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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
5 July 2007 (05.07.2007)

PCT

(10) International Publication Number
WO 2007/074343 A2

(51) International Patent Classification: Not classified

(21) International Application Number:
PCT/GB2006/004946

(22) International Filing Date:
28 December 2006 (28.12.2006)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

0526519.4	28 December 2005 (28.12.2005)	GB
0600417.0	10 January 2006 (10.01.2006)	GB
0602033.3	1 February 2006 (01.02.2006)	GB

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

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: **POPE, Steven** [GB/GB]; 25 Greville Road, Cambridge, Cambridges CB1 3QJ (GB). **RIDDOCH, David** [GB/GB]; 68 Tenison Road, Cambridge CB1 2DW (GB).

(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, GT, 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.

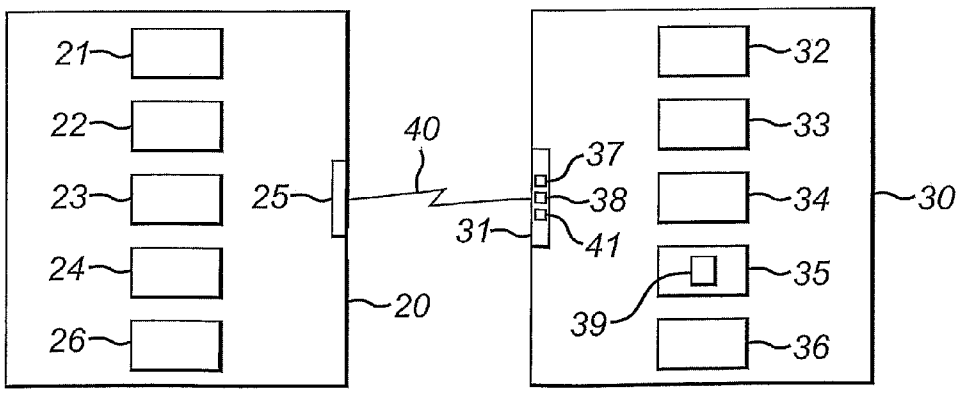
(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IS, IT, 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: PROCESSING RECEIVED DATA



(57) Abstract: A method for controlling the processing of data in a data processor, the data processor being connectable to a further device over a data link, the method comprising the steps of: receiving data at an element of the data processor; if a set interval has elapsed following the receipt of the data, determining whether processing of the received data in accordance with a data transfer protocol has begun, and, if it has not, triggering such processing of the received data by a protocol processing element; sensing conditions pertaining to the data link; and setting the interval in dependence on the sensed conditions.

WO 2007/074343 A2

COMBINED PCT DEC06

The present application relates to data processing systems and discloses three distinct inventive concepts which are described below in Sections A to C of the description.

Claims 1 to 25 relate to the description in Section A, claims 26 to 53 relate to the description in Section B, and claims 54 to 90 relate to the description in Section C.

In the appended drawings, figures 1 to 5 relate to the description in Section A, figures 6 to 12 relate to the description in Section B, and figures 13 to 18 relate to the description in Section C.

Embodiments of each of the inventions described herein may include any one or more of the features described in relation to the other inventions.

Where reference numerals are used in a Section of the description they refer only to the figures that relate to the description in that Section.

SECTION A

PROCESSING RECEIVED DATA

This invention relates to a method and apparatus for controlling the processing of data in a data processor.

In data networks it is important to enable efficient and reliable transfer of data between devices. Data can only reliably be transferred over a connection between two devices at the rate that a bottleneck in the connection can deal with. For example, a switch in a TCP/IP configured connection may be able to pass data at a speed of 10Mbps while other elements of the connection can pass data at, say, 100Mbps. The lowest data rate determines the maximum overall rate for the connection which, in this example, would be 10 Mbps. If data is transmitted between two devices at a higher speed, packets will be dropped and will subsequently need to be retransmitted. If more than one link is combined over a connector such as a switch, then the buffering capacity of the connector needs to be taken into account in determining maximum rates for the links, otherwise data loss could occur at the connector.

Figure 1 shows the architecture of a typical networked computing unit 1. Block 6 indicates the hardware domain of the computing unit. In the hardware domain the unit includes a processor 2 which is connected to a program store 4 and a working memory 5. The program store stores program code for execution by the processor 2 and could, for example, be a hard disc. The working memory could, for example, be a random access memory (RAM) chip. The processor is connected via a network interface card (NIC) 2 to a network 10. Although the NIC is conventionally termed a card, it need not be in the form of a card: it could for instance be in the form of an integrated circuit or it could be incorporated into the hardware that embodies the processor 2. In the software domain the computing unit implements an operating system 8 which supports an application 9. The operating system and the application are implemented by the execution by the processor 3 of program code such as that stored in the program store 4.

When the computing unit receives data over the network, that data may have to be passed to the application. Conventionally the data does not pass directly to the application. One reason for this is that it may be desired that the operating system polices interactions between the application and the hardware. As a result, the application may be required to interface with the hardware via the operating system. Another reason is that data may arrive from the network at any time, but the application cannot be assumed always to be receptive to data from the network. The application could, for example, be de-scheduled or could be engaged on another task when the data arrives. It is therefore necessary to provide an input mechanism (conventionally an input/output (I/O) mechanism) whereby the application can access received data.

Figure 2 shows an architecture employing a standard kernel TCP transport (TCPk). The operation of this architecture is as follows.

On packet reception from the network, interface hardware 101 (e.g. a NIC) transfers data into a pre-allocated data buffer (a) and invokes an interrupt handler in the operating system (OS) 100 by means of an interrupt line (step i). The interrupt handler manages the hardware interface. For example, it can indicate available buffers for receiving data by means of post() system calls, and it can pass the received packet (for example an Ethernet packet) and identify 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).

TCP receive-side processing then 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 the port is marked as holding valid data. Marking could be performed by means of a scheduler in the OS 100, and it could involve awakening a blocked process such that the process will then respond to the presence of the data.

In some circumstances the TCP receive processing may require other packets to be transmitted (step iv), for example where previously transmitted data needs to be retransmitted or where previously enqueued data can now be transmitted, perhaps because the TCP transmit window (discussed below) has increased. In these cases packets are enqueued with the OS Network Driver Interface Specification ("NDIS") driver 103 for transmission.

In order for an application to retrieve data from a data buffer it must invoke the OS Application Program Interface (API) 104 (step v), for example by means of a call such as recv(), select() or poll(). These calls enable the application to check whether data for that application has been received over the network. A recv() call initially causes copying of the data from the kernel buffer to the application's buffer. The copying enables the kernel of the OS to reuse the buffers which it has allocated for storing network data, and which have special attributes such as being DMA accessible. The copying can also mean 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.

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 (described in co-pending application WO2005/074611). 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.

The send process behaves similarly except that there is usually one path of execution. The application calls the operating system API 104 (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 103 for transmission.

If successful, the system call returns with an indication of the data scheduled (by the hardware 101) 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 acknowledgements from the device to which it is transmitting, or pending window updates (discussed below), and the device driver 103 may queue in software pending data transmission requests to the hardware 101.

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 100 provides all OS modules with time and scheduling services (typically driven by interrupts triggered by the hardware clock 102), which enable the TCP stack to implement timers on a per-connection basis. Such a hardware timer is generally required in a user-level architecture, since then data can be received at a NIC without any thread of an application being aware of that data. In addition to a hardware timer of this type, timers can be provided (typically in software) to ensure that protocol processing advances.

The setting of a software timer for ensuring the advance of protocol processing can impact on the efficiency of data transfer over the network. The timer can for example be instructed by a transport protocol library of the application to start counting when a new packet is delivered from the NIC to the transport protocol library. On expiry of a timeout, the timer causes an event to be delivered to an event queue in the kernel, for example by issuing an event from the NIC 101, the event identifying an event queue in the OS. At the same time as the event is delivered, an interrupt is scheduled to be delivered to the OS. According to the interrupt moderation rules in force, an interrupt is raised and the OS will start to execute device driver code to process events in the event queue. Thus, the software timer can be arranged to trigger protocol processing of data received at the data processor over the network, or to trigger protocol processing of data for transmission over the network. Such a timer preferably causes the kernel to be invoked relatively soon (for example within 250ms) after the receipt of data at the NIC.

Figure 3a illustrates a conventional synchronous I/O mechanism. An application 32 running on an OS 33 is supported by a socket 50 and a transport library 36. The transport library has a receive buffer 51 allocated to it. The buffer could be an area of memory in the memory 5 shown in figure 1. When data is received by the NIC 31 it writes that data to the buffer 51. When the application 32 wants to receive the data it issues a receive command (recv) to the transport library via the socket 50. In response, the transport library transmits to the application a message that includes the contents of the buffer. This involves copying the contents of the buffer into the message and storing the copied contents in a buffer 52 of the application. In response to obtaining this data, the application may cause messages to be issued, such as an acknowledgement to the device which transmitted the data. A problem with this I/O mechanism is that if the application fails to service the buffer often enough then the buffer 51 can become full, as a consequence of which no more data can be received.

Figure 3b illustrates a conventional asynchronous I/O mechanism. This mechanism avoids the overhead of copying the data by transferring ownership of buffers between the transport library and the application. Before data is to be received, the application 32 has a set of buffers (B_1-B_3) allocated to it. It then passes ownership of those buffers to the transport library 36 by transmitting to the transport library one or more post() commands that specify those buffers. When data is received it is written into those buffers. When the application wants to access the data it takes ownership of one or more of the buffers back from the transport library. This can be done using a gather() command that specifies the buffers whose ownership is to be taken back. The application can then access those buffers directly to read the data. A problem with this I/O arrangement is that the amount of data that is collected when the gather()command is executed could be very large, if a large amount of buffer space has been allocated to the transport library, and as a result the application may need considerable time to process that data.

Thus, with both of these mechanisms problems can arise if the application services the buffers at too fast or too slow a rate. If the buffers are serviced too infrequently then they can become full (in which case the reception of data must be suspended) or the amount of data that is returned to the application when the buffers are serviced could be very large. However, if the buffers are serviced too frequently then there will be excessive communication overheads between the application and the transport library as messages are sent between the two. One way of addressing these problems is to arrange for the transport library to set a timer that, on reaching a timeout, triggers the operating system to assist in processing any received data. This is particularly useful in the case of a user-level network architecture, where the transport library is normally driven by synchronous I/O calls from the application. The timer could, for example, run on the NIC. This mechanism can improve throughput but it has the disadvantage that it involves interrupts being set to activate the operating system to process the data. Processing interrupts involves overhead, and there may also only be a limited number of interrupts available in the system.

There is therefore a need for a mechanism which can increase the efficiency with which data can be protocol processed.

According to the present invention there is provided a method for controlling the processing of data in a data processor, the data processor being connectable to a further device over a data link, the method comprising the steps of: receiving data at an element of the data processor; if a set interval has elapsed following the receipt of the data, determining whether processing of the received data in accordance with a data transfer protocol has begun, and, if it has not, triggering such processing of the received data by a protocol processing element; sensing conditions pertaining to the data link; andsetting the interval in dependence on the sensed conditions.

The data processor may be connectable to the data link by means of an interface. A timer could reside on the interface, and the said interval could suitably be measured by the timer. The interface could suitably be implemented in hardware.

The step of determining may comprise determining whether processing of the received data by code at user level has begun.

The said protocol processing element is preferably an operating system.

The received data could comprise data received at the data processor over the data link, optionally by means of an asynchronous transmission.

The received data could also comprise data to be transmitted over the data link, optionally by means of an asynchronous transmission over the data link.

The step of triggering processing may comprise issuing an interrupt, preferably to the operating system.

The said element could be a transport library associated with an application running on the data processor.

The method could further comprise the step of in response to receiving the data, sending an instruction from the said element to the timer. The step of sending an instruction to the timer could comprise triggering the timer directly from the said element via a memory mapping onto the said interface.

The step of setting the interval may comprise reducing the interval if the sensed conditions are indicative of an increase in data rate over the data link.

Buffer space could be allocated to the data link for storing data received at the data processor over the data link, and the protocol could be a protocol that employs a receive window in accordance with which a transmitter of data according to the protocol will transmit no further traffic data once the amount of data defined by the receive window has been transmitted and is unacknowledged by the receiver, and the step of setting the interval could comprise reducing the interval in response to sensing that the size of the buffer space allocated to the data link is greater than the

size of the receive window. The method could also comprise the step of varying the size of the buffer space allocated to the data link in response to a request from a consumer of the traffic data. The consumer could be an application running on the data processor.

The step of sensing conditions could comprise sensing the presence in a transmit buffer of data to be transmitted over the data link.

The step of setting the interval could comprise reducing the interval in response to sensing in a transmit buffer data to be transmitted over the data link.

The step of setting the interval could comprise reducing the interval in response to sensing that a congestion mode of the protocol is in operation over the data link.

The protocol could suitably be TCP.

According to a second aspect of the present invention there is provided apparatus for controlling the processing of data in a data processor, the data processor being connectable to a further device over a data link, the apparatus comprising: an element arranged to receive data; and a control entity arranged to, if a set interval has elapsed following the receipt of data, determine whether processing of the received data in accordance with a data transfer protocol has begun, and, if it has not, trigger such processing of the received data by a protocol processing element; wherein the control entity is further arranged to sense conditions pertaining to the data link and set the interval in depend dependence on the sensed conditions.

According to a third aspect of the present invention there is provided a control entity for use with a data processor, the data processor being connectable to a further device over a data link, and comprising an element arranged to receive data, the control entity being arranged to: if a set interval has elapsed following the receipt of data by the said element, determine whether processing of the received data in accordance with a data transfer protocol has begun, and, if it has not, trigger such

processing of the received data by a protocol processing element; and sense conditions pertaining to the data link and set the interval in dependence on the sensed conditions.

The present invention will now be described by way of example with reference to the accompanying drawings.

In the drawings:

figure 1 shows the architecture of a computing system;

figure 2 is a schematic representation of a prior art data transfer architecture;

figure 3a shows a data processing system arranged for synchronous data transfer;

figure 3b shows a data processing system arranged for asynchronous data transfer;

figure 4 illustrates a typical flow of data transmitted in accordance with the TCP protocol; and

figure 5 shows schematically a pair of data processing devices communicating over a data link.

In the present system the entity that is to process received data, which could for example be an application or part of the operating system, is capable of varying the minimum threshold frequency with which it services the receive buffer(s) in dependence on conditions that may be taken as indicative of changes in the rate at which data is being received. In this way the entity can service the buffers more quickly when data is expected to be received in greater volume, and more slowly when little data is expected to be received.

TCP is a common example of a transport protocol, and a discussion of the TCP windows technique will now be given. According to the TCP protocol, each time a certain amount of data is transmitted from one device to another over a network, the transmitting device must await an acknowledgement from the receiving device before sending further data. A TCP window is the amount of unacknowledged data a

sender can send on a particular connection before it must await an acknowledgement from the receiver. To give a simple example, the window could be specified as 10 octets (or bytes). Thus, once 10 bytes have been sent by a sender, no further transmission will be permitted until the sender receives an acknowledgement from the receiver that at least some of those 10 bytes have been received. If the acknowledgement indicates, for example, that all 10 bytes have been safely received then the sender is able to transmit a further 10 bytes. If the acknowledgement indicates that only the first 2 bytes have been received then the sender may transmit 2 further bytes and must then await further acknowledgement.

The TCP window mechanism provides congestion control for a network. The window is generally fixed for a connection according to the properties of the bottleneck in the connection at a given time to ensure that data cannot be sent at a greater rate than the bottleneck can handle without losing data.

A receiver in a network advertises a receive window to devices with which it has network connections, so that the devices can configure themselves accordingly. The sender's send window will be set equal to the receiver's receive window for a particular connection. As an example, the size of the receive window (in bytes) could simply be the size of the buffer space on a receiving device's network interface minus the amount of data currently stored in the buffer.

Window announcements from a receiver can include an acknowledgement of previously sent data. This is an efficient arrangement since it can enable two items of information to be sent in one message.

It is commonly desirable for transmission and reception over a network to be in a steady-state condition whereby packets are periodically sent from a sender and acknowledgements (which may be accompanied by window announcements) are periodically sent from a receiver. This is an efficient mode of data transfer since data packets can be continually in transit over the network.

The window size for a particular network connection can be varied in response to changes within the connection; for example, it could be increased if more buffer space becomes available at the receiver. Changes in window size are indicated to a sender by means of window announcements.

The flow rate of data according to the TCP protocol may be ramped, as illustrated in figure 4. In general, the receive window will initially be defined to be relatively small. If a transmitting application has a large amount of data to be sent, evidenced by the fact that the receiver is receiving data at the maximum rate permitted by the window/acknowledgement technique, in other words if the connection is "busy", then the receiver may wish to permit a greater rate of transmission over the connection by increasing its receive window. Typically, the window will be increased in a stepwise manner for as long as the transmitter continues to transmit at the maximum rate and until packet drop is detected. When packet drop is detected, the window will be decreased (typically halved) in order to avoid further loss of data, and this will be advertised in the next window announcement sent by the receiver. In the exemplary flow shown in figure 4, the shaded regions are where flow rate is increasing and thus fast window increase is desirable so that the transfer rate can be increased accordingly.

It was noted above that the TCP window may be halved suddenly when packet drop is detected. Packet drop may be detected as an absence of an acknowledgement from the receiving device at the transmitting device. Typically, after a predetermined time interval, if no acknowledgement is received at the transmitter then the transmitter will commence retransmission of the packets which have not yet been acknowledged. The absence of an acknowledgement may be due to data actually having been lost over the network connection, or it may be due to a delay in the connection such that the data has not been received at the receiver within the predetermined time interval, causing the transmitter to timeout and commence retransmission.

Considering transmission from a transmitting device, if a send queue associated with a transmitting application is not permitted adequate CPU time then protocol processing of data waiting in the queue cannot occur sufficiently fast for the send queue to be emptied and the data to be sent to the network interface for transmission over the network. If data is awaiting processing and no data is ready to be sent then it is possible that the connection may go idle. Additionally, send queues may become full such that they cannot accept new data. Similarly, for a receiving device it is important that incoming data can be processed at the rate at which it is being received. Regular and timely processing of buffered incoming data is therefore important. In general, timely protocol processing of waiting data is desirable to achieve efficient data transfer over a data link. This is especially true for asynchronous transmission, since large chunks of data tend to be delivered, whereas in synchronous transmission the chunks tend to be smaller.

Additionally, the overall transfer rate of data can be improved by enabling an increasing window to reach its maximum size as quickly as possible. This can be achieved by dedicating more CPU time to flows which are accelerating than to flows which are constant or increasing.

In the discussion of figure 2 above, it was noted that timers may be used to ensure that queues of data are permitted regular attention. In respect of issuing interrupts following timeouts, the inventors of the present invention have appreciated that a long timeout can reduce CPU overhead since it can reduce the number of spurious interrupts. However, a long timeout can be inappropriate in asynchronous transmission mode since it can cause protocol processing to proceed at too slow a rate.

The provision of timeouts for ensuring protocol processing is particularly important in a system incorporating a user-level protocol stack, such as that described in co-pending PCT application number PCT/GB05/001525. In such a system protocol processing (such as TCP processing) generally takes place within the context of an application thread. If the transport library is not given sufficient CPU time (for

example because the thread is performing another operation or is de-scheduled) then by means of a timeout and an interrupt a component of the OS can be engaged to carry out the required protocol processing on behalf of the user-level stack. The question of how to determine an optimal length for a timeout is thus relevant in this environment. It is therefore proposed by the inventors to provide a means of adjusting timeouts dynamically according at least to the behaviour of an application to or from which data is being sent over the network.

In one instance, it may be desirable to provide a timer at a transmitting device, and to cause the timer to start each time data is delivered to an element of the device at which protocol processing must be performed to enable subsequent transmission of the data over the network. When the timer times out, the operating system or user-level code could be triggered to perform protocol processing of the delivered data. This could ensure that data could flow at an acceptable rate from the transmitting device over the network.

In another instance, it may be desirable to provide a timer at a receiving device, and to cause the timer to start each time data is delivered from the network to an element of the device at which protocol processing must be performed for the data to be made available to a consumer, such as an application running on the receiving device. In this instance, when the timer times out, the operating system or user-level code could be triggered to perform protocol processing of the received data. This could ensure that incoming data could be processed at the rate at which it is being received over the network.

The element of the transmitting device could be a transport library or any buffer in which data is stored pending processing to prepare the data for transmission by a network protocol. The element of the receiving device could be a transport library or any buffer in which data is stored following receipt over a network pending processing to enable the data to be read by a consumer.

When the rate of data transmission over a network is ramping up it is desirable to ramp up the size of the TCP window as fast as possible in accordance with the TCP algorithms to keep pace with the amount of data that needs to be sent so that data does not back up at the transmitter. This can be achieved by reducing the timeout on the timer in the transmitting device so that data can be sent at the maximum rate (as defined by the current size of the window), thus triggering window announcements as frequently as possible. More generally, it is desirable to respond as quickly as possible to changes in flow rate.

It will be understood that embodiments of the invention can successfully be applied to both synchronous and asynchronous arrangements.

In general, embodiments of the invention support an algorithm for determining an appropriate timeout length for the present conditions in the connection. The conditions could include the presence or recent occurrence of congestion, the amount of data which an application wishes to send or receive, the amount of other activity in the network, or any other aspect of the network which is relevant to the flow of data over a connection.

Depending on the detected conditions, it could be appropriate to modify the length of the timeout at the receiver or at the transmitter, or at both. The timer could suitably be implemented in hardware, for example at a network interface, or it could be implemented in software. The timer could suitably be arranged to receive instructions to start timing from an element of a data processor at which protocol processing is required. The element could suitably be a transport library.

In one embodiment, a mechanism can be implemented whereby events issued by a transport library can be batched together such that only one interrupt is raised for multiple events. This can avoid a high overhead on the CPU.

Figure 5 illustrates a pair of data processing devices communicating over a data link. In the present example device 20 is transmitting data and the other device 30 is

receiving that data. However, the devices could communicate bi-directionally. The devices could be in direct communication or could communicate indirectly over a network (e.g. via one or more routers).

In this example the protocol in use by the devices is TCP (transmission control protocol) over Ethernet, but the present invention is suitable for use with other protocols.

Device 20 comprises a data store 21 that stores the data that is to be transmitted. Under the control of an application 22 supported by an operating system 23 and running on a processing unit 24 data from the data store 21 is passed to a NIC 25 for transmission to device 30. Once data has been passed to the NIC for transmission it waits in a send queue. When the NIC is able to transmit more data it takes data from the send queue, encapsulates it in accordance with the protocols that are in use, and transmits it. The send queue may be embodied by a dedicated buffer for outgoing data, or it may be embodied by the storing by the NIC of the addresses of other buffers from which it is to retrieve data for transmission. The NIC 25 performs protocol processing either alone or in conjunction with a transport library 26 running on the processing unit 24.

Device 30 comprises a NIC 31 that is connected to a data link 40 by which it can receive data from NIC 25. The data is destined for an application 32 that is supported by an operating system 33 and runs on a processing unit 34. The device further includes a data store 35 for storing received data, in which one or more buffers can be defined, and a transport library 36 which comprises a set of processes that run on the processing unit and can have state associated with them for performing certain networking functions and for assisting in interfacing between the NIC and the application. Co-pending applications PCT/IB05/002639 and WO2005/086448 describe examples of such a transport library having such functions. The NIC 31 performs protocol processing either alone or in conjunction with the transport library 36.

The protocol processing functions that are carried out by the transmitter and the receiver include implementing flow and congestion control. As is well known, TCP provides a number of flow and congestion control functions. One function involves the TCP receive window described above.

When the application 32 is to receive data it requests from the operating system 33 that buffer space is allocated to it for storing that data in the period between it having been received at the NIC 25 and it being accepted by the application for processing. The buffer can be allocated by the operating system 33 or by the transport library 36. Typically the application will maintain a certain amount of receive buffer space for normal usage. However, a well-designed application will, if it requests a substantial amount of data over the network or learns that it is about to be sent a substantial amount of data over the network, request additional receive buffer space to accommodate that data.

In this context there are numerous situations that are indicative of a potential or ongoing change in the volume data flow over the link between the devices 20 and 30. These include, but are not limited to:

1. A sudden increase or decrease in the receive buffer allocated to a particular application, or in total to all the applications running on the receiving device. This could indicate a change in flow because, as explained above, a well-designed application will be expected to modify its allocated buffer space in anticipation of receiving more or less data. It should be noted that the amount of allocated buffer space therefore provides an indirect means of signalling between the application layer and lower layers of the protocol stack. Such an increase could be defined by an increase or decrease in allocated receive buffer space of more than a predetermined amount or more than a predetermined proportion over a predetermined time. Alternatively, such an increase could be indicated by the relative sizes of the TCP receive window for a link and the posted receive buffer size for the application to which that link relates. For example, such an increase could be

deemed to have taken place when the posted receive buffer size exceeds the TCP receive window.

2. That the send queue of the transmitter for a particular link is not empty, or has remained not empty for greater than a predetermined period of time. This may be indicative of congestion on the data link between the transmitter and the receiver. However, it may also be indicative of data having been received at the receiver and not having been protocol processed so that an acknowledgement for that data can be sent to the transmitter.

3. That TCP congestion mode is operating: i.e. that the TCP receive window is reducing (backing off), or subsequently increasing.

Analogous situations will arise under protocols other than TCP and can be sensed in an analogous way. For example, any distributed transport protocol such as SCTP has algorithms similar to the TCP receive window algorithm. The sliding window mechanism is commonly used. Other protocols (often used in conjunction with hardware) such as Infiniband and some ATM link-layer schemes expose credits to the system. The thresholds or rate of change of such credits can be used as indicators of a change in flow rate.

In the present system the NIC 31 implements one or more timers such as timer 37. These timers consist of a counter that is decremented periodically when clocked by a clock 38. The clock could run on the NIC, or the NIC itself could be clocked from the remainder of the data processing device to which it is attached. An initial value is loaded into the timer. When the timer reaches zero the NIC performs an action stored in its state memory 41 in relation to that timer.
One use of such timers is for allowing the NIC 31 to signal the operating system 36 to process data that is waiting in a receive buffer. This will now be described in more detail.

When the device 30 is being configured for reception of data by an application running on it, the application will request one or more areas in memory 35 to be allocated to it for use as receive buffers 39. It then signals the NIC to inform it that the link is to be established and to inform it which receiver buffer(s) are to be used with that link. The buffer allocation and the signalling may be done via the operating system 33 and/or the transport library 36.

When data is received over the network by NIC 31, the NIC identifies which data link that data relates to. This may be indicated by the port number on which the data was received and/or by the source address of the data. The NIC then stores the data in a receive buffer corresponding to that data link.

Since the traffic data components of the received data cannot be identified until protocol processing has been done, the received traffic data cannot be passed to the receiving application until protocol processing has been completed. It is preferred that at least some of the protocol processing that is to be performed on the received data is performed downstream of the NIC, for example by the operating system or the transport library of the receiving device 30. This has been found to permit many potential performance enhancements. The protocol processing that is performed downstream of the NIC can conveniently include the generation of or the triggering of acknowledgement messages for received data. A mechanism whereby this can be done will be described below.

When data has been received by the NIC and written to a receive buffer the operating system or the transport library is to perform protocol processing on that data. Either of these may be instructed by the receiving application to perform the protocol processing. Alternatively either of them may be triggered by a timer that runs separately from the application. Having such a timer running separately from the application is advantageous because it allows the timer to continue to run even if the application becomes de-scheduled. It is preferred that the timer is a timer such as timer 37 running on the NIC. Such a timer can be configured (e.g. by the application or the transport library) when the data link is set up. Conveniently, the

transport library signals the NIC during establishment of the link to have the NIC allocate one of its timers for use with the link. To do this the transport library informs the NIC of the initial value of the timer and the function that is to be performed when the timer reaches zero. The initial value is stored by the NIC in memory 41, and the NIC is arranged to automatically reset the timer to the value stored in that field when it has reached zero. The function may be to instruct the operating system to perform protocol processing for any data received for that link and for which protocol processing has not yet been performed. The function is also stored in association with the respective timer in memory 41 so that it can be recalled and actioned by the NIC when the timer reaches zero. Once the timer has been set up it will continually decrement, and then trigger (for example) the operating system to perform protocol processing each time it reaches zero.

In the present system at least one of the entities implements an algorithm that manages the interval over which the timer 37 operates. The interval is altered in dependence on sensed conditions, particularly communication conditions in the transmitter and/or the receiver for the link with which the timer is associated. Most preferably, the interval over which the timer operates is reduced when the entity in question senses conditions that are indicative of an increase in data rate. Most preferably, the interval over which the timer operates is increased when the entity in question senses conditions that are indicative of a decrease in data rate. Some specific examples are as follows.

1. The interval can be reduced in response to an increase or decrease in the receive buffer allocated to a particular application, or in total to all the applications running on the receiving device. This may be detected in the conditions described above, and most preferably when the posted receive buffer size for the link in question exceeds the size of the TCP receive window for that link. In the opposite conditions the interval can be increased.
2. The interval can be reduced if the transmitter's send queue for the link in question is not empty, or has remained non-empty for a certain period of time. In the opposite condition the interval can be increased.

3. The interval can be increased when congestion mode is operating and/or the transmission scheme is backing off. In the opposite conditions the interval can be increased.

Altering the timer interval can most conveniently be done by altering the initial value for the timer that is stored in memory 41. The new value will be applied to the timer when it is next reset. However, it could also be advantageous to alter the current value of the timer's counter. The latter method is useful where the interval over which the timer operates is being reduced from a high value to a significantly lower value.

The entity that applies the algorithm to cause the timer interval to be altered could be the NIC, the operating system, the transport library or the application, or another entity. It is preferred that it is applied by the transport library since it can communicate directly with the NIC.

The timers on the NIC run outside the scope of scheduling by the main CPU 34 on the device 30, and outside the scope of scheduling by the operating system of the device 30. As a result they can be expected to run continually independently of the load or processing demands on the device 30.

The event that is triggered by the expiry of the timer could be a compound event in which the operating system is triggered to perform a number of functions. For example, the entity that executes the algorithm could be arranged to detect that multiple links are using timers that are set to the same value, and in response to that to group those links so as to use a single timer that triggers protocol processing for all those links using a single interrupt. This saves on interrupts.

The entity that is to perform protocol processing on being triggered by the timer is preferably arranged so that when it is triggered it checks whether protocol processing is already being performed on received data for the respective link, and if it is to not perform protocol processing in response to the trigger. It may conveniently detect

whether protocol processing is being performed by the presence of a flag that any device performing protocol processing on the system is arranged to sent and unset.

The applicant hereby discloses in isolation each individual feature described herein and any combination of two or more such features, to the extent that such features or combinations are capable of being carried out based on the present specification as a whole in the light of the common general knowledge of a person skilled in the art, irrespective of whether such features or combinations of features solve any problems disclosed herein, and without limitation to the scope of the claims. The applicant indicates that aspects of the present invention may consist of any such individual feature or combination of features. In view of the foregoing description it will be evident to a person skilled in the art that various modifications may be made within the scope of the invention.

SECTION B

DATA BUFFERING

This invention relates to the buffering of data, for example in the processing of data units in a device bridging between two data protocols.

Figure 6 shows in outline the logical and physical architecture of a bridge 1 for bridging between data links 2 and 3. In this example link 2 carries data according to the Fibrechannel protocol and link 3 carries data according to the ISCSI (Internet Small Computer Serial Interface) protocol over the Ethernet protocol (known as ISCSI-over-Ethernet). The bridge comprises a Fibrechannel hardware interface 4, an Ethernet hardware interface 5 and a data processing section 6. The interfaces link the data processing section to the respective data links 2 and 3. The data processing section implements a series of logical protocol layers: a Fibrechannel driver 7, a Fibrechannel stack 8, a bridge/buffer cache 9, an ISCSI stack 10, a TCP (transmission control protocol) stack 11 and an Ethernet driver 12. These layers convert packets that have been received in accordance with one of the protocols into packets for transmission according to the other of the protocols, and buffer the packets as necessary to accommodate flow control over the links.

Figure 7 shows the physical architecture of the data processing section 6. The data processing section 6 comprises a data bus 13, such as a PCI (personal computer interface) bus. Connected to the data bus 13 are the Ethernet hardware interface 5, the Fibrechannel hardware interface 4 and the memory bus 14. Connected to the memory bus 14 are a memory unit 15, such as a RAM (random access memory) chip, and a CPU (central processing unit) 16 which has an integral cache 17.

The example of an ISCSI-over-Ethernet packet being received and translated to Fibrechannel will be discussed, in order to explain problems of the prior art. The structure of the Ethernet packet is shown in figure 8. The packet 30 comprises an Ethernet header 31, a TCP header 32, an ISCSI header 33 and ISCSI traffic data 34.

Arrows 20 to 22 in figure 7 illustrate the conventional manner of processing an incoming Ethernet packet in this system. The Ethernet packet is received by Ethernet interface 5 and passed over the PCI and memory buses 12, 13 to memory 14 (step 20), where it is stored until it can be processed by the CPU 15. When the CPU is ready to process the Ethernet packet it is passed over the memory bus to the cache 16 of the CPU. (Step 21). The CPU processes the packet to perform protocol processing and re-encapsulate the data for transmission over Fibrechannel. The Fibrechannel packet is then passed over the memory bus and the PCI bus to the Fibrechannel interface 4 (step 22), from which it is transmitted. It will be appreciated that this process involves passing the entire Ethernet packet three times over the memory bus 13. These bus traversals slow down the bridging process.

It would be possible to pass the Ethernet packet directly from the Ethernet interface 5 to the CPU, without it first being stored in memory. However, this would require the CPU to signal the Ethernet hardware to tell it to pass the packet, or alternatively for the CPU and the Ethernet hardware to be synchronised, which would be inefficient and could also lead to poor cache performance. In any event, this is not readily possible in current server chipsets.

An alternative process is illustrated in figure 9. Figure 9 is analogous to figure 7 but shows different process steps. In step 23 the received Ethernet packet is passed from the Ethernet hardware to the memory 14. When the CPU is ready to process the packet only the header data is passed to the CPU. (Step 24). The CPU process the header data, forms a Fibrechannel header and transmits the Fibrechannel header to the Fibrechannel interface. (Step 25). Then the traffic data 34 is passed to the Fibrechannel hardware (step 26), which mates it with the received header to form a Fibrechannel packet for transmission. This method has the advantage that the traffic data 34 traverses the memory bus only twice. However, this method is not straightforward to implement, since the CPU must be capable of arranging for the traffic data to be passed from the memory 14 to the Fibrechannel hardware in step 26. This is problematic because the CPU would conventionally have received only the headers for that packet, without any indication of where the packet was located in

memory, and so it would have no knowledge of where the traffic data is located in the memory. As a result, the CPU would be unable to inform the bridging entity that is to transmit that data onwards of what data is to be transmitted. Furthermore, if that transmitting entity is to be implemented in software then it could be implemented at user level, for example as an application, or as part of the operating system kernel. If it is implemented at user level then it would not conventionally be able to access physical memory addresses, being restricted instead to accessing memory via virtual memory addresses. As a result, it could not access the packet data in memory directly via a physical address. Alternatively, if the transmitting entity is implemented in the kernel then for software abstraction and engineering reasons it would be preferable for it to interface with the network at a high level of abstraction, for instance by way of a sockets API (application programming interface). As a result, it would be preferred that it does not access the packet data in memory directly via a physical address.

One way of addressing this problem is to permit the Ethernet hardware 5 to access the memory 14 by RDMA (remote direct memory access), and for the Ethernet hardware to be allocated named buffers in the memory. Then the Ethernet hardware can write the traffic data of each packet to a specific named buffer and through the RDMA interface with the bridging application (e.g. uDAPL) indicate to the application the location / identity of the buffer which has received data. The CPU can access the data by means of reading the buffer, for example by means of a post() instruction having as its operand the name of the buffer that is to be read. The Fibrechannel hardware can then be passed a reference to the named buffer by the application and so (also by RDMA) read the data from the named buffer. The buffer remains allocated to the Ethernet hardware during the reading step(s).

One problem with this approach is that it requires the Ethernet hardware to be capable of accessing the memory 14 by RDMA, and to include functionality that can handle the named buffer protocol. If the Ethernet hardware is not compatible with RDMA or with the named buffer protocol, or if the remainder of the system is not

configured to communicated with the Ethernet hardware by RDMA then this method cannot be used. Also, RDMA typically involves performance overheads.

Analogous problems arise when bridging in the opposite direction: from Fibrechannel to ISCSI, and when using other protocols.

There is therefore a need to improve the processing of data units in bridging situations.

According to one aspect of the present invention there is provided a method for bridging between a first data link carrying data units of a first data protocol and a second data link for carrying data units of a second protocol by means of a bridging device, the first and second protocols being such that data units of each protocol include protocol data and traffic data and the bridging device comprising a first interface entity for interfacing with the first data link, a second interface entity for interfacing with the second data link, a protocol processing entity and a memory accessible by the first interface entity, the second interface entity and the protocol processing entity, the method comprising: receiving by means of the first interface entity data units of the first protocol, and storing those data units in the memory; accessing by means of the protocol processing entity the protocol data of data units stored in the memory and thereby performing protocol processing for those data units under the first protocol; and accessing by means of the second interface entity the traffic data of data units stored in the memory and thereby transmitting that traffic data over the second data link in data units of the second data protocol.

According to a second aspect of the present invention there is provided a bridging device for bridging between a first data link carrying data units of a first data protocol and a second data link for carrying data units of a second protocol, the first and second protocols being such that data units of each protocol include protocol data and traffic data and the bridging device comprising: a first interface entity for interfacing with the first data link, a second interface entity for interfacing with the second data link, a protocol processing entity and a memory accessible by the first

interface entity, the second interface entity and the protocol processing entity; the first interface entity being arranged to receive data units of the first protocol, and storing those data units in the memory; the protocol processing entity being arranged to access the protocol data of data units stored in the memory and thereby perform protocol processing for those data units under the first protocol; and the second interface entity being arranged to access the traffic data of data units stored in the memory and thereby transmit that traffic data over the second data link in data units of the second data protocol.

According to a third aspect of the present invention there is provided a data processing system comprising: a memory comprising a plurality of buffer regions; an operating system for supporting processing entities running on the data processing system and for restricting access to the buffer regions to one or more entities; a first interface entity running on the data processing system whereby a first hardware device may communicate with the buffer regions; and an application entity running on the data processing system; the first interface entity and the application entity being configured to, in respect of a buffer region to which the operating system permits access by both the interface entity and the application entity, communicate ownership data so as to indicate which of the first interface entity and the application entity may access the buffer region and to access the buffer region only in accordance with the ownership data.

According to a fourth aspect of the present invention there is provided a method for operating a data processing system comprising: a memory comprising a plurality of buffer regions; an operating system for supporting processing entities running on the data processing system and for restricting access to the buffer regions to one or more entities; a first interface entity running on the data processing system whereby a first hardware device may communicate with the buffer regions; and an application entity running on the data processing system; the method comprising, in respect of a buffer region to which the operating system permits access by both the interface entity and the application entity, communicating ownership data by means of the first interface entity and the application entity so as to indicate which of the first interface

entity and the application entity may access the buffer region and to access the buffer region only in accordance with the ownership data.

According to a fifth aspect of the present invention there is provided a protocol processing entity for operation in a bridging device for bridging between a first data link carrying data units of a first data protocol and a second data link for carrying data units of a second protocol by means of a bridging device, the first and second protocols being such that data units of each protocol include protocol data and traffic data and the protocol processing entity being arranged to cause a processor of the bridging device to perform protocol processing for data units stored in the memory without it accessing the traffic data of those units stored in the memory. The protocol processing entity may be implemented in software. The software may be stored on a data carrier.

The protocol processing entity may be arranged to perform protocol processing for the data units stored in the memory without it accessing the traffic data of those units stored in the memory.

The first protocol may be such that protocol data of a data unit of the first protocol includes check data that is a function of the traffic data of the data unit. The method may then comprise: applying the function by means of the first entity to the content of a data unit of the first protocol received by the first interface entity to calculate first check data; transmitting the first check data to the protocol processing entity; and comparing by means of the protocol processing entity the first check data calculated for a data unit with the check data included in the protocol data of that data unit.

The memory may comprise a plurality of buffer regions. The first interface entity, the second interface entity and the protocol processing entity may each be arranged to access a buffer region only when they have control of it. The method may then comprise: the first interface entity storing a received data unit of the first protocol in a buffer of which it has control and subsequently passing control of that buffer to the protocol processing entity; the protocol processing entity passing control of a buffer

to the second interface entity when it has performed protocol processing of the or each data unit stored in that buffer; and the second interface entity passing control of a buffer to the first interface entity when it has transmitting the traffic data contained in that buffer over the second data link in data units of the second data protocol.

The method may comprise: generating by means of the protocol processing entity protocol data of the second protocol for the data units to be transmitted under the second protocol; communicating that protocol data to the second interface entity; and the second interface entity including that protocol data in the said data units of the second protocol.

The second protocol may be such that protocol data of a data unit of the second protocol includes check data that is a function of the traffic data of the data unit. The method may then comprise: applying the function by means of the second interface entity to the content of a data unit of the second protocol to be transmitted by the second interface entity to calculate first check data; combining that check data with protocol data received from the protocol processing entity to form second protocol data; and the second interface entity including the second protocol data in the said data units of the second protocol.

One of the first and second protocols may be TCP. One of the first and second protocols may be Fibrechannel. The first and second protocols may be the same.

The first and second interface entities may each communicate with the respective data link via a respective hardware interface.

The first and second interface entities may each communicate with the respective data link via the same hardware interface.

The protocol processing may comprise terminating a link of the first protocol.

The protocol processing may comprise: inspecting the traffic data of the first protocol; comparing the traffic data of the first protocol with one or more pre-set rules; and if the traffic data does not satisfy the rules preventing that traffic data from being transmitted by the second interface entity.

The data processing system may comprise a second interface entity running on the data processing system whereby a second hardware device may communicate with the buffer regions. The first and second interface entities and the application entity may be configured to, in respect of a buffer region to which the operating system permits access by the first and second interface entities and the application entity, communicate ownership data so as to indicate which of the first and second interface entities and the application entity may access each buffer regions and to access each buffer region only in accordance with the ownership data.

The first interface entity may be arranged to, on receiving a data unit, store that data unit in a buffer region that it may access in accordance with the ownership data and to subsequently modify the ownership data such that the application entity may access that buffer region in accordance with the ownership data. The application entity may be arranged to perform protocol processing on data unit(s) stored in a buffer region that it may access in accordance with the ownership data and to subsequently modify the ownership data such that the second interface entity may access that buffer region in accordance with the ownership data. The second interface entity may be arranged to transmit at least some of the content of data unit(s) stored in a buffer region that it may access in accordance with the ownership data and to subsequently modify the ownership data such that the application entity may access that buffer region in accordance with the ownership data.

The present invention will now be described by way of example with reference to the accompanying drawings. In the drawings:

Figure 6 shows in outline the logical and physical architecture of a bridge.

Figure 7 shows the architecture of the bridge of figure 6 in more detail, illustrating data transfer steps.

Figure 8 shows the structure of an ISCSI-over-Ethernet packet.

Figure 9 shows the architecture of the bridge of figure 6, illustrating alternative data transfer steps.

Figure 10 illustrates the physical architecture of a bridging device.

Figure 11 illustrates the logical architecture of the bridging device of figure 10.

Figure 12 shows the processing of data in the bridging device of figure 10.

In the bridging device described below, data units of a first protocol are received by interface hardware and written to one or more receive buffers. In the example described below, those data units are TCP packets which encapsulate ISCSI packets. The TCP and ISCSI header data is then passed to the entity that performs protocol processing. The header data is passed to that entity without the traffic data of the packets, but with information that identifies the location of the traffic data within the buffer(s). The protocol processing entity performs TCP and ISCSI protocol processing. If protocol processing is successful then it also passes the data identifying the location of the traffic data in the buffers to an interface that will be used for transmitting the outgoing packets. The interface can then read that data, form one or more headers for transmitting it as data units of a second protocol, and transmit it. In bridging between the data links that carry the packets of the respective protocols, the bridging device receives data units of one protocol and transmits data units of another protocol which include the traffic data contained in the received data units.

Figure 10 shows the physical architecture of a device 40 for bridging between an ISCSI-over-Ethernet data link 41 and a Fibrechannel data link 42. The device comprises an Ethernet hardware interface 43, a Fibrechannel hardware interface 44 and a central processing section 45. The hardware interfaces link the respective data links to the central processing section 45 via a bus 46, which could be a PCI bus. The central processing section comprises a CPU 47, which includes a cache 47a and a processing section 47b, and random access memory 48 which are linked by a memory bus 49 to the PCI bus. A non-volatile storage device 50, such as a hard disc, stores program code for execution by the CPU.

Figure 11 shows the logical architecture provided by the central processing section 45 of the bridging device 40. The CPU provides four main logical functions: an Ethernet transport library 51, a bridging application 52, a Fibrechannel transport library 53 and an operating system kernel 54. The transport libraries, the bridging application and the operating system are implemented in software which is executed by the CPU. The general principles of operation of such systems are discussed in WO 2004/025477.

Areas of the memory 48 are allocated for use as buffers 55, 56. These buffers are configured in such a way that the interface that receives the incoming data can write to them, the bridging application can read from them, and the interface that transmits the outgoing data can read from them. This may be achieved in a number of ways. In a system that is configured not to police memory access any buffer may be accessible in this way. In other operating systems they may be set up as anonymous memory: i.e. memory that is not mapped to a specific process; so that they can be freely accessed by both interfaces. Another approach is to implement a further process, or a set of instructions calls, or an API that is able to act as an intermediary to access the buffers on behalf of the interfaces.

The present example will be described with reference to a system in which the operating system allocates memory resources to specific processes and restricts other processes from accessing those resources. The transport libraries 51, 53 and the bridging application 52 are implemented in a single process, by virtue of them occupying a common instruction space. As a result, a buffer allocated to any of those three entities can be accessible to the other two. (Under a normal operating system (OS), OS-allocated buffers are only accessible if the OS chooses for them to be). The interfaces 43, 44 should be capable of writing to and reading from the buffers. This can be achieved in a number of ways. For example, each transport libraries may implement an API through which the respective interface can access the buffers. Alternatively, the interface could interact directly with the operating system to access the buffers. This may be convenient where, in an alternative

embodiment, one of the transport libraries is implemented as part of the operating system and derives its ability to access the buffers through its integration with the operating system rather than its sharing of an instruction space with the bridging application.

Each buffer is identifiable by a handle that acts as a virtual reference to the buffer. The handle is issued by the operating system when the buffer is allocated. An entity wishing to read from the buffer can issue a read call to the operating system identifying the buffer by the handle, in response to which the operating system will return the content of the buffer or the part of buffer cited in the read call. An entity wishing to write to the buffer can issue a write call to the operating system identifying the buffer by the handle, in response to which the operating system will write data supplied with the call to the buffer or to the part of buffer cited in the write call. As a result, the buffers need not be referenced by a physical address, and can hence be accessed by user-level entities under operating systems that limit the access of user-level entities to physical memory.

The transport libraries and the bridging application implement a protocol to allow them to cooperatively access the buffers that are allocated to the instruction space that they share. In this protocol each of those entities maintains an "owned buffer" list of the buffers that it has responsibility for. Each entity is arranged to access only those buffers currently included in its owned buffer list. Each entity can pass a "handover" message to one of the other entities. The handover message includes the handle of a buffer. On transmitting the handover message (or alternatively on acknowledgement of the handover message), the entity that transmitted the handover message deletes the buffer mentioned in the message from its owned buffer list. On receipt of a handover message an entity adds the buffer mentioned in the message to its owned buffer list. This process allows the entities to cooperatively assign control of each buffer between each other, independently of the operating system. The entity whose owned buffer list includes a buffer is also responsible for the administration of that buffer: for example for returning the buffer to the operating system when it is no longer required. Buffers that are subject to this protocol will be

termed "anonymous buffers" since the operating system does not discriminate between the entities of the common instruction space in policing access to those buffers.

The operation of the device for bridging packets from the Ethernet interface to the Fibrechannel interface will now be explained. The device operates in an analogous way to bridge packets in the opposite direction.

At the start of operations the bridging application 52 requests the operating system 54 to allocate blocks of memory for use by the bridging system as buffers 55. The operating system allocates a set of buffers accordingly and passes handles to them to the application. These buffers can then be accessed directly by the bridging application and the transport libraries, and can be accessed by the interfaces by means of the anonymous APIs implemented by the respective transport libraries.

One or more of the buffers are passed to the incoming transport library 51 by means of one or more handover messages. The transport library adds those buffers to its owned buffer list. The transport library maintains a data structure that permits it to identify which of those buffers contains unprocessed packets. This may be done by queuing the buffers or by storing a flag indicating whether each buffer is in use. On being passed a buffer the incoming transport library notes that buffer as being free. The data structure preferably indicates the order in which the packets were received, in order that that information can be used to help prioritise their subsequent processing. Multiple packets could be stored in each buffer, and a data structure maintained by the Ethernet transport library to indicate the location of each packet.

Referring to figure 12, as Ethernet packets are received Ethernet protocol processing is performed by the Ethernet interface hardware 43, and the Ethernet headers are removed from the Ethernet packets, leaving TCP packets in which ISCSI packets are encapsulated. Each of these packets is written by the Ethernet hardware into one of the buffers 55. (Step 60). This is achieved by the Ethernet hardware issuing a buffer write call to the API of the Ethernet transport library, with the TCP packet as an

operand. In response to this call the transport library identifies a buffer that is included in its owned buffer list and that is free to receive a packet. It stores the received packet in that buffer and then modifies its data structure to mark the buffer as being occupied.

Thus, at least some of the protocol processing that is to be performed on the packet can be performed by the interface (43, in this example) that received the incoming packet data. This is especially efficient if that interface includes dedicated hardware for performing that function. Such hardware can also be used in protocol processing for non-bridged packets: for example packets sent to the bridge and that are to terminate there. One example of such a situation is when an administrator is transmitting data to control the bridging device remotely. The interface that receives the incoming packet data has access to both the header and the traffic data of the packet. As a result, it can readily perform protocol processing operations that require knowledge of the traffic data in addition to the header data. Examples of these operations include verifying checksum data, CRC (cyclic redundancy check) data or bit-count data. In addition to Ethernet protocol processing the hardware could conveniently perform TCP protocol processing of received packets.

The application 52 runs continually. Periodically it makes a call, which may for example be "recv()" or "complete()" to the transport library 51 to initiate the protocol processing of any Ethernet packet that is waiting in one of the buffers 55. (Step 61). The recv()/complete() call does not specify any buffer. In response to the recv()/complete() call the transport library 51 checks its data structure to find whether any of the buffers 55 contain unprocessed packets. Preferably the transport library identifies the buffer that contains the earliest-received packet that is still unprocessed, or if the buffer is capable of prioritising certain traffic then it may bias its identification of a packet based on that prioritisation. If an unprocessed packet has been identified then the transport library responds to the recv()/complete() call by returning a response message to the application (step 62), which includes:

- the TCP and ISCSI headers of the identified packet, which may collectively be considered to constitute a header or header data of the packet;

- the handle of the buffer in which the identified packet is stored;
- the start point within that buffer of the traffic data block of the packet; and
- the length of the traffic data block of the packet.

By means of the headers the application can perform protocol processing on the received packet. The other data collectively identifies the location of the traffic data for the packet. The response message including a buffer handle is treated by the incoming transport library and the bridging application as handing that buffer over to the bridging application. The incoming transport library deletes that buffer handle from its owned buffer list as one of the buffers 55, and the bridging application adds the handle to its owned buffer list.

It will be noted that by this message the application has received the header of the packet and a handle to the traffic data of the packet. However, the traffic data itself has not been transferred. The application can now perform protocol processing on the header data.

The protocol processing that is to be performed by the application may involve functions that are to be performed on the traffic data of the packet. For example, ISCSI headers include a CRC field, which needs to be verified over the traffic data. Since the application does not have access to the traffic data it cannot straightforwardly perform this processing. Several options are available. First, the application could assume that that CRC (or other such error-check data) is correct. This may be a useful option if the data is delay-critical and need not anyway be re-transmitted, or if error checking is being performed in a lower-level protocol. Another option is for the interface to calculate the error-check data over the relevant portion of the received packet and to store it in the buffer together with the packet. The error check data can then be passed to the application in the response message detailed above, and the application can simply verify whether that data matches the data that is included in the header. This requires the interface to be capable of identifying data of the relevant higher-level protocol (e.g. ISCSI) embedded in received packets of a lower-level protocol (e.g. Ethernet or TCP), and to be capable of executing the error-check algorithm appropriate to that higher-level data. Thus, in this approach the

execution of the error-check algorithm is performed by a different entity from that which carries out the remainder of the protocol processing, and by a different entity from that which verifies the error-check data.

Not all of the headers of the packet as received at the hardware interface need be passed in the response message that is sent to the application, or even stored in the buffer. If protocol processing for one or more protocols is performed at the interface then the headers for those protocols can be omitted from the response and not stored in the buffer. However, it may still be useful for the application to receive the headers of one or more protocols for which the application does not perform protocol processing. One reason for this is that it provides a way of allow the application to calculate the outgoing route. The outgoing route could be determined by the Fibrechannel transport library 53 making use of system-wide route tables that could , for example, be maintained by the operating system. The Fibrechannel transport library 53 can look up a destination address in the route tables so as to resolve it to the appropriate outgoing FC interface.

The application is configured in advance to perform protocol processing on one or more protocol levels. The levels that are to be protocol processed by the application will depend on the bridging circumstances. The application is configured to be capable of performing such protocol processing in accordance with the specifications for the protocol(s) in question. In the present example the application performs protocol processing on the ISCSI header. (Step 63).

Having performed protocol processing on the header as received from the incoming transport library, the application then passes a send() command to the Fibrechannel transport library (step 65). The send() command includes as an operand the handle of the buffer that includes the packet in question. It may also include data that specifies the location of the traffic data in the buffer, for example the start point and length of the traffic data block of the packet. The send() command is interpreted by the buffering application and by the outgoing transport library as handing over that buffer to the outgoing transport library. Accordingly, the bridging application deletes

that buffer handle from its owned buffer list, and the outgoing transport library adds the handle to its owned buffer list, as one of the buffers 56.

The Fibrechannel transport library then reads the header data from that buffer (step 66) and using the header alone (i.e. without receiving the traffic data stored in the buffer) it forms a Fibrechannel header for onward transmission of the corresponding traffic data (step 67).

The Fiberchannel transport library then provides that header and the traffic data to the Fibrechannel interface, which combines them into a packet for transmission (step 68). The header and the traffic data could be provided to the Fiberchannel interface in a number of ways. For example, the header could be written into the buffer and the start location and length of the header and the traffic data could be passed to the Fiberchannel interface. Conveniently the header could be written to the buffer immediately before the traffic data, so that only one set of start location and length data needs to be transmitted. If the outgoing header or header set is longer than the incoming header or header set this may require the incoming interface to write the data to the buffer in such a way as to leave sufficient free space before the traffic data to accommodate the outgoing header. The Fiberchannel interface could then read the data from the buffer, for example by DMA (direct memory access). Alternatively, the header could be transmitted to the Fiberchannel interface together with the start location and length of the traffic data and the interface could then read the traffic data, by means of an API call to the transport library, and combine the two together. Alternatively, both the header and the traffic data could be transmitted to the Fiberchannel interface. The header and the start/length data could be provided to the Fiberchannel interface by being written to a queue stored in a predefined set of memory locations, which is polled periodically by the interface.

The outgoing header might have to include calculated data, such as CRCs, that is to be calculated as a function of the traffic data. In this situation the header as formed by the transport library can include space (e.g. as zero bits) for receiving that calculated data. The outgoing hardware interface can then calculate the calculated

data and insert it into the appropriate location in the header. This avoids the outgoing transport library having to access the traffic data.

Once the Fibrechannel packet has been transmitted for a particular incoming packet the buffer in which the incoming packed had been stored can be re-used. The Fibrechannel transport library hands over ownership of the buffer to the Ethernet transport library. Accordingly, the Fiberchannel transport library deletes that buffer handle from its owned buffer list, and the Ethernet transport library adds the handle to its owned buffer list, marking the buffer as free for storage of an incoming packet.

As indicated above, the buffers in which the packets are stored are implemented as anonymous buffers. When a packet is received the buffer that is to hold that packet is owned by the incoming hardware and/or the incoming transport library. When the packet comes to be processed by the bridging application ownership of the buffer is transferred to the bridging application. Then when the packet comes to be transmitted ownership of the buffer is transferred to the outgoing hardware and/or the outgoing transport library. Once the packet has been transmitted ownership of the buffer can be returned to the incoming hardware and/or the incoming transport library. In this way the buffers can be used efficiently, and without problems of access control. The use of anonymous buffers avoids the need for the various entities to have to support named buffers. This is especially significant in the case of the incoming and outgoing hardware since it may not be possible to modify pre-existing hardware to support named buffers. It may also not be economically viable to use such hardware since it requires significant additional complexity – namely the ability to fully perform complex protocol processing e.g. to support TCP and RDMA (iWARP) protocol processing. This would in practice require a powerful CPU to be embedded in the hardware, which would make the hardware excessively expensive.

Once each layer of protocol processing is completed for a packet the portion of the packet's header that relates to that protocol is no longer required. As a result, the memory in which that portion of header was stored can be used to store other data structures. This will be described in more detail below.

When a packet is received the incoming hardware and/or transport library should have one or more buffers in its ownership. It selects one of those buffers for writing the packet to. That buffer may include one or more other received packets, in which case the hardware/library selects suitable free space in the buffer for accommodating the newly received packet. Preferably it attempts to pack the available space efficiently. There are various ways to aim at this: one is to find a space in a buffer that most closely matches the size of the received packet, whilst not being smaller than the received packet. The space in the buffer may be managed by a data structure stored in the buffer itself which provides pointers to the start and end of the packets stored in the buffer. If the buffer includes multiple packets then ownership of the buffer is passed to the application when any of those is to be protocol processed by the application. When the packet has been transmitted the remaining packets in the buffer remain unchanged but the data structure is updated to show the space formerly occupied by the packet as being vacant.

If the TCP and ISCSI protocol processing is unsuccessful then the traffic data of the packet may be dropped. The data need not be deleted from the buffer: instead the anonymous buffer handle can simply passed back to the Ethernet transport library for reuse.

This mechanism has the consequence that the traffic data needs to pass only twice over the memory bus: once from the Ethernet hardware to memory and once from memory to the Fibrechannel hardware. It does not need to pass through the CPU; in particular it does not need to pass through the cache of the CPU. The same approach could be used for other protocols; it is not limited to bridging between Ethernet and Fibrechannel.

The transport libraries and the application can run at user level. This can improve reliability and efficiency over prior approaches in which protocol processing is performed by the operating system. Reliability is improved because the machine can continue in operation even if a user-level process fails.

The transport libraries and the application are configured programmatically so that if their ownership list does not include the identification of a particular buffer they will not access that buffer.

If the machine is running other applications in other address spaces then the named buffers for one application are not accessible to the others. This feature provides for isolation between applications and system integrity. This is enforced by the operating system in the normal manner of protecting applications' memory spaces.

The received data can be delivered directly from the hardware to the ISCSI stack, which is constituted by the Ethernet transport library and the application operating in cooperation with each other. This avoids the need for buffering received data on the hardware, and for transmitting the data via the operating system as in some prior implementations.

The trigger for the passing of data from the buffers to the CPU is the polling of the transport library at step 61. The polling can be triggered by an event sent by the Ethernet hardware to the application on receipt of data, a timer controlled by the application, by a command from a higher level process or from a user, or in response to a condition in the bridging device such as the CPU running out of headers to process. This approach means that there is no need for the protocol processing to be triggered by an interrupt when data arrives. This economises on the use of interrupts.

The bridging device may be implemented on a conventional personal computer or server. The hardware interfaces could be provided as network interface cards (NICs) which could each be peripheral devices or built into the computer. For example, the NICs could be provided as integrated circuits on the computer's motherboard.

When multiple packets have been received the operations in figure 12 can be combined for multiple packets. For example, the response data (at step 62) for

multiple packets can be passed to the CPU and stored in the CPU's cache awaiting processing.

There may be limitations on the size of the outgoing packets that mean that the traffic data of an incoming packet cannot be contained in a single outgoing packet. In that case the traffic data can be contained in two or more outgoing packets, each of whose headers is generated by the transport library of the outgoing protocol.

Since the packets are written to contiguous blocks of free space in the buffers 54, as packets get removed from the buffers 54 gaps can appear in the stream of data in the buffers. If the received packets are of different lengths then those gaps might not be completely filled by new received data. As a result the buffers can become fragmented, and therefore inefficiently utilised. To mitigate this, as soon as the header of a received packet has been passed to the CPU for processing the space occupied by that header can be freed up immediately. That space can be used to allow a larger packet to be received in a gap in memory preceding that header. Alternatively that space can be used for various data constructs. For example, it can be used to store a linked-list data structure that allows packets to be stored discontiguously in the buffer. Alternatively, it could be used to store the data structure that indicates the location of each packet and the order in which it was received. Fragmentation may also be reduced by performing a defragmentation operation on the content of a buffer, or by moving packets whose headers have not been passed to the CPU for processing from one buffer to another. One preferred fragmentation algorithm is to check from time to time for buffers that contain less data than a pre-set threshold level. The data in such a buffer is moved out to another buffer, and the data structure that indicates which packet is where is updated accordingly.

In a typical architecture, when the Ethernet packet headers are read to the CPU for processing by the bridging application they will normally be stored in a cache of the CPU. The headers will then be marked as "dirty" data. Therefore, in normal circumstances they would be flushed out of the cache and written back to the buffer

so as to preserve the integrity of that memory. However, once the headers have been processed by the bridging application they are not needed any more, and so writing them back to the buffer is wasteful. Therefore, efficiency can be increased by taking measures to prevent the CPU from writing the headers back to memory. One way to achieve this is by using an instruction such as the wbinv (write-back invalidate) instruction which is available on some architectures. This instruction can be used in respect of the header data stored in the cache to prevent the bridging application from writing that dirty data back to the memory. The instruction can conveniently be invoked by the bridging application on header data that is stored in the cache when it completes processing of that header data. At the same point, it can arrange for the space in the buffer(s) that was occupied by that header data to be marked as free for use, for instance by updating the data directory that indicates the buffer contents.

The principles described above can be used for bridging in the opposite direction: from Fibrechannel to ISCSI, and when using other protocols. Thus the references herein to Ethernet and Fibrechannel can be substituted for references to other incoming and outgoing protocols respectively. They could also be used for bridging between links that use two identical protocols. In the apparatus of figure 10 the software could be configured to permit concurrent bridging in both directions. If the protocols are capable of being operated over a common data link then the same interface hardware could be used to provide the interface for incoming and for outgoing packets.

The anonymous buffer mechanism described above could be used in applications other than bridging. In general it can be advantageous wherever multiple devices that have their own processing capabilities are to process data units in a buffer, and where one of those devices is to carry out processing on only a part of each data unit. In such situations the anonymous buffer mechanism allows the devices or their interfaces to the buffer to cooperates so that the entirety of each data unit need not pass excessively through the system. One examples of such an application is a firewall in which a network card is to provide data units to an application that is to

inspect the header of each data unit and in dependence on that header either block or pass the data unit. In that situation, the processor would not need to terminate a link of the incoming protocol to perform the required processing: it could simply inspect incoming packets, compare them with pre-stored rules and allow them to pass only if they satisfy the rules. Another example is a tape backup application where data is being received by a computer over a network, written to a buffer and then passed to a tape drive interface for storage. Another example is a billing system for a telecommunications network, in which a network device inspects the headers of packets in order to update billing records for subscribers based on the amount or type of traffic passing to or from them.

The applicant hereby discloses in isolation each individual feature described herein and any combination of two or more such features, to the extent that such features or combinations are capable of being carried out based on the present specification as a whole in the light of the common general knowledge of a person skilled in the art, irrespective of whether such features or combinations of features solve any problems disclosed herein, and without limitation to the scope of the claims. The applicant indicates that aspects of the present invention may consist of any such individual feature or combination of features. In view of the foregoing description it will be evident to a person skilled in the art that various modifications may be made within the scope of the invention.

SECTION C

VIRTUALISATION SUPPORT

This invention relates to supporting virtual instances of data processing entities, particularly hardware devices.

It is normal for a computer to have just a single operating system (OS) running at any one time. That operating system provides support to one or more user-level applications running on the computer. The support provided by the operating system typically includes providing the applications with protocols whereby the applications can communicate with hardware components of the computer. Those components could include input/output (I/O) devices such as a keyboard, a display or a network interface. The protocols can be provided in the form of libraries of procedures that can be called by the applications and which, when executed, communicate with the hardware in the desired way.

Another arrangement, which is illustrated in figure 13, involves running multiple operating systems 1 on a single computer 2. This arrangement has been implemented for example by the IBM 360 and more recently by VMware and Xen. Each operating system supports a respective set of applications 3, which communicate with their operating system in the normal way. Each operating system functions independently of the others, so a supervisory entity or hypervisor 4 is used to manage the operating systems and their interactions with the hardware 5. The hypervisor performs functions such as scheduling the operations of each operating system and ensuring that when the hardware needs to communicate with a particular one of the operating systems its messages are directed correctly. Therefore the hypervisor is itself a form of operating system.

The latter function of the hypervisor is especially significant when the hardware needs to initiate communications with an operating system or even with directly with a user-level entity such as an application. Such communications must be directed to the right one of the operating systems and/or to the right one of the applications.

One example of a hardware configuration in which this could arise is the one described in WO 2004/025477. In that configuration the network interface hardware can pass data that has been received over the network to a buffer at its own initiation, rather than having to wait for the operating system or the application to request any received data. The buffer can be owned by the operating system or the application, so the operating system or the application can access the data stored in the buffer.

Figure 14 illustrates the manner in which network hardware is conventionally configured for operation in a system running multiple operating systems. In figure 14 like components have the same reference numbers as in figure 13. In the arrangement of figure 14 the network hardware 6 implements two virtual network interfaces 7 and 8. Each of those virtual interfaces is operated by common hardware components of the network hardware 6, but each has its own state which is stored in a respective region of state memory 9 of the hardware. Included in the state of each virtual network interface is its MAC (medium access control) address, which it uses in the network 10 to which the interface is connected. Each virtual interface has a different MAC address. This allows the network hardware to identify which of the virtual interfaces an incoming packet should be directed to, by means of the MAC address specified in an incoming data packet. For this purpose a filter 12 is implemented in the network hardware. When an incoming packet arrives the filter 12 checks its content, identifies the destination MAC address that it specifies and directs it to the appropriate virtual network interface based on that MAC address. Each operating system 1 has its own instance of a driver 11 for communicating with the respective virtual interface.

WO 2004/025477 describes a network interface that, when implemented on a platform that that has a single operating system and one or more applications, can advantageously deliver data received over the network directly to a buffer owned by the operating system or the application. There may be multiple buffers that are capable of receiving the data: the operating system could have one or more buffers for receiving the data, as could the or each application. In a typical configuration

there could be one buffer allocated for each communication path or channel that is in use. The network card is therefore expected to direct received data to the appropriate one of those buffers. This can be done by the NIC using a look-up table that is pre-configured to store the address of each buffer and the port number associated with that buffer The NIC can store the virtual interface associated with the host::port information. The data in a received TCP packet contains the host::port information which when looked up determines the virtual interface to which data should be directed. Each virtual interface is associated with a set of buffers and according to information from the recipient application or operating system the NIC is able to determine which of the buffers it should next deliver data into. The network card can then look up the appropriate buffer when data is received on a particular port. Data received on other ports (for example a request from a remote terminal to establish a connection to the local machine) can be sent to a default virtual interface, which is normally associated with the operating system kernel. The kernel can then decide how to handle that data.

When the system described in WO 2004/025477 is implemented on a platform that has multiple operating systems the approach described above cannot be readily implemented. The look-up table does not take account of the MAC address to which data has been directed, so it cannot distinguish between data sent on the same port but to different operating systems or to applications supported by different operating systems. Therefore, the efficiency advantages that stem from filtering and directing incoming data at the network hardware cannot be achieved in that scenario.

There is therefore a need for an improved way of arranging an interface to receive and direct incoming data.

According to one aspect of the present invention there is provided a computer system comprising: hardware including a data interface for interfacing between the computer system and a data source; a memory; a first operating system capable of communicating with the hardware; and a second operating system capable of supporting a user-level application and being configured to communicate with the

hardware via the first operating system, the second operating system being capable of allocating a region of the memory for use as a buffer by such a user-level application; wherein the data interface is configurable to associate a predetermined data format with a region of the memory that has been allocated for use as a buffer by a user-level application supported by the second operating system so as to, on subsequently receiving from the data source a data message of that format, automatically store data of that message in that region of the memory without it passing via the first or second operating systems.

According to a second aspect of the present invention there is provided a computer system comprising: hardware including a data interface for interfacing between the computer system and a data source; a memory; a first operating system capable of communicating with the hardware; and a second operating system capable of supporting a user-level application; wherein at least one of the first and second operating systems is arranged to detect that data of a first data message received by the data interface from the data source has been directed to a destination via the first operating system, and in response to detecting that to configure the data interface to direct data of subsequent data messages having a data format in common with the first data message to that destination without it passing via the second operating system.

According to a third aspect of the present invention there is provided a data interface that is capable of operation in a computer system according to the first or second aspects of the present invention. Such a data interface is preferably configurable to associate a predetermined data format with a region of the memory of the computer system external to the data interface that has been allocated for use as a buffer by a user-level application supported by the second operating system so as to, on subsequently receiving from the data source a data message of that format, automatically store data of that message in that region of the memory without it passing via the first or second operating systems.

According to a fourth aspect of the present invention there is provided a computer program for acting as a driver for communicating with a data interface in a computer system according to the first or second aspects of the present invention. Preferably the computer program is such as to provide an operating system with functionality to, in response to detection in response to detection that data of a first data message received by the data interface from the data source has been directed to a destination via another operating system, configure the data interface to direct data of subsequent data messages having a data format in common with the first data message to that destination without it passing via the second operating system. The computer program may be stored on a data carrier.

According to a fifth aspect of the present invention there is provided a method for operating a computer system comprising hardware including a data interface for interfacing between the computer system and a data source, a memory, a first operating system capable of communicating with the hardware; and a second operating system capable of supporting a user-level application and being configured to communicate with the hardware via the first operating system, the second operating system being capable of allocating a region of the memory for use as a buffer by such a user-level application; the method comprising: configuring the data interface to associate a predetermined data format with a region of the memory that has been allocated for use as a buffer by a user-level application supported by the second operating system; and the data interface on subsequently receiving from the data source a data message of that format, automatically storing data of that message in that region of the memory without it passing via the first or second operating systems.

According to a sixth aspect of the present invention there is provided a method for operating a computer system comprising: hardware including a data interface for interfacing between the computer system and a data source; a memory; a first operating system capable of communicating with the hardware; and a second operating system capable of supporting a user-level application; the method comprising: detecting by means of at least one of the first and second operating

systems that data of a first data message received by the data interface from the data source has been directed to a destination via the first operating system; and in response to detecting that configuring the data interface by means of that operating system to direct data of subsequent data messages having a data format in common with the first data message to that destination without it passing via the second operating system.

Preferably the first operating system is capable of serving as an interface between multiple further operating systems and the hardware.

Preferably the first operating system is a hypervisor. Preferably the second operating system is an operating system that provides direct application support, such as Windows or Linux.

The data interface may be a network interface. The data source may be a data network.

Preferably the data interface has access to a data store for storing a plurality of indications of respective data formats and corresponding to each one an indication of a destination, and the data interface is arranged to, on receiving a data message from the data source identify whether the format of the data message matches a data format an indication of which is stored in the data store, and if it does to pass data of that message to that destination.

Preferably the data format is at least partially defined by a destination address. Preferably the address is an internet layer address. Preferably the address is an IP (internet protocol) address.

Preferably the data format is at least partially defined by a data port, such as a TCP port.

Preferably the data message is a data packet.

Preferably the data interface is configurable automatically by the first operating system to associate the predetermined data format with the region of the memory that has been allocated for use as a buffer by a user-level application supported by the second operating system.

Preferably the data interface is configurable automatically by the second operating system to associate the predetermined data format with the region of the memory that has been allocated for use as a buffer by a user-level application supported by the second operating system.

Preferably the data interface is arranged to, on receiving a data message from the data source identify whether it is configured to associate the format of that message with a region of the memory and if it has to automatically store data of that message in that region of the memory without it passing via the first or second operating systems.

Preferably one of the first operating system, the second operating system and the data interface is arranged to deconfigure the data interface from associating a message format with a region of the memory when a pre-set time has elapsed from when the interface was configured to associate that message format with that region of the memory.

Preferably one of the first operating system, the second operating system and the data interface is arranged to deconfigure the data interface from associating a message format with a region of the memory in response to sensing that data traffic conditions match one or more predefined criteria. The said traffic criteria may include the criterion that the flow of received data of the message format is below a pre-set amount in a pre-set time.

Preferably the second operating system is arranged to perform the step of configuring the data interface to direct data of subsequent data messages having a

data format in common with the first data message only when it detects that the flow of received data of that data format is above a pre-set amount in a pre-set time.

Preferably the second operating system is arranged to perform the step of configuring the data interface to direct data of subsequent data messages having a data format in common with the first data message only for data formats of one or more pre-set types.

The said pre-set types may each be defined by respective port numbers or ranges of port numbers.

The present invention will now be described by way of example with reference to the accompanying drawings. In the drawings:

figure 13 shows the architecture of a computer system supporting multiple operating systems;

figure 14 shows the configuration of a network interface for operation in the system of figure 13;

figure 15 shows the architecture of a computer system supporting multiple operating systems; and

figures 16 to 18 show communication flows in the system of figure 15.

In the system to be described below, the network interface is capable of directing received data to a particular receive buffer in dependence on destination information (e.g. MAC address) to which that data has been transmitted. Furthermore, the system is arranged to automatically configure the network interface to direct the data in that way. In a platform that has multiple operating systems managed by a hypervisor, the hypervisor may be arranged to automatically configure the network interface. If the system is para-virtualised then an operating system or a part of it may be arranged to automatically configure the network interface.

NICs conventionally do not support directing incoming traffic to a particular consumer of data based on destination MAC addresses since that is not normally required in a

single-operating-system environment. The following description sets out a method by which such NICs can be configured to efficiently support multiple-operating-system environments, and also a method by which enhanced NICs that do support filtering of incoming data based on destination MAC addresses can be configured.

The present example will be described with reference to a network that uses TCP/IP over Ethernet. However, the present invention is not limited to use with these and is applicable to networks that use other protocols.

Figure 15 shows a data processing device 20 connected to a network 21 by a data link 22. A further data processor 23 is also connected to the network and can communicate with the data processor 20 over the network. Figure 15 illustrates the components of data processor 20 in the hardware domain (24) and the software domain (25). In the hardware domain it comprises a network interface 26, a central processor 27 (e.g. a CPU), a working memory 28 (e.g. RAM) and a program store 29 (e.g. a hard disc). These are interconnected by a bus 30. In the software domain it comprises an operating system 31 and applications 32. These are provided by the execution by processor 27 of suitable program code stored in the program store 29. The operating system supports the applications by, for example, managing the applications' access to hardware components of the system. For this purpose, the operating system includes a set of drivers 33 for communication with the network interface 26.

The data processor 20 could be a general-purpose computer.

The operating system 31 and applications 32 are in a first environment 34. In the system illustrated in figure 15 multiple such environments are supported, as illustrated at 35 and 36. Each environment has a respective operating system that operates independently of the other operating systems and its own user-level application(s) that operate independently of applications in others of the environments. The operation of the environments is managed by hypervisor 37. The basic functions of the hypervisor could be analogous to those employed in

known multi-operating-system arrangements such as IBM 360 or VMware or Xen. The hypervisor interfaces between the operating systems and the hardware so as to allow each operating system to function correctly without conflicting with the other operating systems on the hardware platform 20.

The network interface 26 may be termed a network interface card (NIC). However, it need not take the form of a card. It could be a stand-alone external device or it could be integrated on to a motherboard of the data processor 20 or even integrated into an integrated circuit on which the processor 27 is implemented. The network interface card comprises a processor 38, a program store 39 and a working memory 40. The processor 38 executes code stored in the store 39 to perform networking functions, and uses the memory 40 as a temporary store for incoming or outgoing data, and to store working data such as look-up tables that are configured during operation.

The network interface 26 is capable of automatically delivering received data directly to buffers owned by an intended recipient of the data, for instance to a user-level buffer. The network interface could perform protocol proceeding on received data before storing it in the appropriate buffer. However, it is preferred that protocol processing is performed by the operating system 31 and/or at user level by transport library 41 after the network interface has delivered the data to the appropriate buffer.

The operating system 31, a user-level transport library 41 and the network interface 26 itself can cooperate to configure a table 43 in the network interface so as to cause the network interface to deliver data to a recipient's buffer in a single-operating-system environment. To this end the transport library 41 provides a routine that can be called by any application 32 that wishes to receive data. The process of configuration of table 43 and its use for receiving data in a single-operating system environment are illustrated in figure 16. When an application calls the routine in the transport library (step 70) the transport library arranges with the operating system for the allocation of one or more buffers 42 in memory 28 in which received data can be stored (step 71), and transmits a message to the network interface 26 to inform it of

the location of the buffer(s) and the TCP port number that will be associated with the buffer(s) (step 72). The TCP port number and/or other address bits of an incoming packet will allow the network interface to identify incoming traffic that is to be directed to the buffer(s). Where more than one contiguous memory buffer is allocated to the application those buffers they form a pool that can be used for receiving incoming data. In the present description data will for simplicity be described as being delivered to a buffer, but in practice it could be delivered to a discontinuous region of memory formed by a pool of buffers, all of which are owned by a particular destination application as a virtual interface. The network interface is informed of the address of the default virtual interface when the system is first configured. The network interface is arranged to, on subsequently receiving such a message, configure a look-up filter table 43 in memory 40 to hold the location of the buffer and the parameters (step 73). In practice the filter table 43 could be split into a first table that maps patterns of address bits to virtual interfaces and a second table that maps virtual interfaces to the physical addresses of the buffers that they are associated with and ownership information indicating which virtual interface has the right to deliver data onto that buffer.. When incoming data is received from the network (step 74) the network interface checks its characteristics against the parameters stored in filter table 43 (step 75). If the data matches a set of stored parameters the network interface directs that data to the buffer 50 whose location is stored in association with those parameters (step 76). The network interface is also arranged to direct incoming data that does not match any of the parameters stored in the table to a default location, which is conveniently the operating system in order that the operating system can process that data.

The network interface is capable of operating so as to support multiple MAC addresses, and in a multi-operating-system environment as illustrated in figure 15 it operates under the control of the hypervisor to use a different MAC address for communications to or from each operating system. When one of the operating systems begins to request network services the hypervisor allocates a MAC address to that operating system, stores in a table 44 the pairing of that MAC address with that operating system, and instructs the network interface to configure itself to

support that additional MAC address. When data is to be transmitted from that operating system the hypervisor instructs the network interface to transmit it from the appropriate MAC address. The hypervisor may also need to instruct the NIC to operate in a "promiscuous" mode, so that it will accept data directed to multiple MAC addresses.

In its default configuration, in order that received data is directed to the appropriate location the network interface forwards received packets to the hypervisor. The hypervisor identifies the destination MAC address in each packet, looks it up in table 44 and forwards the packet to the appropriate operating system. It would be possible for the network interface to be configured to do this. However, this would require the NIC to perform filtering based on the MAC address, which is not desirable. The reason for this is that that capability is not required in a single-OS system, and additionally providing support for choosing a destination based on MAC address would require the table 43 (which is conveniently provided as a content-addressable memory or RAM based hash table) to be larger. This would be expected to make it slower to look up data in the table or to take up excessive amounts of memory on the NIC. Since all incoming traffic is filtered against the filter table 43 it is desirable to keep operation of the table as simple and quick as possible.

The present system therefore provides other mechanisms for configuring the table 43 in a system such as that of figure 15 that has multiple operating systems running side-by-side. These mechanisms are illustrated in figures 17 and 18. In the present system, the network interface can be configured by these mechanisms to direct data directly to the appropriate receive buffer, without the intervention of the hypervisor to route each item of incoming data. This can provide a significant improvement in efficiency, since the amount of processing that the hypervisor is required to perform is significantly reduced.

When the system is fully virtualised the operating systems and the applications are unaware that they are running in a multi-OS environment. In this situation the

mechanism of figure 17 can be used. In a para-virtualised environment either mechanism can be used.

When an application wishes to be able to receive data it calls the routine in the transport library in the same way as if it were in a single-OS system. The transport library obtains a buffer for the data to be received and signals the hypervisor with the details of the connection (including buffer location and port number) in the way that it would normally signal the network interface in a single-OS system. The hypervisor then stores the filtering parameters (e.g. port number) against the address of the appropriate buffer in a table 45 that is specific to that operating system. The tables 44 and 45 could be integrated with each other. Table 45 may also indicate other locations to which data could be delivered. For example it could indicate a default location for each environment, which could conveniently be the operating system of that environment.

Figure 17 shows the signalling in one configuration mechanism. In this mechanism the hypervisor is arranged to configure the NIC's filter table 43 when it has to forward received network data to an operating system. Once the table has been appropriately configured, any further data of the same nature can be forwarded by the network interface without the intervention of the hypervisor.

Figure 17 shows signalling between the network interface 26, the hypervisor 37, an operating system 31 supported by the hypervisor and the buffer 50 of an application running on that operating system. At step 100 a TCP/IP data packet is received by the network interface. The network interface checks whether that packet matches any of the rows in filter table 43 (step 101), and since it does not it forwards the packet to the hypervisor. (Step 102). The hypervisor analyses the packet to identify its destination MAC address and looks that address up in table 44 to identify which of the operating systems it is supporting has been allocated that MAC address. It then looks up in the appropriate one of the tables 45 to identify which destination corresponds to the destination port of the received packet (step 103), and it forwards the packet to that destination (step 104). The destination could be the operating

system 31 or a buffer 50. The hypervisor is configured so as to then automatically send a configuration message to the NIC (step 105) to cause the NIC to configure its table 43 so that future packets having the same destination IP address and port as that previously received packet will be forwarded directly by the NIC to that buffer. In response to that message the NIC configures table 43 accordingly (step 106). This will avoid the need for such packets to be handled by the hypervisor, reducing load on the system.

When such a packet is subsequently received (step 110) the NIC checks its details against the filter table 43, finds a match and retrieves the destination stored in the table (step 111) and then automatically forwards the packet to that destination. This subsequent operation bypasses the hypervisor.

The hypervisor may automatically configure the table 43 in this way in response to the need to forward any received packets to a destination. Alternatively, in some situations it may be preferable for it to configure the table only after it has received a predetermined amount of data (e.g. a certain number of incoming packets) to a particular IP address or to a particular combination of IP address and port number.

This first configuration mechanism is transparent to the operating systems and the applications.

The second configuration mechanism can be used if the system is a para-virtualised system: that is one in which entities in one of the environments 34 to 36 can have knowledge of the fact that they are running in a multiple-OS system. In this mechanism the table 43 of the NIC can be configured by one of the operating systems or by an application running on that operating system.

Figure 18 shows signalling in the second mechanism. At step 200 a TCP/IP data packet is received by the network interface. The network interface fails to match the packet in table 43 (step 201) and forwards it to the hypervisor. (Step 202). The hypervisor analyses the packet to identify its destination MAC address and looks that

address up in table 44 to identify which of the operating systems it is supporting has been allocated that MAC address. (Step 202). It then looks up in the tables 44, 45 to identify where to direct the received packet. (Step 203). In this example the table 45 indicates that the packet is to be sent to operating system 31. The hypervisor forwards the packet to that operating system. (Step 204). The operating system processes the packet, for example by storing it so it is accessible by an application or by protocol-processing it. The operating system also sends a configuration message to the NIC (step 205) to cause the NIC to configure its table 43 so that future packets having the same destination IP address and port as that packet will be forwarded directly by the NIC to that operating system or to the appropriate buffer. In response to that message the NIC configures table 43 accordingly (step 206). This will avoid the need for such packets to be handled by the hypervisor, reducing load on the system. Subsequent operation is as in steps 110 to 112 of figure 17. In this configuration the transport library is unused.

The operating system 31 may automatically configure the table 43 in this way in response to detecting certain pre-programmed traffic flow conditions. These may include traffic flows greater than a set volume in a set period of time to a particular port. It may advantageously monitor particular port numbers that are known to be used by high-performance applications. Which ports those are will depend on the use to which the host computer is being put. In this way the operating system can arrange for all packets to a particular high-priority application, such as a database application, to be accelerated. Alternatively, in some situations it may be preferable for it to configure the table only after it has received a predetermined amount of data (e.g. a certain number of incoming packets) to a particular IP address or to a particular combination of IP address and port number. It may then automatically deconfigure the table to delete a particular stored combination of data format and buffer address. It may do so in response to detecting conditions such as that the combination has been in place for a pre-set time, or that less data than a set threshold (of data volume or number of packets) has been received in a pre-set time. That threshold may be very low, so that the deconfiguration takes place only if no data of the respective type has been received in a certain time.

Entries in the table 43 may be deleted automatically after a pre-set period of time. This can help to avoid the table 43 growing too large and to take account of the possibility of the destination buffers or IP addresses changing. The hypervisor and/or the operating systems could be configured to signal the NIC to delete entries in the table 43 when they become out-of-date.

The multiple operating systems could be multiple instances of a single type of operating system or they could each be different operating systems.

In the data is preferably received as packets and forwarded in the same packetised form. However, the traffic data could be extracted from received packets by the NIC and forwarded to the hypervisor or elsewhere together with a representation of the routing data (e.g. IP address and port) that is needed for identifying the destination of data. This could involve partial or full protocol processing being performed by the NIC.

The data could be conveyed over the network by a protocol other than TCP/IP over Ethernet. Any suitable networking protocol can be used. Instead of MAC addresses, IP addresses and ports analogous identifiers at similar levels of the protocol that is in use can be employed for identifying the required destination of received data.

The functions performed by the hypervisor or the operating system in the mechanisms illustrated in figures 17 and 18 could be performed by the core (kernel) of the hypervisor or the operating system or by a driver of the hypervisor or operating system that allows it to communicate with the NIC.

The applicant hereby discloses in isolation each individual feature described herein and any combination of two or more such features, to the extent that such features or combinations are capable of being carried out based on the present specification as a whole in the light of the common general knowledge of a person skilled in the art, irrespective of whether such features or combinations of features solve any problems

disclosed herein, and without limitation to the scope of the claims. The applicant indicates that aspects of the present invention may consist of any such individual feature or combination of features. In view of the foregoing description it will be evident to a person skilled in the art that various modifications may be made within the scope of the invention.

CLAIMS

1. A method for controlling the processing of data in a data processor, the data processor being connectable to a further device over a data link, the method comprising the steps of:

receiving data at an element of the data processor;

if a set interval has elapsed following the receipt of the data, determining whether processing of the received data in accordance with a data transfer protocol has begun, and, if it has not, triggering such processing of the received data by a protocol processing element;

sensing conditions pertaining to the data link; and

setting the interval in dependence on the sensed conditions.

2. A method according to claim 1 wherein the data processor is connectable to the data link by means of an interface.

3. A method according to claim 2 wherein a timer resides on the interface, and the said interval is measured by the timer.

4. A method according to claim 2 or claim 3 wherein the interface is implemented in hardware.

5. A method according to any preceding claim wherein the step of determining comprises determining whether processing of the received data by code at user level has begun.

6. A method according to any preceding claim wherein the said protocol processing element is an operating system.

7. A method according to any preceding claim wherein the received data comprises data received at the data processor over the data link.

8. A method according to claim 7 wherein the received data comprises data received at the data processor by means of an asynchronous transmission over the data link.

9. A method according to any of claims 1 to 6 wherein the received data comprises data to be transmitted over the data link.

10. A method according to claim 9 wherein the received data comprises data to be transmitted by means of an asynchronous transmission over the data link.

11. A method according to any preceding claim wherein the step of triggering processing comprises issuing an interrupt.

12. A method according to claim 11 as dependent on claim 6 wherein the step of triggering processing comprises issuing an interrupt to the operating system.

13. A method according to any preceding claim wherein the said element is a transport library associated with an application running on the data processor.

14. A method according to claim 2 or any of claims 3 to 13 as dependent on claim 2, further comprising the step of in response to receiving the data, sending an instruction from the said element to the timer.

15. A method according to claim 14 wherein the step of sending an instruction to the timer comprises triggering the timer directly from the said element via a memory mapping onto the said interface.

16. A method according to any preceding claim wherein the step of setting the interval comprises reducing the interval if the sensed conditions are indicative of an increase in data rate over the data link.

17. A method according to claim 16 wherein buffer space is allocated to the data link for storing data received at the data processor over the data link, and the protocol is

a protocol that employs a receive window in accordance with which a transmitter of data according to the protocol will transmit no further traffic data once the amount of data defined by the receive window has been transmitted and is unacknowledged by the receiver, and wherein the step of setting the interval comprises reducing the interval in response to sensing that the size of the buffer space allocated to the data link is greater than the size of the receive window.

18. A method according to claim 17 further comprising the step of varying the size of the buffer space allocated to the data link in response to a request from a consumer of the traffic data.

19. A method according to claim 18 wherein the consumer is an application running on the data processor.

20. A method according to any preceding claim wherein the step of sensing conditions comprises sensing the presence in a transmit buffer of data to be transmitted over the data link.

21. A method according to claim 20 wherein the step of setting the interval comprises reducing the interval in response to sensing in a transmit buffer data to be transmitted over the data link.

22. A method according to any preceding claim wherein the step of varying the predetermined value comprises reducing the predetermined value in response to sensing that a congestion mode of the protocol is in operation over the data link.

23. A method according to any preceding claim wherein the protocol is TCP.

24. Apparatus for controlling the processing of data in a data processor, the data processor being connectable to a further device over a data link, the apparatus comprising:

an element arranged to receive data; and

a control entity arranged to, if a set interval has elapsed following the receipt of data, determine whether processing of the received data in accordance with a data transfer protocol has begun, and, if it has not, trigger such processing of the received data by a protocol processing element;

wherein the control entity is further arranged to sense conditions pertaining to the data link and set the interval in dependence on the sensed conditions.

25. A control entity for use with a data processor, the data processor being connectable to a further device over a data link, and comprising an element arranged to receive data, the control entity being arranged to:

if a set interval has elapsed following the receipt of data by the said element, determine whether processing of the received data in accordance with a data transfer protocol has begun, and, if it has not, trigger such processing of the received data by a protocol processing element; and

sense conditions pertaining to the data link and set the interval in dependence on the sensed conditions.

26. A method for bridging between a first data link carrying data units of a first data protocol and a second data link for carrying data units of a second protocol by means of a bridging device, the first and second protocols being such that data units of each protocol include protocol data and traffic data and the bridging device comprising a first interface entity for interfacing with the first data link, a second interface entity for interfacing with the second data link, a protocol processing entity and a memory accessible by the first interface entity, the second interface entity and the protocol processing entity, the method comprising:

receiving by means of the first interface entity data units of the first protocol, and storing those data units in the memory;

accessing by means of the protocol processing entity the protocol data of data units stored in the memory and thereby performing protocol processing for those data units under the first protocol; and

accessing by means of the second interface entity the traffic data of data units stored in the memory and thereby transmitting that traffic data over the second data link in data units of the second data protocol.

27. A method as claimed in claim 26, wherein the protocol processing entity is arranged to perform protocol processing for the data units stored in the memory without it accessing the traffic data of those units stored in the memory.

28. A method as claimed in claim 27, wherein the first protocol is such that protocol data of a data unit of the first protocol includes check data that is a function of the traffic data of the data unit, and the method comprises:

applying the function by means of the first entity to the content of a data unit of the first protocol received by the first interface entity to calculate first check data;

transmitting the first check data to the protocol processing entity; and

comparing by means of the protocol processing entity the first check data calculated for a data unit with the check data included in the protocol data of that data unit.

29. A method as claimed in any of claims 26 to 28, wherein:

the memory comprises a plurality of buffer regions;

the first interface entity, the second interface entity and the protocol processing entity are each arranged to access a buffer region only when they have control of it; and the method comprises:

the first interface entity storing a received data unit of the first protocol in a buffer of which it has control and subsequently passing control of that buffer to the protocol processing entity;

the protocol processing entity passing control of a buffer to the second interface entity when it has performed protocol processing of the or each data unit stored in that buffer; and

the second interface entity passing control of a buffer to the first interface entity when it has transmitting the traffic data contained in that buffer over the second data link in data units of the second data protocol.

30. A method as claimed in any of claims 26 to 29, comprising:

generating by means of the protocol processing entity protocol data of the second protocol for the data units to be transmitted under the second protocol;

communicating that protocol data to the second interface entity; and

the second interface entity including that protocol data in the said data units of the second protocol.

31. A method as claimed in claim 30, wherein the second protocol is such that protocol data of a data unit of the second protocol includes check data that is a function of the traffic data of the data unit, and the method comprises:

applying the function by means of the second interface entity to the content of a data unit of the second protocol to be transmitted by the second interface entity to calculate first check data;

combining that check data with protocol data received from the protocol processing entity to form second protocol data; and

the second interface entity including the second protocol data in the said data units of the second protocol.

32. A method as claimed in any of claims 26 to 31, wherein one of the first and second protocols is TCP.

33. A method as claimed in any of claims 26 to 32, wherein one of the first and second protocols is Fibrechannel.

34. A method as claimed in any of claims 26 to 32, wherein the first and second protocols are the same.

35. A method as claimed in any of claims 26 to 34, wherein the first and second interface entities each communicate with the respective data link via a respective hardware interface.

36. A method as claimed in any of claims 26 to 35, wherein the first and second interface entities each communicate with the respective data link via the same hardware interface.

37. A method as claimed in any of claims 26 to 36, wherein the protocol processing comprises terminating a link of the first protocol.

38. A method as claimed in any of claims 26 to 37, wherein the protocol processing comprises:

inspecting the traffic data of the first protocol;

comparing the traffic data of the first protocol with one or more pre-set rules; and

if the traffic data does not satisfy the rules preventing that traffic data from being transmitted by the second interface entity.

39. A bridging device for bridging between a first data link carrying data units of a first data protocol and a second data link for carrying data units of a second protocol, the first and second protocols being such that data units of each protocol include protocol data and traffic data and the bridging device comprising:

a first interface entity for interfacing with the first data link, a second interface entity for interfacing with the second data link, a protocol processing entity and a memory accessible by the first interface entity, the second interface entity and the protocol processing entity;

the first interface entity being arranged to receive data units of the first protocol, and storing those data units in the memory;

the protocol processing entity being arranged to access the protocol data of data units stored in the memory and thereby perform protocol processing for those data units under the first protocol; and

the second interface entity being arranged to access the traffic data of data units stored in the memory and thereby transmit that traffic data over the second data link in data units of the second data protocol.

40. A data processing system comprising:

a memory comprising a plurality of buffer regions;

an operating system for supporting processing entities running on the data processing system and for restricting access to the buffer regions to one or more entities;

a first interface entity running on the data processing system whereby a first hardware device may communicate with the buffer regions; and

an application entity running on the data processing system;

the first interface entity and the application entity being configured to, in respect of a buffer region to which the operating system permits access by both the interface entity and the application entity, communicate ownership data so as to indicate which of the first interface entity and the application entity may access the buffer region and to access the buffer region only in accordance with the ownership data.

41. A data processing system as claimed in claim 40, wherein the data processing system comprises a second interface entity running on the data processing system whereby a second hardware device may communicate with the buffer regions and the first and second interface entities and the application entity are configured to, in respect of a buffer region to which the operating system permits access by the first and second interface entities and the application entity, communicate ownership data so as to indicate which of the first and second interface entities and the application entity may access each buffer regions and to access each buffer region only in accordance with the ownership data.

42. A data processing system as claimed in claim 41, wherein:

the first interface entity is arranged to, on receiving a data unit, store that data unit in a buffer region that it may access in accordance with the ownership data and to subsequently modify the ownership data such that the application entity may access that buffer region in accordance with the ownership data;

the application entity is arranged to perform protocol processing on data unit(s) stored in a buffer region that it may access in accordance with the ownership

data and to subsequently modify the ownership data such that the second interface entity may access that buffer region in accordance with the ownership data; and

the second interface entity is arranged to transmit at least some of the content of data unit(s) stored in a buffer region that it may access in accordance with the ownership data and to subsequently modify the ownership data such that the application entity may access that buffer region in accordance with the ownership data.

43. A data processing system as claimed in any of claims 40 to 42, wherein the said protocol processing is protocol proceeding in accordance with a first protocol and system is arranged so that the second interface entity can transmit the said content in accordance with a second protocol.

44. A data processing system as claimed in claim 43, wherein one of the first and second protocols is TCP.

45. A data processing system as claimed in claims 43 or 44, wherein one of the first and second protocols is Fibrechannel.

46. A data processing system as claimed in any of claims 43 to 45, wherein the first and second protocols are the same.

47. A data processing system as claimed in any of claims 43 to 46, wherein the protocol processing comprises terminating a link of the first protocol.

48. A method as claimed in any of claims 43 to 46, wherein the protocol processing comprises:

inspecting the traffic data of the first protocol;

comparing the traffic data of the first protocol with one or more pre-set rules; and

if the traffic data does not satisfy the rules preventing that traffic data from being transmitted by the second interface entity.

49. A data processing system as claimed in any of claims 41 to 48, wherein the first and second interface entities are each configured to communicate with a respective data link via a respective hardware interface.

50. A data processing system as claimed in any of claims 41 to 48, wherein the first and second interface entities are each configured to communicate with the respective data link via the same hardware interface.

51. A method for operating a data processing system comprising:

a memory comprising a plurality of buffer regions;

an operating system for supporting processing entities running on the data processing system and for restricting access to the buffer regions to one or more entities;

a first interface entity running on the data processing system whereby a first hardware device may communicate with the buffer regions; and

an application entity running on the data processing system;

the method comprising, in respect of a buffer region to which the operating system permits access by both the interface entity and the application entity, communicating ownership data by means of the first interface entity and the application entity so as to indicate which of the first interface entity and the application entity may access the buffer region and to access the buffer region only in accordance with the ownership data.

52. A protocol processing entity for operation in a bridging device for bridging between a first data link carrying data units of a first data protocol and a second data link for carrying data units of a second protocol by means of a bridging device, the first and second protocols being such that data units of each protocol include protocol data and traffic data and the protocol processing entity being arranged to cause a processor of the bridging device to perform protocol processing for data units stored in the memory without it accessing the traffic data of those units stored in the memory.

53. A data carrier carrying software defining a computer program for implementing a protocol processing entity as claimed in claim 52.

54. A computer system comprising:

hardware including a data interface for interfacing between the computer system and a data source;

a memory;

a first operating system capable of communicating with the hardware; and

a second operating system capable of supporting a user-level application and being configured to communicate with the hardware via the first operating system, the second operating system being capable of allocating a region of the memory for use as a buffer by such a user-level application;

wherein the data interface is configurable to associate a predetermined data format with a region of the memory that has been allocated for use as a buffer by a user-level application supported by the second operating system so as to, on subsequently receiving from the data source a data message of that format, automatically store data of that message in that region of the memory without it passing via the first or second operating systems.

55. A computer system as claimed in claim 54, wherein the first operating system is capable of serving as an interface between multiple further operating systems and the hardware.

56. A computer system as claimed in claim 55, wherein the first operating system is a hypervisor.

57. A computer system as claimed in any of claims 54 to 56, wherein the data interface is a network interface and the data source is a data network.

58. A computer system as claimed in any of claims 54 to 57, wherein the data interface has access to a data store for storing a plurality of indications of respective

data formats and corresponding to each one an indication of a destination, and the data interface is arranged to, on receiving a data message from the data source identify whether the format of the data message matches a data format an indication of which is stored in the data store, and if it does to pass data of that message to that destination.

59. A computer system as claimed in any of claims 54 to 58, wherein the data format is at least partially defined by a destination address.

60. A computer system as claimed in claim 59, wherein the address is an internet layer address.

61. A computer system as claimed in claim 60, wherein the address is an IP (internet protocol) address.

62. A computer system as claimed in any of claims 54 to 61, wherein the data format is at least partially defined by a data port.

63. A computer system as claimed in any of claims 54 to 62, wherein the data message is a data packet.

64. A computer system as claimed in any of claims 54 to 63 wherein the data interface is configurable automatically by the first operating system to associate the predetermined data format with the region of the memory that has been allocated for use as a buffer by a user-level application supported by the second operating system.

65. A computer system as claimed in any of claims 54 to 64 wherein the data interface is configurable automatically by the second operating system to associate the predetermined data format with the region of the memory that has been allocated for use as a buffer by a user-level application supported by the second operating system.

66. A computer system as claimed in any of claims 54 to 65, wherein the data interface is arranged to, on receiving a data message from the data source identify whether it is configured to associate the format of that message with a region of the memory and if it has to automatically store data of that message in that region of the memory without it passing via the first or second operating systems.

67. A computer system as claimed in any of claims 54 to 66, wherein one of the first operating system, the second operating system and the data interface is arranged to deconfigure the data interface from associating a message format with a region of the memory when a pre-set time has elapsed from when the interface was configured to associate that message format with that region of the memory.

68. A computer system as claimed in any of claims 54 to 67, wherein one of the first operating system, the second operating system and the data interface is arranged to deconfigure the data interface from associating a message format with a region of the memory in response to sensing that data traffic conditions match one or more predefined criteria.

69. A computer system as claimed in of claims 54 to 68, wherein the said traffic criteria include the criterion that the flow of received data of the message format is below a pre-set amount in a pre-set time.

70. A computer system comprising:

hardware including a data interface for interfacing between the computer system and a data source;

a memory;

a first operating system capable of communicating with the hardware; and

a second operating system capable of supporting a user-level application;

wherein at least one of the first and second operating systems is arranged to detect that data of a first data message received by the data interface from the data source has been directed to a destination via the first operating system, and in response to

detecting that to configure the data interface to direct data of subsequent data messages having a data format in common with the first data message to that destination without it passing via the second operating system.

71. A computer system as claimed in claim 70, wherein the first operating system is capable of serving as an interface between multiple further operating systems and the hardware.

72. A computer system as claimed in claim 71, wherein the first operating system is a hypervisor.

73. A computer system as claimed in any of claims 70 to 72, wherein the data interface is a network interface and the data source is a data network.

74. A computer system as claimed in any of claims 70 to 73, wherein the data interface has access to a data store for storing a plurality of indications of respective data formats and corresponding to each one an indication of a destination, and the data interface is arranged to, on receiving a data message from the data source identify whether the format of the data message matches a data format an indication of which is stored in the data store, and if it does to pass data of that message to that destination.

75. A computer system as claimed in any of claims 70 to 74, wherein the data format is at least partially defined by a destination address.

76. A computer system as claimed in claim 75, wherein the address is an internet layer address.

77. A computer system as claimed in claim 76, wherein the address is an IP (internet protocol) address.

78. A computer system as claimed in any of claims 54 to 77, wherein the data format is at least partially defined by a data port.

79. A computer system as claimed in any of claims 54 to 78, wherein the data message is a data packet.

80. A computer system as claimed in any of claims 70 to 79, wherein the data interface is arranged to, on receiving a data message from the data source identify whether it is configured to associate the format of that message with a region of the memory and if it has to automatically store data of that message in that region of the memory without it passing via the first or second operating systems.

81. A computer system as claimed in any of claims 70 to 80, wherein one of the first operating system, the second operating system and the data interface is arranged to deconfigure the data interface from associating a message format with a region of the memory when a pre-set time has elapsed from when the interface was configured to associate that message format with that region of the memory.

82. A computer system as claimed in of claims 70 to 81, wherein one of the first operating system, the second operating system and the data interface is arranged to deconfigure the data interface from associating a data format with a region of the memory in response to sensing that data traffic conditions match one or more predefined criteria.

83. A computer system as claimed in claim 82, wherein the said traffic criteria include the criterion that the flow of received data of the data format is below a pre-set amount in a pre-set time.

84. A computer system as claimed in any of claims 70 to 83, wherein the second operating system is arranged to perform the step of configuring the data interface to direct data of subsequent data messages having a data format in common with the

first data message only when it detects that the flow of received data of that data format is above a pre-set amount in a pre-set time.

85. A computer system as claimed in any of claims 70 to 84, wherein the second operating system is arranged to perform the step of configuring the data interface to direct data of subsequent data messages having a data format in common with the first data message only for data formats of one or more pre-set types.

86. A computer system as claimed in claim 85, wherein the said pre-set types are each defined by respective port numbers or ranges of port numbers.

87. A data interface that is capable of operation in a computer system as claimed in any of claims 54 to 86, and is configurable to associate a predetermined data format with a region of the memory of the computer system external to the data interface that has been allocated for use as a buffer by a user-level application supported by the second operating system so as to, on subsequently receiving from the data source a data message of that format, automatically store data of that message in that region of the memory without it passing via the first or second operating systems.

88. A computer program for acting as a driver for communicating with a data interface in a computer system as claimed in any of claims 54 to 87, the computer program being such as to provide an operating system with functionality to, in response to detection in response to detection that data of a first data message received by the data interface from the data source has been directed to a destination via another operating system, configure the data interface to direct data of subsequent data messages having a data format in common with the first data message to that destination without it passing via the second operating system.

89. A method for operating a computer system comprising hardware including a data interface for interfacing between the computer system and a data source, a memory, a first operating system capable of communicating with the hardware; and a second

operating system capable of supporting a user-level application and being configured to communicate with the hardware via the first operating system, the second operating system being capable of allocating a region of the memory for use as a buffer by such a user-level application;

the method comprising:

configuring the data interface to associate a predetermined data format with a region of the memory that has been allocated for use as a buffer by a user-level application supported by the second operating system; and

the data interface on subsequently receiving from the data source a data message of that format, automatically storing data of that message in that region of the memory without it passing via the first or second operating systems.

90. A method for operating a computer system comprising: hardware including a data interface for interfacing between the computer system and a data source; a memory; a first operating system capable of communicating with the hardware; and a second operating system capable of supporting a user-level application;

the method comprising:

detecting by means of at least one of the first and second operating systems that data of a first data message received by the data interface from the data source has been directed to a destination via the first operating system; and

in response to detecting that configuring the data interface by means of that operating system to direct data of subsequent data messages having a data format in common with the first data message to that destination without it passing via the second operating system.

FIG. 1

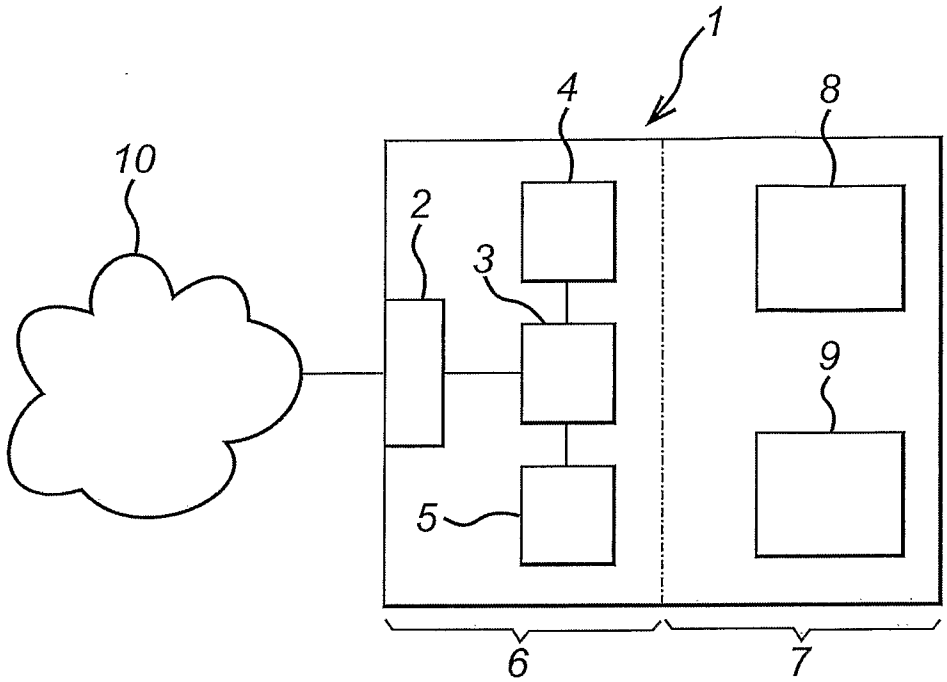
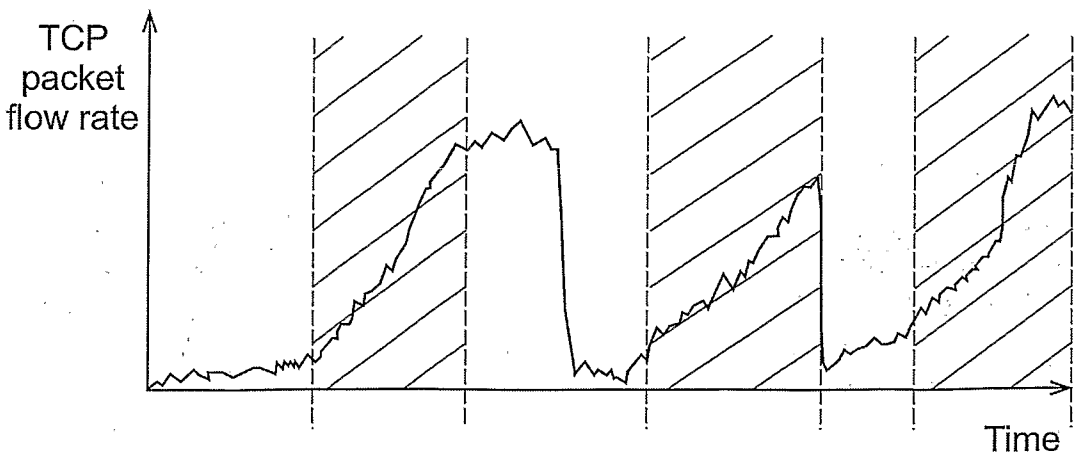


FIG. 4



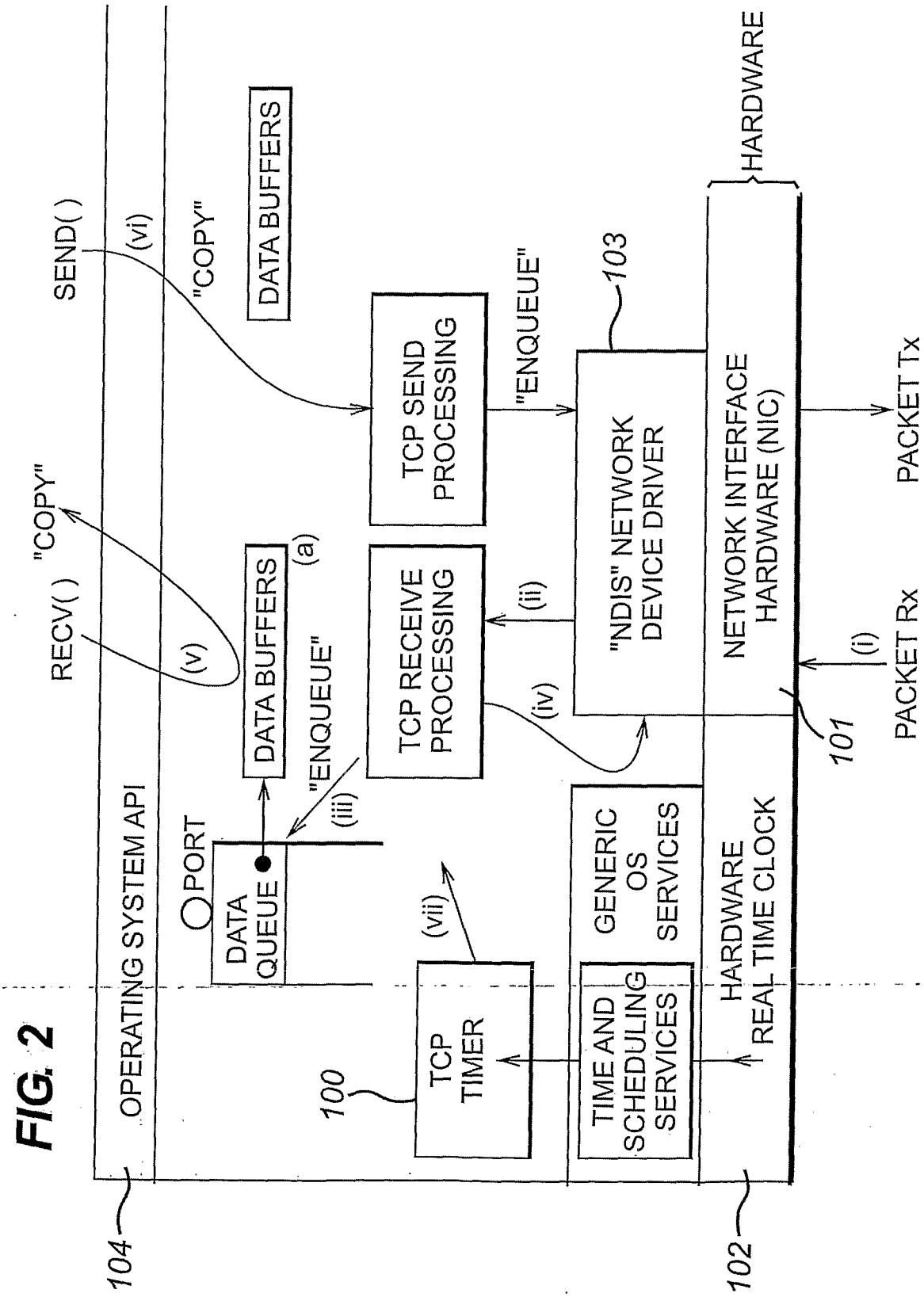

OPERATING SYSTEM API
SEND( )
(vi)
"COPY"
DATA BUFFERS
TCP SEND PROCESSING
"ENQUEUE"
103
HARDWARE
"COPY"
RECV( )
(v)
DATA BUFFERS
(a)
"ENQUEUE"
(iii)
TCP RECEIVE PROCESSING
(ii)
"NDIS" NETWORK DEVICE DRIVER
NETWORK INTERFACE HARDWARE (NIC)
PACKET Tx
(iv)
(i)
PACKET Rx
PORT
DATA QUEUE
(vii)
GENERIC OS SERVICES
101
100
TCP TIMER
TIME AND SCHEDULING SERVICES
HARDWARE REAL TIME CLOCK
104
102

FIG. 2

FIG. 3a

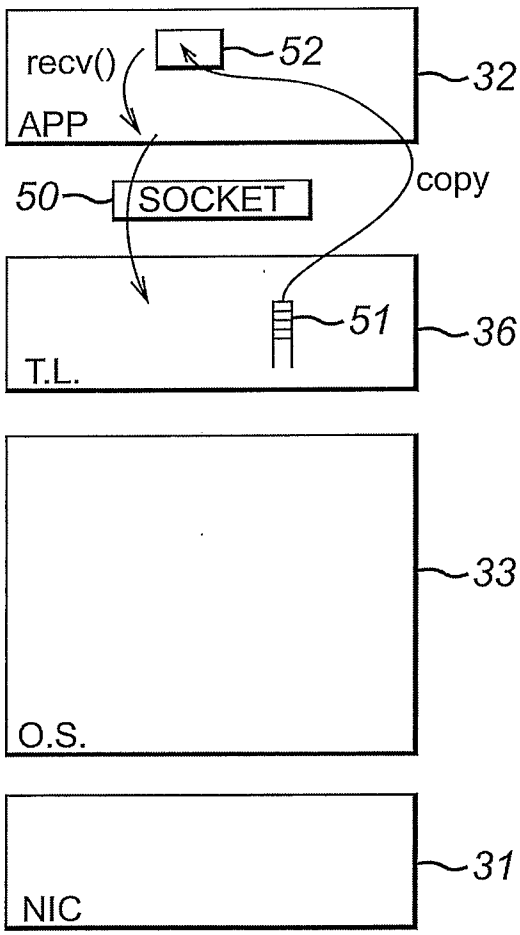


FIG. 3b

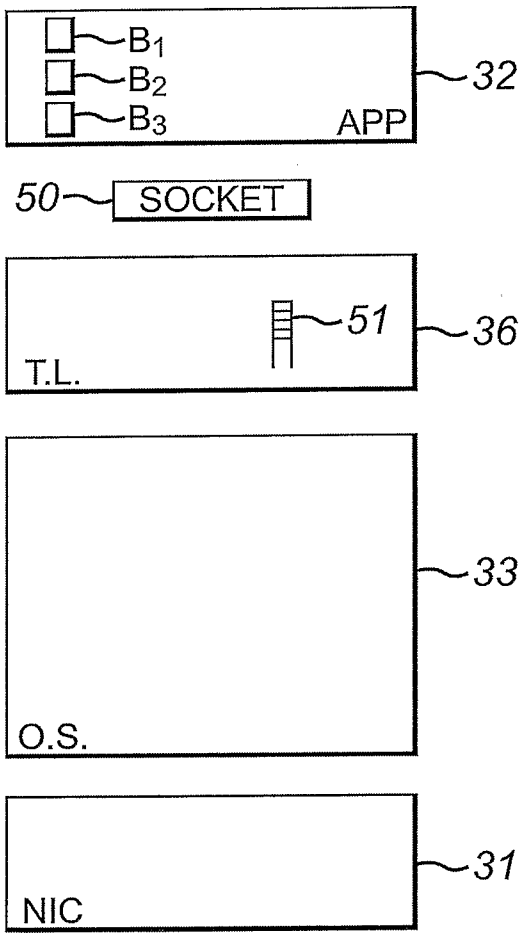


FIG. 5

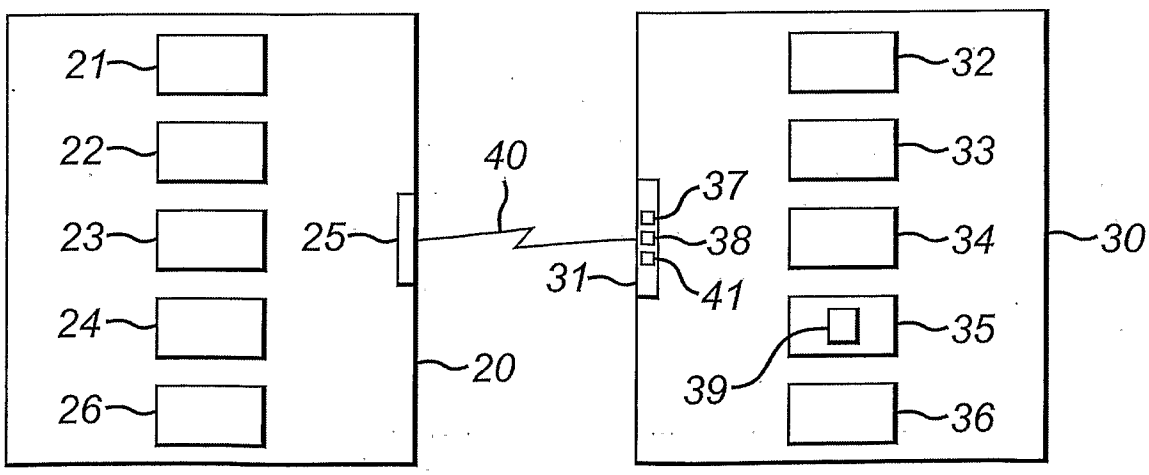


FIG. 6

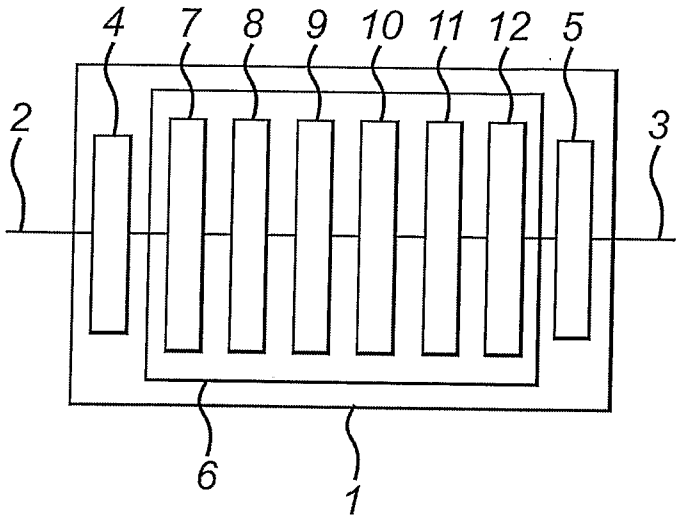


FIG. 7

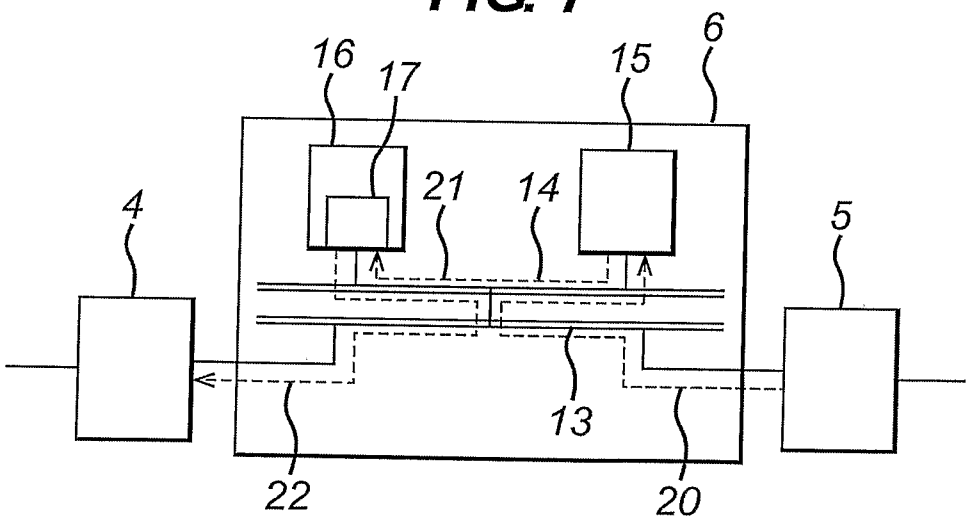


FIG. 8

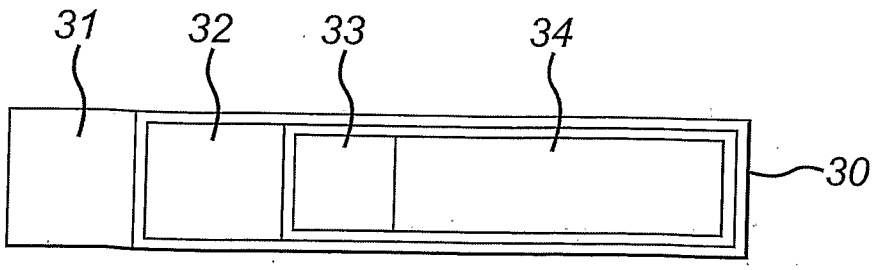


FIG. 9

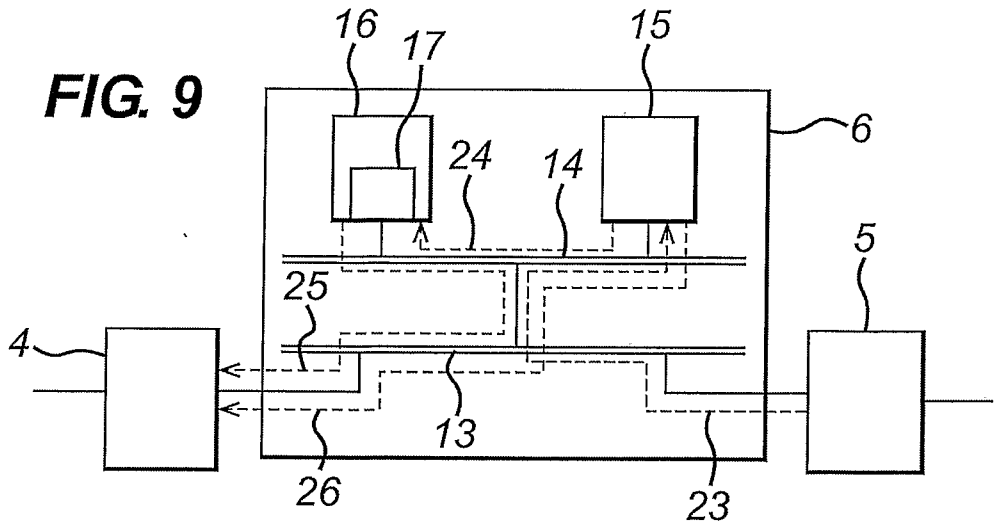


FIG. 10

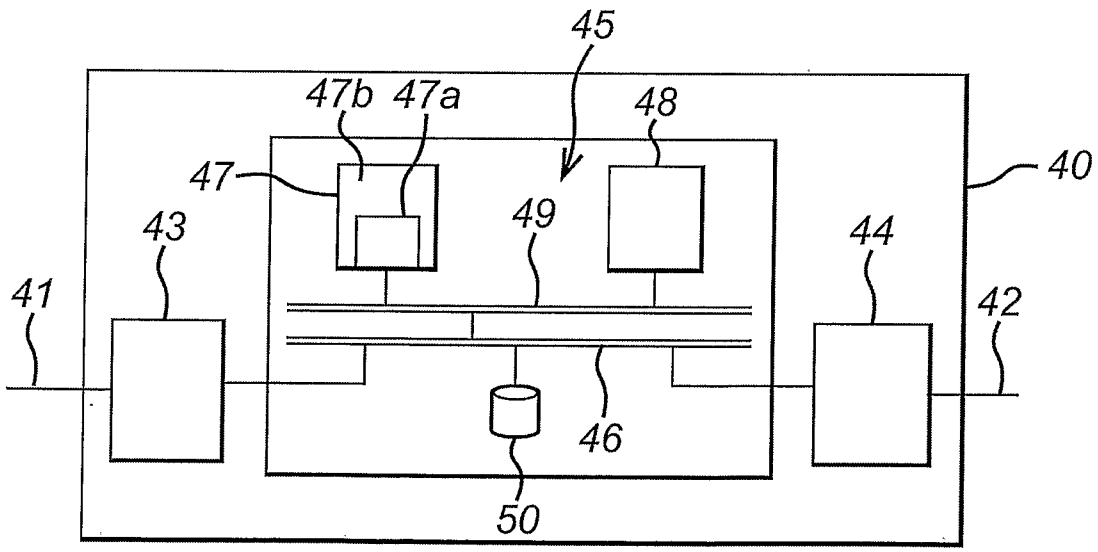


FIG. 11

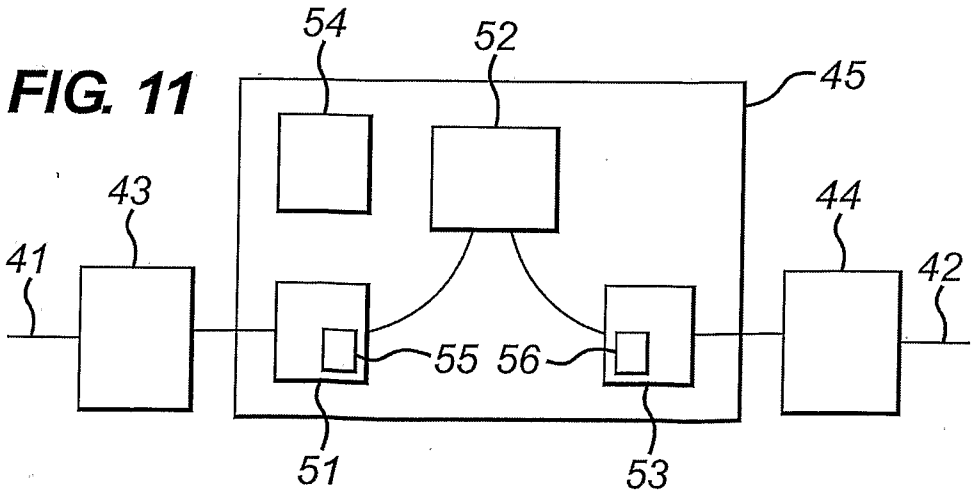


FIG. 12

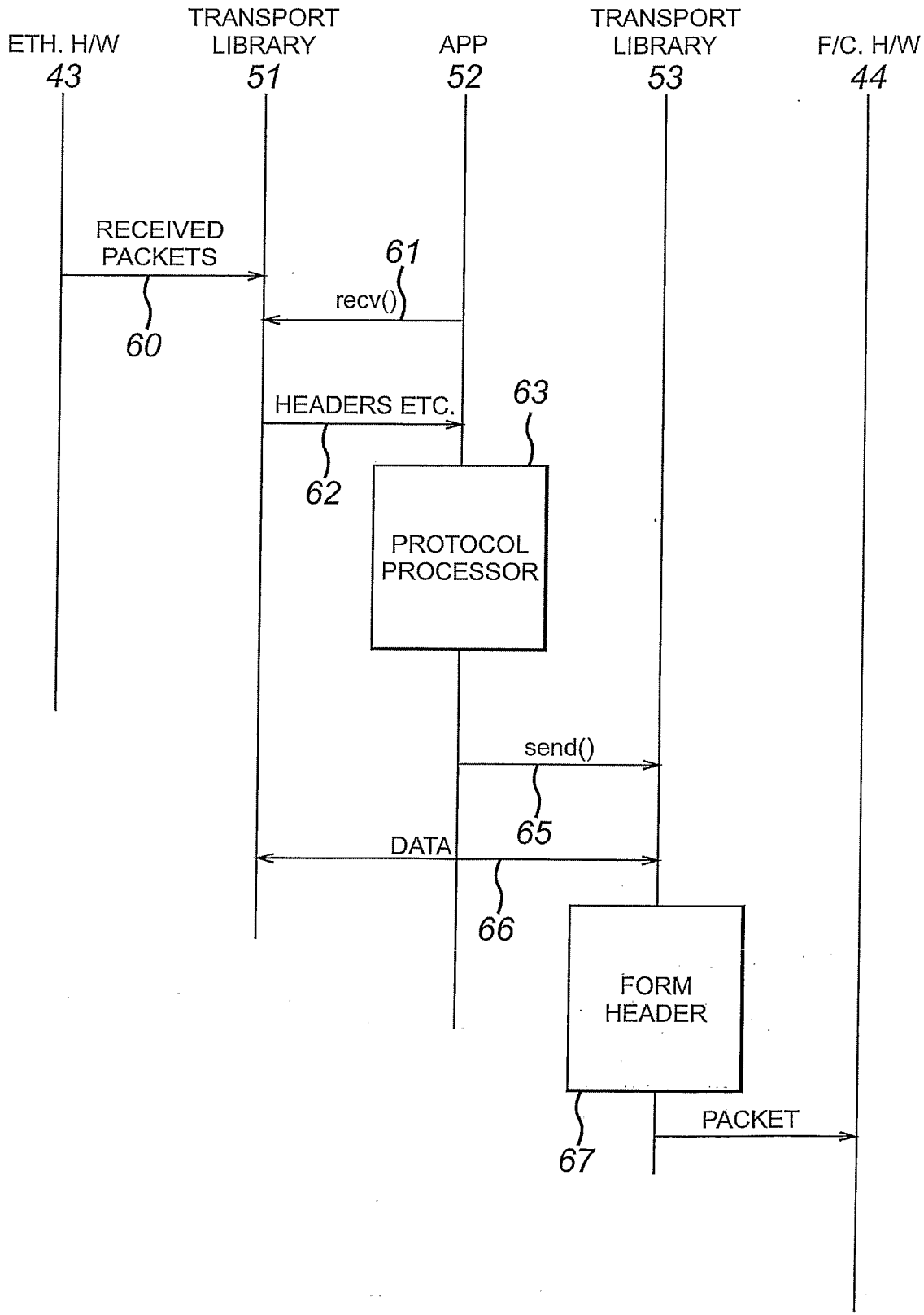

ETH. H/W
43
TRANSPORT LIBRARY
51
APP
52
TRANSPORT LIBRARY
53
F/C. H/W
44
RECEIVED PACKETS
60
61
recv()
HEADERS ETC.
62
63
PROTOCOL PROCESSOR
send()
65
DATA
66
FORM HEADER
67
PACKET

FIG. 13

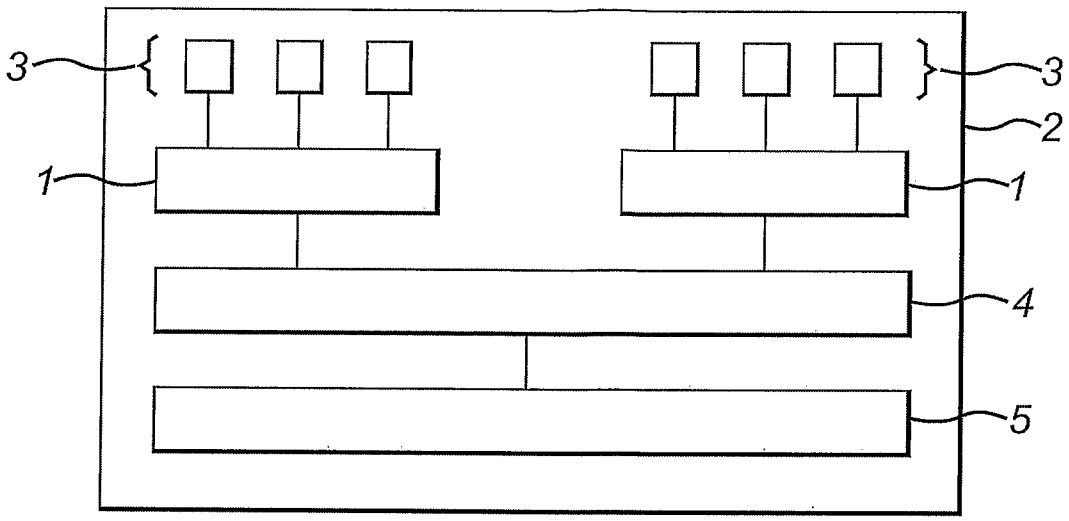


FIG. 14

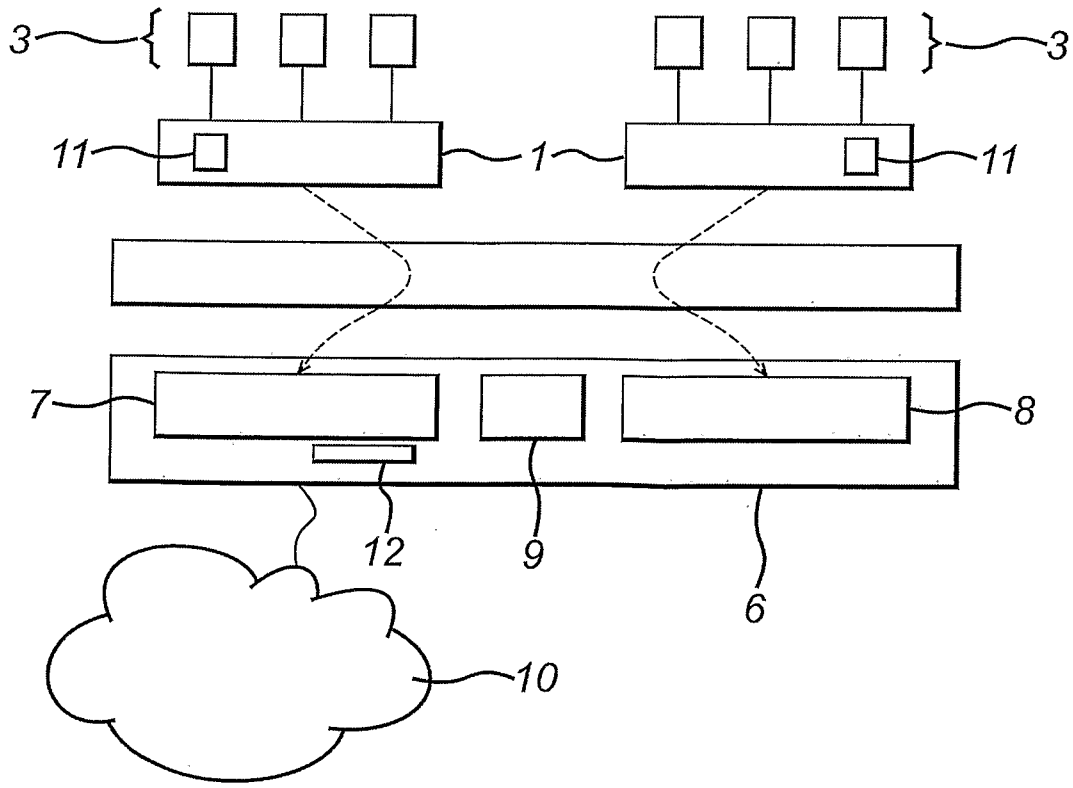


FIG. 15

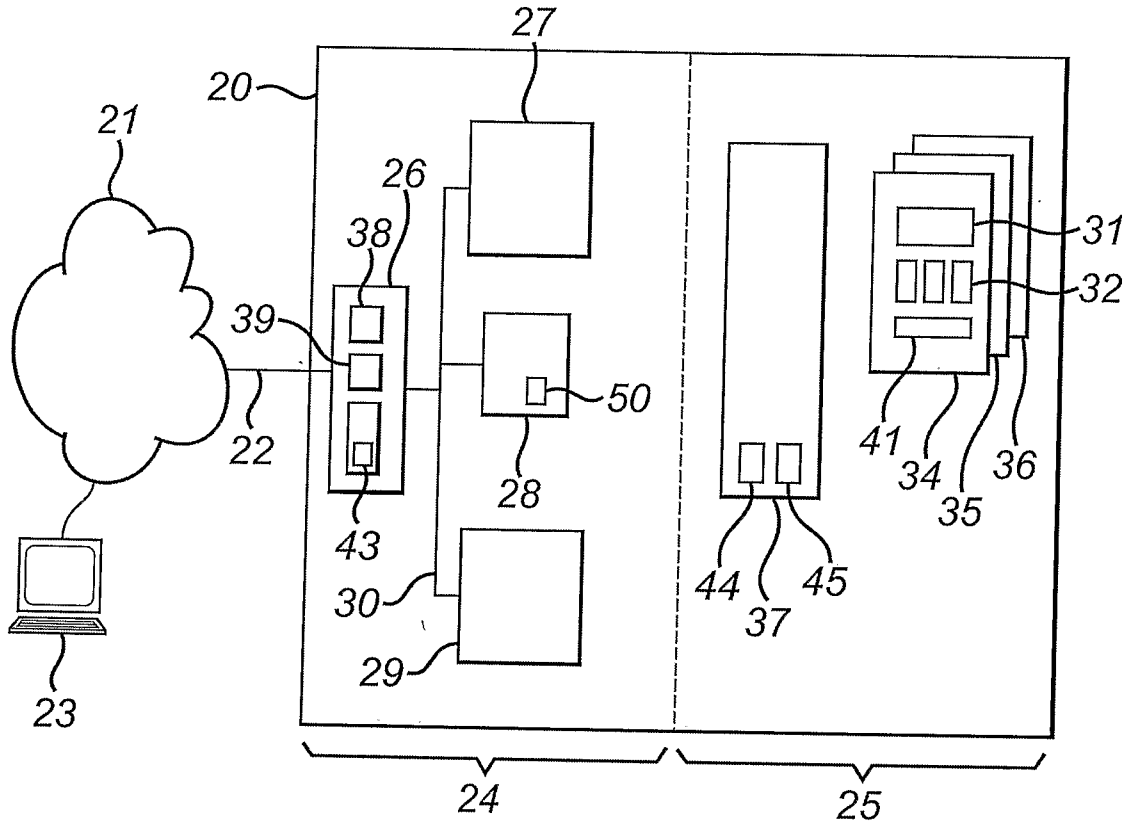

27
20
21
26
38
39
22
50
28
43
30
29
23
31
32
41
34
35
36
44
37
45
24
25

FIG. 16

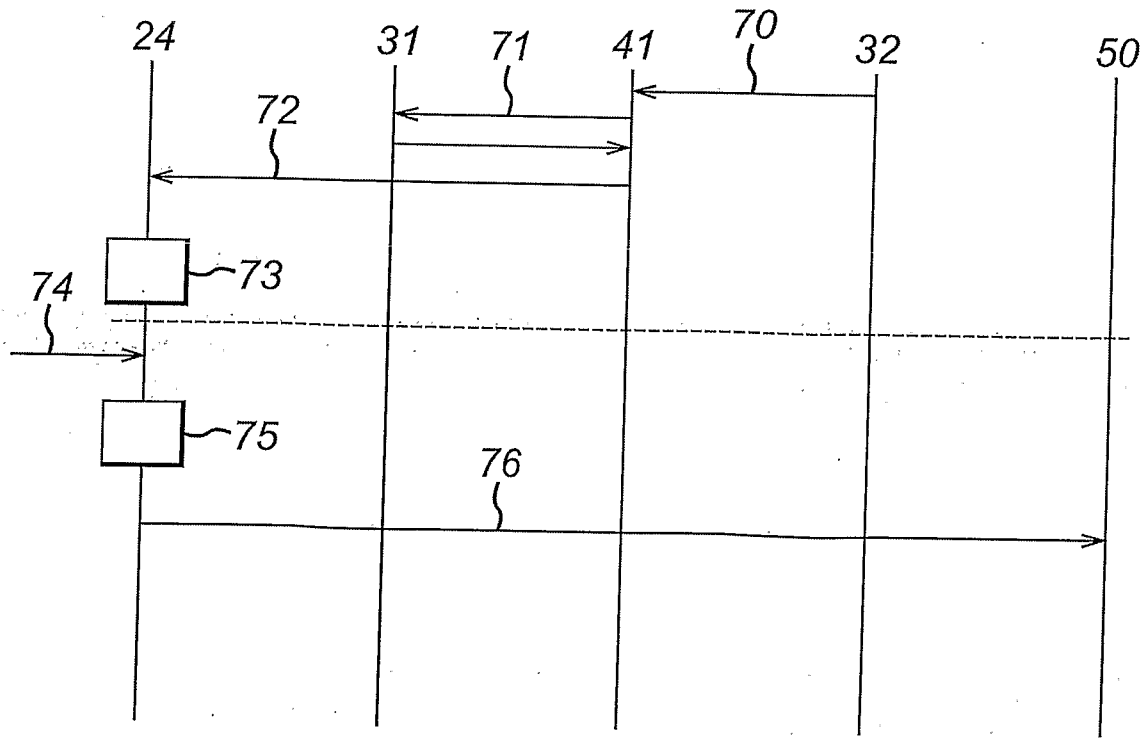

24
31
71
41
70
32
50
72
74
73
75
76

FIG. 17

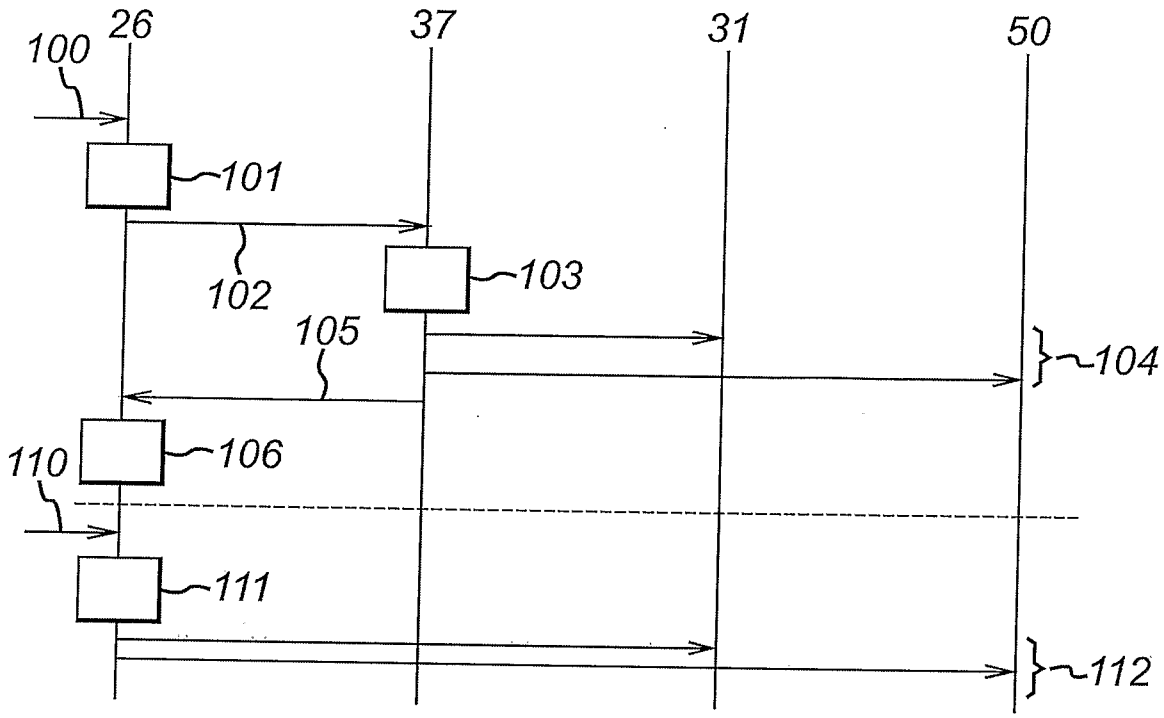
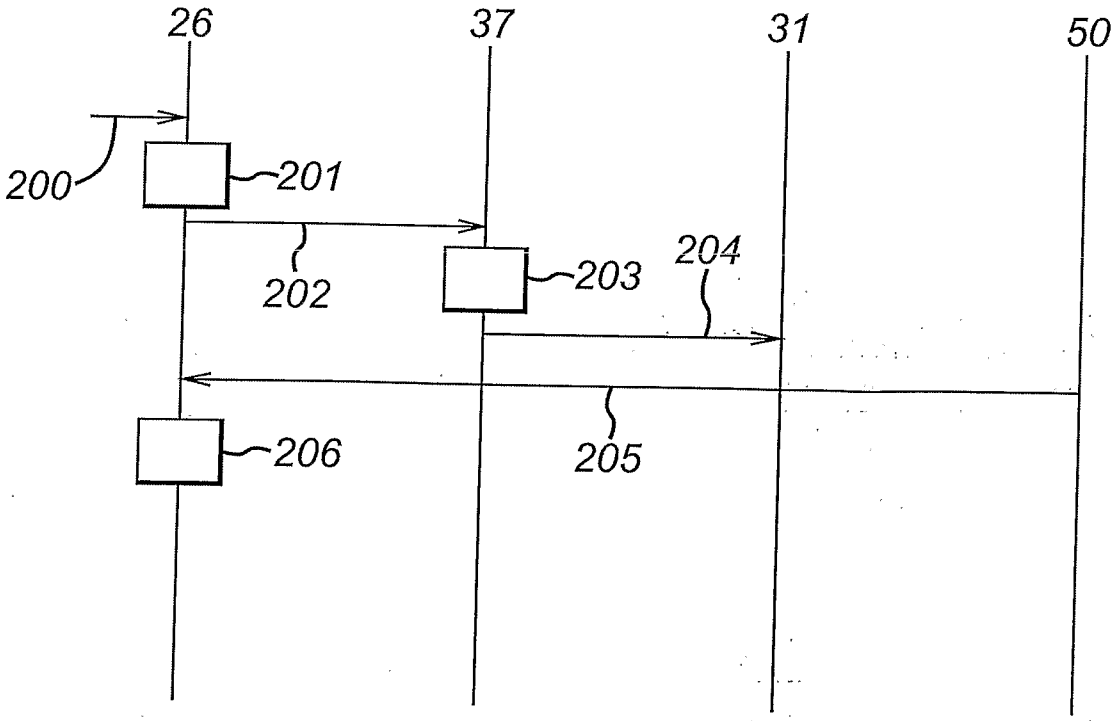

100
26
37
31
50
101
102
103
105
104
110
106
111
112

FIG. 18


26
37
31
50
200
201
202
203
204
205
206