# A Framework for QoS Guarantees for Multimedia Applications within an Endsystem

R. Gopalakrishnan and Guru M. Parulkar

Washington University, St. Louis MO 63130, USA

**Abstract.** We describe a framework for providing quality-of-service guarantees within the endsystem for networked multimedia applications. Our framework integrates four components namely, QoS specification, QoS mapping, QoS enforcement, and a protocol implementation model. The framework allows QoS specifications to be at a high level so that an application can easily specify its requirements using a small number of parameters. The second component of the framework is QoS mapping. The mapping operations derive resource requirements for each end-to-end session from the QoS parameters specified by the application. Important resources considered are the CPU and the network connection. The third component of the framework is QoS enforcement. An important enforcement issue is providing processing guarantees for protocol code. An efficient real-time signal facility (RTSig) has been implemented to achieve this. The last component is an application level protocol implementation model. Protocol code structured according to this model can take advantage of the QoS support facilities in our framework. We also review related work in these areas and show how they fit into the general framework.

## 1 Introduction

There is a growing need to provide support for multimedia processing within endsystems. This will enable the development of a variety of exciting applications such as interactive video, customized news services, virtual shopping malls and many more. A large fraction of the data handled by these applications will be of the continuous media (CM) type. An important aspect of these applications is that they require quality-of-service (QoS) guarantees for the transfer and processing of CM data. Emerging networks such as ATM [18], and the proposed integrated services internet with reservations [24], can provide guarantees on bandwidth and delay for data transfer. Likewise modern computer systems have the computing power and I/O bandwidth for handling continuous media such as video. These developments have provided the motivation for providing QoS guarantees within the endsystem. This will complement the guarantees provided by the network for data transfer, and build upon the increasing processing power of the endsystem hardware.

We have developed a QoS framework from the point of view of application processes that run on the endsystems. Our framework consists of the following four components.

- **QoS Specification:** Specification of QoS requirements is essential to provide QoS guarantees. Since applications have widely varying requirements, it is necessary to provide them a structured and general way to specify QoS. It is important to keep in mind that application users can specify only high level parameters, and other detailed requirements must be derived automatically.
- **QoS Mapping:** As mentioned above, QoS requirements are at the application level. Since several resources such as the CPU, memory, the network adaptor, and network connections are involved in communication, the QoS specifications must be mapped to resource requirements. We consider an important feature of the mapping mechanism to be its ability to hide the endsystem platform characteristics through a standard interface thereby allowing protocol code to be portable. The specification and mapping operations are shown in Figure 1.
- **QoS Enforcement:** The mapping operation derives resource requirements that are allocated by the operating system to each end-to-end session during the setup phase. During the data transfer phase, the QoS enforcement mechanisms must schedule various shared resources to satisfy the allocations. In particular, we focus on CPU scheduling for protocol processing because it determines how the aggregate processing capacity of the endsystem is shared among different end-to-end sessions.
- **Protocol Implementation Model:** Given the mechanisms for QoS specification, mapping, and enforcement, protocol code has to be structured to take advantage of these facilities. The protocol implementation model facilitates the mapping of protocol services to appropriate implementation components provided in the framework. An important objective of the model is to provide operating system mechanisms to improve efficiency of protocol implementations by reducing data movement and context switching.

Our goal is to extend the notion of QoS within the endsystem, starting from the network interface and various protocols, upto the application processes. We further wish to achieve the same efficiency as state-of-the-art protocol implementations.

In the rest of the paper, we describe the four components of the QoS framework, and the problems that we address in each. We present a brief overview of our solutions to realize each component. We also show how other proposed solutions fit into this framework.

## 2 QoS Specification

Several methods have been proposed for specifying QoS requirements at the network layer. However at the application layer, the choice of parameters has not been adequately addressed. There are two criteria that guide the choice of parameters to specify application requirements. These are

- There is a wide diversity in the types of applications and their communication requirements. It would be impractical however to allow the same diversity in

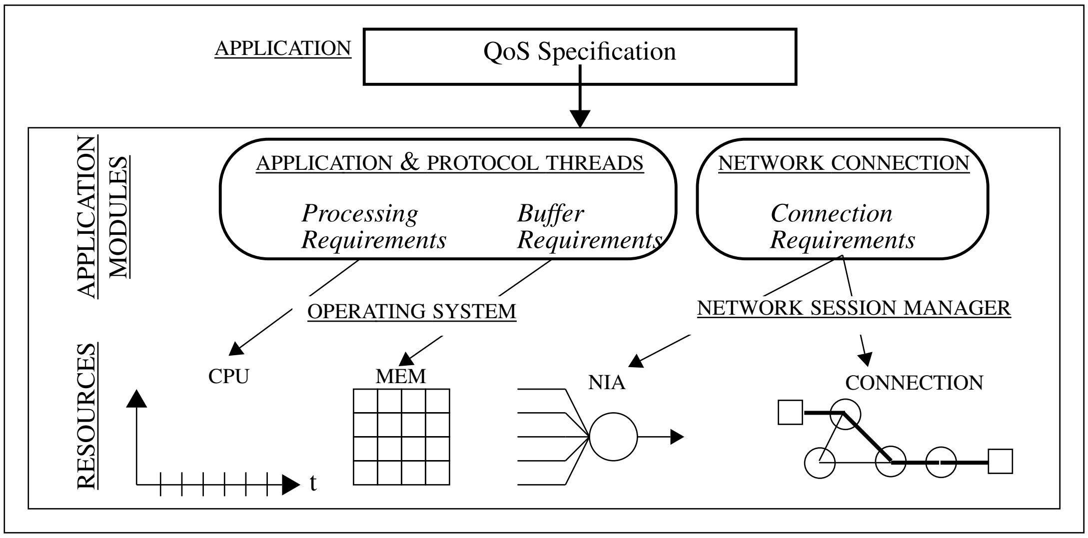


**Fig. 1.** QoS Specification and Mapping

the way QoS requirements are specified because a system can support only limited number of formats. Hence it is necessary to identify a canonical set of specification formats that are representative of a wide variety of application requirements.

– An application user may only be able to specify QoS parameters at a high level. The choice of parameters must therefore be such that they are few in number, and are easily specifiable.

We have chosen a set of representative application classes, and identified QoS parameters for each class. We expect that most of the data streams used by applications will fall in one of these classes differing only in the values of the QoS parameters. Therefore supporting these basic types would meet the needs of most applications. The three application classes are—

1. **Isochronous class:** This includes compressed and uncompressed continuous media data that is generated and consumed at the endsystem at a fixed rate. Traffic of this class is characterized by three parameters namely, the *frame rate*, the *maximum frame size*, and the *average frame size*. In addition, the *maximum delay* between the generation of a frame at the source and its display at the receiver is specified. These QoS parameters help to determine the peak and average bandwidth required for the network connection. The delay parameter is also used to determine the network delay and size of playout buffers.
2. **Burst class:** This includes most cases where bulk data transfer is required. A bulk data source does not have a natural rate and is statically available on secondary storage. QoS parameters therefore consist of the *total size* of

the data unit (such as the size of a digital image), and the desired *bandwidth* at which the data should be sent. This class would be useful in a multimedia explorer kind of an application for transferring images, and bursts of data on request from a remote client where each burst has to be provided the requested bandwidth.

3. **Low Delay class:** This class includes applications that require small response times such as RPC requests, and control packets such as transport layer acknowledgments. Bandwidth is typically not a concern and *message delay* is the dominant QoS parameter. Included in this class are data streams that require a certain rate of message delivery to be provided. An example of such an application is distributed interactive simulation, where a large number of messages can arrive at an endpoint. In such cases, the message rate implies that both network connection bandwidth, and processing resources must be guaranteed.

The three classes and their parameters are shown in Table 1

| | |
|---|---|
| Isochronous | frame rate, maximum frame size, average frame size, delay |
| Burst | total size, required bandwidth |
| Low Delay | maximum message size, delay, message rate |

**Table 1.** Application Classes for QoS Specification

### 2.1 Related Work in QoS Specification

There is a vast amount of literature on QoS specification methods. This section is not an attempt to critically evaluate these approaches, but to direct the reader to related work that present alternative approaches to QoS specification at the transport layer and above.

Transport layer QoS parameters are examined in [19, 4]. The use of application layer QoS parameters has been discussed in [20]. These consist of *media characteristics* and *transmission characteristics* parameters. In [5] application level QoS specifications take the form of "channel types" that encapsulate preconfigured QoS parameters for commonly used media. Finally, we have the LBAP model [1] that characterizes application traffic and processing workload within the endsystem, and has been used in [2, 13, 29].

We take a look at the use of application classes used to facilitate QoS specification. In [22] the *stream* and *message* types have been used that correspond to the *isochronous* and *low delay* classes in our framework. Other classifications [30] define *reliable* and *unreliable* categories of real-time streams. We believe that all these approaches can be readily incorporated in our model, provided mapping procedures are defined to derive resource requirements.

## 3 QoS Mapping

The role of QoS mapping is to determine resource requirements implied by a given QoS specification. For example, a communication session will require one or more network connections, and one or more independently schedulable processing resources[1] to do protocol processing for the data carried over the connections. Therefore mapping operations should derive the parameters for the network connections, and processing requirements for the threads. We describe the steps involved in the QoS mapping operation for a data stream of the application.

1. We scale the data sizes generated at the sender to account for the header and retransmission overheads introduced by the protocol. We also partition the application specified delay between the sending host, the network, and the receiving host. The delay partitioning yields a delay "budget", for each component along the end-to-end path.
2. The delay budget and the scaled data sizes are used to determine the parameters for the network connection such as bandwidth and delay.
3. From the network connection bandwidth and the delay budget for the sender, protocol processing requirements at the sender (and at the receiver) can be derived. This requires knowledge of the computation time for the protocol code to process a protocol data unit (PDU).

These operations are depicted in Figure 2.

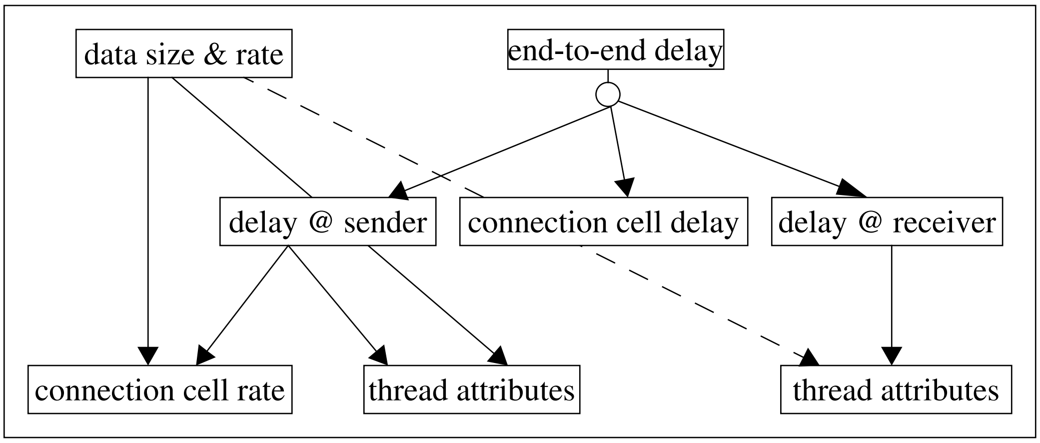


**Fig. 2.** QoS Mapping Operations

We take a closer look at how processing requirements are derived for an application session. These requirements determine the QoS provided within the

[1] In general, a processing resource would be a "thread". However we advocate the use of an alternative mechanism in Section 4.

endsystem and are used by the scheduler to perform QoS enforcement (Section 4).

### 3.1 Deriving Processing Requirements

We take the example of deriving processing requirements of a CM stream such as compressed video. We assume that previous mapping steps have derived the network connection bandwidth, and the portion of the end-to-end delay allocated to the sender. The rate of the network connection and the PDU payload size determine the time duration required by the adaptor to transmit a PDU. To keep up with the adaptor, the protocol processing at the sender must occur at a rate so that at least one PDU can be processed and enqueued at the adaptor within this duration. Equivalently, we could require that a batch of $B_p$ PDUs be processed in every interval of length that is $B_p$ times the original duration. Batching is useful for efficiency reasons, because the overhead of scheduling protocol processing is amortized over a batch of PDU's. However $B_p$ can be increased only upto a limit without exceeding the delay budget allocated to the sending host. The various trade-offs in choice of $B_p$ are described in [11].

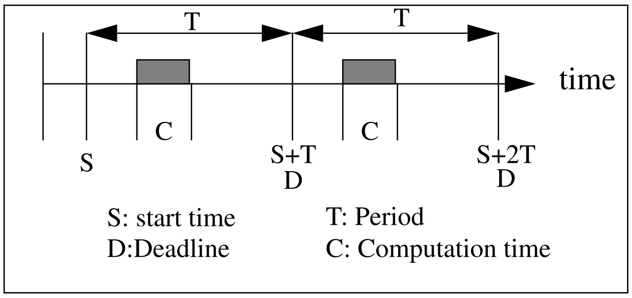


**Fig. 3.** Periodic Processing Model

We thus arrive at a periodic model shown in Figure 3 for expressing protocol processing requirements. The period $T$ is the time to transmit $B_p$ PDUs, and the computation time $C$ in each period is the time to process $B_p$ PDUs. The value of $C$ depends on the protocol code used to process a PDU. Since the protocol processing involved for each PDU is the same, this time should be constant for a given PDU size. For transfer of bulk data and CM streams that extend over durations of a few seconds or more, this characterization is sufficient. For these application sessions, the values of $T$ and $C$ can be used to reserve CPU capacity during connection setup, and processing guarantees can be made during the data transfer phase by appropriate choice of scheduling mechanisms in the operating system.

### 3.2 Platform Independence

An important objective of our framework is to have a uniform way of determining processing requirements for any protocol implementation. Since a given protocol implementation may have a different requirement of $C$ depending on the platform it is running on, it must be able to determine this during session creation time. To facilitate this, the platform must export a set of procedures (possibly organized as a linkable library) that can be used by any protocol implementation to determine processing requirements for a session. Such a library should define a standard interface for QoS mapping, so that protocol implementations are portable across platforms. Therefore if the processing requirements for a communication session are specified in terms of $B_p$ and $T$, the specific protocol code used by the session can make calls to the QoS mapping functions to determine $C$. The only restriction on the protocol code is that it must conform to the periodic processing model that we have adopted in our framework.

### 3.3 Related Work

The related work in this area has addressed all the issues that we have outlined above, but the solutions proposed differ in the mapping methodology used, and the implementation techniques adopted. As before we do not attempt a critical evaluation, but present the main features of the different approaches with respect to these two aspects.

- **Methodology:** Several methods have been used to map QoS specifications to network connection parameters. For example, translation from transport parameters to network connection bandwidth and delay parameters is shown in [19]. In [1] an economic cost model is used to partition the end-to-end delay specified by the application along the data path. Mapping QoS specifications to parameters for a token ring network adaptor is shown in [29]. In [22], processing requirements are expressed according to a periodic model. The period is the inverse of the buffer processing rate, and the computation time required is determined by the scheduler during run-time by making a "dummy" call to the protocol code. We can incorporate these methods in our framework by implementing them in a manner that provides platform independence to protocol code.
- **Implementation:** In most cases, the mapping operations are done by a local server process. The QoS broker mechanism is used in [21] to translate between application QoS requirements and local resources. It includes a tuning facility that lets users adjust their requirements using a visual interface and then translates this into network QoS parameters. Other options are to place the mapping functions in the operating system, or to implement them as a dynamically loaded library.

## 4 QoS Enforcement

In this section we examine OS scheduling issues that are needed to meet the protocol processing requirements of applications. We identify two components in the implementation of OS support for periodic processing. The first is the mechanism used to structure the periodic processing activity within a process. The second is the mechanism used within the OS to schedule these activities in different processes. These two aspects are shown in Figure 4. This organization is used by systems such as RT-Mach [28] and Chorus [22].

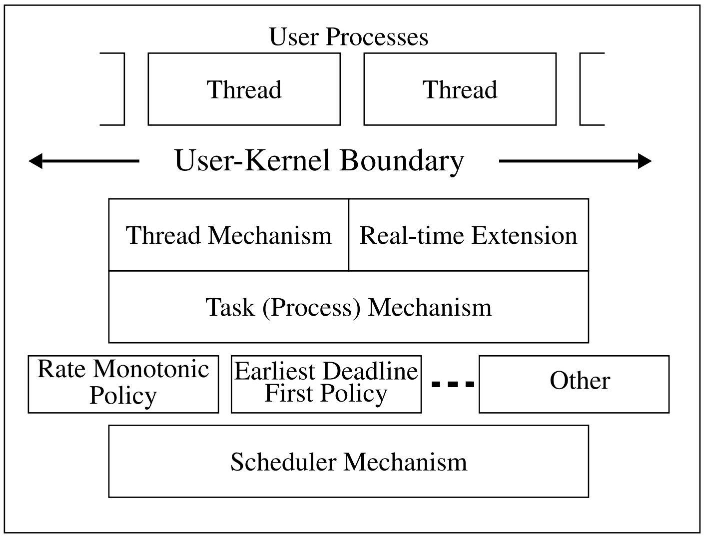


**Fig. 4.** OS Scheduling Mechanisms

Traditionally real-time periodic threads have been used to structure code that must be run periodically. The thread period and the computation time can be set to the values derived by the mapping operation. A separate thread could be assigned to each independently schedulable protocol operation for a communication session. The kernel schedules threads in the system using a real-time priority scheduling scheme. Typical priority schemes used are *rate-monotonic* [15] and *earliest-deadline-first* [15].

However for the purpose of protocol processing, threads are not entirely suitable. This is because of the following reasons–

- **Portability:** A real-time thread facility is complex to implement and is not available in general purpose systems such as UNIX. Therefore protocol code that is structured using threads cannot be ported to these systems. Thus the benefits of providing QoS guarantees cannot be realized in these systems.

– **Efficiency:** Real-time threads are scheduled in a pre-emptive manner as dictated by the priority mechanism. This leads to increased number of context switches. It has been shown that context switching costs can be an order of magnitude more than the cost of processing a PDU [6]. Another consequence of a fully pre-emptive scheme is that threads that share common variables must lock them before access. This requires making a system call to obtain the lock. In addition, the OS must make sure that the locking schemes implemented do not cause unbounded priority inversion [23].

To circumvent the problems with real-time threads, we have implemented a *real-time signal* (RTSig) facility to structure periodic activity. A signal can be regarded as a software interrupt that is delivered to a process by the scheduler. It is associated with a handler function in the application process. Existing signal facilities in UNIX cannot provide processing guarantees. However the RTSig mechanism can ensure real-time behavior, and in this respect is similar to real-time threads. The advantage of the RTSig facility is that it is simple to implement, and no change to the existing process scheduling code is required. In addition, the RTSig scheduling scheme is more efficient than that used in thread based systems and this is discussed in the next section.

### 4.1 Scheduling Scheme for RTSigs

RTSigs are scheduled using a variation of the rate-monotonic scheme called rate-monotonic with delayed pre-emption (RMDP). The scheme exploits the fact that protocol processing is repetitive in nature. In the RMDP scheme, a running RTSig can be pre-empted only at the end of an iteration. One iteration of the handler is assumed to process a single PDU. In addition to $B_p$ which is the number of PDUs that must be processed in a period, we associate a number $b_p$ that is the number of PDUs that a RTSig handler is allowed to process every time it is invoked. Because the scheme ensures that a handler does some minimum amount of work in each run, the number of context-switches is reduced. Another consequence is that since a handler cannot be pre-empted in the middle of an iteration, it need not lock variables shared with other handlers. The implementation of the RMDP scheme shown in Figure 5, uses information in shared memory between each handler and the scheduler. When a higher priority RTSig becomes ready the scheduler notifies the running handler by writing into a shared memory field. The handler checks the field after each iteration and yields the processor if the field is set. The count of PDUs that remain to be processed is maintained in shared memory, so that when the handler resumes it knows how much more work is left to be done. We have modified the schedulability test for the rate-monotonic scheme [15] to account for the effect of delayed pre-emption. Implementation details and experimental results on the RTSig may be found in [12].

### 4.2 Related Work

Several alternate scheduling mechanisms have been used to provide processing guarantees. The *deadline workahead* scheduling scheme [1] is used in [13, 29]. It

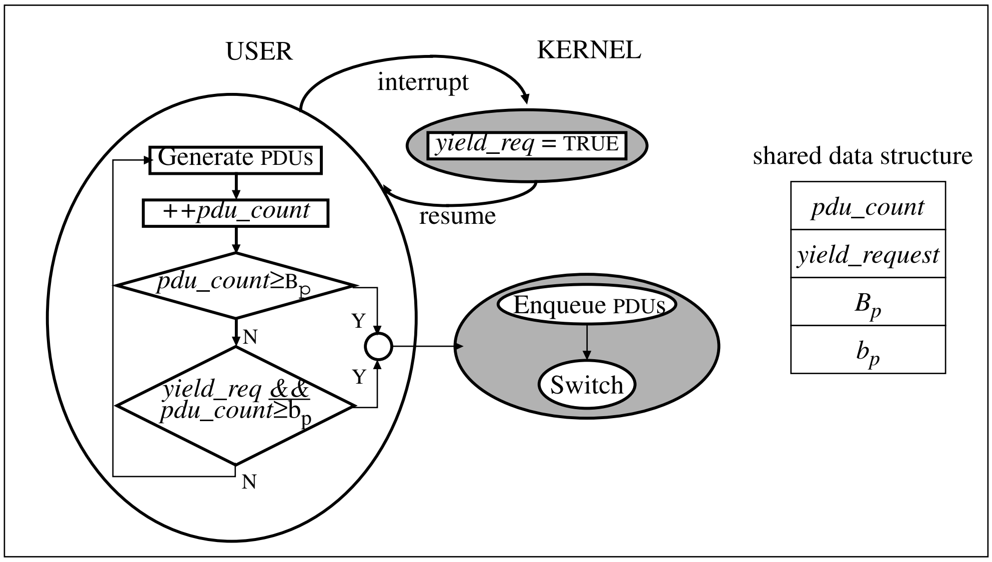


**Fig. 5.** The RMDP Scheme

has been implemented using a technique called *split level scheduling* [13]. The implementation uses shared memory to communicate between the kernel scheduler and a user thread scheduler to reduce the number of domain switches. Other works use a *mixed* scheduling scheme [15] that is a combination of the rate-monotonic and the earliest-deadline-first schemes [20, 22]. Rate based execution has been used in [14] for adaptive applications that have varying processing requirements. Similarly the notion of a *reserve* is used in [17] to monitor and control processor usage by an application.

## 5 Protocol Implementation Model

In current state-of-the-art implementations, the networking subsystem is implemented in the kernel and is accessed by user processes through a high level IPC interface (such as *sockets*). In this model, the protocol processing code is in the kernel address space. State information for all protocol sessions is maintained in protocol control blocks (PCBs) in the kernel space. The protocol code demultiplexes incoming packets to the appropriate session, and moves the data to the application space after protocol processing is completed. 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. Thus protocol processing can be done in the application process, and hence protocol code is part of the application program. This model is referred to as application level protocol (ALP) model [16, 27]. In this model, protocol code maintains the PCBs only for the sessions involving the process. Therefore the user-kernel boundary is now at the network adaptor level rather than at the IPC level. The advantage of the ALP model is that the kernel can be bypassed, and there is greater flexibility when protocol processing is done in user space. The main reason we have adopted the ALP model, is that it leverages the real-time processing facilities available to the application process to provide processing guarantees. We therefore structure the code that implements the data transfer protocol as one or more RTSig handlers. This code would be part of a protocol library that would be linked to the application program. By controlling the RTSig parameters $T$, $B_p$, and $b_p$, we can control the QoS offered to each communication session. This approach effectively separates the two concerns–that of implementing the code that does the protocol operations, and that of scheduling this code on a per connection basis in accordance with the QoS requirements of each connection.

With this basic background on the ALP model, we turn to some of the efficiency issues that arise in protocol implementations.

### 5.1 Performance Issues in ALP

It is recognized that data movement and context switching are the most expensive operations in protocol processing [6]. Since data movement speed is limited by memory bandwidth, copying of data must be avoided whenever possible. Likewise, system calls and interrupts cause context switching, and these operations must be minimized as well. We show how the shared memory mechanism between the RTSigs and the operating system can be used to alleviate these problems. A more detailed description of related issues may be found in [11].

### 5.2 Shared Memory for Data Transfer

Shared memory is used to reduce the number of system calls and avoid copying of data between the user and kernel. The overall organization is shown in Figure 6 and is explained below.

- **Reducing System Calls:** We use data structures in shared memory between each handler and the network adaptor driver to share information about outgoing and incoming PDUs. After a handler generates a PDU, it creates a data structure called a PDU *descriptor* with control information about the PDU. The descriptor is then added to the tail of a linked list of descriptors created earlier. All these data structures reside in shared memory. When the handler yields, either due to a yield request, or when it completes $B_p$ PDUs, the adaptor driver traverses the descriptor list and enqueues the PDUs for transmission. The receive side operations are similar. It can be seen that we combine the operations of passing PDUs and scheduling the handler in a single system call. This saves on the number of context switches.
- **Avoiding data copying:** We use buffers in the shared memory between user process and the network driver to store PDU headers and data for outgoing and incoming PDUs. These pages would typically be "wired" down to prevent unpredictable delays due to page faults. Because the data is available in

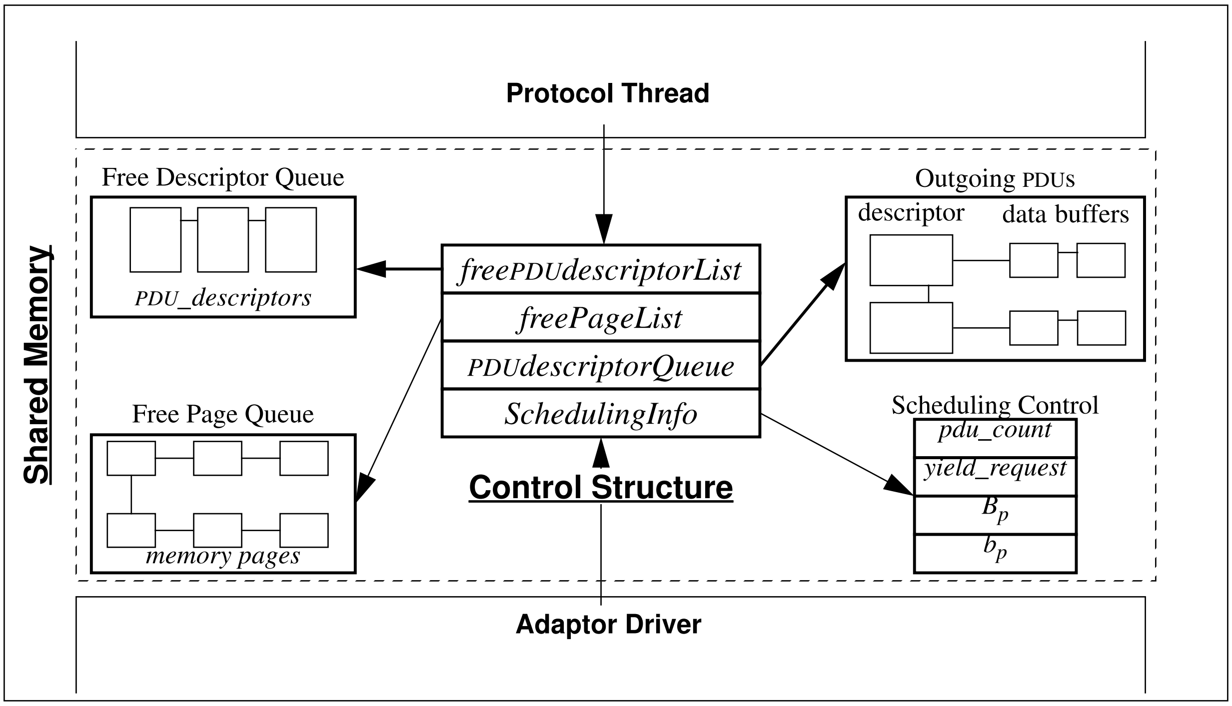


**Fig. 6.** Use of Shared Memory

both the kernel and user address spaces, there is no need to remap pages or copy data. Use of shared memory for data buffers requires changes to the application programming interface to protocols and these issues are being addressed in [7].

### 5.3 Related Work

Most of the work in reducing data movement and context switches has been done for protocols implemented in the kernel. We provide a brief overview of the techniques used to achieve this.

1. *Data Movement.* The use of specialized hardware at the network adaptor to reduce data movement has been described in [26, 25]. In [3, 8] application buffers are located on the adaptor, and checksum hardware is provided so that the protocol code need not read the data. Software solutions such as *fbufs*[9] can reduce the cost of interdomain data transfer. Using shared memory to move data is described in [27].
2. *Minimizing System Calls.* In the *segment streaming* approach [10], once a stream is setup no further system calls are needed to move data. In [19] circular buffers are shared between a process and the kernel to minimize

copying. The kernel polls the shared memory and looks for data to be sent. The objective of the *memory mapped streams* [13] approach is to reduce the control operations in data transfer and again uses a shared memory solution.

## 6 Conclusion

We have presented a framework for providing QoS guarantees within the endsystem. We have addressed four issues in our framework namely, specification, mapping, enforcement, and efficient protocol implementation model. We have implemented a novel real-time signal mechanism that uses shared memory between each signal handler and the scheduler. Our solution reduces context switches and obviates the need for locking variables shared between handlers within a process. Our protocol implementation model uses the same shared memory mechanism to reduce the cost of data movement and context switching. We are currently building upon the real-time signal facility to implement standard protocols according to our implementation model. We have also described related work in all the areas and have shown how they fit into the overall framework.

## References


1. Anderson, D.P.: Metascheuling for continuous media. ACM Transactions on Computer Systems. Vol. 11 No. 3 August 1993 226–252
2. Anderson, D.P., Govindan, R., Homsy, G., Wahbe, R.: Integrated digital continuous media:A framework based on MACH, X11, and TCP/IP.
3. Banks, D., Prudence, M.: A high-performance network architecture for a PA_RISC workstation. IEEE Journal on Selected Areas in Communications, Vol.11 No.2 February 1993 191-202.
4. Campbell, A., Coulson, G., Hutchison, D.: A multimedia enhanced transport service in a quality of service architecture. Internal report MPG-93-22 Dept. of computing, Lancaster University, 1993.
5. Campbell, A., et.al.: Integrated quality of service for multimedia communications IEEE INFOCOM March 1993 732–739.
6. Clark, D.D., Jacobsen, V., Romkey, J., Salwen, H.: An analysis of TCP processing overhead. IEEE Communications Magazine 27(6) 1989 23–29.
7. Cranor, C.D., Parulkar, G.M.: Universal continuous media I/O Technical Report WUCS-94-34, Dept. of Computer Science, Washington University, St.Louis.
8. Dalton, C., et.al: Afterburner. IEEE Network July 1993 36–43
9. Druschel, P., Peterson, L.L.: High performance cross domain data transfer. Technical report TR-92-11, Dept. of Computer Science, University of Arizona, Tucson March 1992.
10. Gong, G., Parulkar, G.M.: Segment streaming for efficient pipelined televisualization. Proc. IEEE Military Communications Conf. MILCOMM 1992.
11. Gopalakrishnan, R., Parulkar, G.M.: Efficient quality of service support in multimedia computer operating systems. Tech. rep. WUCS-94-26, Dept. of Computer Science, Washington University in St.Louis November 1994.

12. Gopalakrishnan, R., Parulkar, G.M.: Real-time Signals: A mechanism to provide real-time processing guarantees Tech. Rep. WUCS-95-06, Dept. of Computer Science, Washington University in St.Louis April 1995.
13. Govindan, R., Anderson, D.P.: Scheduling and IPC mechanisms for continuous media. Thirteenth ACM Symp. on Operating Systems Principles 1991.
14. Jeffay, K., Bennet, D.: A rate based execution abstraction for multimedia computing. 5th International Workshop on Network and Operating System Support for Digital Audio and Video 67–78.
15. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard real-time environment. Journal of the ACM Vol.20 No.1 January 1973 46-61.
16. Maeda, C., Bershad, B.N.: Protocol service decomposition for high performance networking. Fourteenth ACM Symp. on Operating Systems Principles December 1993 244-255.
17. Mercer, C.W., Rajkumar, R.: An interactive interface and RT-Mach support for monitoring and controlling resource management. IEEE RTAS'95 Chicago 134–139.
18. Minzer, S.E.: Broadband ISDN and asynchronous transfer mode (ATM). IEEE Communications Magazine September 1989 17–24.
19. Moran, M., Wolfinger, B.: Design of a continuous media data transport service and protocol. Tech. Rep. TR-92-019, Computer Science Division, University of California, Berkeley, April 1992.
20. Nahrstedt, K.: Network Service Customization: End-point perspective. Tech. Rep. MS-CIS-93-100, University of Pennsylvania, Computer and Information Sciences, December 1993.
21. Nahrstedt, K., Smith, J.M.: The QoS Broker IEEE Multimedia Magazine Spring 1995 53–67
22. Robin, P., et.al.: Implementing a QoS controlled ATM based communications system in Chorus. Internal Rep. MPG-94-05, Dept. of Computing Lancaster University march 1994.
23. Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority inheritance protocols: An approach to real-time synchronization. IEEE Transactions on Computers Vol.39 No.9 September 1990 1175–1185.
24. Shenker, S.: An architecture of the future Internet. Proc. of Hot Interconnects Symposium August 1994.
25. Smith, J.M., Traw, C.B.S.: Giving applications access to Gb/s networking. IEEE network July 1993.
26. Sterbenz, J.P., Parulkar, G.M.: Axon host network interface architecture for gigabit communications Protocols for high speed networks, Elseiver Science Publishers, 1991 211-136.
27. Thekkath, C.A., et.al.: Implementing network protocols at the user level. ACM SIGCOMM September 1993 64–72.
28. Tokuda, H. et.al.: Real-time Mach: Towards predictable real-time systems. Proc. USENIX Mach Workshop, October 1990.
29. Vogt, C., Herrtwich, R.G., Nagarajan, R.: HeiRAT: The Heidelberg resource administration technique design philosophy and goals. Internal Report, IBM European Networking Center.
30. Zitterbart, M. et.al.: A model for flexible high performance communication subsystems. IEEE Journal on Selected Areas in Communication Vol.11 No.4 May 1993 507–518.