Application Level Protocol Implementations to provide Quality-of-Service Guarantees at Endsystems

R.Gopalakrishnan
gopal@dworkin.wustl.edu

Guru M. Parulkar
guru@flora.wustl.edu

Department of Computer Science
Washington University in St.Louis
St.Louis, MO 63130

1. Introduction

Emerging broadband networks can cater to the needs of a variety of media, both continuous (such as video) as well as discrete (data) [11]. Likewise, modern high performance workstations can store and process continuous media (CM), and can send and receive CM data over the network in the same way as discrete data [7,9,13,15]. These two technologies have provided the impetus for the development of a variety of distributed multimedia applications.

An important aspect of multimedia applications is that they require *quality-of-service (QoS) guarantees* from the endsystems (for processing) as well as from the network (for communication). Much research [5, 18] has been conducted on how to provide QoS guarantees in networks. For example in the ATM network model, an application can request its required bandwidth when setting up a connection. If the network accepts the connection, it guarantees to provide the requested bandwidth. Apart from bandwidth, various other QoS parameters such as cell delay and cell loss rate can be guaranteed. Such guarantees will also be provided by the next generation of the Internet protocols [15]. However, to provide *end-to-end* guarantees we must also consider the processing of network data at the endsystem. To keep up with increasing transmission rates, it is necessary that the host must possess sufficient processing capacity and internal datapath bandwidth. In addition to this, the network subsystem and the host operating system(OS) must ensure that this aggregate capacity is shared among the application processes according to their individual QoS requirements. Providing QoS guarantees at the endsystem is an important problem as shown by work in this area [2, 10, 12, 14].

There are four issues that must be considered to provide QoS guarantees for protocol processing. These are:

1. *QoS Specification:* This involves developing mechanisms to specify QoS requirements for a large variety of application types. Since it is not possible to foresee all possible requirement types while designing the networking software, it is necessary to identify a few application classes that can capture the needs of most applications. Within each class, it is necessary to identify parameters to specify QoS requirements.
2. *QoS Mapping:* The QoS specifications are at the application level. Since several resources (such as CPU, memory, and network connections) are involved in communication, the specifications must be mapped to resource requirements. For example, QoS parameters (such as bandwidth) have to be derived for the network connection. Likewise, the amount of processing required must be determined so that the CPU capacity can be allocated to ensure that protocol data units (PDUs) are processed at the desired rate.
3. *QoS Enforcement:* Enforcement has to do with scheduling shared resources during data transfer to satisfy the requirements of each connection. These requirements are derived by the mapping operation. We are mainly concerned with efficient CPU scheduling mechanisms to reduce context-switching, and to integrate these scheduling mechanisms into protocol code.
4. *Protocol Implementation:* The solutions for QoS specification, mapping and enforcement have to be integrated into protocol implementations. We have adopted an application level protocol implementation model. In this model, protocols are part of each application process instead of being implemented within the OS. Since the process is the basic unit of resource management within the endsystem, we can associate a certain amount of resource

capacity with each process as dictated by its QoS requirements. Then, existing resource management mechanisms can be used to enforce these allocations. Having chosen the protocol implementation model, we must develop efficient mechanisms to reduce the overhead associated with data movement and context-switching since they dominate protocol processing costs [3].

2. Overview of Solutions

We will give an overview of our solutions to the problems mentioned above. Figure 1 shows the QoS specification and mapping operations.

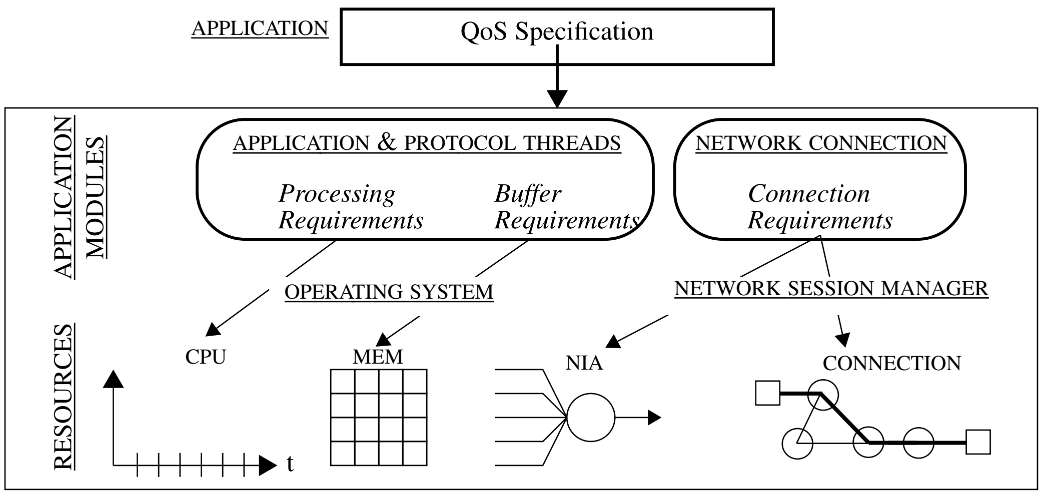

FIGURE 1. QoS Mapping

These two operations are performed during connection setup, and determine the performance during the data transfer phase. For each connection, the application specifies values for QoS parameters. The parameters and their interpretation depend on the class of the data stream and are described in [6]. Given the specification, the mapping operation determines requirements for resources. In Figure 1, the mapping operation derives the processing requirements for a connection as follows. Each connection is associated with a *thread*. The thread represents the processor resource since it does all the processing of PDUs over its connection. For bandwidth critical applications, we reserve processor capacity by associating a time period with each thread and scheduling it to run for some amount of time in each period [17]. This guarantees that in each period the thread processes a certain number of PDUs. The choice of thread period and the number of PDUs to be processed in each period are derived based on QoS parameter values, and measured parameters such as the time to process a PDU of a given size. The OS is then requested to create the thread and bind it to the protocol code. Similarly, the mapping operation derives the bandwidth required for the network connection and specifies it to the network in terms of the connection attributes [5]. Details of the mapping methodology are in [6].

The QoS enforcement and protocol processing operations come into play during the data transfer phase. This is shown in Figure 2 which depicts the organization of the host communication subsystem. It shows two schedulable resources in the endsystem that are involved in supporting a connection: the network interface adaptor that sends and receives *frames*, and the thread that processes the protocol data units(PDUs) contained in these frames.

The PDUs are exchanged on connections supported by the underlying ATM network. For the sake of understanding, we have shown connections as half-duplex. The adaptor takes outgoing PDUs enqueued for each connection, segments it into cells and sends these cells at the rate determined by the connection bandwidth. This action is referred to as *pacing*. Likewise, the adaptor reassembles incoming frames on each connection into PDUs and places them in the host memory to be processed by the application process. The adaptor is controlled by its device driver. The driver maps memory pages containing network data between the process address space and per connection queues. The driver also sets up the adaptor to DMA data for each connection from (or to) its queues in host memory. Physical copying is avoided whenever possible.

The CPU resource is shown with a single application process that has periodic threads that do protocol and application level processing. The application thread does operations such as presentation layer functions and other functions that could depend on the nature of the data stream. QoS enforcement is mainly concerned with scheduling threads for all the connections so that they obtain their required share of the CPU in each period. This is the same as ensuring that all threads meet their deadlines. We have developed a scheduling mechanism (described in Section 3.4.2) that is especially suitable for scheduling protocol processing threads efficiently. More details can be found in [6].

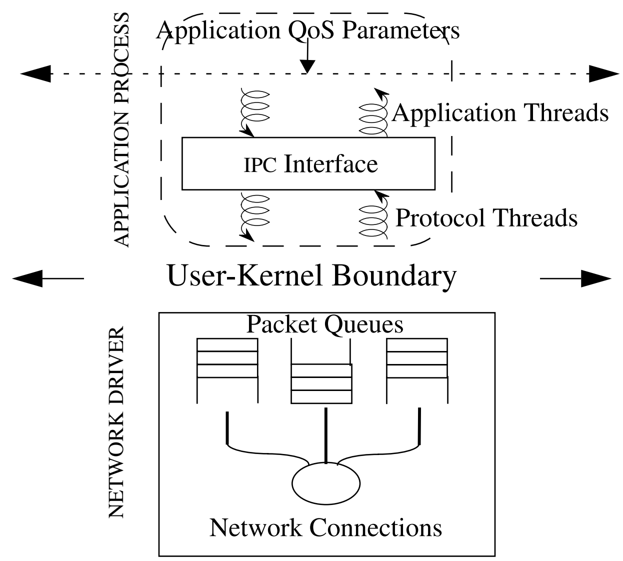

FIGURE 2. IPC Model

The CPU and the adaptor execute concurrently. The PDUs processed by a thread in each period are paced out by the adaptor within one period so that they leave the host before the next batch of PDUs are enqueued. The rate at which cells are paced out is determined by the mapping operation and the connection bandwidth is determined accordingly.

This paper deals only with the protocol implementation aspects that relate to data transfer. Details about other aspects can be found in [6]. In the next section, we introduce the application level protocol (ALP) implementation model and show why it is useful in providing QoS guarantees for protocol processing. We then discuss some of the problems that must be overcome to realize efficient ALP implementations. The rest of Section 3 describes our solutions to the these problems.

3. Application level Protocol Implementation

In current state-of-the-art implementations[1], the networking subsystem is implemented in the kernel and is accessed by user processes through a high level IPC interface (such as *sockets*). This model is referred to as the kernel resident protocol model. In this model, the kernel maintains state for all protocol sessions in protocol control blocks (PCBs). It schedules protocol operations, handles network events and moves data across protection domains for all sessions.

With the emergence of connection oriented networks such as ATM, it is possible to move network data directly to/from the process space based on the connection identifier and so there is no need for the kernel to demultiplex data based on transport header fields. This allows protocol processing to be performed in the user process itself and so the protocol code can be part of the user program. This model is referred to as application level protocol (ALP) model [16]. In this model, protocol code maintains the PCBs only for the sessions involving the process. It also handles protocol events such as packet arrivals and time-outs for these sessions. Therefore the user–kernel boundary is now at the network interface driver level rather than at the IPC level.

The advantages of the ALP model is that there is no central agent (such as the kernel) that handles all protocol activity. Besides it has the flexibility that comes with user level implementations. However, the main implication of the ALP model is that it leverages the existing resource management paradigm that treats the process as a unit for accounting and resource allocation, to solve the problem of providing QoS guarantees. Accordingly, the ALP model locates protocol processing within each process (or thread). By controlling the resource allocation to each thread, we can control the QoS offered to each protocol session. This approach cleanly separates the two concerns of implementing the protocol specific code, and managing resources on a per connection basis in accordance with the QoS requirements of each connection. The latter is automatically taken care of by the OS. This allows the protocol implementor to be unaffected by the platform specific resource management details. This cannot be done quite as easily when protocol processing is done in the kernel because there is no "isolation" between connections. The kernel must therefore explicitly include code to schedule protocol sessions on the CPU. While multi-threaded kernels will be able to overcome this shortcoming, they are still far from becoming readily available and may not support mechanisms such as periodic threads.

While the ALP model seems attractive for the above reasons, attention must be paid to implementation details so that it is as efficient as state-of-the-art kernel implementations. Section 3.1 lists the factors that strongly affect performance of protocol implementations and the ALP model in particular. Section 3.2 gives an overview of our solutions to overcome these problems and describes how these solutions relate to each other. These solutions are explained in greater detail in the remaining sections.

3.1 Performance Issues in Protocol Implementations

It is recognized that a major portion of protocol processing time is spent in moving data [3]. Moving data involves accessing memory and this can be a limiting factor in performance as transmission speeds become comparable to memory bandwidth [4]. Another expensive operation is context switching [3]. Context switching occurs when the process makes system calls and when interrupts occur (due to packet arrival and timer expiration). A naive ALP implementation that does not reduce these operations will perform poorly. For the ALP model, we discuss when these operations occur during data transfer and how they affect performance. We will draw comparisons with the kernel protocol model because this is the state-of-the-art. Our objective is to achieve at least as good performance as existing implementations, while also providing QoS guarantees. These operations are the following:

1. *Data Passing*: In the ALP model the data units that move across the user–kernel boundary are PDUs. When protocols are in the kernel the units of data that move across the user-kernel boundary are much larger and they are passed as parameters in system calls. If the same system call mechanism is used to move PDUs across the user kernel boundary, then the number of calls needed will be significantly higher.
2. *Data Copying*: Apart from the overhead of system calls, moving the contents of a PDU across the user–kernel boundary is also expensive. On the send side, a PDU is either copied or mapped[1] (by modifying page tables) into a kernel buffer from where the adaptor can read it. On the receive side, data received into kernel buffers is mapped to user address space. Because the cost of page mapping operations can be high, mapping data pages every time a PDU is sent or received by the protocol thread will be inefficient. This problem is also present in kernel resident protocols and several solutions have been proposed to overcome this.
3. *Asynchronous Event Processing*: Processing of asynchronous protocol events such as packet arrival and timer expiry are usually triggered by interrupts. Interrupts cause context switches because the appropriate protocol thread must be scheduled to handle them. In addition, scheduling threads for asynchronous events disrupts the periodic scheduling of threads and interferes with the QoS enforcement mechanism (to be described in Section 3.4.)

1. The data has to be page aligned for mapping

4. *Protocol Thread Scheduling:* Since periodic threads are used for processing each connection, they have to be scheduled using a priority mechanism. This is necessary to ensure that each thread gets its required CPU time in each period. Most schemes[8] developed so far are meant for hard real-time guarantees. They assume that a process is pre-empted as soon as a higher priority process becomes ready. This exacerbates the context-switching overhead even further compared to quantum based scheduling schemes.
5. *Control Operations*: In the ALP model all protocol processing is done in user space and therefore protocol operations that require kernel privileges have to be done using system calls. Because system calls are expensive, ALP implementations are put at a disadvantage compared to in-kernel implementations. The operations that need system support are—
   a) Retransmission: A transport PDU that is passed to the network driver for transmission may need to be retransmitted later due to errors. In the ALP model, the user protocol code that buffers the PDUs will have to move the PDU again to the kernel, whereas this is not necessary if the protocol in the kernel maintains the PDUs.
   b) Timer Operations: Protocol code often needs to do time related operations on a per-packet basis such as setting a retransmission timer, or putting timestamps in the header. In the ALP case, system calls may have to be made to perform these operations. Similarly asynchronous timer events such as timer expiry that are handled by the kernel must now be passed on to the user protocol thread and this could cause context switching.
   c) Page Remapping: A protocol that receives misordered data needs to remap pages to make the data contiguous. Misordering happens when a PDU is lost and its successor is written in its place by the adaptor. If the protocol is in the user space, system calls will be necessary to do remapping.

A solution to these problems is necessary to make ALP implementations an attractive alternative for system designers.

3.2 Solution Outline

- We use data structures in shared memory between each process and the driver to share information about outgoing and incoming PDUs. In each thread period, a batch of PDUs need to be processed. Control information about these PDUs is written in fields in the shared data structures. These structures are organized as a linked list and are examined by the driver when it gets control. Likewise, the driver creates a linked list of such data structures that describe incoming PDUs. This list is examined by the protocol thread when it gets scheduled by the scheduler.
- We combine the operations of passing PDUs across the user-kernel boundary and thread scheduling in a single system call. Whenever a running thread yields the processor, it ensures that it has processed some number of PDUs that can be enqueued for transmission. The driver enqueues these PDUs at the adaptor before scheduling the next eligible thread. In addition, we also use the shared memory to implement the scheduling mechanism as described in Section 3.4.3.
- We use buffers in the shared memory between user process and the network driver to store PDU headers and data for both outgoing and incoming PDUs. These pages would typically be pinned down to prevent unpredictable delays due to page faults. This reduces the need to remap pages between user and kernel on every send and receive operation. With DMA capability of the adaptor, this mechanism can potentially achieve *zero-copy* semantics.
- We avoid generating interrupts to process asynchronous events. Instead, we rely on the thread scheduler to dispatch the thread for which the event is destined. The scheduling of threads is done according to the attributes derived by the mapping operation. We therefore describe our event handling policy as *specification driven* rather than *event driven*. This approach has the potential benefit of achieving *interrupt free* operation.
- We provide retransmission support, timer support, and page remapping support for user protocol threads using a command interface in shared memory. As before, the scheduling system call is used to trigger these operations thereby avoiding extra system calls.

As can be seen, all these solutions involve using shared memory between user space and kernel to co-ordinate protocol processing operations. The main idea is that we have dispensed with the "heavyweight" system call mechanism to implement the handshake between the user and the kernel for most protocol operations in the critical path. These critical operations are data movement and scheduling. Using a single shared memory mechanism to implement the cooperation between protocol threads and kernel reduces implementation complexity. In cases where there is a need for performing operations with kernel privileges, we "piggyback" these operations with the scheduling trap.

Figure 3 shows the overall organization of the shared memory between a sending protocol thread and the adaptor driver. The control structure is accessible to both the protocol thread as well as the driver. This structure has a pointer to the queue of outgoing PDUs. It also has pointers to free descriptors (that contain information about PDUs) and free pages. Similarly a pointer to scheduling information is stored in the control structure. The following sections provide more details on how each structure is used.

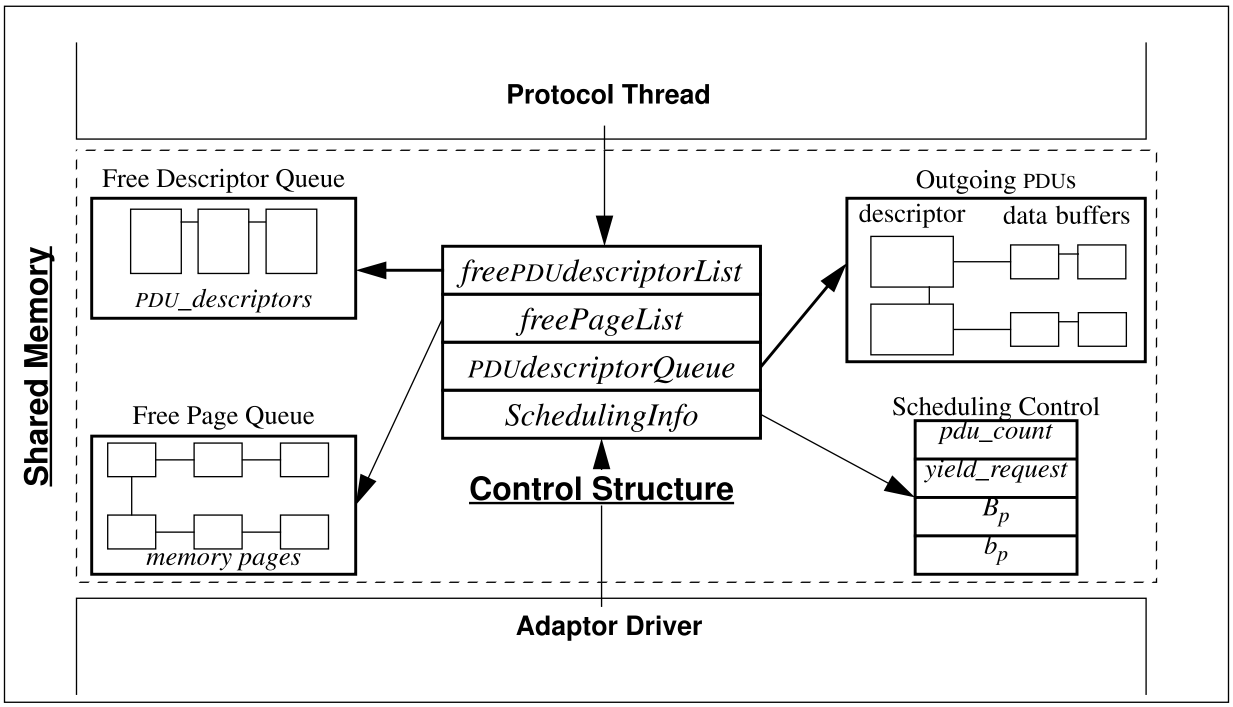

FIGURE 3. Using Shared Memory for Handshaking

3.3 PDU Movement without System Calls

As mentioned earlier, a sending protocol thread prepares each PDU in buffers in shared memory. Similarly, the adaptor writes (using DMA) incoming PDUs into free pages in this shared memory. The shared memory is also used for the protocol thread and driver to share control information about outgoing and incoming PDUs. We refer to these data structures as *PDU descriptors*.

The descriptor contains fields that hold buffer addresses, commands, status and various arguments that are passed between the driver and the thread. The address field holds a pointer to a linked list of buffer addresses that contain the PDU. The command field is used by the protocol to request the driver to perform operations on the PDU. Typical operations are retransmission timer setup and release operations. Arguments needed by these requests are also provided. Similarly the driver can set status flags to indicate the results of various operations that it performed on the PDU. Typically a batch of PDU descriptors are linked together and passed on to the driver. The usage of these descriptors for the send and receive cases is described below.

Send Side. The protocol thread prepares a descriptor for each PDU from a free list of descriptors in shared memory and places it on an outgoing queue. When the driver gets control, it converts these addresses to physical addresses and enqueues the PDUs at the adaptor. The adaptor traverses the queue, and paces out data in each PDU without involving the CPU. After sending out a PDU completely, the driver may indicate a status and possibly other local error conditions if any (e.g., DMA underrun). The protocol thread when it gets scheduled again, reclaims the descriptors of PDUs that have been sent out.

Receive Side. The driver prepares a list of PDU descriptors that point to free buffers and enqueues it at the adaptor. As PDUs are received, the adaptor writes the data into these buffers and marks the descriptors as containing data. When the receiving protocol thread is scheduled in the next period, the driver removes these descriptors from the free list of the adaptor, and places it in the control structure. When the protocol thread runs, it goes through the list and processes them.

An advantage with this approach is that since the receiving thread is scheduled periodically, there is no need for an interrupt to trigger the processing of received PDUs. An interrupt may be needed at the beginning of a burst in order to activate the receiving thread. Subsequently, the presence of data would be checked and would serve as an enabling condition for a thread to be scheduled but would not by itself trigger the running of thread.

Using shared memory for moving data as opposed to making system calls has several advantages. Firstly, a sending thread does not have to make a separate system call for passing control information about PDUs to the driver but can use the scheduling system call to do so. Likewise, the driver does not need to make an upcall to notify the receiving thread for every incoming PDU. Secondly, it works well with the batching approach where a batch of PDU descriptors corresponding to the outgoing and incoming PDUs can be formed into a linked list. When the control is given to the network driver (or to the protocol thread) this batch is processed. In addition, the descriptor fields can also be used to implement a generalized command-response mechanism between the driver and the protocol thread to do operations other than data movement.

3.4 Thread Scheduling

There are two issues that we address in this area. They are:

1. *Pre-Emption Control*: We have adopted a real-time scheduling policy based on rate-monotonic priorities in order to provide performance guarantees. In this policy, the priority of a thread is proportional to its rate (i.e inversely proportional to its period). It is a pre-emptive scheduling policy, and is intended to provide hard real-time guarantees. The main problem with pre-emptive scheduling is that a higher priority process must be scheduled as soon as it becomes ready. As a result there could be a large number of pre-emptions as compared to quantum based scheduling schemes. Each scheduling operation involves a context switch and the cost of context switching has been shown to dominate protocol processing even in current implementations where there is no explicit scheduling[3]. Therefore minimizing pre-emption is important to attain efficiency.
2. *Tracking Usage*: In the periodic thread model, the scheduler must ensure that each thread conforms to its allotted execution time in each period. One way to do this is for the scheduler to track the time the thread runs on the CPU (across multiple context switches if necessary)[10] and forcibly terminate it after its time quota is over. Since a protocol thread must enqueue the PDUs at the adaptor before terminating, the kernel must ensure that it does not terminate the thread before the enqueue operation is performed. This is difficult because the kernel is unaware of the thread status. An alternative would be to let the user threads to track their own usage. However using time as a measure of usage in this case may involve system call overheads and lead to awkward programming style. Therefore the problem is to determine a measure of CPU usage other than time, and to eliminate the need for explicit policing and enforcement of CPU usage by the kernel scheduler.

3.4.1 Solution outline

The main aspects of our solution to the problems mentioned above are:

- We preempt a running thread only after it has processed some minimum amount of data since it was last scheduled. The operations of enqueueing/dequeueing of PDUs at the driver (as well as yielding the processor) can be done in a single context switch. We call this technique *delayed pre-emption*.

- We use the *count* of PDUs processed by the threads as a usage measure rather than time. Furthermore, the kernel scheduler does not preempt a running thread when it loses priority, but only notifies it of this condition using a field in shared memory. The protocol thread then yields the CPU by itself after it has processed the minimum number of PDUs.

3.4.2 Delayed Pre-emption

The idea behind our proposed solution is that whenever a periodic thread is scheduled, it runs until it generates (or consumes) a minimum number of PDUs even though a higher priority thread may become ready while it is running. The rationale behind this is that, if a thread at a sending host has to yield the CPU, then it can make a single system call that combines the operations of enqueueing PDUs processed so far and yielding the CPU. Likewise, for a receiving thread, the PDUs that have been completely processed during its current run could be put back in the free list. We have seen how the mapping operation determines the number of PDUs B_p, that need to be processed every period. Similarly, in order to ensure that useful work is done for every scheduling operation, we specify the minimum number of PDUs b_p, that must be processed before a thread yields the CPU. This number is chosen during thread creation time to be a certain fraction of the batching factor B_p.

We have derived criteria for making the choice of b_p for a thread without violating deadlines of higher priority threads. We therefore have a schedulability test that takes into account the modification to the basic rate-monotonic policy. Also, our scheme allows us to exploit the trade-off between utilization and pre-emption overhead. If the utilization of periodic threads are low, then we can ensure that all threads meet their deadlines while reducing the number of pre-emptions compared to the standard priority mechanism implementation. This feature is useful in general purpose computer systems where some CPU capacity is kept aside for doing "best-effort" processing. Another benefit of this scheme is that it increases the concurrency in the operation of the CPU and the adaptor because when a thread yields the CPU to a higher priority thread, the PDUs that it has processed so far can be paced out by the adaptor.

3.4.3 Shared Structure for Scheduling Protocol Threads

The solutions proposed above require a closer degree of co-operation between the user process (protocol thread) and the kernel (scheduler and network driver). For example, the scheduler must know if any PDUs have been received on a connection in order to make a scheduling decision. This information is typically known to the driver. More importantly, the scheduler must be able to inform a running thread whether a higher priority thread is waiting for the CPU. Likewise, the thread must be able to inform the scheduler how many PDUs it has processed since it got the CPU.

Our implementation uses a data structure in *shared memory* between each user protocol thread and kernel to maintain control information about scheduling as shown in Figure 3. The protocol writes into the *pdu_count* field in the structure to indicate how many PDUs have been processed so far. The kernel writes into the *yield_request* field to indicate that a higher priority thread has become ready. These fields are further explained below:

1. *pdu_count*: This field shows the number of PDUs that have been processed by the running thread since it was scheduled on the CPU. This field is cleared when the thread is scheduled to run. Whenever it generates a PDU it increments this count. If a higher priority thread becomes ready while the thread is running, the scheduler reads this field. If the value is greater than b_p, then the scheduler enqueues the set of PDUs processed so far at the adaptor and schedules the higher priority thread to run. The case when the count is less than b_p is discussed below.
2. *yield_request*: If a higher priority process has become ready but the running thread has not yet generated b_p PDUs, the kernel writes into this field to indicate to the running thread that it must yield the CPU when it completes processing b_p PDUs. This is done in the scheduling interrupt routine. The thread checks this field every time it completes processing a PDU. If the field is set, it continues to run until b_p PDUs have been processed and yields the CPU.

3.5 Support For Control Operations in Protocol Processing

Control operations that need kernel privileges need to be implemented without additional system calls in order to make ALP implementations efficient. These operations include retransmission support, timer management and page remapping operations. The details of how these operations are carried out is given below.

1. *Retransmission*: Every time period a sending protocol thread generates a batch of PDUs and enqueues it for transmission. If the protocol needs to retransmit a PDU at a later time, it must not free the PDUs until it receives an acknowledgment (ACK) from the recipient. The sending protocol may use a control connection to receive ACKs and we assume a control thread processes these ACKs.

   When the sending thread is invoked during its next period, it takes the descriptors for PDUs that have been marked as sent by the adaptor and puts them on a list awaiting ACKs. There is a timer associated with each PDU descriptor that is loaded with the maximum time that the sender will wait for an ACK before retransmitting the packet. When an ACK is received, the thread of the control connection frees the PDUs that have been successfully received. It also requests the driver to clear the timers associated with acknowledged PDUs when it yields the CPU. If a retransmission timer does expire, the kernel sets a status flag in its PDU descriptor. Every time the sending thread runs, it checks the status field and enqueues the PDU to be retransmitted. The benefit of this scheme is that timer set and reset operations do not need system calls. Furthermore, the timer expiry event need not be handled by the user protocol threads and the retransmitted PDU need not cross the user kernel boundary again.

2. Timer Operations: Timer operations need to be supported through the command interface if the user processes cannot access system time without making system calls. When a sending thread calls into the driver with a list of PDU descriptors it can request the retransmit timer to be set for a certain interval by setting a fields in the descriptor. The sender may also specify if the timer is to be set after the last byte of the PDU is sent or when its first byte is read out. In any case, the driver sets the timer and if it expires, it handles the timer interrupt and writes the status field as described earlier. Another service related to time, is to set timestamps on PDUs that are sent and received. The driver may be requested to insert a timestamp at a fixed offset within a PDU before passing it to the adaptor (in the case of send) and to the receiving thread (in the receive case).

3. Remapping Support: As mentioned earlier, if the adaptor can separate headers from data and no PDU is dropped by the network, the data gets written contiguously in the recipient's address space and no further remapping is necessary[1]. However, when a PDU gets lost, the adaptor writes the next one in its place thereby destroying contiguity. When the protocol thread runs and detects this, it must remap the data units that were sent after the lost PDU. Also when the lost PDU is received correctly, it must be mapped to its right position. When the protocol needs to remap pages, instead of making a system call, it creates a remap request structure. The structure is a pair of addresses where the first field is the current address and the second field is the new address. When the receive thread calls into the driver to schedule itself out, it passes a list of remap structures. The kernel does this remapping before scheduling in the next thread.

4. Conclusion

We have described the issues involved in providing QoS support at the endsystems. We have also presented our design for the host networking subsystem that can provide QoS guarantees. It is important to note that one has to optimize the entire data path, from the application interface, upto the network connection to be able to provide end-to-end QoS. It is also important not to lose sight of efficiency issues in protocol implementation because they can detract from the benefits obtained from providing QoS support. We are currently engaged in implementing this design and we expect to obtain a lot of experience from this exercise.

References.

[1] Banks, D.,Prudence, M., "A High-Performance Network Architecture for a PA-RISC Workstation," *IEEE*

1. Of course PDU size must be some multiple of page frame size.

Journal on Selected Areas in Communications, Vol.11, No.2, February 1993, pp.191-202.

[2] Campbell, A., Coulson, G., Hutchison, D., “A Multimedia Enhanced Transport Service in a Quality of Service Architecture,” Internal Report MPG-93-22, Department of Computing, Lancaster University, 1993.

[3] Clark, D. D., Jacobson, V., Romkey, J., Salwen, H., “An analysis of TCP processing overhead,” *IEEE Communications Magazine*, 27(6) 1989, pg. 23-29.

[4] Dittia, Z.D., Cox, J.R., Parulkar, G.M., “Using an ATM Interconnect as a High Performance I/O Backplane,” *Proceedings of the Hot Interconnects Symposium*, August 1994.

[5] Ferrari, D., Verma, D.C., “A Scheme for Real-Time Channel Establishment in Wide-Area Networks,” *IEEE Journal on Selected Areas in Communications*, Vol.8, No.3, April 1990, pp. 368-*379.*

[6] Gopalakrishnan, R., Parulkar, G.M., “Efficient Quality of Service Support in Multimedia Computer Operating Systems,” Technical Report WUCS-94-26, Dept. of Computer Science, Washington University in St.Louis.

[7] Hayter, M., McAuley, D., “The desk area network,” *ACM Operating Systems Review,* Oct 1990.

[8] Herrtwich, R.G., “An Introduction to Real-Time Scheduling,” Rechnical Report TR-90-035, International Computer Science Institute, Berkeley, California, July 30 1990.

[9] Houh, H.H., Adam, J.F., Ismert, M., Lindblad, C.J., Tennenhouse, D.L., “The VuNet Desk Area Network: Architecture, Implementation and Experience,” *IEEE Journal on Selected Areas in Communications*, First Quarter, 1995 (To appear).

[10] Mercer, C.W., Stefen Savage, Tokuda H., “Processor Capacity Reserves for Multimedia Operating Systems,” Technical Report CMU-CS-93-157, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, May 1993.

[11] Minzer, S.E., “Broadband ISDN and Asynchronous Transfer Mode(ATM),” *IEEE Communications Magazine,*Sep 1989, pp. 17-24.

[12] Nahrstedt, K., Smith, J.M., “An Application Driven Approach to Networked Multimedia Systems,” *18th Conference on Local Computer Networks,* Minneapolis, MN, September 1993.

[13] Richard, W. D., Cox, J. R., Gottlieb, B., Krieger, K., “The Washington university multimedia system,” Technical Report WUCS-93-01, Dept. of Computer Science, Washington University in St. Louis, Jan 1993.

[14] Robin Philippe, et al., “Implementing a QoS Controlled ATM Based Communications System in Chorus,” Internal Report MPG-94-05, Department of Computing, Lancaster University, March 1994.

[15] Shenker, S., “An Architecture of the Future Internet,” *Proceedings of the Hot Interconnects Symposium*, August 1994.

[16] Sterbenz, J., Parulkar, G.M., “Axon Host-Network Interface Architecture for Gigabit Communications,” Protocols for High-Speed Networks, II, Marjory Johnson (Editor), Elsevier Science Publishers B.V. (North Holland), 1991, pp.211-236.

[17] Thekkath, C.A., Nguyen T.D., Moy Evelyn, Lazowska, E.D., “Implementing Network Protocols at User Level,” ACM SIGCOMM, Ithaca, NY, September 1993, pp.64-72.

[18] Tokuda, H., Nakajima, T., Rao, P., “Real-Time Mach: Towards predictable real-time systems,” *Proceedings, USENIX 1990 Mach Workshop*, Oct 1990.

[19] Zhang, Hui, Keshav,S., “Comparison of Rate-Based Service Disciplines,” *Proceedings of the ACM SIG-COMM*, 1991, pp. 113-121.