1

QUALITY OF SERVICE SUPPORT FOR PROTOCOL PROCESSING WITHIN ENDSYSTEMS

R. Gopalakrishnan, Guru M. Parulkar

gopal@dworkin.wustl.edu, guru@arl.wustl.edu

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

ABSTRACT

We describe a framework for providing quality-of-service (QoS) guarantees within the endsystem for networked multimedia applications. There are four components in this framework namely—QoS specification, QoS mapping, QoS enforcement, and protocol implementation. QoS specifications are at a high level, and use a small number of parameters to allow applications greater ease in specifying their requirements. Based on the QoS specifications, QoS mapping operations derive resource requirements for each end-to-end session of the application. Important resources considered are the CPU and the network connection. The third component of the framework is QoS enforcement. Enforcement is mainly concerned with providing real-time processing guarantees for protocol code during data transfer. We have implemented a real-time upcall (RTU) facility which is an enhancement to the well known upcall mechanism for structuring protocols. RTUs are scheduled using a policy called rate monotonic with delayed pre-emption (RMDP) that takes advantage of the iterative nature of protocol processing to reduce context switching overhead and increase scheduling efficiency. The last component of our framework is an application level protocol implementation model. Protocol code is structured as RTUs with attributes that are derived from high level specifications by the QoS mapping operations. We describe techniques to reduce the cost of data movement and context switching in such implementations. The RTU mechanism currently runs in the NetBSD OS on Sparc and Pentium based platforms. Initial measurements show that it is an effective means to implement protocols with QoS support.

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 [20], and the proposed integrated services internet with reservations [26], 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.

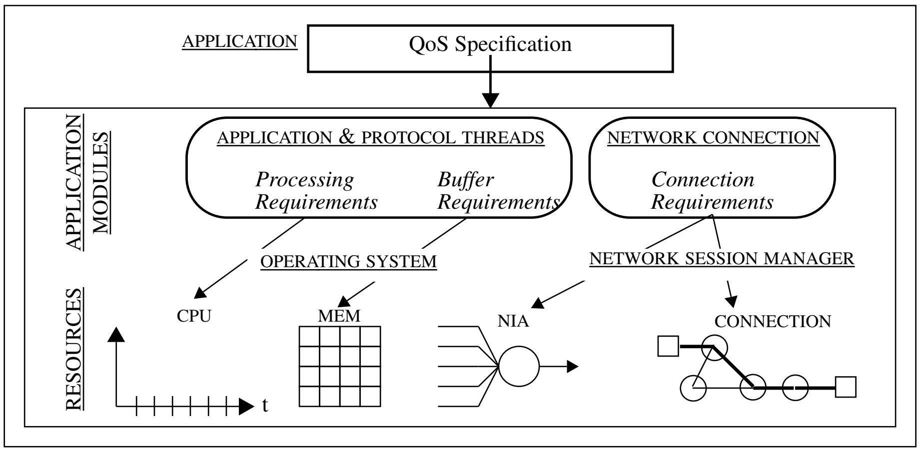

Figure 1 QoS Specification and Mapping

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 they pose. We present a brief overview of our solutions to realize each component.

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 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 as shown in Table 1 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 used in a multimedia explorer application that displays images, or other bulk data from a remote server and each transfer 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

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

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.

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. Accordingly, we list some of these approaches and their main ideas.

Transport layer QoS parameters are examined in [21, 4]. The use of application layer QoS parameters has been discussed in [22]. 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, 15, 31].

We also examine the use of application classes used to facilitate QoS specification. In [24] the *stream* and *message* types have been used that correspond to the *isochronous* and *low delay* classes in our framework. Other classifications [32] 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 threads[1] to do protocol processing for the data carried over the connections. Therefore QoS mapping operations should derive the parameters for the network connections, and processing requirements for the threads. The steps involved in the QoS mapping operation for a data stream are—

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 network connection parameters 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. We illustrate some of the steps mentioned above with an example of a CM stream such as compressed video.

3.1 Deriving Connection Bandwidth

In our example, we interpret the delay parameter specified by the application as the maximum time that can elapse between generation of a frame at the sender and its display at the receiver. This *end-to-end* delay is partitioned into three parts—the delay at the sender, the network cell delay, and the delay at the receiver. The delay at the sender is called the smoothing interval I_{sm} because the transmission of data is spread out over this interval. To meet the end-to-end delay bound, data units generated at an instant of time can wait for a duration of at most I_{sm} before they are transmitted. Thus all the data generated in

[1] The term "thread" is used in a generic sense. In Section 4 we describe a mechanism called real-time upcalls that supplant threads.

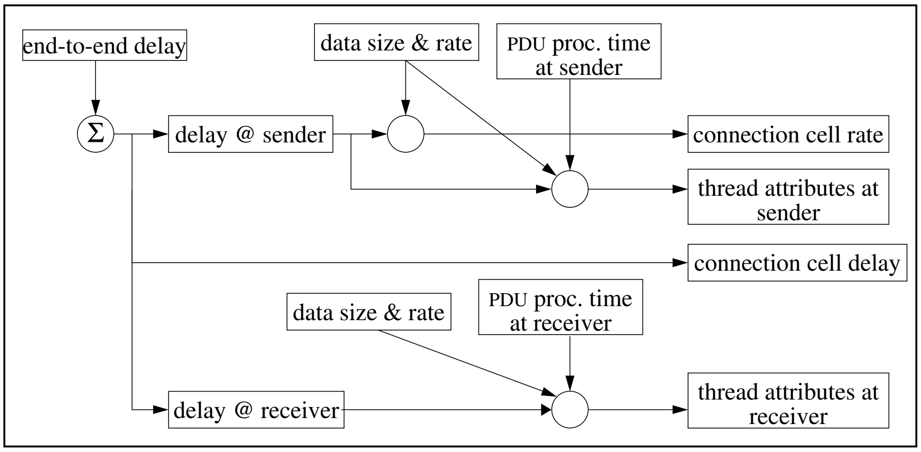

Figure 2 QoS Mapping Operations

an interval I_{sm} must be sent by the end of this interval. If we can determine the maximum amount of data S_{max} that can be generated in time I_{sm}, then the connection bandwidth is given by $R = S_{max}/I_{sm}$. In Figure 3 compressed video frames of different sizes have been generated at instants $2T, 3T$ and $4T$ and the smoothing interval is 3 frame periods. If the connection bandwidth (represented by the slope of the dashed line in Figure 3) is chosen as given above, then we can see that the data generated in any 3 consecutive frames can be drained by the adaptor within the smoothing interval.

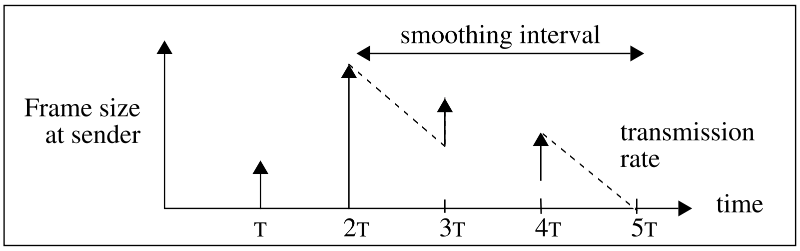

Figure 3 The Smoothing Interval

3.2 Deriving Processing Requirements

As shown in Figure 2 the processing requirement depends on the network connection bandwidth, the PDU size, and the time to process a PDU. If R is the rate of the network connection, and S is the PDU size, then the time duration P required to transmit a PDU is given by $P = S/R$. To keep up with the adaptor, the protocol processing at the sender must proceed 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 a duration $T = B_p * P$. Batching is useful for efficiency reasons, because the overhead of scheduling protocol processing is amortized over a batch of PDU's. The batch size should be some fraction of the number of PDUs that are generated in the smoothing interval. Since it can take upto time P for generated data to be enqueued at the adaptor, a larger P (B_p) reduces the smoothing interval and increases the bandwidth requirement. Thus the choice of B_p involves a trade-off between scheduling efficiency and the peak network bandwidth requirement.

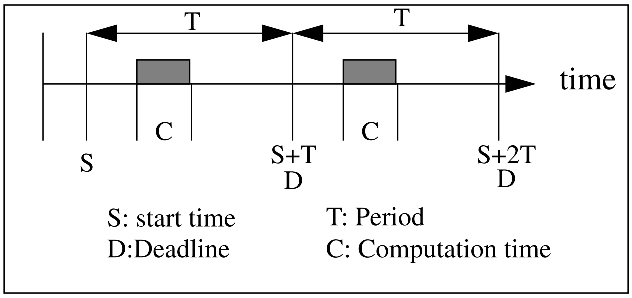

Figure 4 Periodic Processing Model

We thus arrive at a periodic model shown in Figure 4 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.

3.3 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) with standardized interfaces that implement QoS mapping operations. Thus if the processing requirements for a communication session are specified in terms of B_p and T, the QoS mapping functions can be used to determine the value of C for the specific code that is used by the session.

3.4 Related Work

The related work in this area has addressed the issues that we have outlined above, but the solutions proposed differ in terms of the mapping methodology used, and the implementation techniques adopted. We 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 [21]. 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 [31]. In [24], 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.

- **Implementation:** In most cases, the mapping operations are done by a local server process. The QoS broker mechanism is used in [23] 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 5. This organization is used by systems such as RT-Mach [30] and Chorus [24].

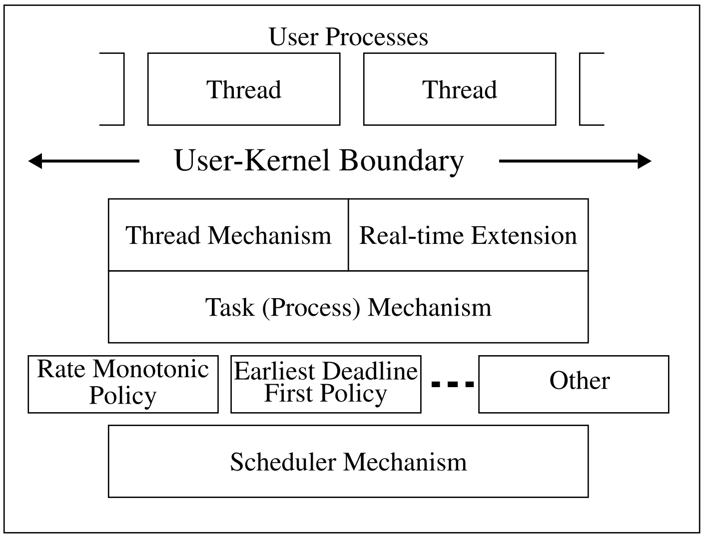

Figure 5 OS Scheduling Mechanisms

Traditionally real-time periodic threads have been used to structure code that must be run periodically. A session could be supported by multiple threads, and thread attributes (period and computation time) can be derived by the QoS mapping operation. The kernel schedules these threads using a real-time priority scheduling scheme. Typical priority schemes used are *rate-monotonic* [17] and *earliest-deadline-first* [17].

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 must be supported in the kernel scheduler leading to considerable implementation complexity. General purpose systems such as UNIX do not provide such facilities. Thus protocol code that is structured using real-time threads cannot be ported to these systems. Hence the benefits of protocols that provide 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 due to scheduling. It has been shown that context switching costs can be an order of magnitude more than the cost of processing a PDU [7]. Another consequence of a fully pre-emptive scheme is that threads that share common variables must lock them before access. For real-time threads, real-time locking operations must be implemented in the kernel to avoid unbounded priority inversion [25]. To acquire/release locks, protocol code must make system calls which leads to context switches.

To circumvent the problems with real-time threads, we have implemented a *real-time upcall* (RTU) facility to structure periodic activity. An upcall is a well known mechanism [6] to structure layered protocol code. A user process can create an RTU and register a handler function to be called. The handler function implements some component of the protocol. The kernel invokes the upcall every period and guarantees that each handler processes its batch of PDUs before its next period. We compare RTUs with the *signal* and *alarm* facilities in UNIX to bring out some major differences. Firstly only one alarm timer is available per process. Secondly there could be arbitrary delays before a signal gets delivered. Thus real-time behavior cannot be ensured. Moreover signals are suitable only to handle exceptions and other asynchronous events. The RTU mechanism on the other hand can ensure real-time behavior, and in this respect is similar to real-time threads. The advantage of the RTU facility is that it is simple to implement, and can be easily ported to different CPUs. No changes to the existing process scheduling code is required. In addition, the RTU mechanism is more efficient than threads because it reduces the number of context switches and eliminates the need to lock variables shared between RTUs in a process.

4.1 Scheduling Scheme for RTUs

RTUs 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 RTU can be pre-empted only at the end of an iteration. Thus 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 (i.e one iteration of the handler). 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 of this scheme 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 6, uses information in shared memory between each handler and the scheduler. When a higher priority RTU 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 [17] to account for the effect of delayed pre-emption. Implementation details and experimental results on the RTU facility may be found in [14].

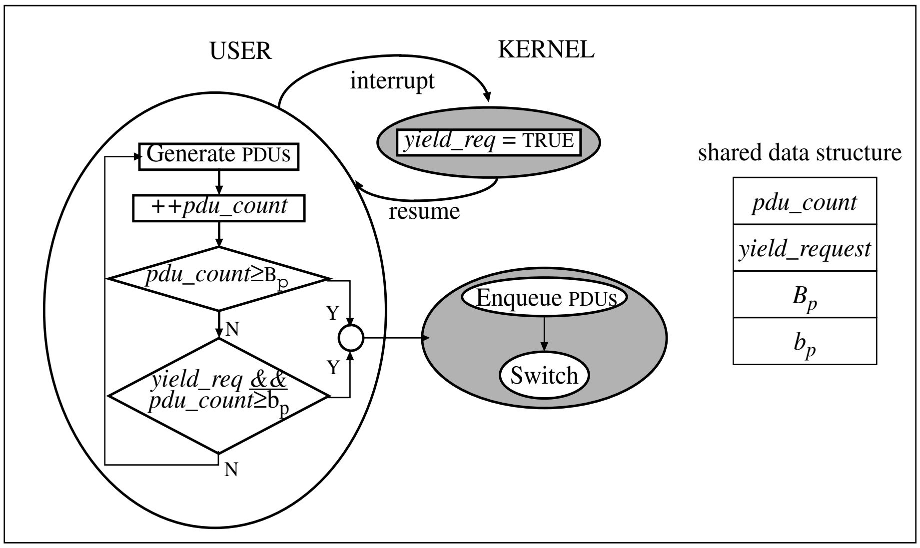

Figure 6 The RMDP Scheme

4.2 Related Work

Several alternate scheduling mechanisms have been used to provide processing guarantees. The *deadline workahead* scheduling scheme [1] is used in [15, 31]. It has been implemented using a technique called *split level scheduling* [15]. 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 [17] that is a combination of the rate-monotonic and the earliest-deadline-first schemes [22, 24]. Rate based execution has been used in [16] for adaptive applications that have varying processing requirements. Similarly the notion of a *reserve* is used in [19] to monitor and control processor usage by an application.

5 PROTOCOL IMPLEMENTATION MODEL

In current state-of-the-art implementations, protocols are implemented in the kernel, and accessed by user processes through a high level IPC interface (such as *sockets*). The kernel maintains state information for all protocol sessions. Processes make *write/read* system calls to move user data to/from protocol buffers. The protocol code multiplexes data from different protocol buffers to the network, and demultiplexes incoming packets to the appropriate process after protocol processing. For 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 higher layer protocol processing can be done in the application process instead of in the kernel. This model is referred to as the *application level protocol* (ALP) model [18, 29]. Protocol code is compiled with the application and maintains state information only for the sessions involving the process. We adopt the ALP model, because it allows protocol code to be structured as RTUs and thus facilitates the provision of QoS guarantees. By controlling the RTU parameters T, B_p, and b_p, we can control the QoS offered to each communication session. The ALP model is shown in Figure 7 where there are two processes each with one protocol session that is supported by two RTUs. One RTU processes data PDUs to be sent, and the other processes ACKs from the receiver. The network interface adaptor (NIA) paces out PDUs enqueued by different RTUs at their respective connection rates.

In the remaining sections, we describe mechanisms to efficiently perform crucial protocol operations in the ALP model. These operations include moving data

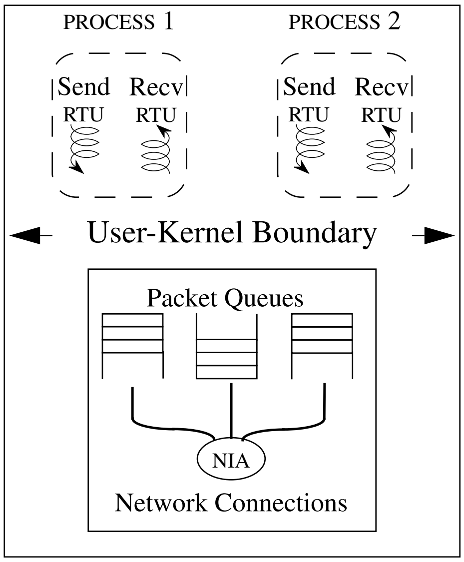

Figure 7 The ALP Model

between protocol buffers and the NIA, handling protocol events such as packet arrival, and RTU scheduling. Improving the performance of these operations is important to overcome the protocol processing bottleneck at the endsystem.

5.1 Performance Issues in ALP

It is recognized that data movement and context switching are the most expensive operations in protocol processing [7]. Since data movement speed is limited by memory bandwidth, copying of data must be avoided whenever possible. Likewise, system calls and interrupts involve switching to the kernel context, and must be minimized as well. Protocol processing operations that involve data movement and context switching are discussed below.

1. *Data Passing*: In the ALP model the data units that move across the user-kernel boundary are PDUs. Using a system call to initiate data movement on a per PDU basis will be inefficient because the number of PDUs in an application buffer can be large. In addition to the system call overhead, copying or remapping the contents of the PDU can be expensive if it is done on a per PDU basis.

2. *Asynchronous Event Processing*: Processing of asynchronous protocol events such as packet arrival and timer expiration are usually triggered by interrupts. Scheduling an RTU to handle each event as it arrives will be inefficient since each event could cause a context switch. Furthermore, scheduling RTUs based on event arrival would interfere with the priority mechanism used to schedule RTUs and violate the QoS guarantees.

Our solution to improve the efficiency of data movement and event processing is to use the shared memory mechanism we have implemented for scheduling RTUs to share information with the kernel. An RTU can use shared memory to store arguments for system calls instead of passing them on stack as in conventional system calls. The kernel can use the shared memory to store information for each event instead of upcalling the RTU for each event. Whenever the RTU is upcalled by the scheduler it examines the shared memory, processes data and events enqueued, and modifies the shared memory to initiate the next set of system calls. Thus the shared memory acts as a command-control interface between each RTU and the kernel. We describe a few mechanisms involving shared memory in greater detail in Section 5.2.

5.2 Shared Memory for Data Transfer

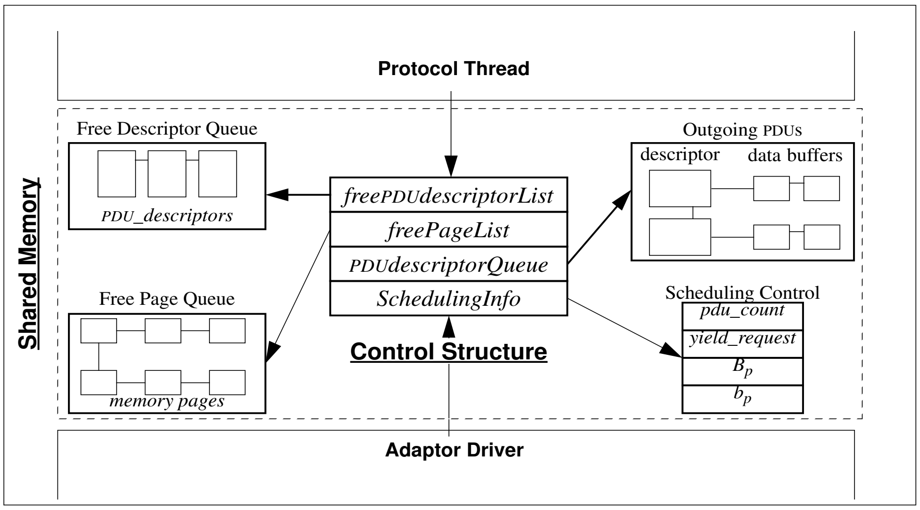

Figure 8 Use of Shared Memory

The overall organization of the shared memory mechanism is shown in Figure 8. The shared memory may be wired down to prevent page faults during access. It contains lists of PDU descriptors and PDU buffers that are used to store control information and data associated with each PDU respectively. These structures are allocated during session setup and placed on their respective free lists. The use of these structures to reduce system calls and interrupts is explained below—

- **Reducing System Calls:** We consider an RTU that implements the sender side protocol of a data transfer session. The RTU attributes determine the number of PDUs in a batch B_p that must be processed each period. After generating each PDU, the handler obtains a PDU *descriptor* from the free list and stores control information about the PDU. The descriptor is then added to a linked list of descriptors maintained 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. We see that the operations of passing the PDUs to the kernel and scheduling the next RTU are accomplished with a single entry to the kernel. This clearly saves on the number of system calls required for data movement. Furthermore, since the pages containing the PDUs are in shared memory they need not be remapped (or copied) to the kernel space. This mechanism entails changes to the API for accessing the network subsystem and these issues are being addressed in [8].

- **Reducing Interrupts:** In most protocol implementations, interrupts are used to initiate the processing of incoming PDUs. In our case however, the RTU scheduler initiates the processing of incoming PDUs. We refer to RTU based PDU processing as *specification driven* rather than *event driven*. Thus the rate at which events are processed is determined by the QoS specification. The specification driven approach does not require interrupts on a per packet basis, and uses the clock interrupt to initiate protocol event processing. Since a batch of PDUs are processed every period, the number of interrupts required are fewer. Furthermore, since clock interrupts occur anyway and are used for other purposes as well, the overall overhead is much lesser.

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 [28, 27]. In [3, 9] 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*[11] can reduce the cost of interdomain data transfer. Previous work in implementing protocols in user space [29] describes the use of shared memory for data movement. They also use batching to reduce the number of system calls. However, they do not provide processing guarantees.

2. *Context Switches.* In the *segment streaming* approach [12], once a stream is setup no further system calls are needed to move data. In [21] 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* [15] approach is to reduce the control operations in data transfer and again uses a shared memory solution.

   Several techniques have been used to reduce the number of interrupts. One method is to interrupt once per burst rather than per packet[10]. The technique of clocked interrupts [27] is very similar to our specification driven approach for event processing.

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 upcall mechanism that uses shared memory between each upcall 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 upcall 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, pp. 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," Tech. Report, University of California, Berkeley.

[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, pp. 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, pp. 732–739.

[6] Clark, D.D., "The Structuring of Systems using Upcalls," ACM Symposium on Operating Systems Principles, 1985, pp. 171–180.

[7] Clark, D.D., Jacobsen, V., Romkey, J., Salwen, H., "An Analysis of TCP Processing Overhead," IEEE Communications Magazine 27(6) 1989, pp. 23–29.

[8] Cranor, C.D., Parulkar, G.M., "Universal Continuous Media I/O," Technical Report WUCS-94-34, Dept. of Computer Science, Washington University, St.Louis.

[9] Dalton, C., et.al, "Afterburner," IEEE Network July 1993, pp. 36–43

[10] Dittia, Z.D., Cox, J.R., Parulkar G.M., "Using an ATM Interconnect As a High Performance I/O Backplane," Proc. Hot Interconnects Symp. August 1994.

[11] 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.

[12] Gong, G., Parulkar, G.M., "Segment Streaming for Efficient Pipelined Televisualization," Proc. IEEE Military Communications Conf. MILCOMM 1992.

[13] 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.

[14] Gopalakrishnan, R., Parulkar, G.M., "Real-time Upcalls: A Mechanism To Provide Real-time Processing Guarantees," Tech. Rep. WUCS-95-06, Dept. of Computer Science, Washington University in St.Louis April 1995.

[15] Govindan, R., Anderson, D.P., "Scheduling and IPC Mechanisms for Continuous Media," Thirteenth ACM Symp. on Operating Systems Principles 1991.

[16] 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 1995, pp. 67–78.

[17] 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, pp. 46-61.

[18] Maeda, C., Bershad, B.N., "Protocol Service Decomposition for High Performance Networking," Fourteenth ACM Symp. on Operating Systems Principles December 1993, pp. 244-255.

[19] Mercer, C.W., Rajkumar, R., "An Interactive Interface and RT-Mach Support for Monitoring and Controlling Resource Management," IEEE RTAS'95 Chicago, May 1995, pp. 134–139.

[20] Minzer, S.E., "Broadband ISDN and Asynchronous Transfer Mode (ATM)," IEEE Communications Magazine September 1989, pp. 17–24.

[21] 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.

[22] Nahrstedt, K., "Network Service Customization: End-point perspective," Tech. Rep. MS-CIS-93-100, University of Pennsylvania, Computer and Information Sciences, December 1993.

[23] Nahrstedt, K., Smith, J.M., "The QoS Broker," IEEE Multimedia Magazine Spring 1995, pp. 53–67

[24] 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.

[25] 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, pp. 1175–1185.

[26] Shenker, S., “An Architecture of the Future Internet,” Proc. of Hot Interconnects Symposium August 1994.

[27] Smith, J.M., Traw, C.B.S., “Giving Applications Access to Gb/s Networking,” IEEE network July 1993.

[28] Sterbenz, J.P., Parulkar, G.M., “Axon Host Network Interface Architecture for Gigabit Communications,” Protocols for high speed networks, Elseiver Science Publishers, 1991, pp. 211-136.

[29] Thekkath, C.A., et.al., “Implementing Network Protocols at the User Level,” ACM SIGCOMM September 1993, pp. 64–72.

[30] Tokuda, H. et.al., “Real-time Mach: Towards Predictable Real-time Systems,” Proc. USENIX Mach Workshop, October 1990.

[31] Vogt, C., Herrtwich, R.G., Nagarajan, R., “HeiRAT: The Heidelberg Resource Administration Technique: Design Philosophy and Goals,” Internal Report, IBM European Networking Center.

[32] 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, pp. 507–518.