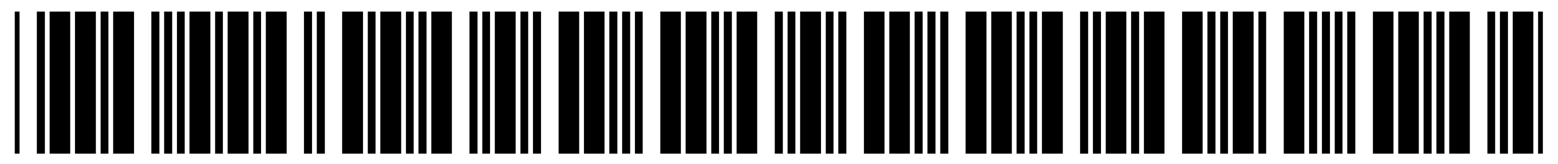


(19) **United States**

(12) **Patent Application Publication** (10) **Pub. No.: US 2002/0026502 A1**

Phillips et al. (43) **Pub. Date: Feb. 28, 2002**

(54) **NETWORK SERVER CARD AND METHOD FOR HANDLING REQUESTS RECEIVED VIA A NETWORK INTERFACE**

(76) Inventors: **Robert C. Phillips**, Northbrook, IL (US); **Caitlin B. Bestler**, Chicago, IL (US)

Correspondence Address:
LEYDIG VOIT & MAYER, LTD
TWO PRUDENTIAL PLAZA, SUITE 4900
180 NORTH STETSON AVENUE
CHICAGO, IL 60601-6780 (US)

(21) Appl. No.: **09/928,235**

(22) Filed: **Aug. 10, 2001**

Related U.S. Application Data

(63) Continuation-in-part of application No. 09/638,774, filed on Aug. 15, 2000.

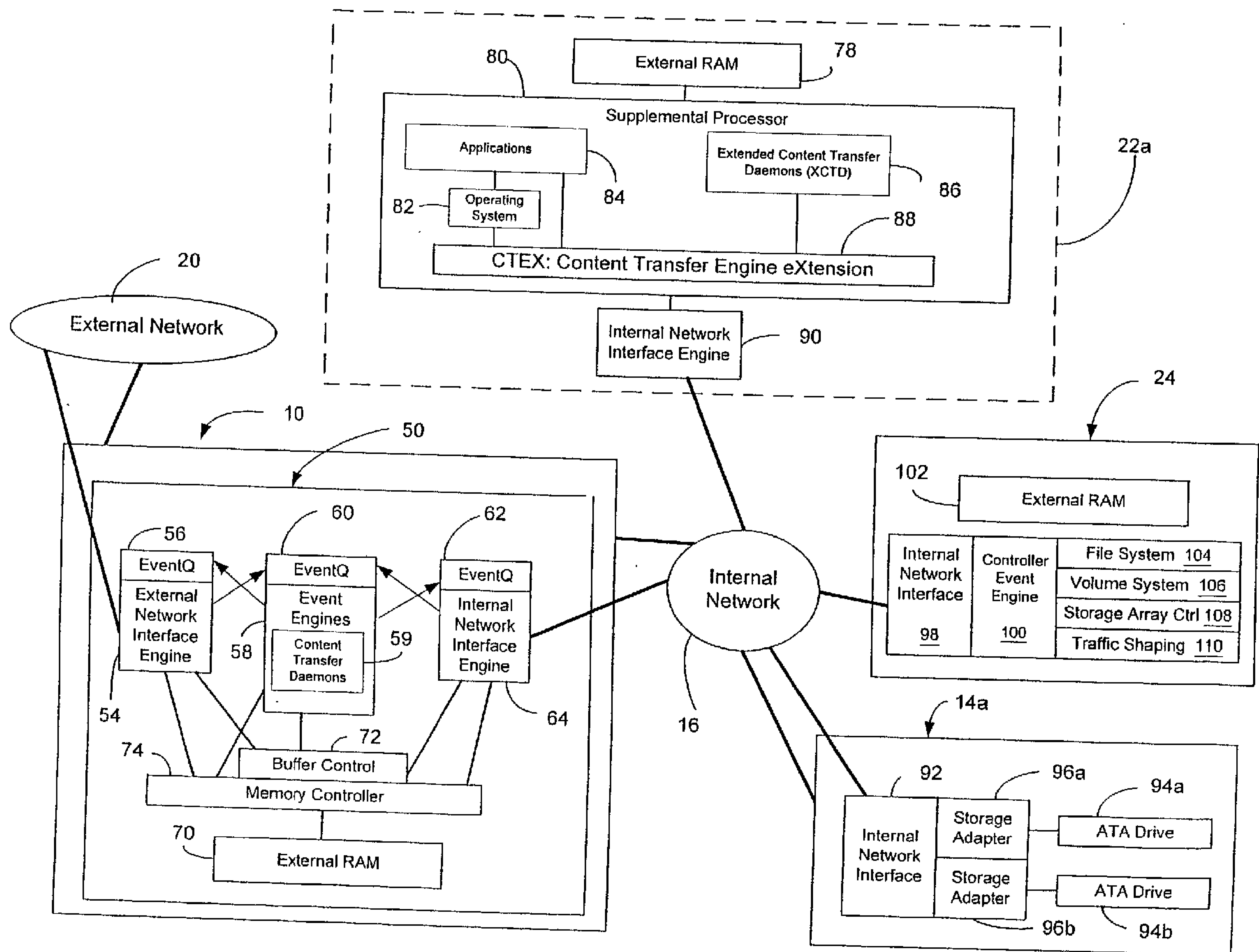
Publication Classification

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

(52) **U.S. Cl.** **709/219**; 709/249

(57) **ABSTRACT**

An information asset (e.g., audio/video data) server system and a set of steps performed by the data server system are disclosed that facilitate efficient handling of a potentially large workload arising from request messages received from users via a communicatively coupled network link. The network data server system comprises content transfer node including an external network interface and a set of event engines. A workload request received by the interface is delegated to one of the set of event engines. The delegated event engine executes the request (or a portion thereof) based upon the request type. In an embodiment of the invention further requests that are part of the same logically related group of communications are identified explicitly or implicitly by header fields and associated with a single set of context data and state tracking maintained by the content transfer node.



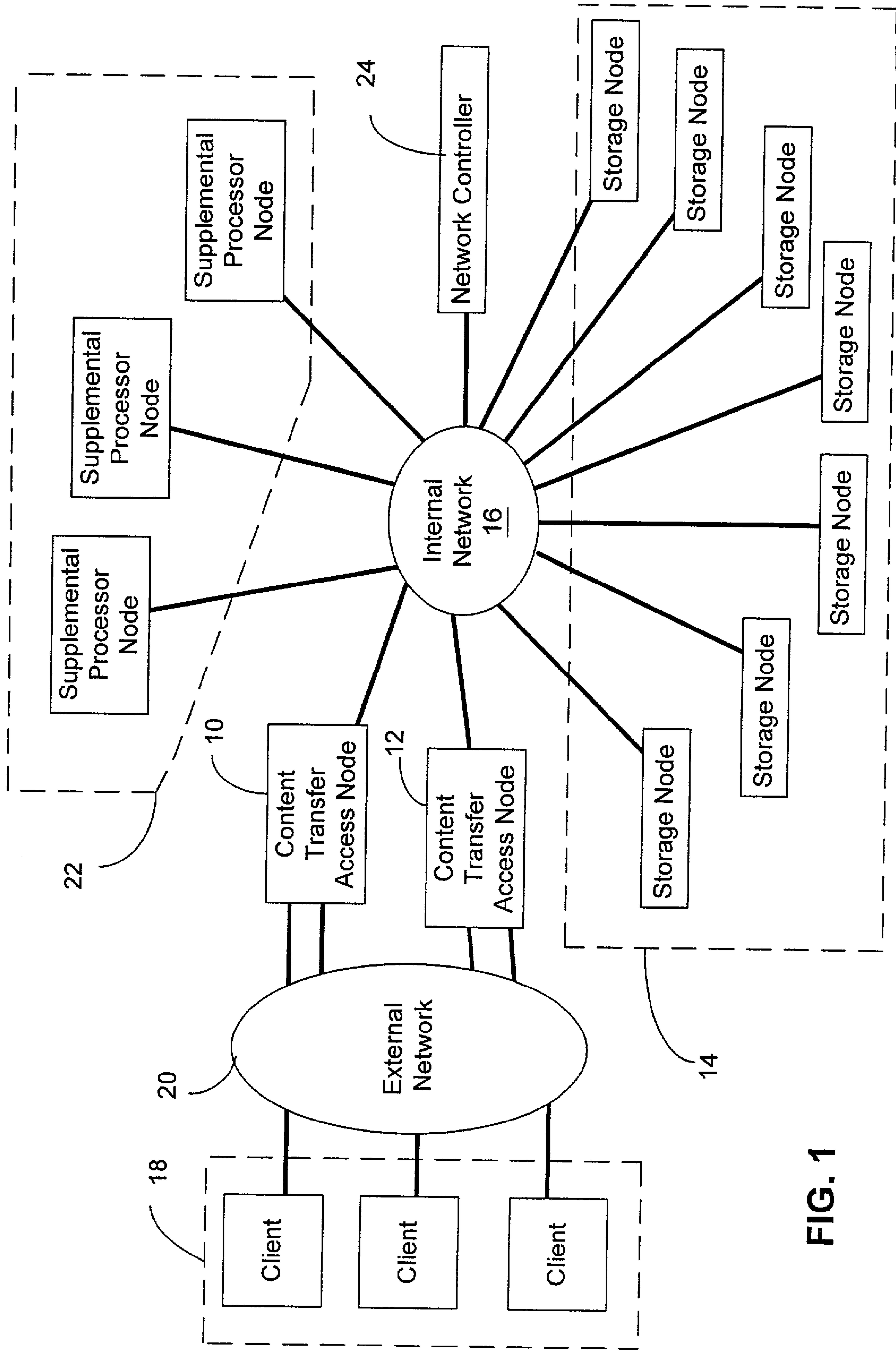


FIG. 1

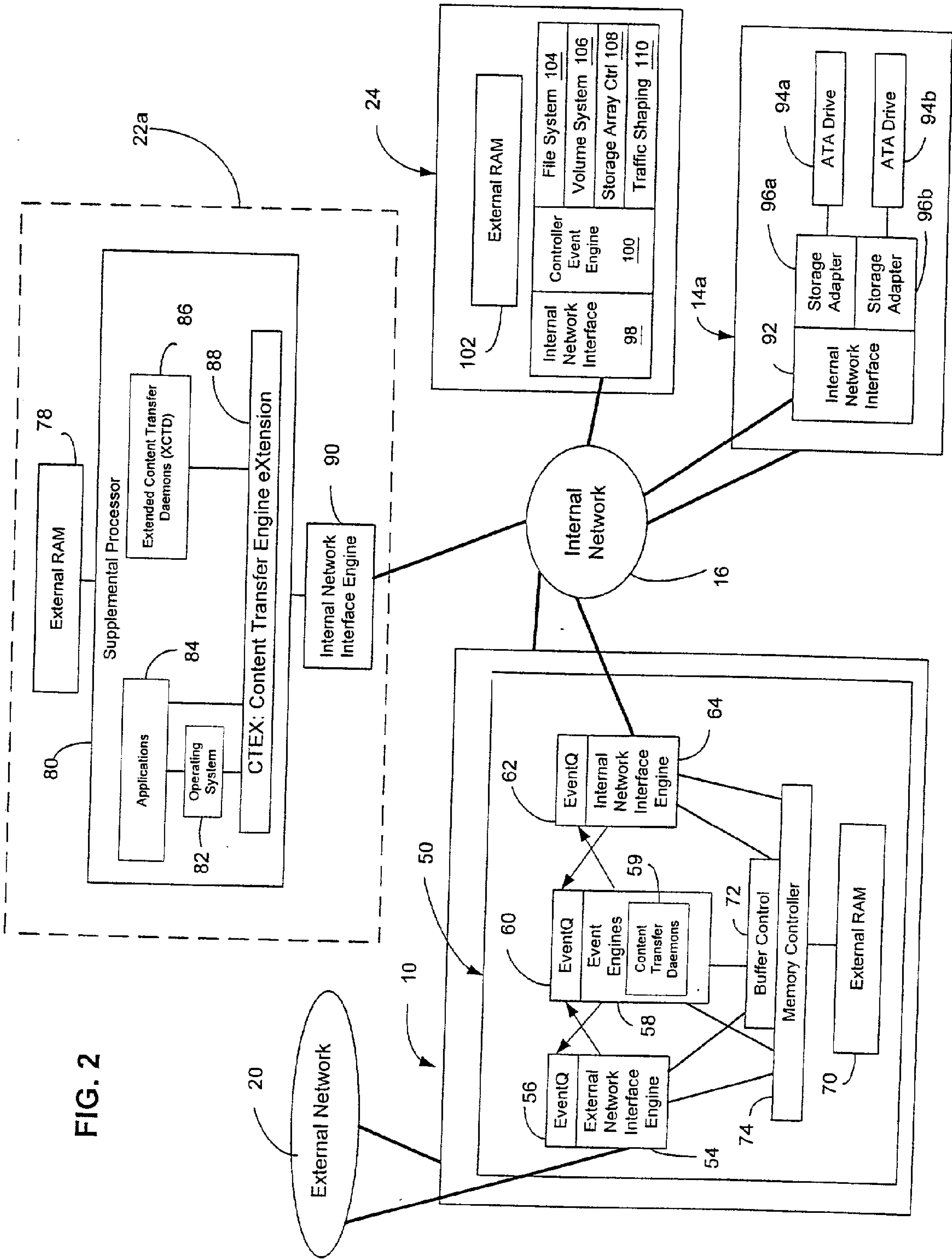


FIG. 2
22a
External RAM
78
Supplemental Processor
Extended Content Transfer Daemons (XCTD)
86
88
CTEX: Content Transfer Engine eXtension
84
Applications
Operating System
80
82
Internal Network Interface Engine
90
24
External RAM
102
Internal Network Interface
98
Controller Event Engine
100
File System 104
Volume System 106
Storage Array Ctrl 108
Traffic Shaping 110
14a
92
Internal Network Interface
Storage Adapter
Storage Adapter
96a
96b
ATA Drive
ATA Drive
94a
94b
Internal Network
16
64
62
EventQ
Internal Network Interface Engine
50
60
EventQ
Event Engines
Content Transfer Daemons
59
72
Buffer Control
Memory Controller
External RAM
10
58
56
EventQ
External Network Interface Engine
54
74
70
20
External Network

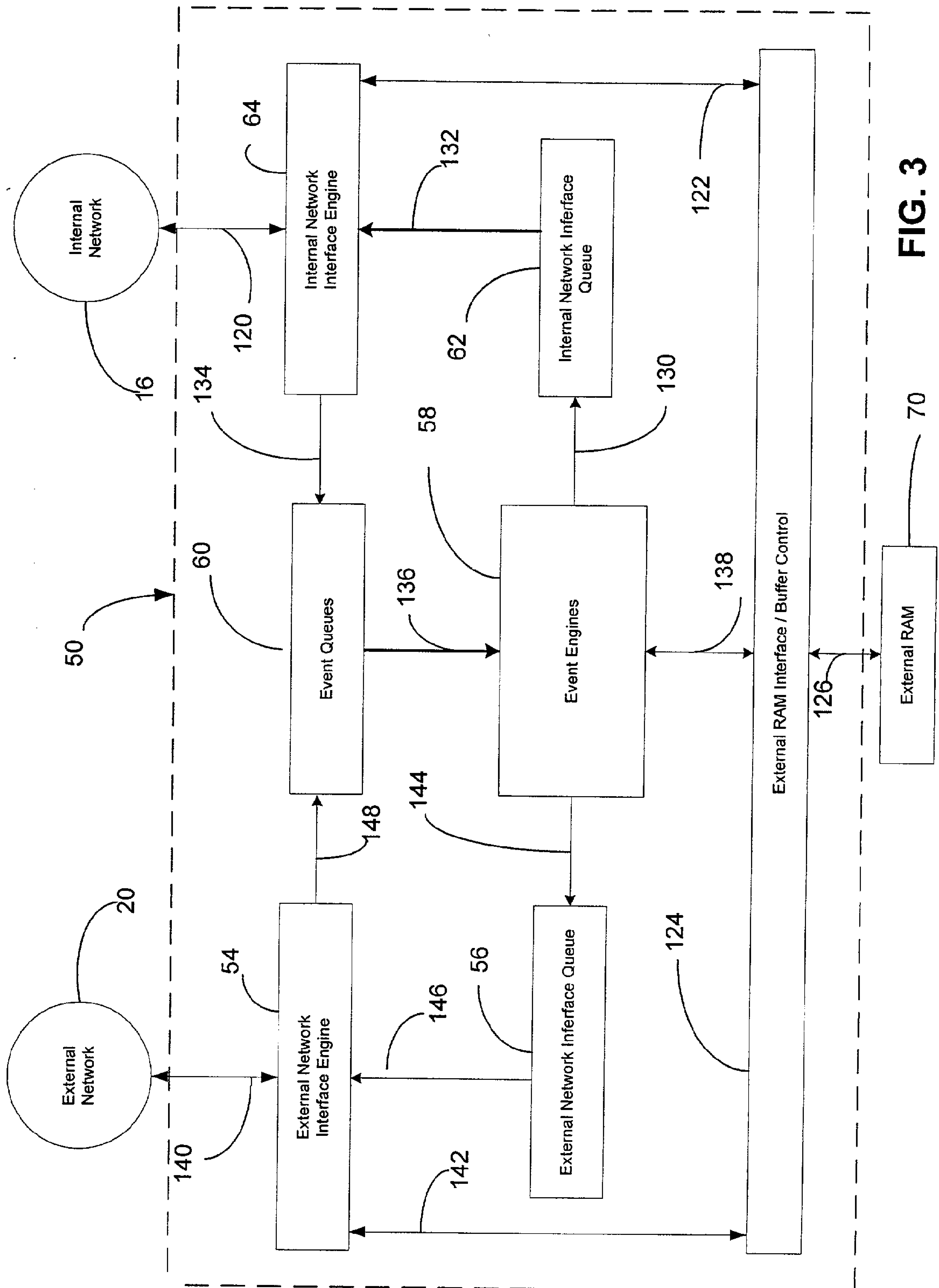


FIG. 3

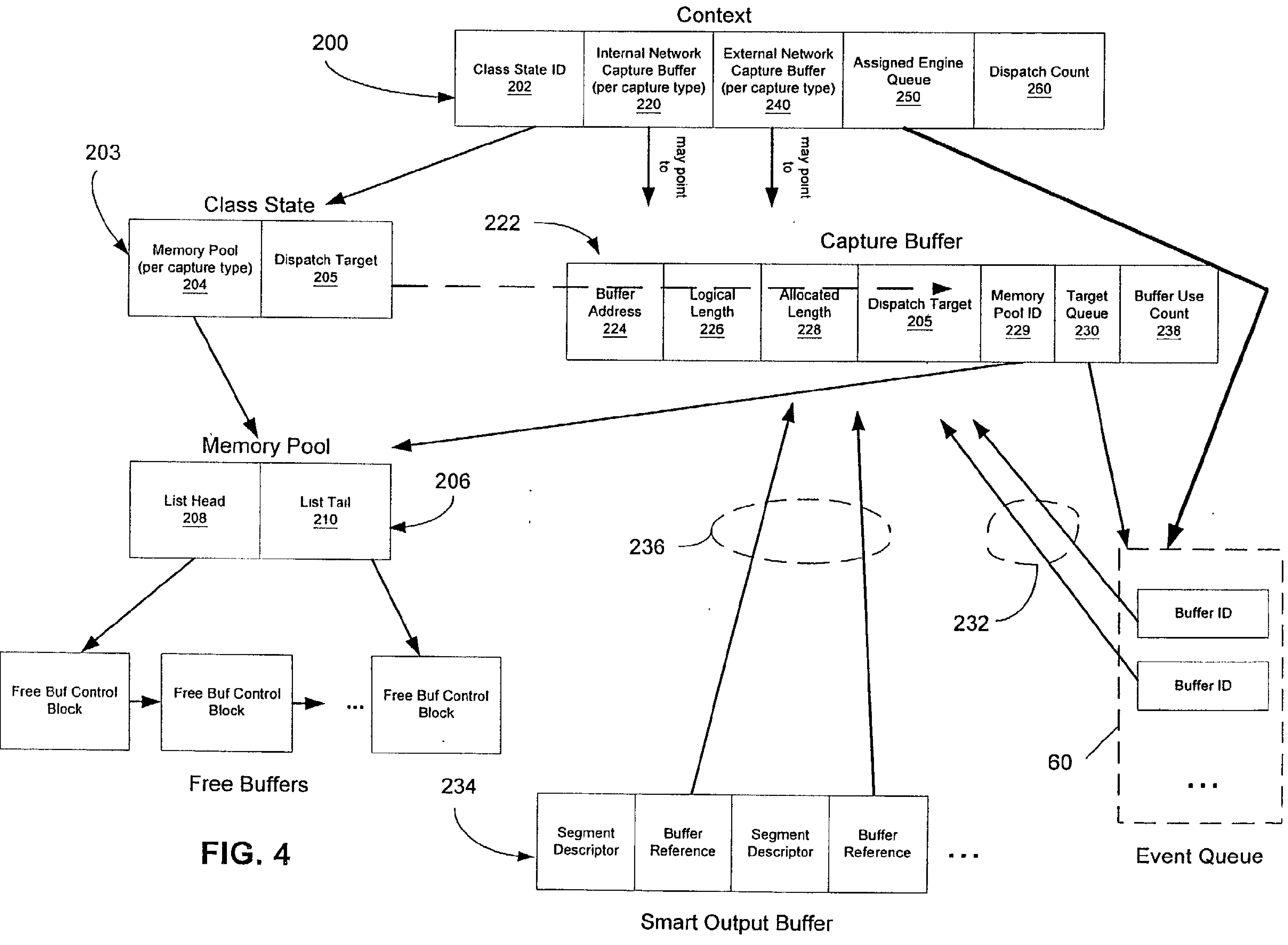


FIG. 4

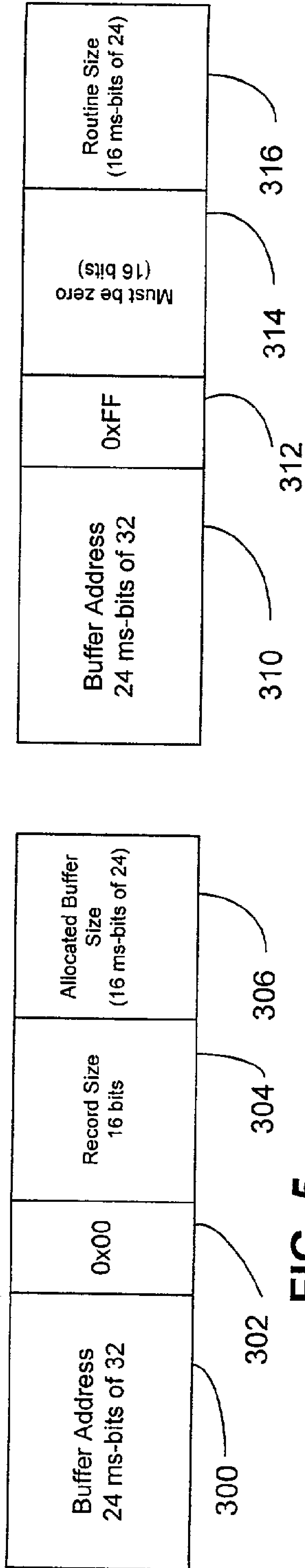


FIG. 5

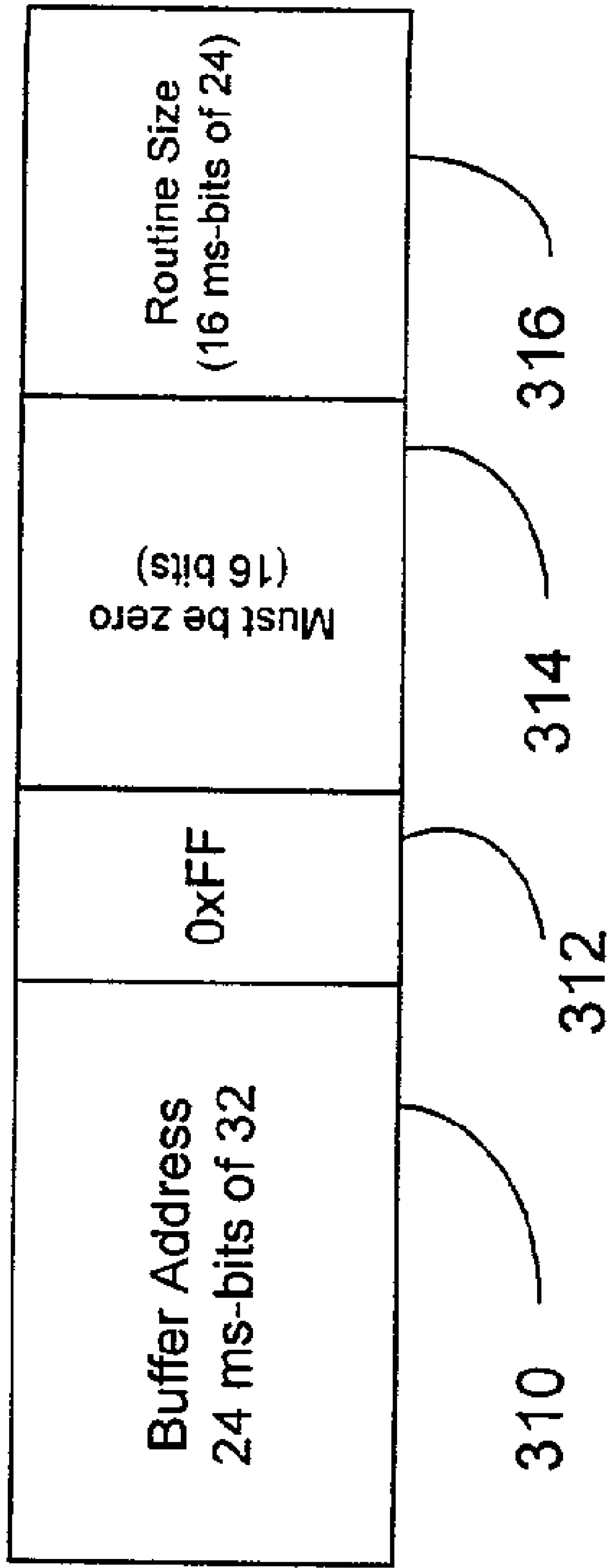


FIG. 6

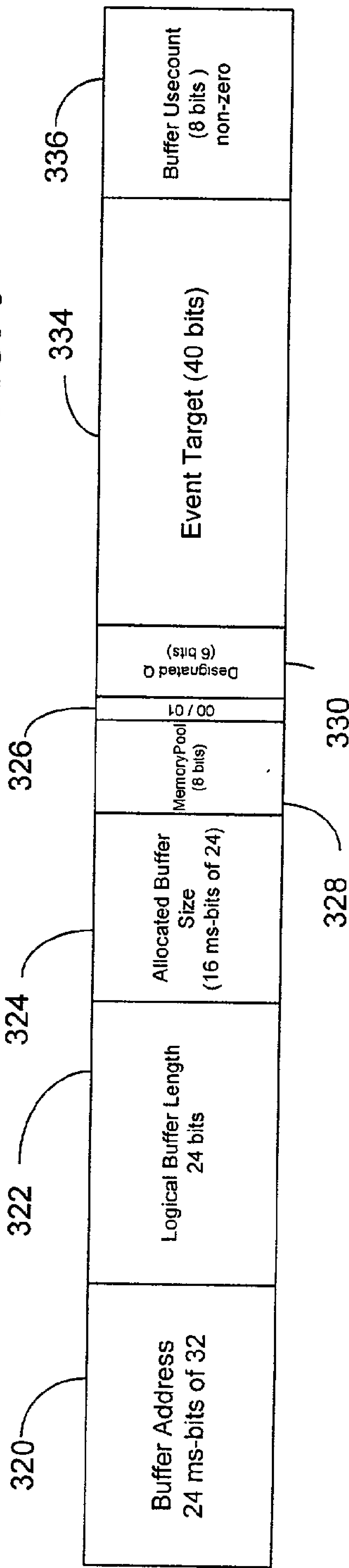


FIG. 7

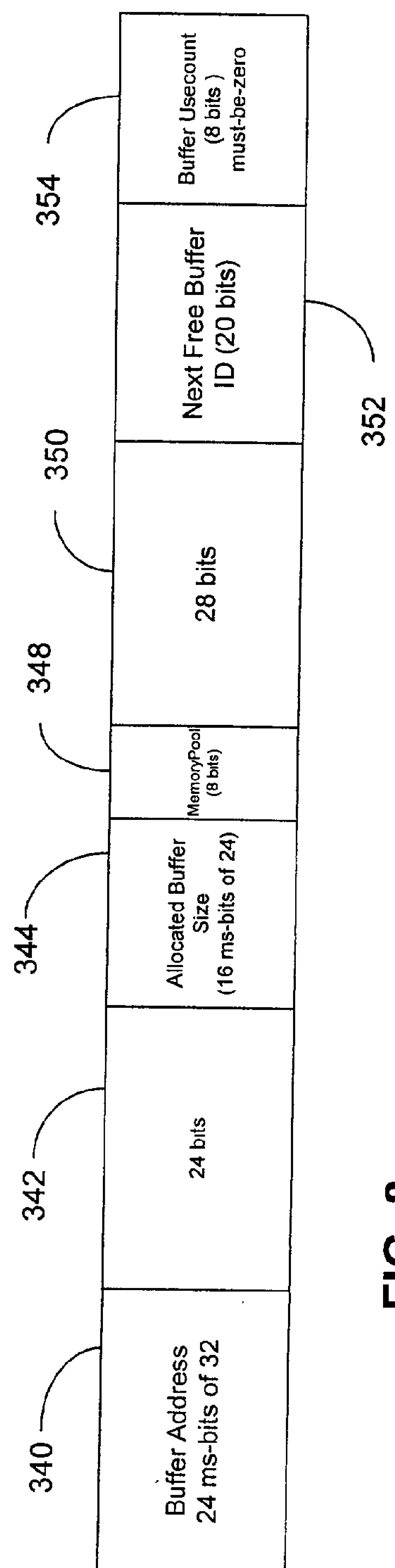


FIG. 8

400

Internal Network Target

0	ITMHdr[0] 3 msbits	ls-nibble of byte 5 of ITM header	byte 6 of ITM header	byte 1 of ITM header	byte 2 of ITM Header	byte 3 of ITM Header

401

Table Update Target

1	0	sel	clear	Table ID (16 bits)	Base Record Index (20 Bits)

402

Ticker Target

1	1	1	0	Delay before post (15 bits) - units are implementation dependent	Context to be notified of ticker expiration (20 bits)

403

External Network Target

1	1	1	1	1	1	1	0	External Network Port Select (32 bit eligible port set)

404

Context Target

1	1	1	1	1	1	1	1	1	1	1	1	1	0	capture type (2 bits)	generation (4 bits)	Context (20 bits)

405

Simulated Input Target

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	Simulated Input Channel (5 bits)

FIG. 9

410 412

Completion Type	Buffer ID (20 bits)

FIG. 10

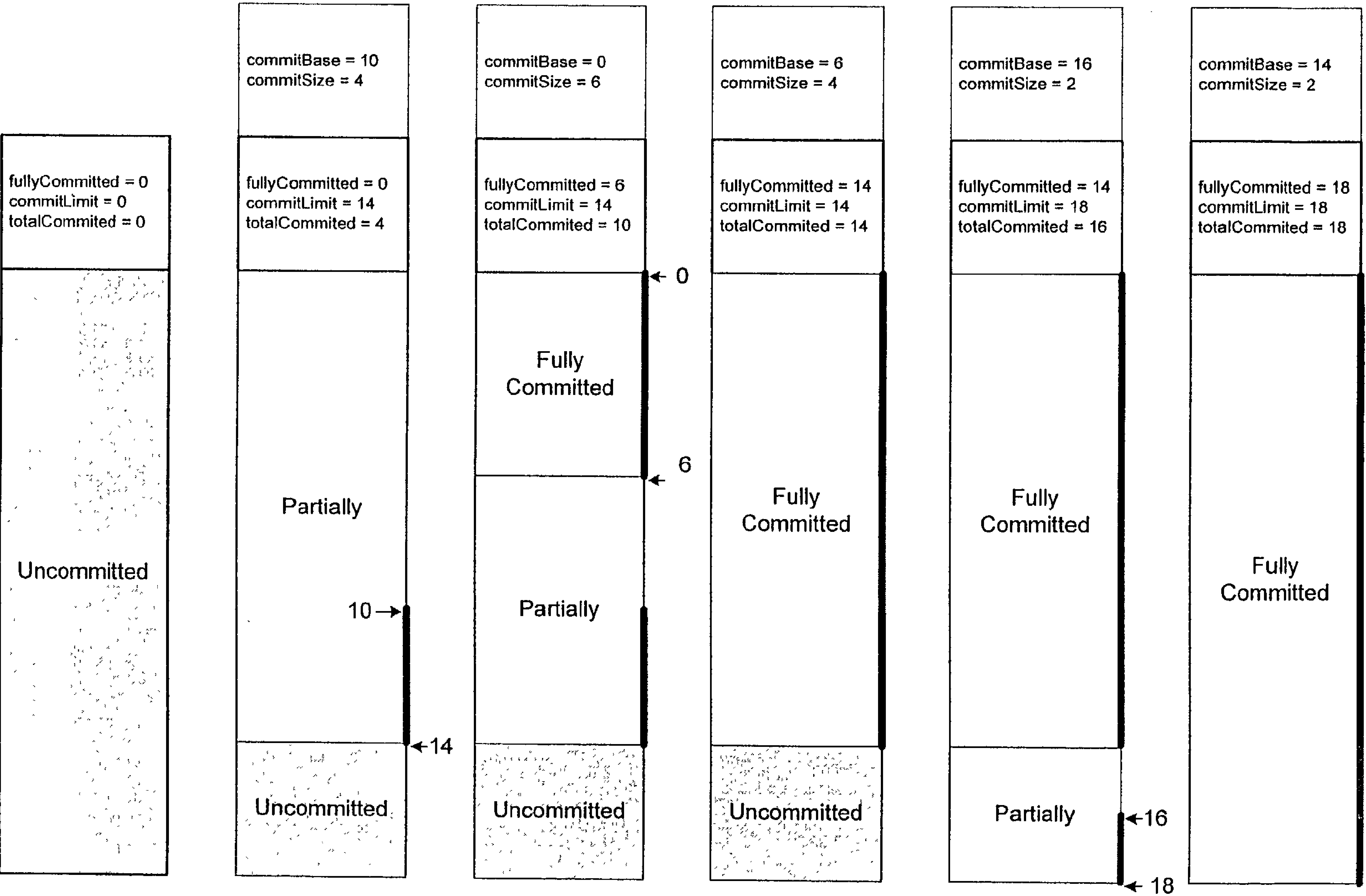


FIG. 11

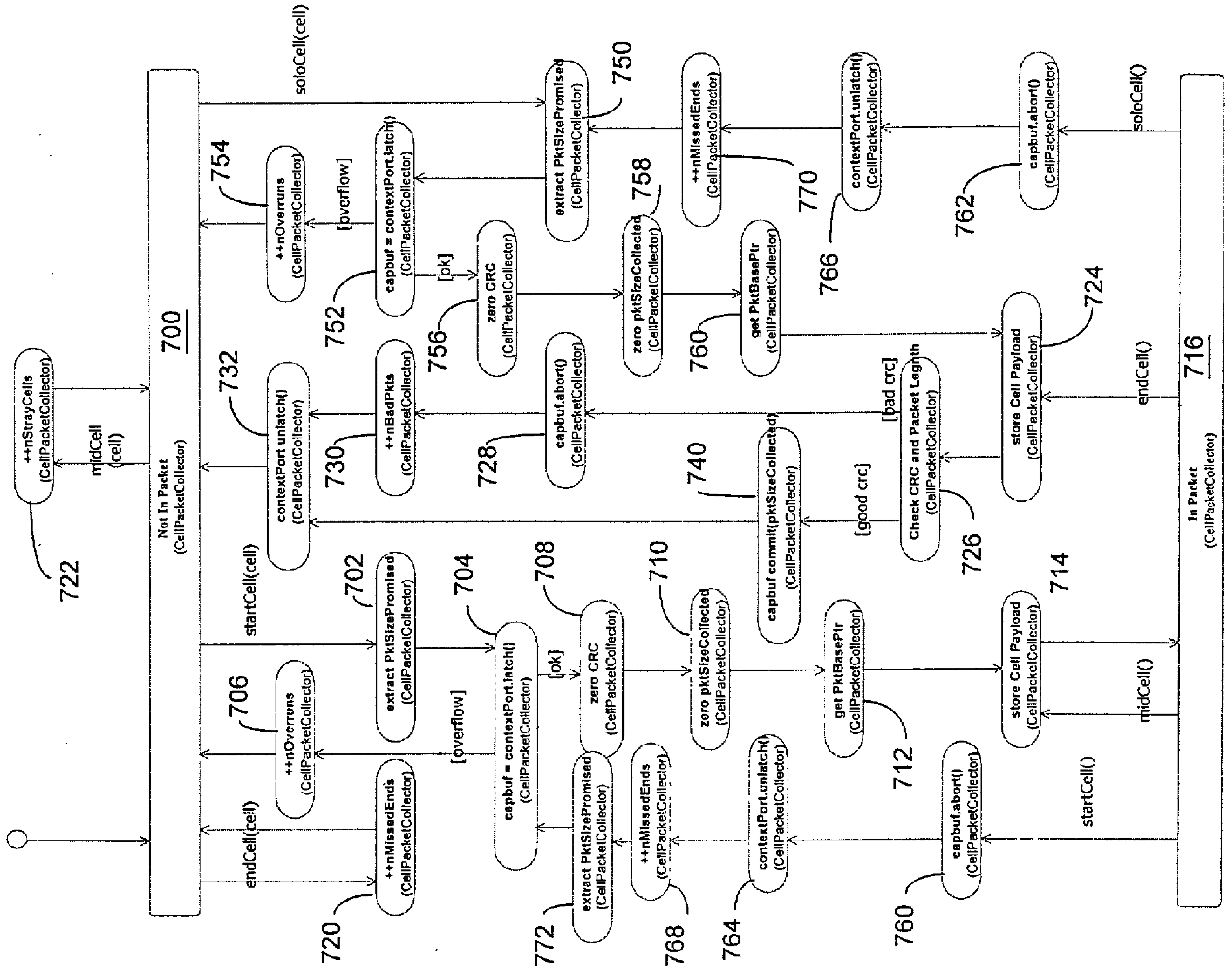


FIG. 12

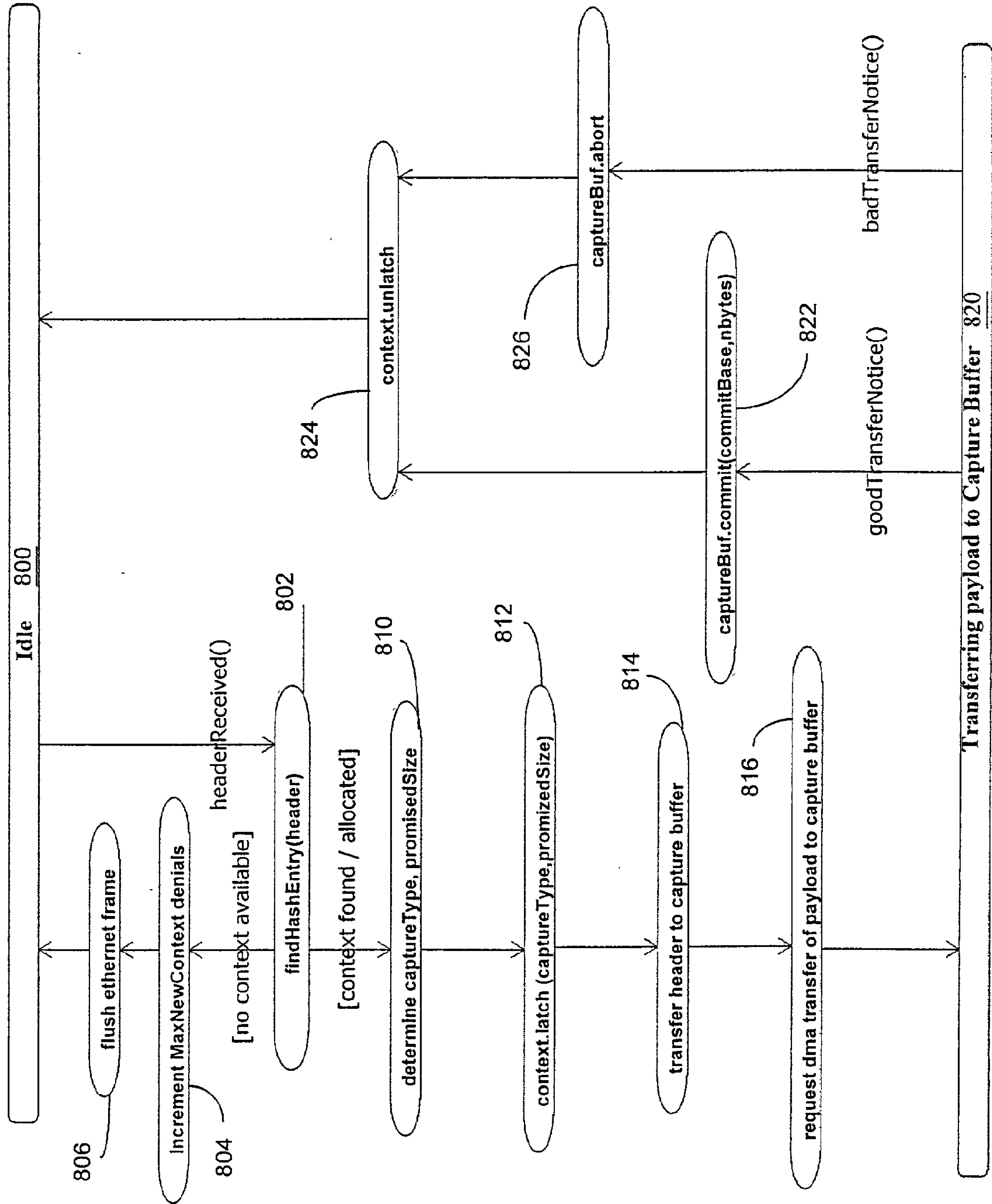


FIG. 13

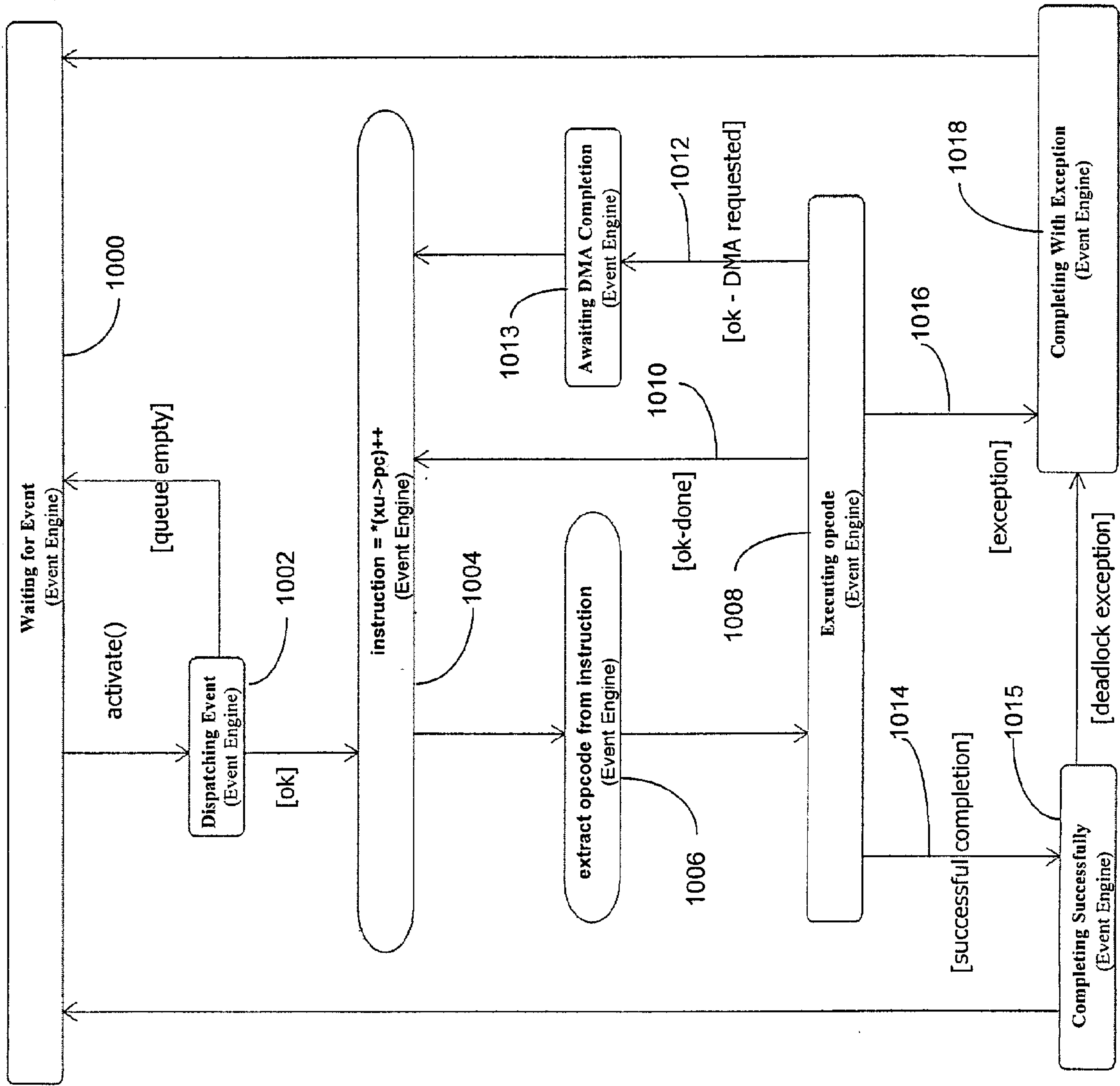


FIG. 14

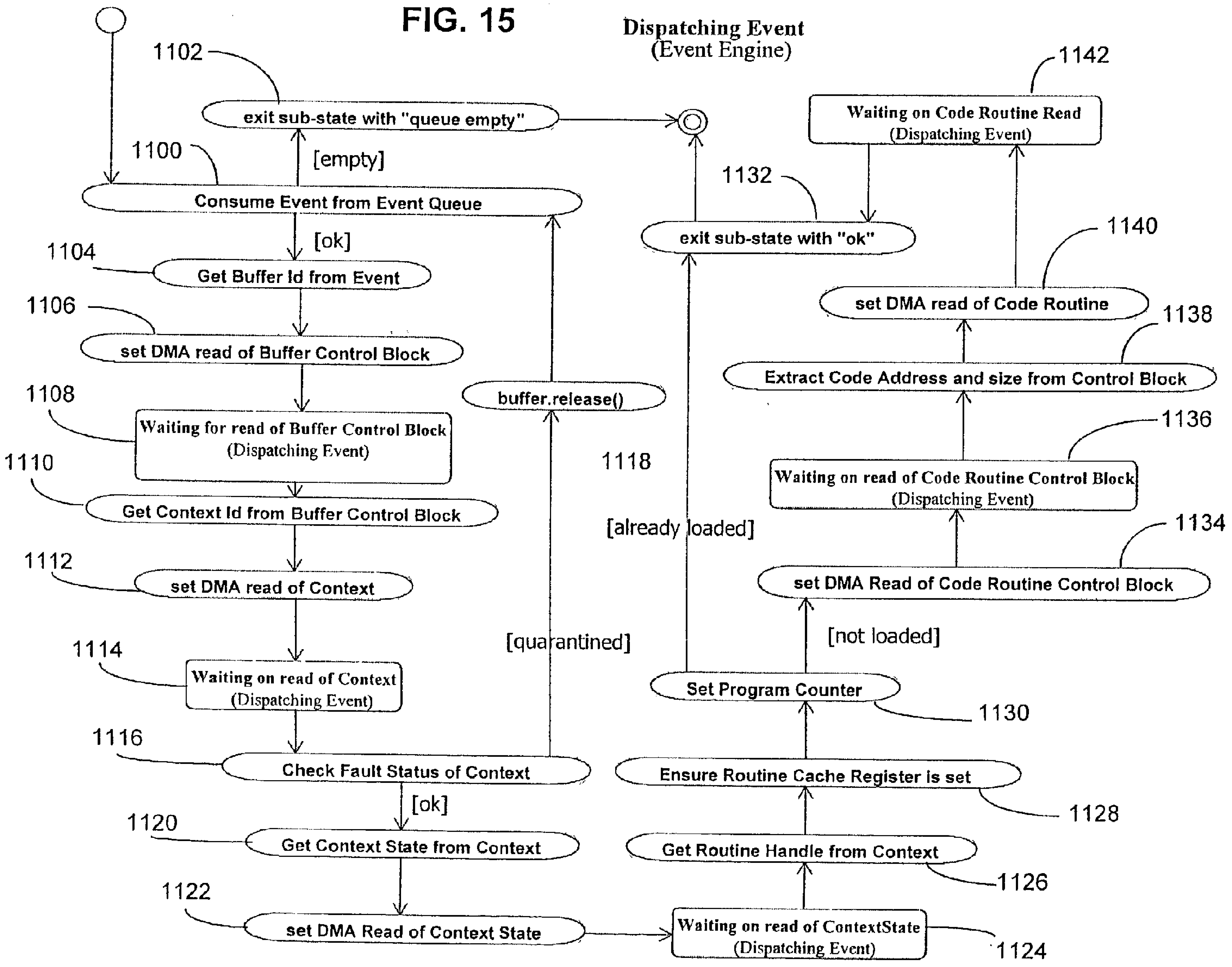


FIG. 15
Dispatching Event
(Event Engine)
1102
exit sub-state with "queue empty"
[empty]
1100
Consume Event from Event Queue
[ok]
1104
Get Buffer Id from Event
1106
set DMA read of Buffer Control Block
1108
Waiting for read of Buffer Control Block
(Dispatching Event)
1110
Get Context Id from Buffer Control Block
1112
set DMA read of Context
1114
Waiting on read of Context
(Dispatching Event)
1116
Check Fault Status of Context
[ok]
1120
Get Context State from Context
1122
set DMA Read of Context State
buffer.release()
1118
[quarantined]
Waiting on read of ContextState
(Dispatching Event)
1124
Get Routine Handle from Context
1126
Ensure Routine Cache Register is set
1128
Set Program Counter
1130
[already loaded]
[not loaded]
1132
exit sub-state with "ok"
set DMA Read of Code Routine Control Block
1134
Waiting on read of Code Routine Control Block
(Dispatching Event)
1136
Extract Code Address and size from Control Block
1138
set DMA read of Code Routine
1140
Waiting on Code Routine Read
(Dispatching Event)
1142

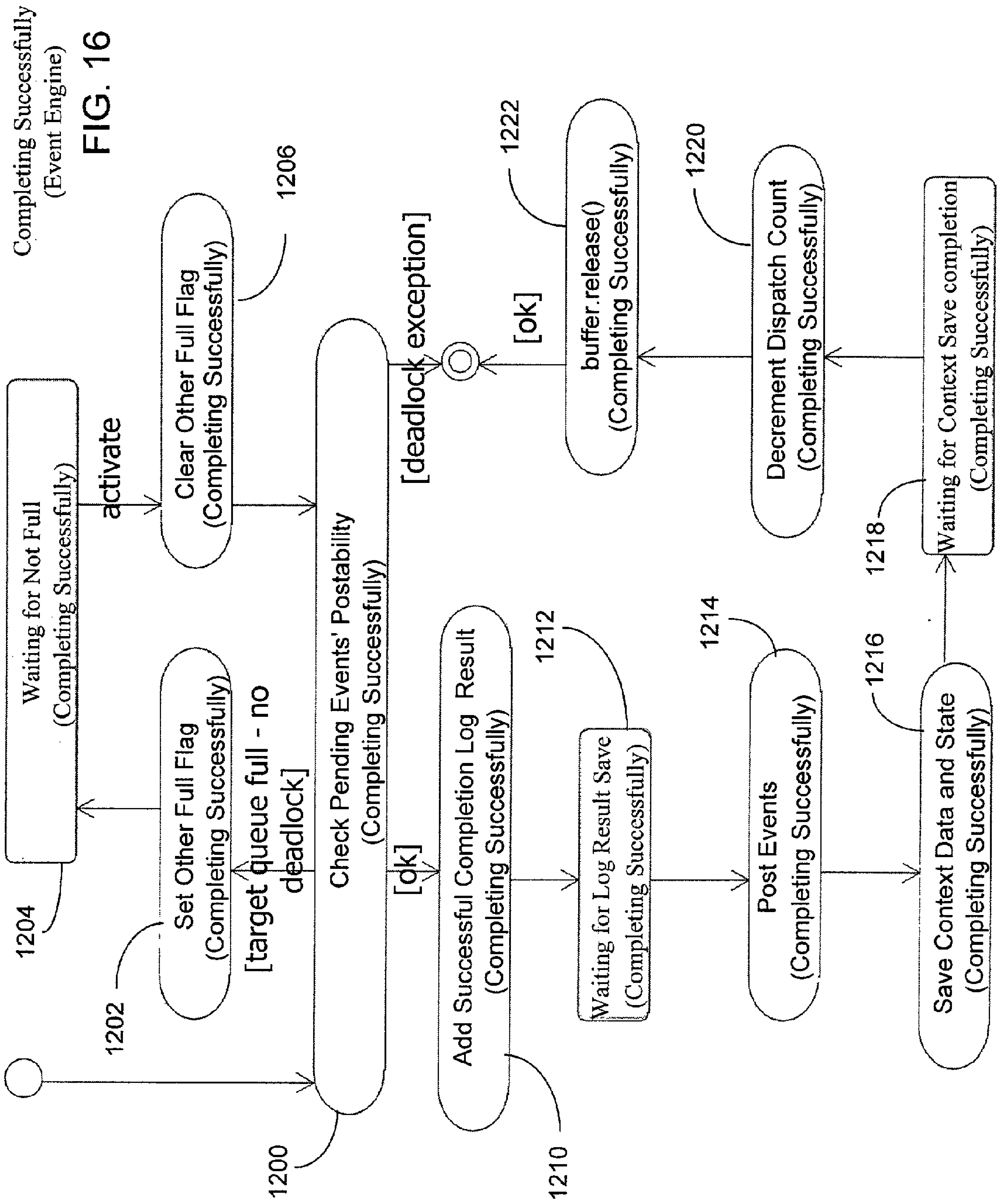


FIG. 16

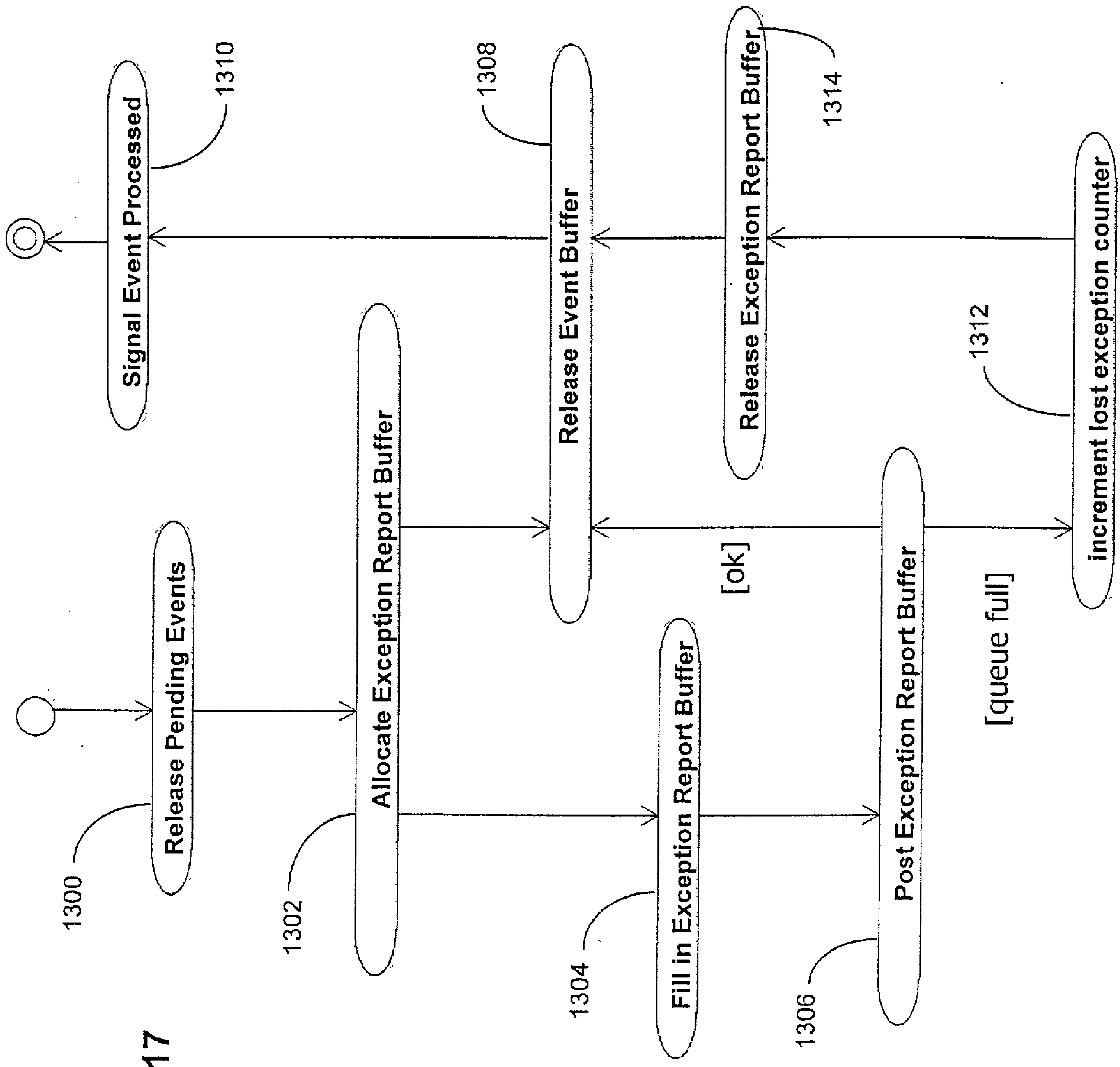


FIG. 17

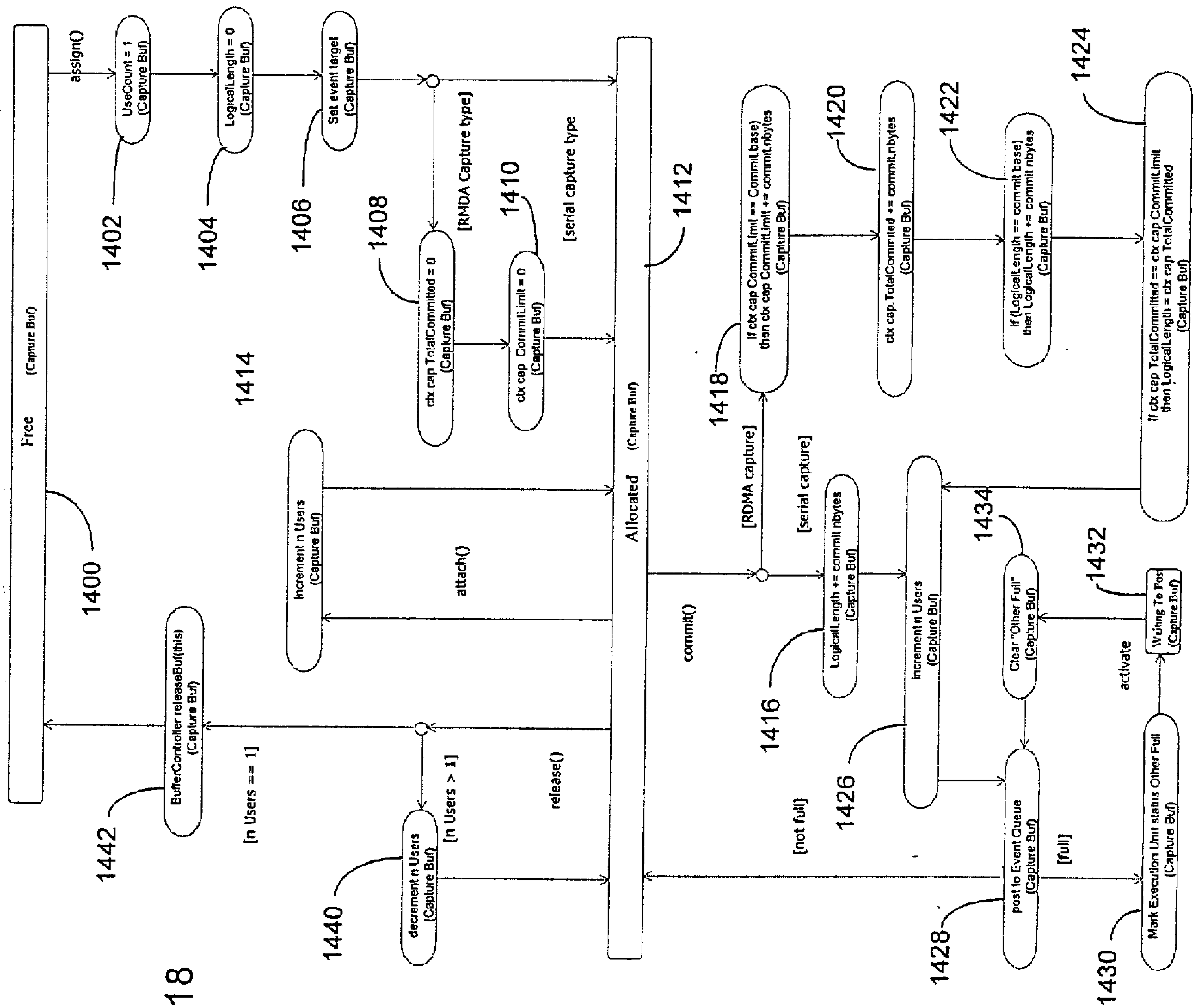


FIG. 18

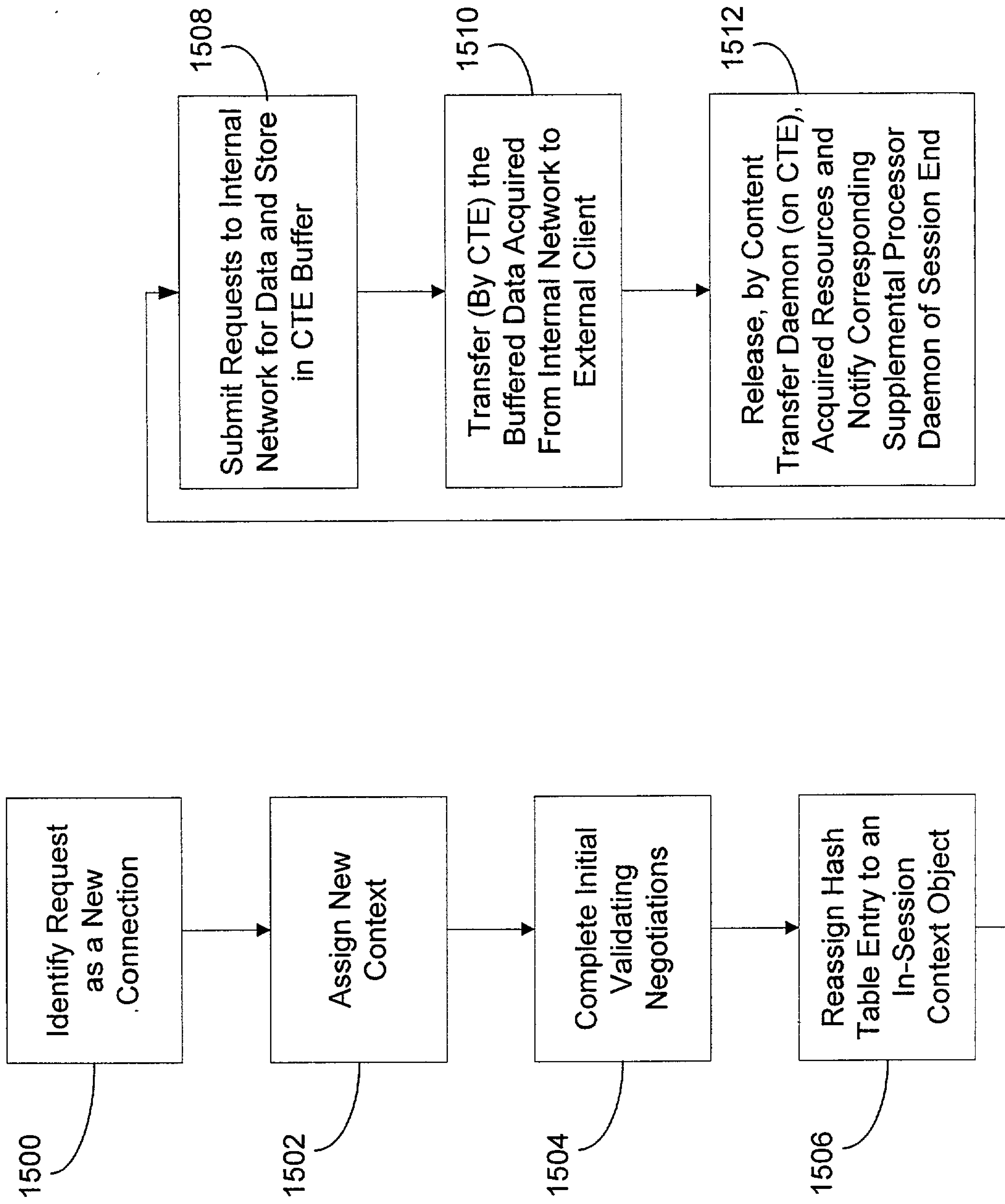


FIG. 19

NETWORK SERVER CARD AND METHOD FOR HANDLING REQUESTS RECEIVED VIA A NETWORK INTERFACE

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] The present application is a continuation-in-part of Phillips et al. U.S. patent application Ser. No. 09/638,774 filed on Aug. 15, 2000, entitled "Network Server Card and Method for Handling Requests Via a Network Interface" that is expressly incorporated herein by reference in its entirety including the contents of any references contained therein.

FIELD OF THE INVENTION

[0002] The present invention generally relates to the field of server systems for handling requests from multiple users in a network environment. More particularly, the present invention concerns apparatuses and methods for efficiently providing specialized services to requesters in a network environment through request distribution.

BACKGROUND OF THE INVENTION

[0003] As the Internet expands, so too are the potential number of users that simultaneously seek access to particular Internet resources (e.g., a particular Web site/address). Thus, operators of Internet sites are well advised to arrange their systems in a manner such that the current and future expanded versions of the system hardware relied upon to deliver site resources to users are capable of responding to a potentially high volume of user requests.

[0004] As the population of web users grows, the number of concurrent clients that a single web server must support similarly grows. It is now routine for a "single site" to have peak load access demands far exceeding the capacity of a single server. Users of such a site desire the illusion of accessing a single server that has consistent information about each user's past transactions. Furthermore, every distinct user should access the apparent single server using a same name. The challenge is to handle request processing load such that users are unaffected by any undesirable side-effects of handling high request volume and high data traffic volume associated with responding to the requests.

[0005] Web servers today are faced with increasing demand to provide audio/video materials, such as MP3, MPEG and Quicktime files. Such files are considerably larger than simple web pages. The combination of more users demanding larger files presents a potentially overwhelming demand for vastly increased data volume handling capabilities in web servers. More servers are needed to handle the increasing data retrieval and transmission workload.

[0006] Solutions have been implemented that share a common goal of dividing the workload of the single virtual server over many actual servers. It is desired that the workload of responding to multiple user requests be divided in a transparent manner so that the division is not visible to the customer. However, load-balancing mechanisms have limited or no knowledge with regard to the context or prior history of the messages they are trying to distribute. Existing solutions have chosen between overly restricting the requests that can be processed by particular servers, thereby limiting the ability to evenly balance the traffic load. In other instances work is divided between servers operating in ignorance of each other.

[0007] There are at least four known approaches for dealing with the aforementioned problems encountered as a result of high user load. The oldest is server mirroring that involves replicating content across multiple sites. Users are encouraged to select a specific server that is closest geographically to them. Because the load distribution is voluntary it is not very efficient. Synchronizing the content of all servers across all sites is problematic and time consuming. Generally this approach is considered suitable only for non-commercial information distribution.

[0008] Second, a distributed naming service (DNS) approach accepts requests from many clients and translates a single Domain Name in the requests to an Internet Protocol (IP) address. Rather than returning a same address to all clients, multiple servers are designated, with each client receiving an IP address of only a particular single server from the available servers. The IP addresses supplied within the DNS answers are balanced with the goal of evenly dividing the client load amongst the available servers. Such distributed processing methods are inexact due to remote caching. Furthermore, problems arise with regard to ensuring that related requests from the same client issued at different times result in a connection to the same server. Traffic to the DNS server is increased because DNS queries are increased. This second approach demonstrates that merely duplicating hardware will not meet a need for additional throughput in a server resource/system. Additionally, because later queries from the same user could easily go to a different server, all work must be recorded on shared storage devices. Such additional storage devices could be additional database or file servers, or devices on a storage area network.

[0009] Third, OSI layer **3** (network) and layer **4** (transport) switching solutions distribute connections across multiple servers having similar capabilities (though possibly differing load handling capacity). Entire sessions are distributed to a particular server regardless of the type of request. Because a layer **3** or **4** switch is not an actual participant in any of the protocol sessions going through it, and because it is of a simpler and more specialized design than the actual servers, it cannot fully understand the protocols that it implements. It cannot with full certainty understand the types of requests it sees. Though limited examination of packet contents may occur, the majority of such switches do not examine the content of packets that pass through them.

[0010] An FTP (File Transfer Protocol) file transfer is one example where limited examination may occur. Some layer **3** switches are configured with enough knowledge of the FTP protocol to recognize the description of a second connection buried within the payload of a packet transferred via a first connection. Others merely assume that all connections between a pair of IP addresses are for the same user. The former approach requires the switch to stay current with all new application protocols, and cannot work with encrypted payloads. The latter presents the problem of requiring all users working behind a single firewall using Port Network Address Translation (PNAT), or masquerading, to be considered as a single user. Given that the goal of

load balancing is to evenly distribute the workload, unpredictability in how much work is being committed to a single server with a given dispatch presents a problem. This example illustrates the deficiencies in the prior known systems that are limited to switching functionality rather than delegating execution of requests for resources to specialized processors/processes.

[0011] Once a session is initiated it is assigned to a particular server based on very little context information. Typically the context information includes only the addressing information within the initial packet itself without the benefit of knowing the substance of any queries to customer databases or other records. In particular it is not practical for the switch to distinguish between a single user and an entire building of users sharing a single IP address. Thus, an initial assignment may be supplemented only by limited analysis of later packets to identify packets belonging to the same session. This distribution scheme is carried out by a switch having very limited analytical capabilities.

[0012] Other prior load distribution solutions conduct only limited analyses to determine the type of a request received by the server system after a session is initiated and assigned to a particular server. Furthermore, providing a set of equally capable servers is a potentially expensive solution that is likely considered too expensive for many potential Internet service providers. Sharing results through direct back channel communications or sharing of database, file or storage servers is still required for these solutions.

[0013] In a fourth known attempt to distribute requests for resources over distributed network servers, server clusters distribute work internally over an internal communications bus that is typically a switched access bus. Communications to the external network interface are performed via a single centralized server. Thus, while distributing the computational load, the fourth known option introduces a potential data communications bottleneck at the centralized communication server.

[0014] The first three solutions divide a single virtual server's workload into multiple user sessions. Such prior known solutions attempt to ensure that all traffic for a given user session is handled by a single actual server while attempting to distribute the work load evenly over the entire set of actual servers. These goals are incompatible in the prior known systems. Mirroring, because it relies upon the user identifying the targeted server, is excellent at ensuring only one actual server deals with a given user, but the only mechanism for balancing load between the servers is the process of users shifting between servers out of frustration. Layer **3** switch solutions achieve load balancing, but at the cost of failing to identify all parts of a single user's interaction with the server. The fourth solution avoids these problems by having a single server delegate work, but limits the scope of this optimization by remaining a single communications bottleneck.

[0015] Thus, while a number of solutions have been implemented to deal with the problem of explosive growth in the popularity and volume of use of the Internet and the resident Web sites, none provide a solution to the increased cost and overhead associated with distributing user workload addressed to a single apparent resource that is, in actuality, distributed across multiple processors.

[0016] It is further noted that today many Internet servers are deployed almost exclusively to distribute stored content. In other words, such servers execute one task-delivering data stored on a memory drive to a requesting client over the Internet. Such servers are structured on a request-load-process-deliver model. The server application on a server host machine accepts a request for content from a client. In response, the server application loads the requested content from a memory drive into process memory space reserved for the server application on the host. After loading the content into process memory space, the server application processes the content, and then the server delivers the processed content to the client.

[0017] Ideally, content providers would prefer utilizing as few server machines as possible to deliver content to Internet clients. However, in the case of streaming data (e.g., video, audio, etc.) the process of accepting a request, loading content and then delivering the content consumes nearly all of the processor's capacity serving only a single Gigabit Ethernet port. When multiple Gigabit Ethernet ports are provided for a single server application, the processor becomes a system bottleneck. As a result, streaming content servers often require multiple replicas to provide satisfactory bandwidth to client/users.

[0018] Many factors contribute to this bottleneck. Packets on both the internal network (where the stored content is accessed) and external network (where the clients are located) arrive interleaved, fragmented and even out-of-order. The received packets must be processed by the host operating system. In addition to the inherent work of de-fragmenting the incoming packets, processing the packets by the host operating system involves switching between user and system memory maps and quite likely copying data between the user memory space and system memory space. Furthermore, because of successive layering of protocols, it is common to apply a variety of checksum or CRC algorithms to different portions of the payload with different degrees of reliability. For example, when a higher layer protocol specifies an end-to-end 32-bit checksum, it cannot eliminate an inadequate 16-bit checksum specified by a lower level protocol. Hence both must be generated and checked. A number of system architectures attempt to address the processor bottleneck problem by optimizing the path between a network interface and a processing application. These include Virtual Interface Architecture (VIA), Infiniband Architecture (IBA), Warp and Microsoft's Winsock Direct.

[0019] The following comprises a discussion of the classical method for a server responding to client requests in an IP environment. In classic server design, IP datagrams arrive at multiple network interface cards (NICs). Each NIC deals with only one communication protocol. A typical server includes NICs for dealing with the external IP network (almost always Ethernet) and separate ones for dealing with an internal storage-oriented network. The internal storage-oriented network may be Ethernet/IP oriented, in which case it is possible to apply a NAS (network attached storage) strategy to access network file servers over the internal network. Alternatively the internal network may be a specialized Storage Area Network (SAN), such as Fibre Channel. The servers on these networks typically provide block-level services, rather than file-oriented services.

[0020] For an Ethernet/IP interface, the classic solution exhibits the following characteristics. Buffers for incoming

traffic are allocated from a system memory pool on a per device basis, without regard to the eventual destination. After the Ethernet frame has been collected and validated, it is passed in FIFO order to the host operating system's protocol stacks. The host operating system's protocol stacks perform all required segmentation and re-assembly to deliver messages to the requesting application process. This must be based upon information within the reassembled packet. Typically this involves one or more copy operations to create an in-memory image of the complete buffer. Alternatively, more complex interfaces can be used to pass a "scatter/gather list" to the application. This results in more complex application code, or postponing coalescing the message fragments to the application code.

[0021] The work involved in transferring network payload through a Host Operating System protocol stack can be so time consuming, especially with switches between system and user memory space, that a single processor server can be almost totally consumed merely getting network traffic between a Gigabit Ethernet NIC and to applications. This leaves almost no processing resources for an application to actually accomplish any true data processing.

[0022] A virtual interface (VI) allows a NIC to directly post results to user memory. This allows a complete message to be assembled directly in the application's buffers without kernel/user mode switching or intermediate copying. VI interfaces support two delivery modes: send and remote direct memory access (RDMA). Send delivery mode is based upon a connection. The incoming data is paired with application read requests and transferred directly to the application memory. Successive reads consume requests, and reads are pre-issued. During RDMA delivery the request supplies a memory key validating its access to the target application memory space and an explicit offset within that target memory space. An RDMA interface allows a file server to utilize an out-of-order delivery strategy. Fragmented files are read from disk drives in the most convenient order. Being restricted to reading them in the "correct" order would require server-side buffering and/or more passes of the drive heads. Out-of-order delivery also enhances/supports striping of material across multiple drives. Without the need to synchronize multiple sources it is easier to get the multiple sources to deliver requested data in parallel.

[0023] VI and similar interfaces (such as InfiniBand) provide improvements over the classic via-kernel methods. However, they still exhibit significant drawbacks. First, VI and InfiniBand (IB) require buffers to be pre-allocated for each pending read. When applied only over the internal storage network this creates only relatively minor problems. However, when dealing with remote clients over the public external network long delays can be expected. Furthermore, pre-allocating distinct buffers for 10,000 active clients is wasteful when, for example, it can be predicted with near statistical certainty that no more than 500 of them will respond in the immediate future. The time between each client action and the time required to process each client request allows a single buffer to be re-used potentially many times. However, under VI and other similar RDMA protocols a buffer is locked down for a single client from before the time the request is issued until the request is fully satisfied. Second, under VI and IB the NIC obtains from a kernel agent the mapping data required to translate virtual memory addresses to physical memory for each client. Obtaining this information, and ensuring that it is "locked down" so that it cannot be moved or swapped out of memory, involves a time-consuming negotiation with the kernel agent. One VI-derived solution, the Infiniband Architecture, has added "Memory Windows" to provide logical slices of memory that are only kernel registered once but have finer-grained rights access administration on the NIC itself. Third, the requirements of VI cause significant portions of "application memory" to be pinned down and effectively function as system memory. Pinning down application memory forces other buffers to be swapped more often and makes it harder for the kernel to optimize application performance. This is especially true if the kernel has not been specifically re-engineered for VI. Fourth, while VI enables bypassing the host processor's operating system, the host processor is not bypassed. Therefore, all data traffic flows into the host processor's memory, is examined by an application, and then typically flows back out to another NIC connected to a network.

[0024] The two above-described prior data transmission approaches adopted by servers (i.e., classic and VI) are now described with reference to how an incoming Ethernet frame of data from, for example, an internal data storage network is handled by the server. In the classic server approach, the Ethernet frame is DMA transferred to a system buffer. Buffer selection is steered by the device type upon which the data is to be transmitted—not by the ultimate target. Intermediate software, typically the host operating system's protocol stacks, assembles messages and then copies them to user memory. This process is reversed to send the loaded payload as part of an HTTP response via an external network.

[0025] In the VI approach, when the Ethernet Frame arrives from a data storage network it is paired either with an application read request (in the case of a "send" operation) or an application memory space (in the case of an "RDMA" operation). Because the memory (and/or request) was pre-registered with the NIC and kernel, the NIC determines the destination address(es) in physical memory and deposits the payload there. This process is reversed to send the loaded payload as part of an HTTP response via the external network.

[0026] It is also interesting to note the number of memory transfer operations associated with processing an incoming Ethernet frame. In the classic server approach, the frame passes into the NIC, into the host processor system buffer, into the host processor user buffer, back to the host processor system buffer, back to a different NIC, and then out to a destination. In a VI type system, the frame passes into the NIC, into a host processor user buffer, back to a different NIC, and out to a destination. In both cases, buffering consumes considerable system resources.

SUMMARY OF THE INVENTION

[0027] While the above-described server architectures will improve carrying out requests under the request-load-process-deliver model, such architectures seek to optimize getting stored content to a server application process and then transferring the data from the server application to the client via network communication interface processes. In accordance with an aspect of the present invention, a server architecture facilitates optimizing transfer of requested content from data storage drives to requesting clients.

[0028] In accordance with the present invention, a data asset server handles content delivery traffic where there is no need to process the stored content, but rather merely a need to package the data and control its delivery to a designated location. Thus, the server carries out a "request-select-deliver" model wherein a request is received by the server from a client. Next, content to be delivered to the client in response to the request is selected. In an embodiment of the invention, notification, but not the selected data itself, is delivered to an application running on the server system that received the request. The content is delivered via an external network interface engine (with necessary protocol wrappers but no transformation of the actual content) to the requesting client.

[0029] In a particular embodiment of the invention, to which the invention is not limited, a content transfer engine receives an Ethernet Frame and a destination message buffer is determined either by a pre-existing read or based on the state of the connection. The destination is content transfer engine controlled memory. Notification, but not the data itself, is delivered to a content transfer daemon in the content transfer engine. The content transfer daemon controls transfer of content for a single end-user session. The content transfer daemon executes within the context of the content transfer engine itself. The content transfer daemon optionally extends its processing scope in conjunction with an extended content transfer daemon that runs on a conventional processor. Buffer capture events are typically sent to the content transfer daemon using event queues, but may be sent over the internal network directly to an extended content transfer daemon. Because dispatching notices should be prompt and highly reliable it would be highly unusual for an external network interface engine to be used. The content transfer daemon then, by way of example, composes an HTTP response referencing payload already in the content transfer engine's memory. The HTTP response, including the payload in the content transfer engine's memory is sent as the HTTP response via the external network.

[0030] With regard to memory operations, the received Ethernet frame is stored in the data buffer physically located on an access node. The Ethernet frame data is not moved into system buffer space managed by a conventional processor, nor is the frame processed by the conventional processor. Instead, the stored data is output to a connected network interface engine associated with an identified target destination without ever entering the conventional processor's data space.

[0031] In an embodiment of the present invention, multiple event engines execute in parallel or as co-routines upon the content transfer engine to serve client requests. Division of responsibilities between a CTE, resident CTDs, and conventional processor resident XCTDs is an application and implementation specific tradeoff. A conventional processor favored strategy would use the Content Transfer Engine to capture and transmit buffers, while making most protocol decisions on the conventional processor. A CTE favored strategy would handle virtually all normal cases itself, and only forward exceptional cases for conventional processor handling. The placement of wrapping protocol headers and trailers around the stored content is performed in any of a variety of possible locations, including both on and off the CTE. Finally, it will be understood by those skilled in the art that the present invention is applicable to a variety of external network communications protocols.

BRIEF DESCRIPTION OF THE DRAWINGS

[0032] The appended claims set forth the features of the present invention with particularity. The invention, together with its objects and advantages, may be best understood from the following detailed description taken in conjunction with the accompanying drawings of which:

[0033] **FIG. 1** is a diagram depicting an exemplary network environment into which the invention is advantageously incorporated;

[0034] **FIG. 2** is a block diagram identifying primary logical components within a set of network nodes arranged in an exemplary physical arrangement in accordance with an information asset server system embodying the present invention;

[0035] **FIG. 3** depicts a set of functional/logical components of a content transfer engine within a server system depicted in **FIG. 2**;

[0036] **FIG. 4** depicts a set of data structures for external buffer control and facilitating execution of operations in accordance with an exemplary embodiment of the present invention;

[0037] **FIG. 5** depicts a memory control block for a table data structure facilitating carrying out an exemplary embodiment of the present invention;

[0038] **FIG. 6** depicts an exemplary code routine memory control block facilitating carrying out the present invention;

[0039] **FIG. 7** depicts an exemplary memory control block for an allocated buffer;

[0040] **FIG. 8** depicts an exemplary memory control block for a free buffer;

[0041] **FIG. 9** depicts a set of exemplary event queue target descriptors in accordance with an exemplary embodiment of the present invention;

[0042] **FIG. 10** depicts an exemplary event queue entry format for queue entries placed within the event queues of a content transfer engine embodying the present invention;

[0043] **FIG. 11** is a sequence of buffer representations depicting the state of a capture buffer as data arrives in an out of order fashion using RDMA mode capture;

[0044] **FIG. 12** is a state diagram specifying how cells arriving on a node channel are collected into a packet;

[0045] **FIG. 13** is a state diagram specifying how Ethernet frames are collected into a packet;

[0046] **FIG. 14** is a state diagram specifying the life cycle of an event engine;

[0047] **FIG. 15** is a state diagram that expands the "Dispatching Event" state of **FIG. 14**;

[0048] **FIG. 16** is a state diagram that expands the "Completing Successfully" state from **FIG. 14**;

[0049] **FIG. 17** is a state diagram that expands the "Completing With Exception" state from **FIG. 14**;

[0050] FIG. 18 is a state diagram specifying the life cycle of a Capture Buffer; and

[0051] FIG. 19 is a flow diagram depicting the general operation of the disclosed data server.

DETAILED DESCRIPTION OF AN ILLUSTRATIVE EMBODIMENT

[0052] A new network server architecture is described below in the form of a preferred embodiment and variations thereof. At the heart of this new network server architecture is a hybrid multiprocessor server. The hybrid multiprocessor server includes a data transmission engine comprising a set of micro-engines. The micro-engines are processors with limited capabilities for executing a particular set of tasks relating to, for example, data communication. The network server also includes at least one supplemental processor, a general purpose processor operating a standard operating system. The network is, by way of example, the Internet. Thus, communication between the server and an external network is preferably conducted in accordance with the Internet Protocol (IP).

[0053] When a user requests to initiate a session with the network server, a session request is received by a network interface engine. The network interface engine passes the request to either a micro-engine or the supplemental processor. After establishing a session/connection with a client, fulfillment of the client's data requests are carried out primarily by one or more of the micro-engines. The supplemental processor, in an embodiment of the invention, receives notification of a data transfer. However, data is transmitted between data storage drives (e.g., disk drives) managed by the network server and the requesting clients without transferring the transmitted data into the supplemental (host) processor's memory space.

[0054] As mentioned above, the network server of the present invention includes a set of one or more specialized micro-engine processors, including for example a set of micro-engines referred to herein as event engines that are configured to accept and execute content transfer requests. Each connection is tracked as a separate context. Each connection belongs to a specified class. The class specifies, for example, a state transition model to which all instances of that class will conform. The state model specifies the default buffer allocation behavior, and how event capture is to be processed (to what event queue and to what target routine) for each state. Instances deal with specific user sessions. An example of a specific class is one specifically configured to control delivery of an html file according to the HTTP delivery protocol using TCP/IP. Each instance actively transfers a specific html file to an end user/client via a specific TCP/IP connection.

[0055] An interface to an internal network provides communication paths between one or more storage servers (comprising multiple storage media devices such as hard drives) and the data transmission engine. In an exemplary implementation the internal network is an internal network of the type generally disclosed in U.S. application Ser. No. 09/579,574, filed on May 26, 2000, and entitled: "Information Distribution System and Method" which is explicitly incorporated herein in its entirety by reference. However the invention disclosed here is intended to be applied to other storage area networks, including Fibre Channel and Infiniband.

[0056] The architecture of a network server system including the above-described hybrid-multiprocessor data transmission interface is extensible, and thus multiple hybrid-multiprocessor network servers having the above general arrangement are integrated to provide access to a shared data resource connected via a switching fabric. Each of the network servers includes its own network interface engines. In the case of an Internet connection, each network server is assigned at least one unique Internet address. The invention is not limited to an Internet environment and may be incorporated into virtually any type of server environment including intranet, local area network, and hybrid combinations of various network interface engines to the network servers. Thus, while generally referred to herein as a network server, the architecture is applicable to network configurations that include both WAN and LAN interfaces.

[0057] The present invention is not intended to be limited to a particular arrangement of distributed handler processors on a Network server. The "server" may take any of many different forms. While preferably the default and specialized handler processors are arranged upon a single, multi-layer printed circuit board, the server is implemented in various embodiments as a set of cards connected via a high speed data/control bus.

[0058] In accordance with and exemplary embodiment of the present invention, an Internet Protocol (IP) access node, comprising a network server and associated stored content requested by clients on an IP network, includes a data access interface and a host. The host provides a conventional application program execution environment including off-the-shelf operating system software such as NetBSD, FreeBSD, Linux or Windows 2000. The data access interface comprises a combination of limited/specific purpose micro-engines that perform a limited set of operations associated with retrieving requested data from a set of data storage drives and delivering the requested data to clients. When a request for data is received by the IP access node, subsequently delivered data bypasses the host (e.g., supplemental) processor thereby avoiding a potential bottleneck to data delivery. While the invention is not limited to any particular protocol, in an exemplary embodiment of the invention, the data access interface is specifically programmed to handle high volume Worldwide Web and streaming media protocols such as the well known HTTP, FTP, RTSP and RTP data transfer protocols.

[0059] With regard to data transmission buffering in contrast to above-described known systems, the disclosed content transfer engines have the ability to defer the actual allocation of a capture buffer until the first packet arrives. The context has a current state. In that state, there are defined memory pools for the internal and external networks. When a packet actually arrives, the first free buffer in that memory pool is allocated. The system only needs to dedicate as many buffers to that pool as are statistically required to meet a desired safety margin to ensure that buffer overflow does not occur. Furthermore, because the pools are state sensitive, allocation rules are incorporated to ensure they cannot be exhausted by denial-of-service (DOS) attacks. A pool is configured to only provide on-demand buffers to established sessions. The pool specified for each state varies based on the peak demand for connections in that state, and the required data collection size. For example, during the life span of an inbound FTP session, there are times when a

simple command is expected (requiring a relatively small buffer), and there are times when a large file transfer is expected (requiring large allocated buffer capacity).

[0060] Turning now to **FIG. 1**, in accordance with an exemplary embodiment of the present invention, a content transfer engine is deployed in a server system environment including a set of content transfer access nodes **10** and **12**. In the exemplary embodiment of the present invention, the content transfer access nodes **10** and **12** access storage devices **14**. The access storage devices are represented in **FIG. 1** as a set of storage nodes accessed by the content transfer access nodes **10** and **12** over an internal network **16** that preferably exhibits at least RDMA capabilities. The internal network **16** is preferably fully non-blocking switch fabric operating under an ATM protocol or a variation thereof, and data is packaged within cells transmitted over the internal network. However, such an internal network is not required to realize the advantages of the present invention.

[0061] In addition, the server system provides Internet content distribution services, e.g., HTTP, to an extensible set of clients **18** over an external network **20** such as the Internet. The interfaces between the content transfer access nodes **10** and **12** and the external network **20** are, by way of example, Gigabit or 100 Mbit Ethernet ports. A particular embodiment of the invention includes dual Gigabit Ethernet ports. The exemplary server system includes one or more supplemental processor (SP) nodes **22**, that reside on the host portion of the server system containing the content transfer access nodes **10** and **12**. The supplemental processor nodes **22** handle portions of protocols and requests that cannot be handled by the limited state models supported by the content transfer engines within the content transfer access nodes **10** and **12**. The SP nodes **22** are accessed directly, or alternatively (as shown in **FIG. 1**) via the internal network **16**. The non-blocking paths of the internal network **16** are governed by a server controller **24**. A number of internal networks are contemplated in accordance with various embodiments of the present invention. Examples of such internal networks include a "DirectPath" architecture of Ikadega, Inc., of Northbrook, Ill., disclosed in U.S. application Ser. No. 09/579,574, filed on May 26, 2000, and entitled: "Information Distribution System and Method"; and Phillips et al. U.S. patent application Ser. No. 09/638,774 filed on Aug. 15, 2000, entitled "Network Server Card and Method for Handling Requests Via a Network Interface" the contents of which are explicitly incorporated herein by reference. Content transfer engines executed within the content transfer access nodes **10** and **12** perform/execute the role of an access node fabric interface in the server system.

[0062] Other suitable internal network types include Infiniband Architecture (IBA) wherein the content transfer access nodes **10** and **12** serve the roles of a hardware channel adapter (HCA); and Fibre Channel wherein the content transfer access nodes **10** and **12** are host adapters.

[0063] Having generally described an exemplary network environment, an embodiment of the invention and exemplary alternatives, attention is now directed to **FIG. 2** that contains a schematic block diagram depicting components of a server system including the content transfer access node **10** in a network server system embodying the present invention as well as logical connections between the identified components. The architecture of the content transfer access node **10** is particularly well-suited for providing Internet oriented services, especially those involving streaming video or other real-time extended data delivery systems. The content transfer access node **10** features multiple execution units (engines/micro-engines). There is an execution unit associated with each network interface (interface engines) and multiple execution units to process protocol events (event engines). Each execution unit is fed by a corresponding dedicated queue. Such queues are described herein below. Execution units handle requests from their corresponding queue on a FIFO basis. Because of the limited processing capabilities of the set of micro-engines, the content transfer access node **10** is preferably utilized to control high-speed data transfers rather than to process (e.g., performing calculations upon) the transferred data itself. In an exemplary embodiment of the invention, the event engines only examine very small portions of the transferred data—typically only protocol headers.

[0064] The architecture of the content transfer access node **10** facilitates dividing tasks associated with processing client requests between the content transfer access nodes **10** and **12** and the SP nodes **22**. The content transfer access node **10** is implemented, by way of example, as a combination of field programmable gate arrays (FPGAs), programmable logic devices (PLDs), and/or application specific integrated circuits (ASICs) that perform a limited set of task types at a relatively high speed and with minimal overhead. The SP nodes **22**, on the other hand, comprise conventional processors executing application programs on an off-the-shelf operating system. The SP nodes **22** execute tasks that are outside the limited scope of tasks executable by a content transfer engine **50**. The SP nodes **22** execute the tasks delegated by, for example, the content transfer access nodes **10** and **12** with the full overhead associated with processing requests in an off-the-shelf operating system environment.

[0065] The exemplary content transfer access node **10** includes the content transfer engine (CTE) **50** for communicating data between the external network **20** and the internal network **16**. The CTE **50** includes an external network interface engine **54**. The external network interface engine **54** passes request notifications to one of a set of event engines **58** via an event queue **60**. The event engines **58** execute content transfer daemons **59** (CTDs) that execute, by way of example, a single instance of a specific protocol. Each instance maintains data to facilitate tracking the state of the protocol interchange and generating responses after being invoked with new packets received for the particular protocol interchange. Each content transfer daemon belongs to a class that specifies the behavior/code common to all instances executing the same protocol. An external interface event queue **56** temporarily stores request notifications for transmitting content over the external network **20** received from one of the set of event engines **58**. The request notifications in the external interface event queue **56** concern transmissions from the server system to the external network **20**. Similarly, the event engines **58** communicate to the internal network **16** via notifications placed within an internal network interface engine queue **62**. Request notifications are passed from the event engines **58** to the internal network interface engine queue **62**. Thereafter, the requests are propagated via an internal network interface engine **64** to the

internal network **16**. The operation of the internal network interface engine **64** is described herein below with reference to **FIG. 3**.

[**0066**] In an exemplary embodiment of the invention, an external RAM **70**, is physically located upon the content transfer access node **10** (located, for example on a network interface card). The external RAM **70** operates as a buffer between external nodes and nodes connected to the internal network **16**. During data transfers from a node on the internal network **16** to one of the external clients **18**, data is placed within an external buffer within the external RAM **70** rather than an application buffer space located in one of the supplemental processor nodes **22**. Thus, in accordance with an embodiment of the invention, transferred data bypasses application space during the data transfers. Access to the external RAM **70** is controlled by a buffer control **72** and memory controller **74** that maintain a context for data transfer operations executed by the CTE **50**. An exemplary context arrangement is depicted, by way of example, in **FIG. 4**.

[**0067**] The CTE **50** includes multiple event engines that process notifications of buffer captures within the CTE **50** itself. There are many possible implementations for the event engines **58**. The capabilities of the event engines **58** include: having access to their own high-speed memory (referred to herein as "Internal Memory"), accepting work via the event queue **60** on the CTE **50**, allocating and de-allocating buffers using a buffer control **72** of the CTE **50**, performing DMA transfers to/from those buffers, posting events to interface engines (e.g., interface engines **54** and **64**), and posting events to other event engines. Multiple event engines **58** are implemented as co-routines on shared hardware and/or through true parallel processing. If implemented as co-routines, event engines which are not engaging in DMA transfers to/from external memory may not be blocked because another engine is performing a DMA transfer. The life cycle of one of the event engines **58** is summarized herein below with reference to **FIG. 14**.

[**0068**] Buffer Control **72**

[**0069**] The assigned usage of internal (on-chip) and external memory within the CTE **50** differs from the usage in a conventional processor. The conventional solution is for on-chip resources to be utilized for dynamic caching of content from the external memory. Generally the conventional processor is unaware of the operation of the cache, and operates as though it were truly working from the external ram. The CTE **50**, by contrast, considers its internal memory to be its primary random access memory resource. The external memory is viewed as a random access device. Asynchronous memory transfers are executed to load the internal memory from the external memory, or to store back to the external memory. Additionally, DMA input and output is executed to/from external memory.

[**0070**] The CTE **50** also manages buffer allocation, event queuing and context switching.

[**0071**] The buffer control **72** manages allocation and de-allocation of memory within the external RAM **70** corresponding to event buffers. The buffer control **72** allocates a capture buffer for capturing input for a specified context. Each context represents a distinct content transfer and an associated content transfer daemon instance. On the internal network **16** contexts are explicitly identified in the first portion of a packet. In an embodiment of the invention the context is specified within the header of the first cell.

[**0072**] Furthermore, for data transmitted over the external network **20**, application of a hash algorithm to the header fields of a packet is performed to identify a context. For a TCP/IP connection the context is identified by the combination of a VLAN (Virtual LAN) field with the source and destination IP address and source and destination TCP Port.

[**0073**] The allocation of a capture buffer within the CTE **50** follows any of a variety of scenarios including, by way of example, the following: a buffer may have been pre-loaded for the context by a content transfer daemon or extended content transfer daemon, the current state of the context specifies that a buffer should be allocated from a specified memory pool (described herein below), or the buffer may already be allocated as a result of collecting packets for the same connection that are part of a larger message. This set of options is in distinct contrast to prior solutions, which would only have pre-loaded read descriptors or the option of allocating buffer space from a common context-insensitive buffer pool.

[**0074**] One way to control allocation of buffer space is to pre-allocate memory pools that are used for particular request types or requesting entities. Allocation of a buffer from a specified memory pool is accomplished by simply removing a buffer control block from the head of a list of free buffer control blocks. Each memory pool has its own distinct free list. Allocating like-sized buffers from a linked list is a well-known practice in embedded software, although more typically employed within the context of a conventional processor.

[**0075**] Upon allocating a buffer control block, its associated use count is incremented. As will be described later herein, there are other actions that can "attach" or make claims on this buffer control block. The number of claims eventually reaches zero. When the number of claims reaches zero the buffer control block is returned to the buffer control **72**'s free pool. For a default memory pool, returning the allocated buffer to the free pool is accomplished by placing the buffer control block at the end of the free pool linked list of free memory blocks.

[**0076**] The buffer control **72**, by way of example, is invoked during routine execution to allocate a new derived or "smart" output buffer. Smart buffers do not contain the actual output, but rather describe the output desired by reference to other buffers. Each of the buffer references is one "attachment" to the referenced smart buffer. For each segment of a "smart" output buffer, the sender may request that the payload for that section be applied to the calculation of a specific CRC or checksum. Other segments may request that the accumulated CRC or checksum be placed as output and then the current value reset. In this manner CRCs dependent on the payload content can be calculated without requiring either one of the event engines **58** or supplemental processor nodes **22** to examine the payload. However, the event engines **58** and/or supplemental processor nodes **22** retain full control over what payload contributes to the calculation of a particular CRC. Without this feature the network interface engines **54** and **64** would have to understand the CRC algorithms for every protocol the CTE **50** supports. This would be particularly difficult because it is

common for the same payload to contribute to multiple CRCs at different OSI layers.

[**0077**] The buffer control **72** is also responsible for allocating and releasing any general free-standing data buffers. Free-standing buffers are used by content transfer daemons **59** to pre-stock boilerplate portions of responses. These may be included by reference in later responses. In this fashion common message elements, such as server identification strings and error codes do not have to be generated for each message that require them.

[**0078**] Supplemental Processor Node *22a*

[**0079**] A supplemental processor (SP) node **22***a,* also referred to as a "host," is capable of performing a wide variety of tasks delegated to it by the content transfer access node **10**. SP node **22***a* comprises, by way of example, its own dedicated external RAM **78**. The exemplary SP node **22***a* also includes a conventional processor **80** executing an off-the-shelf operating system **82** (e.g., WINDOWS, UNIX, LINUX, etc.). The supplemental processor **80** also executes one or more application programs **84** within the operating environment of the off-the-shelf OS **82**. The SP node **22***a* also includes the following program components: extended content transfer daemons (XCTDs) **86**, a content transfer engine extension (CTEX) **88**, and an internal network (fabric) interface **90**. Other included components of the SP node **22***a* include a FLASH memory device enabling bootstrapping the conventional processor **80**. The SP node **22***a* may additionally have its own external interfaces for administrative purposes. Such external interfaces include, by way of example, Ethernet and serial interfaces.

[**0080**] With regard to the programs executed upon the SP node **22***a,* the operating system **82** includes a kernel and drivers. A set of socket daemons execute the operating system **82** and act as Internet daemons by accepting connections through a BSD (Berkeley Software Distribution) socket application program interface. The BSD sockets API is the predominant interface used for applications running on Unix or Unix-derived operating systems that wish to interact via the Internet Protocols. The socket daemons are launched at system start-up or on-demand via a gate-keeping daemon such as INETD (under most Unix systems) which manages the process of launching daemons in response to external connection requests An example of a socket daemon is an APACHE web server. The SP node **22***a* executes the CTEX **88** as a background process/task. The only special kernel support required is permission to interface directly to the internal network interface. The SP node **22***a*'s internal network interface engine **90** performs the same function as the CTE's internal network interface engine **64**, but it may have differing performance requirements/capabilities for dealing with concurrent packet reception. The CTEX **88** bridges events/packets between the content transfer access node **10** and the XCTDs **86**.

[**0081**] Socket daemons are a specific form of applications **84** (as shown in **FIG. 2**). The methods and conventions related to writing this sort of application are well known to those skilled in the art. Indeed, a major goal of the invention is to preserve the ability to execute the vast repository of existing applications code under this paradigm while enabling the actual data transfers to proceed in a more efficient manner.

[**0082**] In an embodiment of the invention the CTEX **88** operates closely with the SP node **22***a*'s internal network interface engine **90** and access is not filtered on a per-message basis by the SP node **22***a* operating system **82**. Such access is provided, for example, by memory mapped locations and FIFO structures maintained directly by the CTEX **88** with the permission of the operating system **82**. The socket daemon applications of the applications **84** may communicate with the CTEX **88** either via the host operating system **82** kernel or via a kernel bypass interface (represented by line **91**) that avoids calls to the operating system. Examples of such bypass interfaces would include the DAFS (Direct Access File System) API and VIPL (Virtual Interface Programming Library).

[**0083**] The XCTDs **86** comprise object oriented Internet daemons operating within the context of the CTEX **88** using an event dispatch-process model. The CTEX **88** invokes XCTDs **86** with an event, and accepts results to be effected upon successful completion. The results include messages to be sent, updates to shared data regions and creation/deletion of contexts. Thus, the relationship between XCTDs **86** and the CTEX **88** is similar to the relationship that exists between the CTE **50**'s event engines **58** and their content transfer daemons (CTDs) **59**. However, the XCTDs **86** have access to the SP node **22***a*'s memory and are capable of more complex computations. The XCTDs **86** are capable of handling protocols without assistance, or alternatively act as pre- and post-filters to conventional socket-oriented application daemons.

[**0084**] In summary with regard to the role of the SP node **22***a* in the proposed streaming data server application environment, the content transfer access node **10** relieves the SP node **22***a* of the task of executing the actual transfer of requested content. The shift from "processing" the stored content to "selecting" and "wrapping" the content, changes the role of the SP node **22***a*. The SP node **22***a* is not even required for handling simple requests for data. Within the context of a server system including the content transfer access node **10**, a "host processor" node containing a conventional processor is more correctly referred-to as a "supplemental processor" due to its supporting role in content transfer operations. The SP node **22***a* is still able to control all protocol handling and participate in handling portions of protocols that are not handled by the content transfer access node **10**. To accomplish these objectives, the SP node **22***a* receives notifications that input has arrived. Under the content transfer approach, the SP node **22***a* is notified that the content is loaded within the content transfer access node **10**'s memory. The SP node **22***a* does not directly examine or process the transferred content. The SP node **22***a* is capable of reading the content transfer access node **10**'s memory when it actually must see the content. Finally, the SP node **22***a* is capable of creating buffers in the content transfer access node **10**'s memory.

[**0085**] A data storage node **14***a* is communicatively coupled to the internal network **16** via an internal network interface **92**. In an exemplary embodiment of the invention, the data storage node **14***a* comprises a set of ATA Drives **94***a* and **94***b* coupled to the internal network interface **92** via storage adaptors **96***a* and **96***b*. An example of the structure and operation of the storage adaptors **96***a* and **96***b* is provided in Phillips et al. U.S. patent application Ser. No. 09/638,774 filed on Aug. 15, 2000, entitled "Network Server Card and Method for Handling Requests Via a Network

Interface" incorporated herein by reference in its entirety including all references incorporated therein.

[**0086**] As mentioned previously herein above, the internal network **16** is preferably a non-blocking switch fabric. The data paths of the switch fabric are controlled by the server controller **24**. The server controller **24** includes an internal network interface **98**. The internal network interface **98** is, by way of example, a switch fabric interface for communicating with the other server components via the internal network **16**. The server controller **24** includes, in a particular embodiment, an event engine **100** driven by notifications presented upon an event queue (not shown). The server controller **24**, includes its own allocated external RAM **102**. The server controller **24** maintains a list of files and volumes in file system **104** and volume system **106** to facilitate access to requested content on the data storage node **14***a*. The server controller **24** also includes a storage array control **108** responsible for issuing instructions to the data storage nodes so as to fulfill the requests of other nodes and higher level components. The server controller **24** also includes a traffic shaping algorithm and controller **110** that establishes and maintains control over distribution of tasks to multiple available processing engines within the server system depicted in **FIGS. 1 and 2**. In the exemplary embodiment the storage array control and traffic shaping responsibilities are implemented by an algorithm that schedules their work jointly in a single software subsystem.

[**0087**] One of the primary tasks of a data server system embodying the present invention, and one for which it is especially well-suited, is delivery of content in the form of streaming data. In the exemplary embodiment of the present invention depicted in **FIG. 1**, the internal network **16** interfaces content transfer access nodes **10** and **12** directly to a set of data storage nodes **14** (e.g., a set of hard disk drives supplying streaming media content to a set of requesting clients). The content transfer access nodes **10** and **12** include interfaces to the external network **20**. The intention of this architecture, and alternative network data access architectures embodying the present invention, is to provide a data path between a data storage device and an external network interface for transmitting data retrieved from the data storage device to requesting clients, and that bypasses a general processor (e.g., the supplemental processor nodes **22**). In accordance with embodiments of the present invention, once a data stream commences in response to a networked client's request, the stream of requested data bypasses conventional processors and operating systems (in the supplemental processor nodes **22**) that add delay-inducing, resource-consuming overhead to such transmissions.

[**0088**] In accordance with an exemplary embodiment of the present invention incorporating the general logical arrangement of components of FIGS. **1** and/or **2**, the content transfer access nodes **10** and **12** receive requests for content stored upon the set of data storage nodes **14**. In response to the requests, possibly after initial setup procedures implemented by the supplemental processor nodes **22**, retrieved content/data passes through the internal network **16** to the content transfer access nodes **10** and/or **12** that operate without the overhead of conventional processors running conventional, off-the-shelf operating systems. The retrieved content/data is handled by one or more of the micro-engines operating within the content transfer access nodes **10** and **12**.

[**0089**] Handling is distinguished from "processing" the retrieved data. Handling, in an illustrative embodiment, includes basic packaging (or wrapping) of the retrieved data with, for example, a header, and transmitting the data according to rules specified by communications protocols. In other instances, the data is packaged by other entities such as the data storage-to-internal network interfaces. During handling the content of the retrieved data is not examined. On the other hand during processing, an application executes to substantively examine and/or modify the retrieved data. Processing is generally executed by an application executing upon an off-the-shelf operating system.

[**0090**] After performing any specified handling, the packaged retrieved data is transmitted over the external network **20** to one of the connected clients **18**. Similarly, the content transfer access nodes **10** and **12** facilitate storing data received via external network **20** onto the data storage nodes **14** without passing the data through a conventional processor/operating system incorporated into the supplemental processor nodes **22**.

[**0091**] The above-described processing differs from that of conventional processors in several ways. First, the content transfer engine **50** has no general purpose caches. Instead the internal memory is used as the primary working area, and the external RAM is viewed as a storage device. Typically processors view the external RAM as the primary working area, and on-chip memory is used to dynamically cache portions of the working area in a manner that is transparent to the software. Given that conventional processors need to support a wide range of application architectures this generalized approach is clearly superior. But the content transfer engine supports a highly specialized processing pattern, and applications can be designed to simply work from the equivalent of the internal cache directly, with the software taking explicit responsibility for transferring to from the external RAM. Second, the context switching required by the CTE **50** to support this type of transfer is extremely minimal compared to those under a conventional processor. Switching execution units only occurs between interpretive instructions (cooperative multi-tasking as opposed to preemptive) and switching between connection contexts only occurs at event dispatch. In conventional processor terms, switching to a connection context is just an application directed load and flush of a data cache. Loading and flushing data caches occurs constantly in conventional processor architectures.

[**0092**] Additionally, conventional processor architectures must save and restore registers, switch stack frames and possibly shift memory maps. Third, the CTE **50** has no need for memory mapping support. Conventional processors, and the entire VI family of protocols, go through many steps to ensure that user-mode applications can use the buffers they want to, at the memory addresses they desire. The CTE **50** provides buffers to content transfer daemons (CTDs). The CTDs do not attempt to control their own memory allocation, and hence do not really care where their data is stored, or how the buffers they are handling are addressed. The aforementioned content request handling is carried out, for example, by the content transfer access node **10** including the content transfer engine (CTE) **50** having a general arrangement depicted in **FIG. 3**. In addition to identifying a set of functional components within the CTE **50**, **FIG. 3** depicts a set of logical paths linking the identified functional

components of the CTE **50** to one another. Links are also depicted between the CTE **50** and the external RAM **70**, the external network **20** and the internal network **16**.

[**0093**] The CTE **50** includes a physical interface linking to the internal network **16**. In particular, a bi-directional data path **120** links the internal network interface engine **64** to the internal network **16**. In an embodiment of the invention, the internal network interface engine **64** supports cell-based data transmissions over the bi-directional data path **120**. However, in alternative embodiments, the data transmitted over the data path **120** is arranged in other formats in accordance with a variety of supported data transmission protocols.

[**0094**] In accordance with an embodiment of the present invention, data is transferred between the external RAM **70** and the internal network interface engine **64**—thereby bypassing costly conventional processor overhead. A bi-directional data path **122** between the internal network interface engine **64** and external RAM interface/buffer control **124** (corresponding to the buffer control **72** and memory controller **74** in **FIG. 2**) facilitates content data transfers between the internal network interface engine **64** and the external RAM **70**. A bi-directional address/data bus **126** links the external RAM interface/buffer control **124** to the external RAM **70**. The bi-directional data path **122** and bi-directional address/data bus **126** facilitate transmitting content data directly between the internal network interface engine **64** and external RAM **70**.

[**0095**] Data transfers to the internal network **16** from the external RAM **70** are executed according to commands generated by the event engines **58**. Upon generation by the event engines **58**, the commands are placed within the internal network interface engine queue **62**. Transfer of generated commands from the event engines **58** to the internal network interface engine queue **62** are represented by line **130**. The queued commands are read by the internal network interface engine **64** (represented by line **132**). The internal network interface engine **64** executes read and write operations involving the external RAM **70** and internal nodes connected via the internal network **16** (e.g., storage nodes **14** and/or the supplemental processor nodes **22**).

[**0096**] The internal network interface engine **64** is also capable of generating events (e.g., commands or messages) that drive the operation of the event engines **58**. The internal network interface engine **64** submits events (described herein below) to the event queue **60** as represented by line **134**. The queued events are read and executed by appropriate ones of the event engines **58** (represented by line **136**). A bi-directional data path **138** between the event engines **58** and the external RAM interface/buffer control **124** facilitates submission of read and write requests by the event engines **58** to the external RAM interface/buffer controller **124**. Such requests concern transfer of data between the external RAM **70** and the internal network interface engine **64** and/or the external network interface engine **54**. When capturing buffers for storing incoming data this interface is used to locate a target memory location in the external RAM **72** to store the received data, and then to store it. The posted event merely refers to the buffer collected.

[**0097**] The output command posted to the external network interface engine queue **56** will typically contain references to this data. Thus content transfer is controlled by the CTD, without requiring examination or processing of the actual packet contents.

[**0098**] Internal Network Interface Engine

[**0099**] Converting Between Packets and Cells

[**0100**] One of the functions performed by the CTE **50** is converting data between a packetized form (for transmission over the external network) and a cell form (for transmission over the internal network). As previously mentioned, in the illustrative embodiment of the present invention, data is transmitted in the form of data cells over the internal network **16**. In an exemplary embodiment of the present invention the cell/packet and packet/cell conversions are executed within the internal network interface engine **64**. Such conversion functionality is embedded into other components of the CTE **50** (e.g., the external network interface engine **54**) in alternative embodiments of the invention.

[**0101**] In it's outbound role (sending data over the internal network **16**), the internal network interface engine **64** receives event buffers from the internal network interface engine queue **62**. In response, the internal network interface engine **64** fetches a buffered packet from the external RAM **70** (via the buffer control **124**) based upon a location pointed to by the contents of an event retrieved from the queue **62**. The internal network interface engine **64** generates a set of cells from the retrieved buffered packet, and transmits the cells to the internal network **16** via lines **120**. Generating the cells includes generating and appending a CRC-**32** for the data within the buffered packet. At the end of transmitting a series of cells corresponding to the buffered packet to the internal network **16**, a buffering outbound depacketizer within the internal network interface engine **64** releases the event buffer back to the buffer control **124**.

[**0102**] An output request is made by posting an event that references a "smart buffer" to the external network interface engine queue **56**. As previously described for the external interface, the internal interface transmits each portion of the described output to a buffer set aside in the external RAM **70**. After each segment is transmitted, the claim on that portion of the smart buffer set aside in RAM **70** is released. Finally the claim on the smart buffer itself is released when the requested data has been transmitted entirely. With regard to data transmission to the internal network, the internal network interface engine **64** translates each segment into a series of internal network cells.

[**0103**] In the exemplary embodiment of the invention, the internal network interface engine **64** receives a sequence of one or more cells from the internal network **16** in accordance with known data cell delivery specifications and data communications techniques—or variations of such standards. The internal network interface engine **64** converts the received cells into packets. The internal network interface engine **64** supports concurrent construction of one packet per assigned reception channel. Each received cell corresponds to a unique reception channel that distinguishes a received cell for a particular packet from other cells received by the CTE **50** that are associated with other incomplete packets being constructed within the CTE **50**. In view of the need to distinguish packets under construction, the internal network interface engine **64** reassembles a maximum of one packet per reception channel at any given instance in time. To maintain such support, the buffering inbound packetizer of the internal network interface engine **64** stores a packet reception state for each designated reception channel (e.g., expecting packet start/middle packet, accumulated packet

length, accumulated CRC-**32** value, etc.). The buffering inbound packetizer latches a context buffer in internal RAM associated with each reception channel based upon a start cell for each packet to be constructed. The internal network interface engine **64** also invokes the external RAM interface/buffer control **124** to establish a data buffer within the external RAM **70** to the extent needed to handle an incoming packet. The options for allocating the buffer are as previously described for the external network interface engine.

[**0104**] Thereafter, the data payload in the corresponding received cells is stored within the established data buffer. When a packet is complete, the internal network interface engine **64** creates an event message. The packet completion event message is placed within a designated one of the event queues **60** associated with a particular event engine of the set of event engines **58** that will handle the completed packet.

[**0105**] Regardless of the manner in which a packet is accumulated, once it is complete, the constructed packet's CRC is validated. If the constructed packet's CRC is good, the payload is delivered to a dedicated destination. Otherwise CRC failure is tallied for statistical tracking, and the packet is dropped. For buffered inbound packetizing, when a packet is invalid (e.g., a failed CRC-32 check), the buffering inbound packetizer of the internal network interface engine **64** releases the current buffer associated with the incoming data packet.

[**0106**] External Network Interface Engine

[**0107**] Having described the portions of the CTE **50** most closely associated with data transfers over the internal network **16**, attention is now directed to portions of the CTE **50** components that facilitate data transfers between the external RAM **70** and the external network **20**. Such transfers are performed, for example, by the CTE **50** to provide retrieved data from one of the data storage nodes **14** to a requesting client via the external network **20**. The external network interface engine **54** receives and transmits packets to/from the external network. In an exemplary embodiment the external network interface engine **54** manages a device bus and the network interface engine devices on the device bus. In accordance with an embodiment of the present invention, data is transferred between the external RAM **70** and the external network interface engine **54**. A bi-directional data path **142** between the external network interface engine **54** and external RAM interface/buffer control **124** facilitates content data transfers between the internal network interface engine **64** and the external RAM **70**. The bi-directional data path **142** and bi-directional address/data bus **126** facilitate transmitting content data directly between the external network interface engine **64** of the CTE **50** and external RAM **70**.

[**0108**] To capture packets, the external network interface engine **54** responds to a packet ready condition (typically in the form of a status line and/or interrupt from an actual network interface engine device) by transferring just the packet header from the actual interface. This is illustrated in **FIG. 13** as the "headerReceived" transition from the "idle" state.

[**0109**] Fields of the header are used to find the hash entry for this connection. If none exists a new context is allocated and placed in the hash table. Fresh context allocations come from a reserved pool of "unknown connection" contexts. This limits the total resources that can be tied down responding to new external network connection requests.

[**0110**] Such limits represent yet another manner in which the system defends against Denial-of-service attacks. If the context resource pool is exhausted, then the incoming frame is discarded. A diagnostic tally of all such discards is kept. After assigning a context, the header is examined to determine the capture type (serial or RDMA) and the total packet size.

[**0111**] Having determined the capture type and packet size, the context is latched for the identified capture type. As detailed herein, latching the capture buffer for a context finds or allocates a buffer to capture the incoming data. Once a capture buffer is allocated, the fetched header is stored in that buffer. Then a DMA transfer from the external network device to the capture buffer is requested.

[**0112**] The DMA transfer either completes successfully (as shown by the "goodTransferNotice" transition in **FIG. 13**) or an error is detected, such as an invalid CRC (as shown by the "badTransferNotice"). If the DMA transfer is successful the captured bytes are committed. At this point the immediate collection is complete, and the context is unlatched. In an external protocol dependent fashion the interface engine determines whether use of the current buffer has ended. If use of the current buffer is done, then the buffer is released. Otherwise the engine returns to an "idle" state. The one condition where a buffer is always "done" is when it has been completely filled. On a bad transfer notice the context is unlatched and the buffer is aborted.

[**0113**] External Network Output Requests

[**0114**] Data transfers between the external network **20** and the external RAM **70** are executed according to commands generated by the event engines **58**. Upon generation by the event engines **58**, the commands are placed within the external network interface engine queue **56**. Transfer of generated commands from the event engines **58** to the external network interface engine queue **56** are represented by line **144**. The queued commands are read by the external network interface engine **54** (represented by line **146**). The external network interface engine **54** executes read and write operations involving the external RAM **70** and external nodes connected via the external network **20** (e.g., clients **18**).

[**0115**] The external network interface engine **54** is also capable of generating events (e.g., commands or messages) that drive the operation of the event engines **58**. The external network interface engine **54** submits events (described herein below) to the event queue **60** as represented by line **148**. The queued events are read and executed by appropriate ones of the event engines **58** (represented by line **136**).

[**0116**] Context Mutual Exclusion by Serialization

[**0117**] One requirement of object-oriented network processing is that the system must ensure that the events for any given object must be processed serially. The processing of event N for object/connection X must be completed before the processing for event N+1 for object/connection X starts. Conventional approaches to solving this problem include assigning each object to a single processing thread, and use of semaphores or locks to ensure that if two threads do try to update the same context, that the second one will be

forced to wait for the first's completion. These conventional solutions consume system resources and can result in delays in handling events. Static assignment of threads is low overhead, but can result in some threads being idle while others are backed up in processing their input queue.

[0118] The CTE **50** solves this problem by dynamically assigning a context/object to a specific execution unit when it makes a dispatch. If there are no currently dispatched events for that context, one of the event engines is picked arbitrarily. To ensure that this selection is statistically balanced, and that it can be done without requiring references to tables stored in external memory, the preferred implementation is to make the selection based upon a hash of a dispatched routine handle. However there are a wide range of algorithms to make a balanced selection with low overhead.

[0119] When a dispatch is made the CTE increments a count of dispatched events for that context, and records a selected event engine of the event engines **58**. As long as that count remains non-zero, all future dispatches for this context/object will be placed on the same queue. At the completion of dispatching each event, the selected event engine will decrement the dispatch count for the context/object. When the count returns to zero, the choice of event queue is once again unlatched and can be dynamically assigned to a later event associated with the context/object.

[0120] Queues

[0121] In an exemplary embodiment, queues are a fundamental element for interfacing the execution units (engines) and the CTDs **59** and device control routines that they run. Each of the queues depicted in **FIGS. 2 and 3** feeds a single execution unit. When a calling component seeks to send a message to another component of the CTE **50**, the calling component posts a message to the target queue. Posting may be a result of buffer capture or may be the result of a successful event dispatch by an event engine. The receiving component is responsible for fetching the received messages from its queue.

[0122] Furthermore, events need not always be posted immediately to a particular queue when they arise. In an exemplary embodiment of the invention, a ticker within the internal network interface engine **64** posts events received from a time deferred request queue (not shown in the figures). The ticker accepts event messages and buffers them for a designated time period. The ticker reposts the event message to an appropriate one of the event queues **60** after a designated wait period specified in the received event message.

[0123] In an embodiment of the invention, message queues are implemented entirely upon a chip containing the processing components of the CTE **50** (i.e., no external, off-chip memory is utilized), and there is no access delay for making an external call to off-board memory. Because, in the exemplary embodiment, the message queue storage is on-chip, the capacity of each queue is not dynamically modifiable. Instead, changes to queue size are carried out in FPGA core or based upon a parameter accessed when the FPGA is initialized.

[0124] The various queues **56**, **60**, and **62** store request "notifications" rather than the requests themselves. The request notifications, an exemplary format of which is discussed herein below with reference to **FIG. 10**, include a command and a pointer to additional instructions and/or data stored within external random access memory **70** managed by the buffer control **72** and memory controller **74**. Transferring notifications, rather than the buffered requests themselves, between components reduces memory transfers within the data access server system embodying the present invention and streamlines data transfer procedures.

[0125] As those skilled in the art will readily appreciate, the above described hardware/firmware arrangement depicted in **FIG. 3** is exemplary of an architecture for facilitating transfer of large amounts of data, retrieved from a storage device via the internal network **16** (e.g., switch fabric), to a requesting external device connected via an external network. The architecture of the content transfer access node **10** depicted in **FIGS. 2 and 3** has only limited processing capabilities for facilitating the data transfers. The result is a relatively inexpensive high-throughput server interface for providing information assets to connected clients over an external network.

[0126] Turning now to **FIG. 4**, an exemplary context format **200** and related data structures are depicted. The context represents the state of a particular task being performed by the CTE **50**. A class state ID **202** provides a pointer to a class state record **203**. The class state record **203** includes a memory pool ID **204** that corresponds to a particular buffer pool from which buffer space is allocated to a particular context object. A dispatch target **205** describes a particular type of action to be performed by the particular context object to which the context is assigned. In an embodiment of the invention, the class state also includes an event engine routine handle that identifies a code routine executed in association with the particular class state.

[0127] A set of memory pools are associated with each of a set of capture buffer types. The memory pool ID **204** points to a particular memory pool record **206** associated with one of the set of capture buffer types. A memory pool record includes two fields: a list head field **208** and a list tail field **210**. The list head field **208** stores a value corresponding to the location of a first free buffer control block in a linked list of free buffer control blocks representing unused ones of memory blocks allocated to the particular memory pool. The list tail field **210** stores a value corresponding to the location of a last free buffer control block in the linked list of free buffer control blocks. In an illustrative embodiment the list head **208** and list tail **210** comprise a total of 48 bits. The first four bits are reserved, the next twenty bits specify a location in the external RAM **70** where the first free buffer control block is located, the next four bits are zero, and the final twenty bits specify a location in the external RAM **70** where the last free buffer control block for a particular memory pool is located.

[0128] For each network interface engine (e.g., internal and external), the context includes a corresponding capture buffer descriptor **220**, **240**. Each internal network capture buffer descriptor **220** and external network capture buffer descriptor **240** includes an optional handle (pointer or other reference) referencing a capture buffer having a format of the type depicted by capture buffer control block **222**. Non-null handles indicate that the process of capturing a buffer for this context on that network interface engine is in-progress. Each instance of the capture buffer control

block **222** includes, by way of example, a copied to buffer address **224** that identifies a location where a buffered data block commences. A logical length **226** identifies the portion of the buffer that has fully captured (or otherwise logically valid) content. By contrast, an allocated length field **228** identifies the total length of the memory allocated to a particular capture buffer. As discussed previously above, the dispatch target field **205** includes one of an extensible set of various target descriptors (see, **FIG. 9**). As indicated by target queue **230**, in addition to pointing to a particular memory pool (field **229**), the capture buffer can also point to a particular event queue that receives notifications/requests for a particular class of tasks. As indicated by arrows **232**, the event queues **60** in turn include queue entries that specify buffer ID's pointing back to an instance of the capture buffer control block **222**. Furthermore, a smart output buffer **234**, in an embodiment of the present invention, includes a list of segment descriptor/buffer reference pairs that reference multiple instances of the capture buffer control block **222**. In such cases, the segment descriptor specifies an operation/task, and the buffer reference (as depicted by lines **236**) specifies a particular instance of the capture buffer control block **222** upon which the operation/task is to be performed.

[**0129**] Sub-fields within the internal network capture buffer descriptor **220** of the context **200** also track a commit limit and total committed fields. These sub-fields support reporting of partial delivery over the internal network when the out-of-order capability of RDMA transfers is invoked. An exemplary buffer allocation scheme involving partial delivery is described herein below. A buffer use count **238** stores a value indicating a the capture buffer capture buffer is released.

[**0130**] The external network capture buffer descriptor **240**, as mentioned above, stores a reference (optional) to an instance of a capture buffer control block **222**. Instances of external network capture buffers correspond to data transfers from the external network **20** into the external RAM **70**. An assigned engine queue **250** specifies one of the set of event engines **58** that has been assigned to handle the task associated with the particular context.

[**0131**] As explained herein above, the CTE **50** dynamically assigns a context/object to a specific execution unit when it makes a dispatch. The dispatch count **260** identifies the number of dispatches associated with a particular context. If there are no currently dispatched events for that context, one of the event engines is picked arbitrarily. When a dispatch is made the CTE increments a count of dispatched events for that context, and records a selected event engine of the event engines **58**. As long as that count remains non-zero, all future dispatches for this context/object will be placed on the same queue. At the completion of dispatching each event, the selected event engine will decrement the dispatch count for the context/object. When the count returns to zero, the choice of event queue is once again unlatched and can be dynamically assigned to a later event associated with the context/object.

[**0132**] In an exemplary embodiment of the present invention all buffers are referenced by twenty-bit buffer IDs that are indexes to buffer control blocks. Four distinct types of buffer control blocks are supported: tables, code routines, allocated buffers and free buffers. The buffer control blocks for tables and code routines are **64** bits each, and **128** bits are required for allocated and free buffers.

[**0133**] A pre-defined location holds the buffer control block for buffer ID zero. This is pre-defined to be a table that holds the buffer control blocks for all tables and code routines. Buffer ID "one" is reserved for the table holding the buffer control blocks for allocated and free buffers. The first element in the table holding the buffer control blocks will have a buffer ID higher than the buffer ID of the last entry in the table/code-routine table.

[**0134**] Turning now to **FIG. 5, a** 64-bit table memory control block format is defined. A table buffer address **300** points to the first entry in a table containing memory control blocks (both 64 and 128-bit). The next 8 bits, field **302** of the memory control block are zeroes (a format marker). A sixteen-bit record size **304** specifies the size of each record. An allocated table size **306** (only 16 of 24 bits are specified) indicates the number of bytes allocated to hold the records; this must be a multiple of the record size.

[**0135**] An exemplary 64-bit code routine memory control block format is defined in **FIG. 6**. While their content and usage is different, code routine memory control blocks are identical in format to table control blocks. Code routine memory control blocks include eight 1's as a format marker **312** and all zeroes in the record size **314**. Rather than a buffer size, routine size **316** stores a size of a code routine.

[**0136**] Turning to **FIG. 7**, the fields are depicted for an allocated buffer control block. Types of buffers include capture (described herein above with reference to **FIG. 4**), direct and smart output. Each instance of a 128-bit memory control block includes a copied to buffer address **320** that identifies a location where a buffered data block commences. A logical length **322** identifies the portion of the allocated space that is logically usable. For output buffers this would indicate to the network interface engine how much of the buffer should be examined for output specifications (or data). For capture buffers it indicates how large the fully committed portion of the buffer is. An allocated length **324** identifies the total length of the memory allocated to a particular allocated buffer. A format marker **326** indicates whether a particular buffer is a smart output buffer, simple output buffer, or a capture buffer. A memory pool ID **328** indicates which one of a total of 256 potential memory pools with which a particular memory buffer is associated. A designated queue **330** identifies the event queue of the set of event queues **60** with which the memory buffer is associated. An event target **334** comprises a description of a particular event and specifies how to process the buffer's contents. Examples of event formats are described herein below with reference to **FIG. 9**. A buffer use count field **336** identifies the number of users of a particular buffer. The buffer use count field **336** is incremented when a buffer is allocated or attached. It is decremented each time a claim on it is released. When it is decremented back to zero it will be placed back in its assigned home memory pool by appending it to the tail of that pool's free list.

[**0137**] A free buffer control block format (one that has not been allocated to an event) is summarized in **FIG. 8**. The free buffer control block format is similar to the buffer control block depicted in **FIG. 7**. The identifying difference is that a free buffer has a zero use count, while an allocated buffer has a non-zero use count. The memory address, allocated buffer length and use count fields are aligned, so as to allow references to buffer control blocks without prior

knowledge as to whether the specific control is free or in use. Each instance of a 128-bit memory control block includes a copied to buffer address **340** that identifies a location where a buffered data block commences. A next set of 24 bits **342** are reserved (all zero). An allocated length **344** identifies the total length of the memory allocated to a particular free buffer. A memory pool **348** indicates which of a total of 256 potential memory pools with which a particular memory buffer is associated. A next set of twenty-eight bits **350** are undefined. They may have leftover content from when this buffer control block was last allocated. A next free buffer **352** (20 bits) identifies the address of a next free buffer control block (thereby enabling chaining of free buffer control blocks). A buffer use count field **354** identifies the number of users of a particular buffer. In a "free buffer" this value is zero by definition—if there were claims upon the buffer then it would not be a free buffer.

[**0138**] Turning now to **FIG. 9, a** set of exemplary event targets (40 bits) are depicted. In general, the first section of an event comprises a format marker field specifying an event type. This is a variable length "Huffman" style encoding, which is well-known technique in the field.

[**0139**] The first type **400** (Internal Network Target) specifies the packet header information required for an Internal Network Target. This format is used for output requests placed on the internal network event queue. Normally this is only used for output requests from CTDs, but a context state may specify forwarding event notices directly to an XCTD via the internal network.

[**0140**] The second type **401** (Table Update) is used by a CTD or XCTD to send updates to specific tables. This mechanism is used to bootstrap the CTE. The target specifies which table is being updated (16 bits) and a record offset of the new data (20 bits). Normally both the 'set' and 'clear' flags are set, causing the entire content of the addressed records to be replaced. By setting only 'set' or 'clear' the updater may request bit-wise setting or clearing of bits in the target record.

[**0141**] The third type **402** (Ticker Target) is used by CTDs to request notification after a specific period of time has elapsed. The target specifies the context that is to receive the event and the minimum number of implementation-defined clock ticks that must transpire first.

[**0142**] The fourth type **403** (External Network Target) is used by the CTD to request output on the external network. The target details the set of external network ports that are acceptable for sending this traffic. In most configurations the CTD finds all ports acceptable, but there could be configurations where different ports lead to different sub-networks.

[**0143**] The fifth type **404** (Context Target) is used upon buffer capture and for communication between CTDs. It specifies the target context (20 bits), the generation (4 bits) and the capture type (2 bits). The generation is zero for packet capture. For CTD generated events it is any number greater than that of the event currently being processed. This limitation prevents "chain reactions" in faulty software where a single event results in posting two events, which result in posting four events, etc.

[**0144**] The sixth type **405** (Simulated Input Target) allows a CTD to simulate input with a supplied buffer that is to be treated as though it were raw input received upon the specified channel. For example, the internal network interface engine could be told to simulate reception on a given Node Channel and then be supplied an array of cells. This feature is intended to support debugging and diagnostics.

[**0145**] As mentioned herein above, the CTE **50** includes a set of message queues **56**, **60** and **62** that buffer request/command transmissions between internal components of the CTE **50**. Referring to **FIG. 10**, in an embodiment of the invention, each message is 3 bytes (24 bits). The first four bits **410** identify a specific event completion notification type. The remaining twenty bits **412** comprise a pointer to buffer in the external RAM **70** associated with the event and created by the external RAM interface/buffer control **124** of the CTE **50**.

[**0146**] Turning now to **FIG. 11, a** set of buffer depictions illustrate another aspect of an embodiment of the present invention enabling buffers to fill in an out-of-order manner while still enabling prompt use of totally delivered portions of the buffer. An important feature of an exemplary embodiment of the invention is the ability to issue large read requests and then receive partial completion notifications as the requested content is delivered. This approach allows the storage network to exercise flexibility in scheduling deliveries.

[**0147**] The more conventional approach would use more, smaller buffers so as to allow for a steady flow of completion notifications. The conventional approach, however, reduces the flexibility of the scheduler. Depending on the internal network protocols, the requests may have to be dealt with in sequential order. At a minimum, scheduling would be constrained to respect the boundaries between the requests. A single transfer could not be scheduled that crossed the artificial boundaries between sequential requests.

[**0148**] When files are stored in non-contiguous sectors on storage media maintained on the storage nodes **14**, optimum scheduling of the drive head may require reading blocks "out of order", i.e. in an order that differs from the logical arrangement of the data in the file. Requiring the server to deliver the blocks in order either causes more disk head motion or storage-side buffering (until the complete file is retrieved). Out-of-order delivery facilitates the greatest flexibility for disk data delivery scheduling while eliminating the need for extra disk side buffering. Augmenting out-of-order delivery with early completions allows the use of larger buffers without the increasing pipeline delay that waiting for complete delivery of those larger buffers would entail. As shown in **FIG. 11**, the ability to support partial completions with out-of-order delivery is achieved by logically dividing each capture buffer into three zones (uncommitted, partially committed and fully committed).

[**0149**] **FIG. 11** depicts a sequence of commits to a data capture buffer. Initially, the buffer is completely empty. The first commit operation adds a data block having a size of "4" and an offset "10." Thus, since a gap exists at the beginning of the buffer, the fully committed value remains at zero. The value of the highest location in the buffer containing data, "14", is stored in the total committed field. Total committed data size is set to 4.

[**0150**] Next, a data block of size "6" is added to the buffer starting at location zero. The total committed is increased to 10, the highest filled location remains "14," and the fully

committed range in the data buffer is increased to "6." The next commit is size "4" beginning at offset "6." The commit limit remains unchanged at "14." At this point the total committed is increased from "10" to "14" and now equals the commit limit (now "14"). Therefore, the fully committed field can be updated to "14. " A partial completion event can now be posted that specifies the readiness of data up to location **14** to be transmitted from the data capture buffer to a specified destination.

[0151] The remaining two diagrams of **FIG. 11** depict the completion of two more commits. The first of the two adds data to the end of the buffer and increases the total committed and the commit limit. The second of the two commits adds data to fill a gap and complete the space of the data capture buffer.

[0152] Turning now to **FIG. 12, a** set of states and transitions are depicted that summarize the creation of data packets from a set of received cells in accordance with an embodiment of the present invention. There are two primary states "Not In Packet" state **700** and "In Packet" state **716** between these two states, cell collectors perform sequences of operations in association with state transitions. From the "Not In Packet" state **700** the normal course of events is to receive a "startCell". The startCell is processed as follows. At action **702** the packet size is extracted from the start cell header. Next, at action **704** the capture buffer is latched for the type of capture (serial or RDMA) and the packet size promised. If during action **704** no capture buffer is available, an ovverun is tallied at action **706** and the collector returns to the "Not In Packet" state **700**.

[0153] Otherwise if a capture buffer is available, then at action **708** the CRC accumulation field is zeroed. Next, at action **710** the packet size collected field is zeroed. Next, at action **712**, the base address for the collection (within the capture buf) is obtained. For RDMA packets this is the RDMA offset from the packet base. For serial packets it is after any previous packets already collected for this buffer. Thereafter, at action **714** the cell's payload is stored, with the CRC and length being accumulated. The collector transitions to the "In Packet" state **716**.

[0154] From the "Not In Packet" state **700** the following error handling reactions are required. On receipt of an "endCell" the "nMissedEnds" tally is incremented at action **720** and then the Not In Packet state **700** is reentered. On receipt of a "midCell" the "nStrayCells" tally is incremented at action **722** and then the Not In Packet state **700** is reentered.

[0155] From the "In Packet" state **716** the following is done in response receipt of a "midCell". In response action **714** stores the cell's payload, with the CRC and length being accumulated. The In Packet state **716** is reentered.

[0156] From the "In Packet" state the following operations are executed in the following identified states in response to receiving an "endCell." The cell's payload is stored per action **724**, with the CRC and length being accumulated. Action **726** checks the accumulated CRC. If it is bad, the capture buffer is aborted at action **728**, the bad packet counter is incremented at action **730**, the context port is unlatched at action **732** and the collector returns to the "Not In Packet" state **700**. Otherwise, at action **726** if the CRC is valid, the collected bytes are committed at action **740** and the context port is unlatched at action **732** before the collector returns to the "Not In Packet" state **700**.

[0157] From the "Not In Packet" state a "solo cell" may also be received. A solo cell is processed as though it were both a start and end cell. After extracting the packet size at action **760**, at action **752** a capture buffer is latched for the packet size promised. If during action **752** no capture buffer is available, an ovverun is tallied at action **754** and the collector returns to the "Not In Packet" state **700**. Otherwise, action **756** is entered wherein the CRC accumulation field is zeroed.

[0158] While in the "In Packet" state **716** either a "soloCell"or "startCell" may be received. After aborting processing of a current packet, the solo and start cells are processed as though the collector was in the "Not In Packet State"**700** after the current packet has been aborted. Aborting the current packet accurs by aborting the current capture buffer at action **760** and **762** for start and solo cells respectively. The context port is unlatched at action **764** and **766**, and the "nMissedEnds" tally is incremented at actions **768** and **770** for start and solo cells respectively. The remaining steps concern processing the cell as though the cell was received while the collector was in the "Not In Packet State"**700**. In particular, a promised packet size is extracted from a start cell or a solo cell at stages **772** and **750**, respectively.

[0159] Turning now to **FIG. 13, a** set of states summarize how Ethernet frames are collected into a packet in accordance with an aspect of an illustrative data server interface. Starting from an "Idle" state **800** the collector responds to the "headerReceived" notice as follows. At action **802** the header contents are read, and used to calculate a hash value. The hash value is used at **802** to find the entry in the hash table where this context should be. If it does not already exist, then a new context is allocated if there are any available in the context buffer pool for that purpose.

[0160] If at action **802** no context existed or could be allocated ("[no context available]") then the "MaxNewContextDenials" tally is incremented at action **804** and the Ethernet frame is flushed at action **806**. The collector then returns to the "Idle" state **800**.

[0161] Otherwise if a context existed or can be created, then at action **810** the capture type (serial or RDMA) and packet size is determined from the header. Based upon the acquired information, at action **812** the capture buffer is latched for the port and capture type. If required by the external protocol, the read header information is transferred into the capture buffer at action **814**. Next, at action **816** an RDMA transfer is requested of the frame payload to the capture buffer and the "Transferring payload to Capture Buffer" state **820** is entered. The "transferring payload to Capture Buffer" state **820** completes with either a successful transfer or an error. In either case the collector returns to the "Idle" state **800** (albeit via differing paths of sub-states/operations). In the case of a good transfer, at action **822** the collected bytes are committed and at action **824** the context is unlatched. Otherwise, in the case of a bad transfer the capture buffer is aborted at action **826**, and the context is unlatched at action **824**.

[0162] **FIG. 14** summarized states and substates/operations within the life cycle of an exemplary one of the event engines **58**. Starting with a Waiting for Event state **1000**, the

event engine is activated. The Waiting for Event state **1000** occurs continuously in a parallel execution architecture, or by having another event engine block in a co-routine implementation. As a result of activation the event engine will enter the Dispatching Event state **1002**. The Dispatching Event state **1002** state will be explored in more detail later herein with reference to **FIG. 15**. Normally this state is exited with an "ok" status, indicating that an event has been dispatched and the execution of its response can now begin at Instruction state **1004**. Alternately, state **1002** can exit with a "queue empty" status and return to the "Waiting for Event" state.

[**0163**] Execution proceeds from the Instruction state **1004** by fetching the next instruction from internal memory, the opcode is decoded at state **1006** and the implementation of that opcode is invoked during state **1008**. The execute opcode state **1008** exits in one of four available methods. The result state transitions are labeled as "ok-done"**1010**, "ok-DMA requested"**1012**, "successful completion"**1014** and "exception"**1016**.

[**0164**] The "ok-done" transition **1010** that returns to the Fetch Execution state **1004** indicates that the instruction has completed, and the event engine is ready to execute the next instruction. In co-routine implementations there is a possibility that the CTE **50** could shift execution to another unblocked event engine between any instruction.

[**0165**] The "ok-DMA requested"transition **1012** indicates that the engine has requested a DMA transfer to or from external memory from buffer control. The event engine will be blocked in wait state **1013** until that transfer has completed. At that point the event engine transitions to the fetch instruction state **1004**. In a co-routine implementation, the CTE shifts execution to an unblocked event engine.

[**0166**] The "successful completion" transition **1014** indicates that the event has been fully processed without an exception being raised. This transfers to a Completing Successfully state **1015** which is explained with reference to **FIG. 16**.

[**0167**] The "exception" transition **1016** occurs when an instruction raises an exception during the execution of a routine instruction. Causes of exceptions include divide by zero, failure to complete before the dispatch deadline or use of an invalid index. A "Completing with Exception" state **1018** is described herein below with reference to **FIG. 17**.

[**0168**] Referring now to **FIG. 15**, to dispatch an event an event engine first consumes an event from its event queue at state **1100**. If the event queue is empty, state **1100** is exited and state **1102** is entered with a "queue empty status". Otherwise at action **1104** a buffer ID is extracted from the consumed event. At action **1106** a DMA read is requested of a buffer control block indexed by the buffer ID. A wait state **1108** is entered.

[**0169**] When the DMA read completes, at action **1110** a target context ID is extracted from the buffer control block. This is used as the key to initiating a DMA read of the Context itself at action **1112**. The event engine waits at state **1114** for the DMA read to complete.

[**0170**] When that read is complete at action **1116** the fault status of the retrieved context is checked. If the context has been quarantined due to uncorrected faults control passes to action **1118** and the current buffer is released. The event engine returns to state **1110** and attempts to fetch an event from its input queue.

[**0171**] Otherwise at action **1120** the event engine gets the current context state from the retrieved context and starts a DMA read of the context state's data at action **1122**. The event engine waits at state **1124** for the context data read to finish.

[**0172**] Upon completion at action **1126** a routine handle is extracted from the context. At action **1128** the event engine ensures that the cache register is set and sets the program counter during action **1130**.

[**0173**] If the code routine is already found within the engine's code cache then during action **1130** the program counter merely needs to be pointed at that location and the "ok" exit can be taken by the event engine during state **1132** to start execution of the routine.

[**0174**] Otherwise after the program counter is initialized during action **1130**, the code routine is DMA read into the code cache. A DMA read is initiated during action **1134** for the code routine's control block. After waiting at state **1136** for the DMA read to complete, at action **1138** the event engine extracts the address and size of the code routine. Next, at action **1140** a DMA read of the code routine is initiated and the event engine waits for its completion during state **1142**. Thereafter, the event engine enters the exit state **1132** and execution of the loaded code routine is enabled.

[**0175**] Referring to **FIG. 16**, upon successful completion the event engine determines at state **1200** if all events that it must post are currently postable. If not, and none of the target queues have their matching execution units in the "Other Full" state then at state **1202** the event engine sets its own "Other Full" status flag and at state **1204** waits until it is time to recheck the postability of the dispatch results. Thereafter, at state **1206** the event engine clears its "Other Full" flag and returns to state **1200**. If at state **1200** waiting could create a deadlock (a target "Other Full" flag is already set) then the event engine must raise the deadlock exception and complete handling of the dispatch as described for the "Completing with Exception" state described below with reference to **FIG. 17**.

[**0176**] Otherwise the event engine creates any required log entries at state **1210**, DMA write them to external memory at wait state **1212**. At state **1214** the event engine posts the events to the other queues, and at state **1216** the event engine DMA writes its context data and state back to external memory. The event engine waits for the DMA write to complete at state **1218**. When the DMA write is complete at state **1220** the event engine decrements the context's dispatch count, and then at state **1222** releases the buffer associated with the event.

[**0177**] Referring to **FIG. 17**, when an exception has been raised, all pending results of the current dispatch must be discarded. This requires at state **1300** releasing the buffers associated with the events that would have been posted upon a successful completion. When possible, at state **1302** the event engine allocates an exception report buffer. There is a pre-designated report buffer for each type of event. At state **1304** that report buffer is filled and placed at the head of the event engine's input queue. At state **1306**, the event will report to the object itself that it faulted, and require it to

validate that its internal data is corrupt. If it cannot do so, or faults while attempting to do so, the context will be marked as quarantined. No further events will be dispatched to it. The exception report buffer will typically have a reference to the original buffer, which will require that it be attached. After handling the exception report buffer, at state **1308** the engine will release its claim on the event buffer that caused the exception and at state **1310** a signal is provided that the event was processed (albeit unsuccessfully).

[**0178**] If the report buffer is full, then the event engine passes from state **1306** to state **1312** wherein it increments a counter that tallies lost exceptions. Next, at state **1314**, the event engine releases its claim upon the exception report buffer. Lastly the event's context dispatch count must be decremented, just as it would have been with a successful completion.

[**0179**] **FIG. 18** illustrates the life cycle of a buffer control block. It starts in the "Free" state **1400**. After buffer control **72** removes the buffer control block from its memory pool's free list, it is assigned to perform a specific capture type for a specific context. This involves setting the initial use count to 1 at state **1402**, initializing the logical length to zero at state **1404**, and setting the event target at state **1406**. The Event Target encodes the target context, the selected target queue, the generation and the capture type (for generation zero events). Target context and queue selection have been described previously herein. The generation field is set to zero when the buffer is being allocated for capture. When events are sent by CTDs to other CTDs the generation field must be set to a value higher than that of the event which is currently being processed. The four types of capture are internal network serial mode, internal network RDMA mode, external network serial mode and external network RDMA mode. A given context may have at most one active capture in-progress for each capture type.

[**0180**] For RDMA captures two additional fields (Total Committed and CommitLimit) must be initialized to zero at states **1408** and **1410**. These fields are found in the context data, rather than buffer control block to avoid wasting their space on buffers that are not supporting RDMA capture. The buffer is now in the Allocated state **1412**. From the allocated state **1412** the buffer may be further attached, which causes the use count to be incremented at state **1414**. The buffer returns to the Allocated state **1412**.

[**0181**] Also from the allocated state **1412** the buffer may be committed at state **1416** or **1418**. The act of committing a portion of a capture buffer indicates two things. First it indicates that the packet collector has received a portion being committed and the portion has been validated by a capture specific method, most typically a CRC**32**. Second, by committing a portion of the capture buffer, the packet collector yields its permission to further modify those bytes. Committed bytes are ready for processing by an event engine.

[**0182**] For serial captures the Logical Length is simply incremented by the number of newly committed bytes at state **1416**. For RDMA (out-of-order) captures the Commit Limit, Total Committed and Logical Length fields must be updated as follows: if the base of the newly committed data (Commit.base) matches the current CommitLimit, then CommitLimit is incremented by the newly committed size (Commit.size) at state **1418**, TotalCommitted is incremented by the newly committed size (Commit.size) at state **1420**, if the LogicalLength (which is the portion fully committed) is equal to the newly committed base (Commit.base) then the Logical Length is incremented by the newly committed size (Commit.size) at state **1422**, and lastly if all bytes below the CommitLimit have been committed (TotalCommitted is equal to CommitLimit) then the LogicalLength can be set to the TotalCommitted at state **1424**. The rationale for out-of-order commits was discussed more fully with reference to **FIG. 11**.

[**0183**] After byte counter maintenance steps are completed the number of users is incremented, at state **1426** and the event is posted to the target queue at state **1428**. If the target queue is full, then at state **1430** the execution unit's status is marked as "Other Full" before the execution unit suspends itself at state **1432**. The "Other Full" status will prevent other execution units from suspending while trying to post output requests to this execution unit's queue. This prevents the well-known "deadly embrace" deadlock, where two execution units are each waiting for the other to empty its input queue, but neither can because they are waiting for the other to act first. The execution unit will be activated from this "waiting to post" state **1432** after any execution unit has completed processing an event. The flag is cleared at state **1434** and the post re-attempted at state **1428**.

[**0184**] As each claim on the buffer is released, at state **1440** the number of users is decremented. Alternatively when the last claim is released at state **1442** the buffer is restored to the tail of its home memory pool and returns to the "free" state **1400**.

[**0185**] Turning now to **FIG. 19, a** set of steps are depicted that summarize an exemplary sequence of events/operations for transmitting requested data from a data server to an external client node. It is noted that in general, the disclosed architecture of the CTE **50** enables applications, through the use of status/control/notification messages, to control the transfer of data from a data storage node to a requesting client for Internet, NAS, SAN and similar protocols without incurring overhead associated with placing transferred data payloads in the memory space of application software.

[**0186**] In an embodiment of the present invention applications are developed as a set of object classes. The resulting objects are invoked either as CTDs **59** on the CTE **50** itself or as XCTDs **86** on a companion CTEX **88** running on the supplemental processor node **22***a*. The CTE **50** captures incoming packets and dispatches corresponding events to a context object to which a particular client request is assigned. By way of example, the context objects access shared tables, generate output requests, generate messages to other objects and modify their own data. The CTEX **88** performs the same function for extended content transfer daemons (XCTDs) on the supplemental processor node **22***a*.

[**0187**] With reference now to **FIG. 19**, an exemplary connection and corresponding client request includes the following operations/steps. Initially at step **1500** an incoming request from the external network is identified as being a new connection. In response, at step **1502** a new context (context object) is assigned and an entry is created in the external network interface's packet identifying hash tables associating the new connection with the new context.

[**0188**] Thereafter, at step **1504** the context object completes initial validating negotiations and establishment of the

connection with a requesting client. An example of such negotiations is completing the TCP three-part handshake. Due to the wide variety and/or potential complexity of this step, finishing establishment of the connection, particularly validation of the user and any supplied credentials (such as user password) will typically be passed by an event engine within the CTE **50** to an XCTD running on the CTEX **88**.

[**0189**] After the CTEX **88** validates the session, at step **1506** the CTEX **88** reassigns the external network packet identifying the connection/context hash table entry to a new context object that handles the in-session protocol for responding to the client request(s). The original context object is returned to the available pool of context objects for processing/validating unknown connections.

[**0190**] Next, at step **1508** the new context object executing on the CTE **50** and associated with an in-session protocol issues/passes requests over the internal network to obtain content requested via the validated client connection. This is performed, by way of example, on a read-ahead basis that is primarily regulated by available buffering rather than as a direct response to a sequence of incoming requests. The requested data is transferred from one or more of the storage nodes **14** via the internal network **16** directly to a content transfer access node **10**, thereby bypassing the supplemental processor nodes **22**. The content transfer engines generally keep supervisory applications executing upon the supplemental processors aware of the status of data transfers and connections. However, the status knowledge is acquired through status notifications passed to the supplemental processor rather than direct observation of the transferred data content by the supervisory applications executing on the supplemental processor.

[**0191**] During step **1510** the in-session context object issues protocol-specific output messages (requests/responses) to the connected external client as material is available and subject to any pacing specifications. The potential applicable data transfer protocols include both client-based pacing (in response to acks) and time-driven pacing (aiming at a specific rate until nacked). Generally there will be fewer internal network fetches (transferring data directly from a data storage node to a content transfer engine residing on the content transfer access node) than external network transmits. For example, it would be common to fetch a 48 KByte HTML file in a single read over the internal network, but it would take at least 32 separate TCP segments to deliver it. For extremely large deliveries, such as streaming media, pacing drives transmission from the buffer space assigned in the CTE **50** for the connection while reads are issued whenever the buffer drops below a configured threshold.

[**0192**] It is noted that execution of steps **1508** and **1510** can overlap. As illustrated in the examples discussed herein above, this is especially true in cases where relatively large files are transferred. In such instances, the transfer of data over an external interface to a requesting client commences prior to completing transfer of the file from a storage node to a buffer in the CTE **50**.

[**0193**] It is further noted that during both steps **1508** and **1510** the new in-session context object on the CTE **50** reports aggregate progress and exception conditions (if present) to a corresponding XCTD executing on the CTEX **88** associated with an application executing upon the supplemental processor node **22***a*. The application, executing upon the supplemental processor **22***a* and potentially supervising client data requests and corresponding responses, does not directly access the transferred data. Instead, the application executing upon the supplemental processor **22***a* observes, via notifications from the CTE **50**, the progress of data transfers. Based upon the notifications, the application issues control instructions via the XCTD to the CTE **50** performing the actual data transfers.

[**0194**] After completing a response to a file request or alternatively when a session is completed, during step **1512** a hash entry for the associated CTD in the CTE **50** is removed from the hash table, the CTD notifies a corresponding XCTD of the completion, and the CTD returns to the available pool of CTDs for its object type.

[**0195**] Illustrative embodiments of the present invention and certain variations thereof have been provided in the Figures and accompanying written description. The present invention is not intended to be limited to these embodiments. Rather the present invention is intended to cover the disclosed embodiments as well as others falling within the scope and spirit of the invention to the fullest extent permitted in view of this disclosure and the inventions defined by the claims appended herein below.

What is claimed is:

1. A network server system for efficiently processing requests for information assets stored upon a set of storage drives, wherein the requests are received via a communicatively coupled network link, the server system comprising:

an internal network communicatively coupling nodes within the network data server system;

a supplemental processor node communicatively coupled to the internal network and comprising a general purpose processor and operating system, and wherein the supplemental processor supports executing application programs;

a data storage node communicatively coupled to the internal network, the data storage node comprising storage media and conversion circuitry for packaging retrieved data from the storage media to a format for transmission over the internal network; and

an external network access node supporting network connections between the network server system and client nodes via an external network, the external network interface comprising:

an external network interface comprising an external network interface engine for executing data transfers between the external network access node and the external network,

an internal network interface comprising an internal network interface engine for executing data transfers between the external network access node and the internal network, and

one or more event engines for executing information asset transfers between the data storage device and the external network in accordance with contexts, maintained by the external network access node,

describing a present state of executing information asset transfers performed by the one or more event engines.

2. A method for processing requests for information assets stored upon a set of data storage drives by a network server system, wherein the requests are received via a communicatively coupled external network link, the method comprising the steps of:

receiving, by an external network access node via the external network link, a request for an information asset;

creating, by the external network access node, a context for the request wherein the context includes a buffer identification and a processing engine on the external network access node assigned to execute the request;

submitting, by the external network access node, a request for data from a storage node connected to the external network access node by an internal network; and

receiving, by the external network access node from the storage node, data corresponding to the request for data from the storage node, and storing the received data within memory on the external network access node corresponding to the buffer identification, wherein data transferred from the storage node to the receiving external network node bypasses application memory space on a general processor node; and

transmitting, by the external network access node, the data stored within memory corresponding to the buffer identification, over the external network link.

3. A network server system for efficiently processing requests for information assets stored upon a set of storage drives, wherein the requests are received via a communicatively coupled network link, the server system comprising:

a supplemental processor node;

a network interface node comprising:

a network interface communicatively coupled to the network link and configured to receive requests from clients via the network link;

delegation logic facilitating: associating a request type with at least a portion of a request, identifying a handler from a set of processing elements for executing at least the portion of the request based upon the request type, and creating a data structure linking at least the portion of the new request to the identified handler processor; and

a data path from the set of storage drives to the network interface, the data path facilitating data transfers between the set of storage drives and the external data access node containing the set of processing elements that bypass the supplemental processor node.

* * * * *