLE
2020
WOULD REMOVAL OF IDENTIFIED ENTRY AND ALL TOMBSTONES REDUCE MAX CHAIN LENGTH?
NO
2018
TOMBSTONE IDENTIFIED ENTRY IN NIC THEN HOST RX FILTER TABLES
YES
2022
REMOVE IDENTIFIED ENTRY AND ALL TOMBSTONES IN CHAIN IN NIC THEN HOST RX FILTER TABLES
2024
PROGRAM REDUCED SEARCH LIMIT IN NIC THEN IN KERNEL
2026
RETURN

FIG. 20

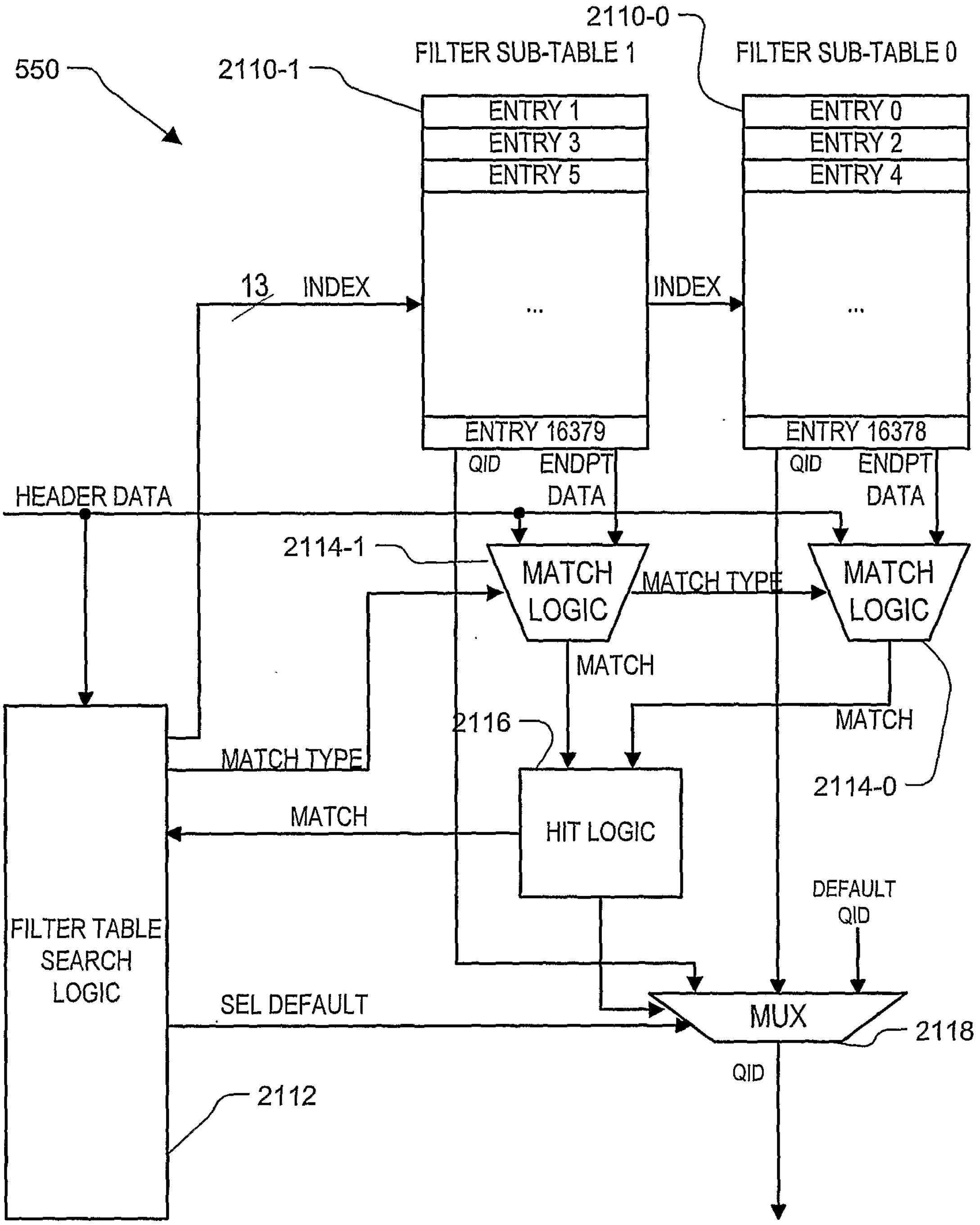
550
2110-1
FILTER SUB-TABLE 1
2110-0
FILTER SUB-TABLE 0
ENTRY 1
ENTRY 3
ENTRY 5
...
ENTRY 16379
ENTRY 0
ENTRY 2
ENTRY 4
...
ENTRY 16378
13
INDEX
INDEX
QID
ENDPT DATA
QID
ENDPT DATA
HEADER DATA
2114-1
MATCH LOGIC
MATCH TYPE
MATCH LOGIC
MATCH
MATCH
2116
2114-0
MATCH TYPE
MATCH
HIT LOGIC
DEFAULT QID
FILTER TABLE SEARCH LOGIC
SEL DEFAULT
MUX
2118
QID
2112

FIG. 21

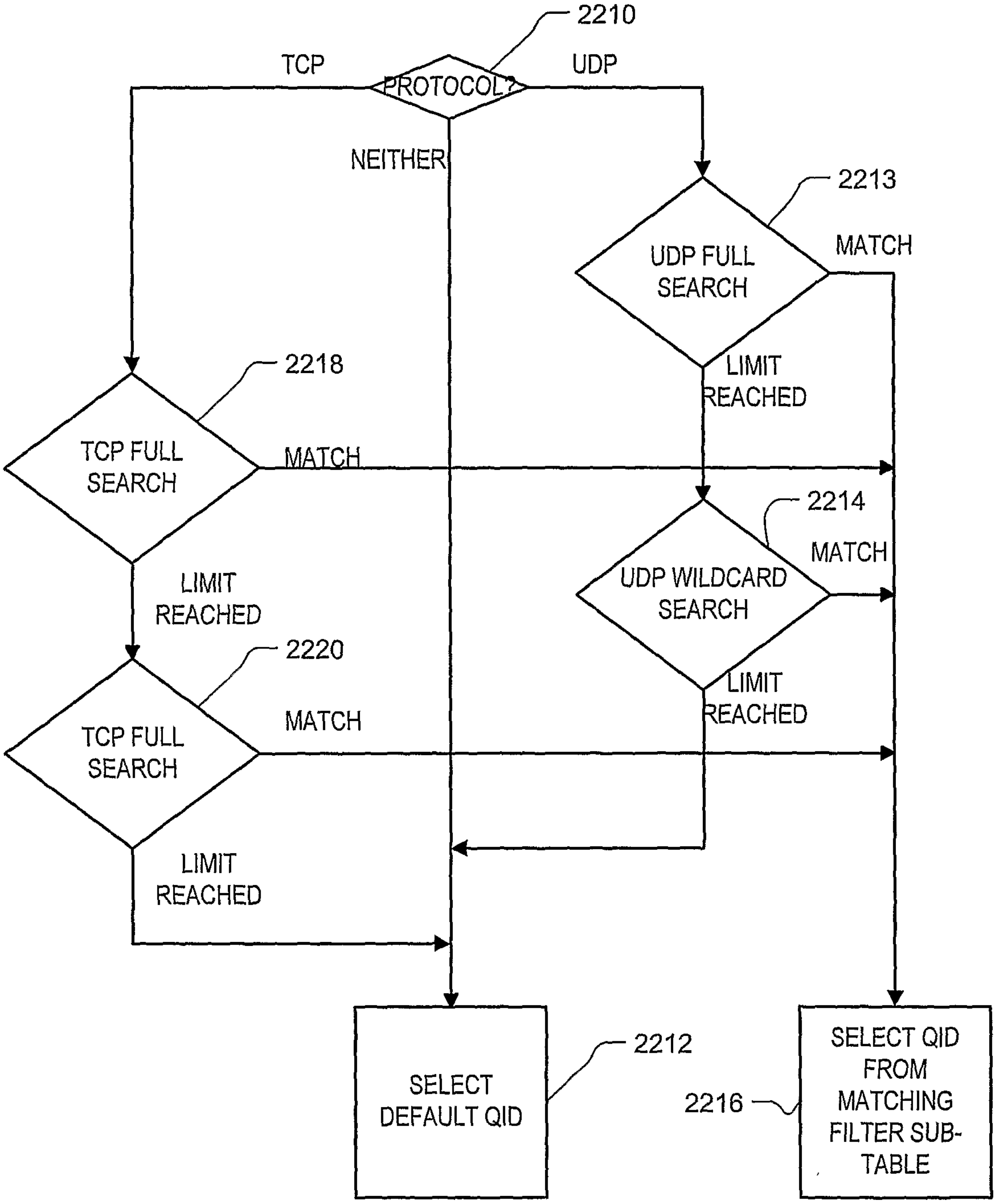
2210
TCP
PROTOCOL?
UDP
NEITHER
2213
UDP FULL SEARCH
MATCH
LIMIT REACHED
2218
TCP FULL SEARCH
MATCH
2214
UDP WILDCARD SEARCH
MATCH
LIMIT REACHED
LIMIT REACHED
2220
TCP FULL SEARCH
MATCH
LIMIT REACHED
LIMIT REACHED
SELECT DEFAULT QID
2212
2216
SELECT QID FROM MATCHING FILTER SUB-TABLE

FIG. 22

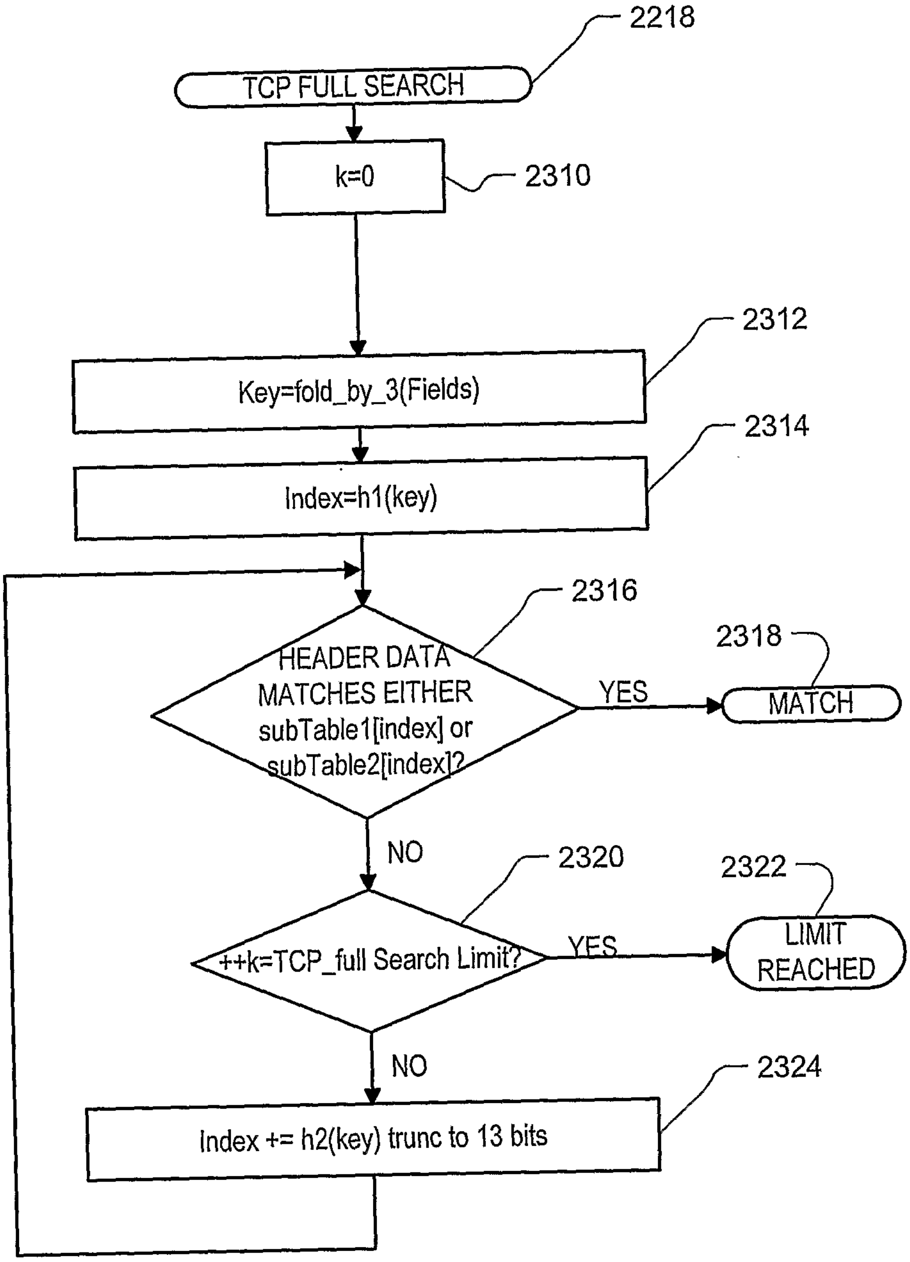
2218
TCP FULL SEARCH
k=0
2310
2312
Key=fold_by_3(Fields)
2314
Index=h1(key)
2316
HEADER DATA MATCHES EITHER subTable1[index] or subTable2[index]?
YES
2318
MATCH
NO
2320
++k=TCP_full Search Limit?
YES
2322
LIMIT REACHED
NO
2324
Index += h2(key) trunc to 13 bits

FIG. 23

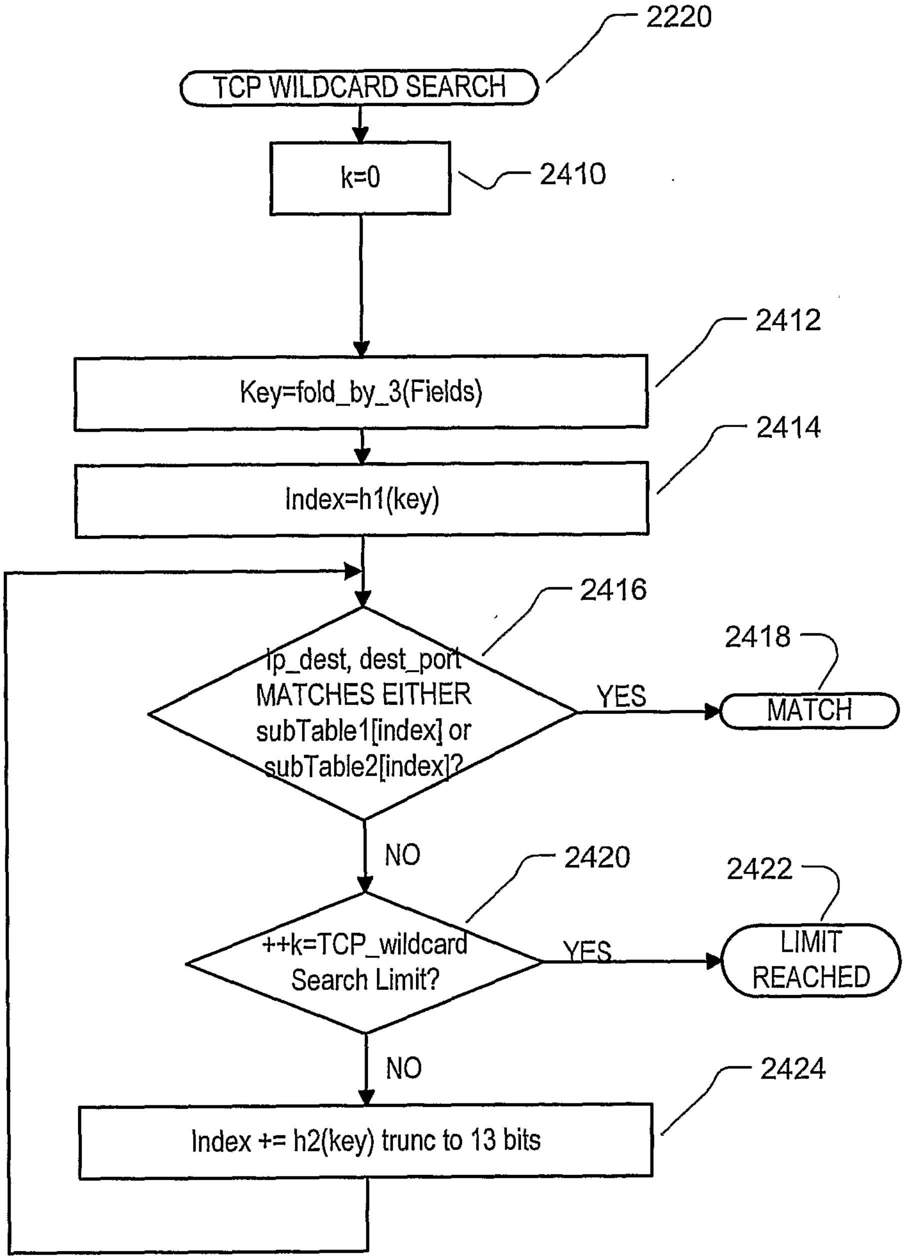
2220
TCP WILDCARD SEARCH
k=0
2410
2412
Key=fold_by_3(Fields)
2414
Index=h1(key)
2416
Ip_dest, dest_port MATCHES EITHER subTable1[index] or subTable2[index]?
YES
2418
MATCH
NO
2420
++k=TCP_wildcard Search Limit?
YES
2422
LIMIT REACHED
NO
2424
Index += h2(key) trunc to 13 bits

FIG. 24

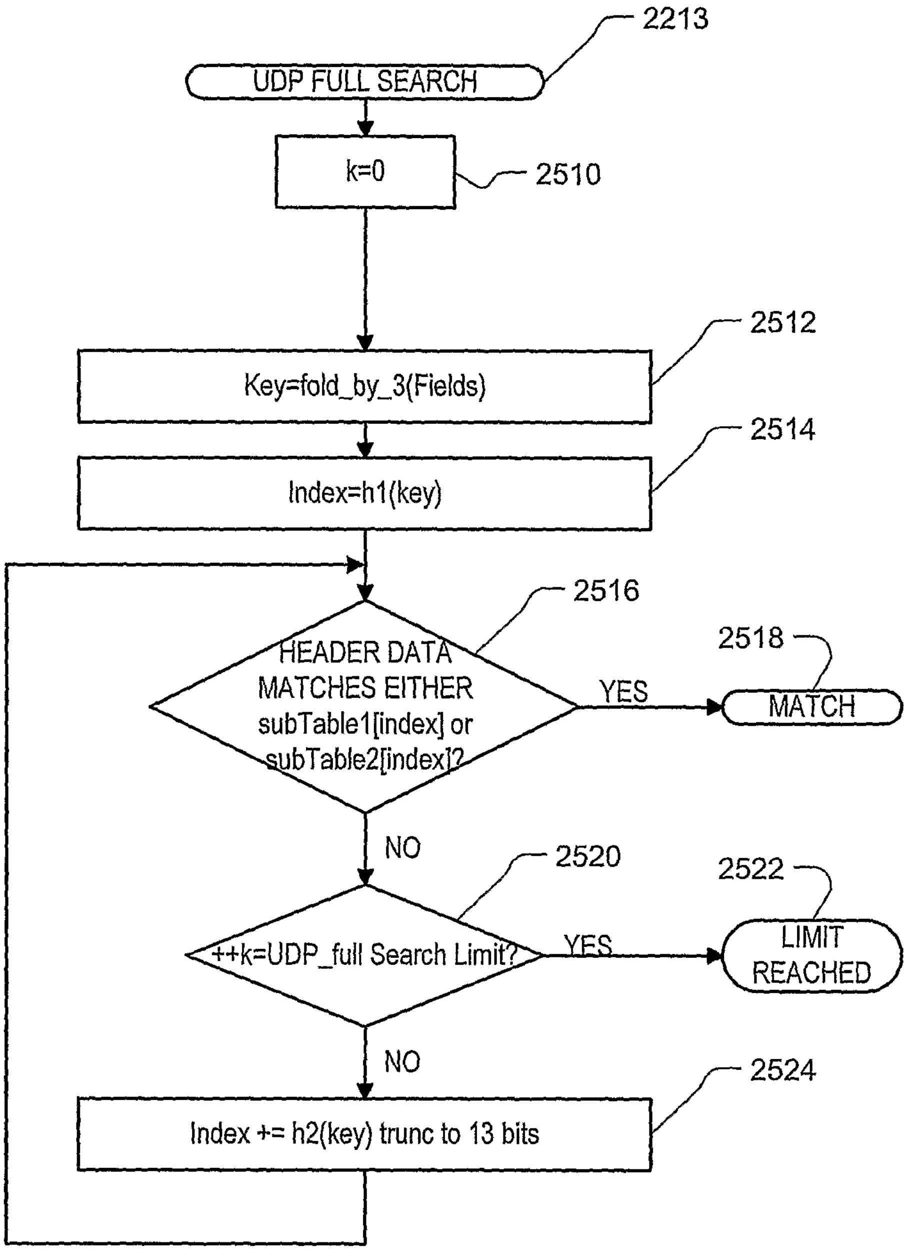
2213
UDP FULL SEARCH
k=0
2510
Key=fold_by_3(Fields)
2512
Index=h1(key)
2514
2516
HEADER DATA MATCHES EITHER subTable1[index] or subTable2[index]?
YES
2518
MATCH
NO
2520
2522
++k=UDP_full Search Limit?
YES
LIMIT REACHED
NO
2524
Index += h2(key) trunc to 13 bits

FIG. 25

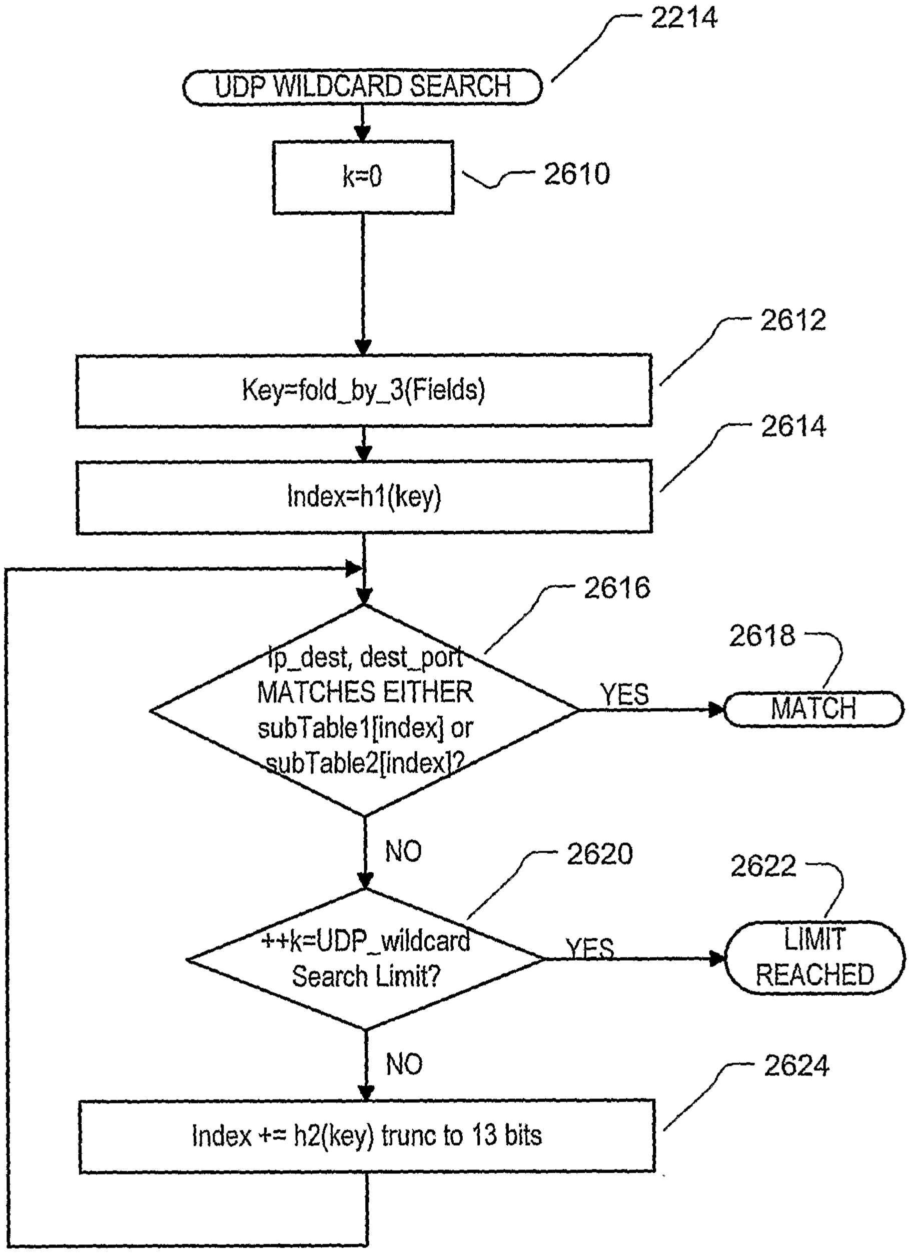
2214
UDP WILDCARD SEARCH
k=0
2610
2612
Key=fold_by_3(Fields)
2614
Index=h1(key)
2616
Ip_dest, dest_port MATCHES EITHER subTable1[index] or subTable2[index]?
YES
2618
MATCH
NO
2620
++k=UDP_wildcard Search Limit?
YES
2622
LIMIT REACHED
NO
2624
Index += h2(key) trunc to 13 bits

FIG. 26

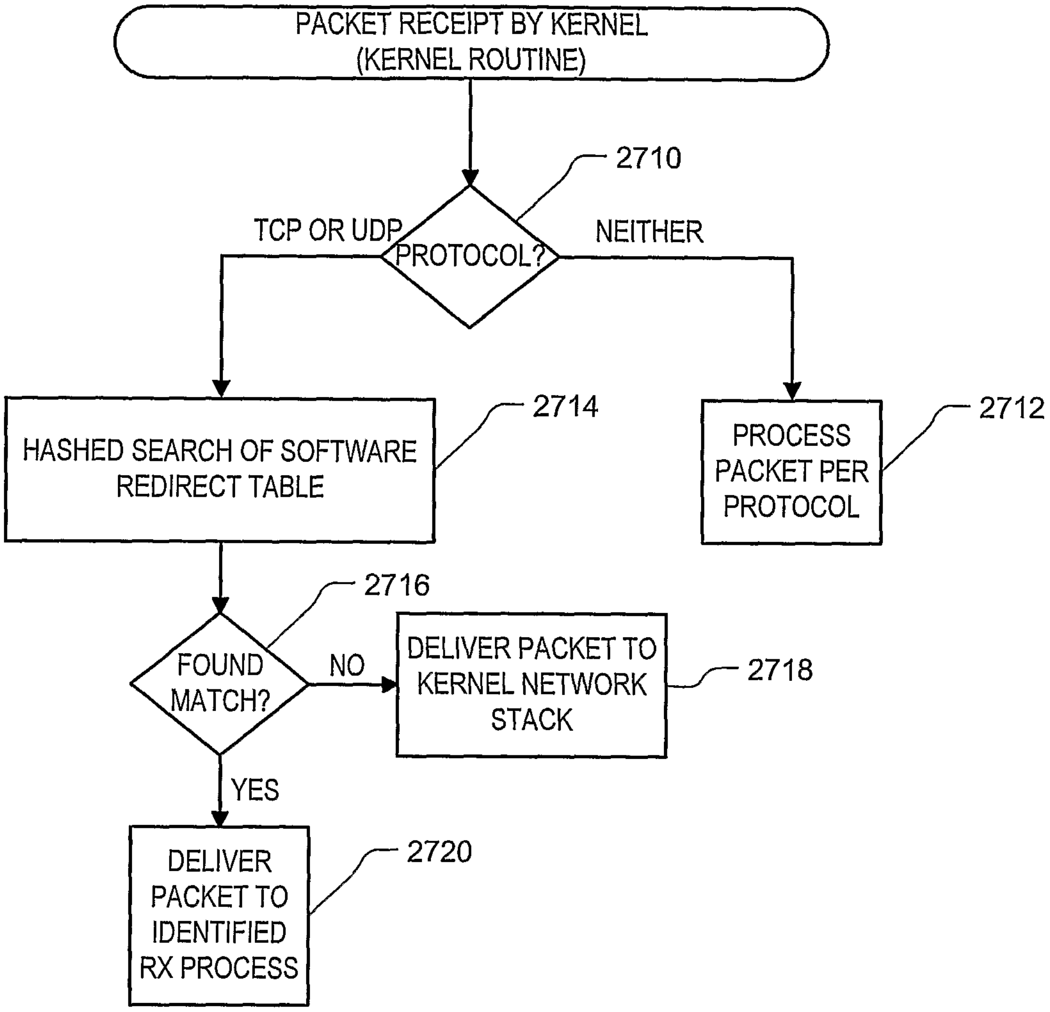
PACKET RECEIPT BY KERNEL
(KERNEL ROUTINE)
2710
TCP OR UDP
PROTOCOL?
NEITHER
HASHED SEARCH OF SOFTWARE REDIRECT TABLE
2714
PROCESS PACKET PER PROTOCOL
2712
2716
FOUND MATCH?
NO
DELIVER PACKET TO KERNEL NETWORK STACK
2718
YES
DELIVER PACKET TO IDENTIFIED RX PROCESS
2720

FIG. 27