Efficient Quality of Service Support in Multimedia Computer Operating Systems

Raman Gopalakrishna

Guru M. Parulkar

WUCS-TM-94-04

August 14, 1994

Department of Computer Science
Campus Box 1045
Washington University
One Brookings Drive
St.Louis, MO 63130-4899

Abstract

This report describes our approach towards providing *quality of service* (QoS) guarantees for network communication within the *endsystems* to support multimedia applications. We first address the problem of QoS *specification* by identifying a set of application classes and their QoS parameters that cover the communication requirements of most applications. We then describe the QoS *mapping* problem, and show how requirements for resources (such as the CPU, the network interface adaptor and network connections) can be automatically derived from the application QoS parameters. We then deal with the QoS *enforcement* issue in which we describe techniques for scheduling protocol processing threads in order to reduce context switching overhead, as well as derive sufficiency conditions in order to provide predictable performance. We integrate all these solutions in a protocol implementation model. The key feature of the model is that protocols are part of the application process and are processed using protocol threads with individual scheduling attributes derived using our QoS mapping method. We propose several performance improvement techniques for application level protocol implementations that can reduce the high cost of data movement and context switching in these implementations. A significant component of this work will consist of implementation and experimentation which will result in significant contributions of practical utility.

1. Introduction

The integration of continuous and discrete media (or *multimedia*) is an important trend, and recent developments in the areas of high-speed broadband networks, multimedia computer workstations, and applications such as hypermedia navigation, bear witness to this fact. The term *distributed multimedia* has been used to describe the emerging scenario in which a single integrated network will carry a wide variety of media (such as video, audio and traditional data) and endsystems i.e host computers will process these different media much in the same way as discrete data. This report is concerned with the support needed at endsystems (such as multimedia workstations) in order to provide performance guarantees for network communication to multimedia applications. Our work is targeted towards networks that can provide guaranteed service on a per connection basis, such as broadband networks based on ATM[1]. Such networks are able to support a mix of traffic types and are said to have *multiservice* capability. In particular, the report explores how the host *communication subsystem* must be implemented so that multimedia applications can benefit from the multiservice capabilities of the network.

1.1 Motivation

Emerging broadband networks [27] will be capable of transporting a variety of media, both continuous (such as video) as well as discrete (data). This capability is primarily due to the high transmission rates that are possible using optical fiber, and the use of fast packet switching technology. Meanwhile, enhancements to endsystem hardware allow them to store and process continuous media (CM) data and communicate them over network connections much in the same way as discrete data [18, 25, 33]. These two developments have opened up a plethora of multimedia applications that use the capability of the network and the endsystems to provide services to users that were not possible before.

An important aspect of multimedia applications is that they require *performance guarantees* from the endsystems (for processing) as well as from the network (for communication). For example, the broadband network provides each user a *connection* (or a *virtual circuit*) over which it carries user data, and to each connection it provides performance guarantees that are referred to as *quality-of-service (QoS) guarantees*. The term QoS refers to parameters such as bandwidth, delay, and loss that determine the "goodness" of the network service, and the values of which can be negotiated between each connection (user) and the network. The term guarantee refers to the ability of the network to honor its obligation of ensuring that each connection gets its negotiated QoS throughout its lifetime. QoS guarantees are necessary in order to be able to support CM traffic that have time-critical transport requirements. It must be noted that multimedia applications have diverse (and often conflicting) processing and transport requirements that need to be supported. For example, high bandwidth is required by applications such as visualization, imaging and video distribution. Applications such as distributed computing and distributed interactive simulation (DIS) are not so demanding in terms of bandwidth but are very intolerant to delays. Therefore supporting these applications in the emerging network and endsystem environments is an important priority of many research efforts.

While a lot of research has been done on how to provide QoS guarantees in networks (such as ATM) [14, 45] relatively less has been done to provide performance guarantees for processing in endsystems. In order to meet the processing requirements of multimedia applications, enhancements are needed in both the host hardware and software [5]. Firstly, the data paths in the host must have sufficient capacity to move time-critical, high volume CM data between hardware components such as storage units, processing elements, memory and the network interface. Furthermore, enough processing power must be available to process the data. The software components, especially the communication subsystem (communication protocols, network device driver etc.), acting under the control of the host operating system (OS), must ensure that the aggregate hardware capacity is shared between the application processes according to their individual QoS requirements. Both these problems present considerable technical challenges, and it is widely recognized that an *integrated* design of the hardware and software components is necessary to meet end to end QoS requirements of applications.

1. ATM stands for Asynchronous Transfer Mode and is the technology of choice for the broadband network.

Until recently, research in QoS guarantees has been focussed on how to multiplex network connections over network components (switches and links) in a manner that satisfies QoS parameters negotiated with the network. However, these guarantees are not “end-to-end”, because they are preserved only between the network access nodes that the endsystems are connected to. Unless we can effectively control the *processing* of data sent/received on these connections by the communication subsystem *within* the endsystem, QoS guarantees cannot be made to the application.

The main motivation for this work is that emerging multimedia applications will need end-to-end performance guarantees for network communication, and effective support at the endsystems is necessary in order to achieve this. We briefly indicate some problems that we are faced with in reaching this goal.

1.2 Brief Statement of Problems

Specification of QoS requirements is necessary to provide performance guarantees. QoS Specification is a difficult problem because of several reasons. Firstly, an endsystem must support applications with widely varying communication needs and QoS parameters. Some applications (their programmers or users) may only be aware of high level parameters that are application specific. They may be unable to translate these into parameters that directly express communication QoS parameters or processing requirements. The *specification* of QoS is therefore the first problem that we need to consider.

An endsystem has several components (or resources) involved in network communication. These resources are managed by the OS so that they can be shared by all applications. If the QoS requirements of an application can be expressed in terms of requirements for these resources, the problem of maintaining QoS is converted to a resource management problem. To be able to do this, the OS has to be aware of the relationship between QoS parameters requested by an application and resource requirement parameters. Examples of resources are the CPU, memory, and the network interface adaptor (or NIA). The derivation of resource requirements from QoS requirements is referred to as the *mapping* problem and is an important component of the solution to providing performance guarantees in endsystems.

The resources mentioned above, are shared among different application processes each with different requirements. For example, a resource such as the CPU has to be *scheduled* so as to satisfy the requirements of each user. Memory has to be partitioned amongst applications and managed during the course of execution. The goal of scheduling and enforcement is to ensure that each user gets its share of the resource, while keeping the resource utilization high. Because multimedia applications must handle time-critical data, scheduling must be in real-time. *Scheduling* endsystem resources is therefore a crucial part of maintaining QoS.

Studies of protocol performance have shown that the major portion of the cost of protocol processing is due to *data movement* and *context switching* [4, 7]. This is because memory bandwidth and interrupt latency have not improved as much as processor performance. Data movement in protocol implementations occurs when data has to move between protection domains or between the memory and network devices. Context switching occurs when control passes from one thread of execution to another (either in the same or different address space) or due to hardware events such as network interrupts and scheduling interrupts. The cost of these operations mainly depends on the *protocol implementation model* (also referred to as *transport system architecture* [35]). Coming up with an implementation model that maintains QoS as well as keeps data movement and context switching costs low is crucial to the success of multimedia applications and therefore is an important problem. Since real-time scheduling requirements exacerbate context-switching, we have to ensure that the solutions we propose to maintain QoS are not at the expense of efficiency or utilization.

1.3 Statement of Goals

In brief, our goal is to extend the notion of QoS *within* the host—from the network interface driver, through the protocol layers, and upto the application threads that generate/consume data. We want our solution to be integrated in the sense that it takes into account the application communication requirements expressed as QoS parameters, uses these to determine local and network resource requirements, and schedules the local resources that implement communication activities. Furthermore, we want to achieve the same efficiency, if not better, of state-of-the-art protocol implementations so that end-to-end QoS guarantees can be realized without reducing system utilization.

As part of the deliverables of this research, we are developing a prototype protocol suite implementation that incorporates our solutions to each of the problems mentioned in Section 1.2. The software deliverables will include an implementation of the automated mapping procedure described in Section 3.4, the implementation of the scheduling mechanism described in Section 4.2, and the implementation of our solutions to overcome the overheads in data movement and context switching described in Section 5.3. The plan of work at the end of Section 3, 4, and 5 gives a more detailed description of what kind of results we are looking for.

1.4 Outline of Report

We begin by briefly stating our assumptions about the endsystem model in Section 2. The details of the model are not part of the research and are included only for introducing terminology and for the sake of completeness. The readers may choose to skip those portions that they are already familiar with. The next three sections deal with the three components of the problem statement—Section 3 deals with the QoS specification and mapping problem, Section 4 deals with CPU scheduling, and Section 5 deals with the problems in data movement and context switching in the protocol implementation model we have adopted. In each of these sections, we present the statement of the problem, highlight the innovative ideas in our solution, and describe each solution in some detail. This is followed by discussion of trade-offs that we will study in our research, and a review of related work in that area. Section 6 presents an example that illustrates various aspects of our solution. Certain details have been left to the appendices that appear in the end.

2. End-system Model

We begin by presenting a model for the endsystem that will serve as a reference for our discussions. We describe the application model, the process model and the IPC model. We identify the set of resources that are primarily involved in the communication activity, and which need to be scheduled in order to preserve QoS. For each of these resources, we specify the resource model that we will use and provide some justification for the choices we make.

2.1 Application Model

We model a multimedia application as a pipeline consisting of several *modules*. A module represents some functionality implemented in software/hardware. A software module resides in a *protection domain* that could either be the OS kernel or a user process. A module can be regarded as a *pipeline stage* that has data units coming in at a certain rate, uses certain compute time to process a unit of data, and has data units going out at some rate. We assume that the compute and communication requirements of modules can be estimated, or derived from application specifications. In the case of network communication for example, a sender transport protocol can be regarded as a module that takes in application data, and generates protocol data units (PDUs) to be passed on to a downstream module (such as the network interface). In this case, the incoming bandwidth is determined by the rate at which the application generates data units and the processing time is the time to do transport protocol processing. The outgoing bandwidth is determined by the rate at which PDUs are generated and may be higher than the incoming bandwidth because a PDU has a protocol header addition to data. We show a typical pipeline with three stages in Figure 1.

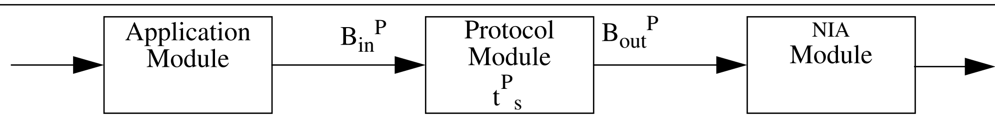

FIGURE 1. Application Model

2.2 Process Model

The process is the basic unit of resource management and consists of an address space and one or more of *threads* of execution that share the address space. There are two kinds of threads namely *kernel threads* and *user threads*. Kernel threads (KTs) are known to the kernel and managed by the system scheduler. On one (or more) KTs several *user threads* (UTs), each associated with its private execution context (a stack, register values), can be scheduled. UTs are scheduled by a *user level scheduler* that is linked to the user program but the management of UTs is transparent to the kernel. The system scheduler tracks the CPU usage of KTs in each process but not of UTs. Context switching between UTs is an order of magnitude less expensive than between KTs, and so for efficiency reasons UTs are often preferable. We assume protocol processing is done using threads which implies that communication protocols are part of the user process (as opposed to being in the kernel) and application and protocol buffers are part of the same address space [24, 39]. This has implications on data movement and context switching that will be discussed in Section 5.

2.3 IPC Model

The IPC model defines the abstraction seen by applications for sending/receiving data. The IPC services are implemented in the communication subsystem (CS). In our model, the sending(receiving) application process is viewed as a source(destination) of end-to-end data units such as for example a video frame, an image file, or an RPC message. The application specifies QoS parameters in terms of these data units to the CS on the source (and destination) which must allocate and manage resources so that these parameters are satisfied. We assume that application specific functionality of the CS is implemented by an *application thread* in the application process. These functions include application oriented error control [16] and presentation functions for example. The application thread also does segmentation(reassembly) of the end-to-end data units that are moved across the CS interface into application data units (ADUs) before passing them on to the transport protocol. In most OSs (such as UNIX) the interface to the protocols is referred to as the IPC interface because protocols are implemented in the kernel. The IPC interface functions (such as `send/recv`) are standardized so that an application can use a wide range of communication protocols without having to be rewritten for each protocol. In our model, we retain the standardized IPC interface, but the protocol code is part of the user process and is run by a *protocol thread*. Typically a network connection that carries end-to-end data with a given QoS has a separate protocol thread that processes PDUs carried on that connection. Finally the protocol threads for all connections (in different processes) interact with the network driver in the kernel. The driver maintains the incoming and outgoing packet queues for all connections and does other control operations (such as programming the NIA) that require kernel privileges. The IPC model is shown in Figure 2.

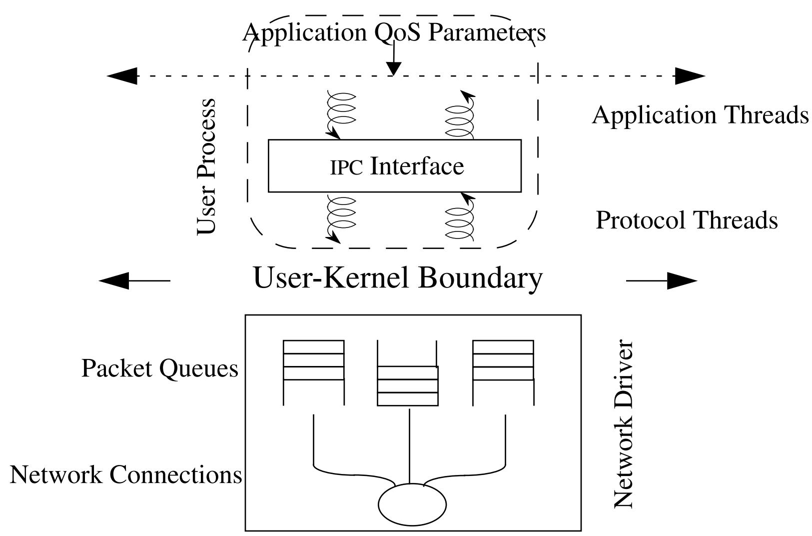

FIGURE 2. IPC Model

2.4 Schedulable Resources

In order to provide QoS guarantees, resources required to perform communication subsystem activities must be reserved for each application. Furthermore, these resources must be scheduled among applications so that the reservations made are honored. We will be concerned with two schedulable resources that are *local* to the host — the CPU, and the NIA. The OS is responsible for managing these resources. The network connection is a *non-local* resource that must be setup by the OS on behalf of applications. Each resource has a *resource model* that defines the syntax (such as parameters and permissible values) for making requests, and the meaning (or semantics) of the request parameters.

In the next three sections we present the model we assume for the CPU, NIA, and the network connection resources. For truly end-to-end QoS maintenance, every local schedulable resource (e.g., disk, I/O bus, etc.) that the application uses to generate/consume data from the network must be reserved and scheduled according to application QoS requirements. However, we focus only on the CPU and the NIA because they must be involved in any network communication, and considering other resources would considerably enlarge the scope of our problem.

2.4.1 The CPU Scheduler Model

The CPU does the application and protocol processing associated with communication, and is therefore crucial to maintaining QoS. The fundamental abstraction or resource model for the CPU that is provided by the OS is the *process* or *thread*. For our purposes of providing guaranteed share of processing time we consider a *real-time* thread model. While a lot of real-time models have primarily been developed for process control and other embedded systems, real-time schedulers intended for general purpose computing are becoming available primarily to provide support for multimedia applications. These schedulers provide *real-time periodic* threads with different priority schemes for controlling pre-emption. A typical candidate is the Mach 3.0 RT-threads package [40] that will be used as a reference model for our examples. The salient features of the scheduler are described below.

Real-time Thread Model. A periodic RT-thread has two main attributes that describe its CPU requirements. The first is the *period* T, that is the interval between successive invocations of the thread. The second is the execution time C, that is the time a thread needs to run in each period or invocation. The scheduler creates (invokes) a new thread at the beginning of every period and sets it up to begin executing at a fixed entry point in its program. The *deadline* D of a thread is the time by which it should complete its task starting from when it becomes ready, and is usually assumed to be the time when the next period begins. The utilization of a thread is equal to C/T and the sum of all thread utilizations is the total CPU utilization. There is also a *priority* associated with each RT-thread. A running thread, can be pre-empted by a higher priority thread any time during its period. Several priority schemes for real-time scheduling are known [19]. Priority schemes are classified as static or dynamic depending on whether the priority of a thread is fixed during its lifetime or changes over time. The thread model is shown in Figure 3.

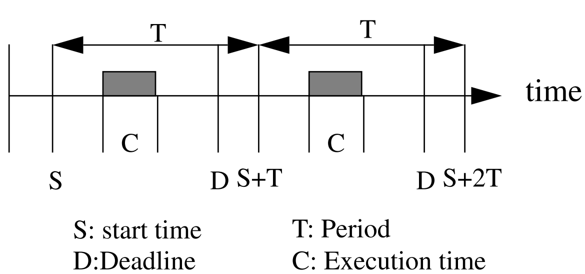

FIGURE 3. Mach 3.0 Real-Time Periodic Thread

Rate Monotonic (RM) Priority. In the RM scheme, the priority is proportional to the rate of the thread. The smaller the period, the higher is the priority. Since the period of a thread is fixed at creation time, the priority is static. An example of a dynamic priority scheme is the earliest deadline first (EDF) scheme in which the thread that has the earliest deadline has highest priority. Both these schemes have been shown to be optimal in the sense that if any static or dynamic scheme can schedule a set of threads so that they all meet their deadlines, then they can also be scheduled by the RM or EDF schemes respectively. For the RM and EDF schemes, there exists a *schedulability criterion* that can decide based on the total utilization, whether all threads can be guaranteed to meet their deadlines [23]. We choose to use the RM priority scheme for implementing the application and protocol threads and the justification for this is provided below.

Justification for Choice of Model. We choose periodic threads because they can easily express CPU requirements over an extended time duration. Periodic threads are a natural choice for CM traffic such as video although the thread period may be much smaller than the frame period. Even in the case of "bursty" sources (large file transfers), because the time to transfer the burst is much larger than typical thread periods, a periodic thread can be setup to do the processing once the burst starts. After the burst is completely processed the thread can be made inactive until the next one arrives (maybe after some random interval). The periodic model is not suited for intermittent short messages that arise in RPC and distributed algorithm applications because the processing time per message (within the communication subsystem) is comparable to a typical thread period. However if the message traffic extends over a longer duration then periodic threads can again be used.

The RM priority scheme has been chosen because of its simplicity. Dynamic schemes such as EDF require threads to be sorted according to deadlines and so a fixed priority scheme such as RM is preferable. Both RM and EDF are meant for providing strict real-time guarantees and a choice among them must consider trade-offs in the cost of context switching and complexity of implementation. Our objective is to take one of these schemes and modify it in order to reduce the context switching overhead. The solutions however can be applicable to both cases. Our focus is therefore not on coming up with a new priority scheme but rather to choose one that can potentially give good performance and adapt it for the specific requirements of protocol processing. This is the rationale behind our choice of RM priority scheme.

2.4.2 The NIA Model

Access to the network from the host (and vice-versa) is through the network interface adaptor (NIA) or adaptor for short [4, 8, 9, 36]. The NIA is controlled by the network driver as shown in Figure 2. The driver programs the NIA according to the requirements of each connection, responds to interrupts generated by the NIA, and does other control operations required of a device driver. The NIA operates concurrently with the CPU and can read and write main memory without any CPU intervention using DMA. For example, while a protocol thread is generating PDUs to be sent on its connection and placing them in memory, the NIA could be transmitting data enqueued earlier on this (or another) connection. Therefore send and receive operations from memory can proceed concurrently with application and protocol processing.

An important task performed by the NIA is scheduling the transmission of outgoing data. Data is sent in the form of fixed size *cells* on network connections and controlling the timing of cell transmissions on connections is referred to as *pacing*. Pacing ensures that the rate at which data is sent on a connection conforms to the peak (and in some cases average) bandwidth reserved for it from the network. This is important because the QoS guarantees provided by the network are contingent upon this rate being honored by the sending host. Pacing is implemented by the NIA hardware on a per connection basis. To preserve QoS, the choice of pacing parameters are important, and the parameters depend on the pacing scheme used. We consider a pacing scheme below, that can control both the rate at which cells are sent out on a connection and the queueing delay at the NIA for each connection.

Hierarchical Round Robin (HRR) Pacing. This scheme was proposed for doing pacing at the output ports of ATM switches and has also been demonstrated in a hardware implementation [21]. The pacing scheme enforces the peak bandwidth for each connection. When an application process generates a burst (a video frame for example), the HRR scheduler in the NIA paces the transmission of cells in the burst at the peak rate allocated to the connection. To maintain the average rate, the host must control the rate at which the bursts are generated by scheduling the application appropriately. In the HRR scheme, there are a number of *levels* and each connection is assigned to one of these levels. The total link bandwidth is partitioned among these levels by assigning a *frame time* for each level, and allocating a certain number of slots for a level in its frame time. A slot represents a cell transmission, and in each frame time the number of slots allotted places a limit on the number of cells belonging to connections in this level that can be sent. For a given level, the connections in it get cell slots in a round-robin manner during each frame time. Lower levels have smaller frame times than higher levels. Since a connection at each level is given some slots every frame time, connections assigned to lower levels have to wait for a shorter time before they are serviced. The frame time for level 1 is the basic cycle time which is the smallest in the system. If there are n_1 slots in a level 1 frame, then b_1 slots are given away to higher levels and the remaining (n_1-b_1) slots are used for the level 1 connections.

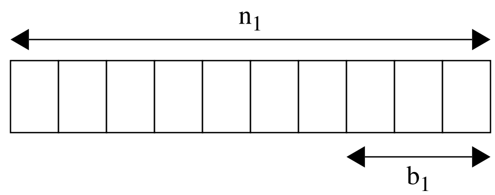

FIGURE 4. Out of a frame time of 10 slots in level 1, 3 slots are allocated to higher levels

Similarly, level 2 has n_2 slots and a frame time $FT_2 = (n_1/b_1) \cdot n_2$. This is because every n_1 slot times it gets b_1 slots, and it needs n_2 slots in all. Out of these n_2 slots, b_2 slots would be given away to levels higher than 2 and so on. In general, the frame time at level i is the interval between serving a slot in successive frames and is given by $FT_i = (n_1/b_1) \cdot (n_2/b_2) \cdot ... (n_{i-1}/b_{i-1}) \cdot n_i$. The bandwidth allotted for level i is $(n_i - b_i)/FT_i$. Thus the scheme statically partitions bandwidth among levels by appropriate choice of n_i and b_i for each level. A connection at level i that gets a_j of the n_i slots, gets a bandwidth of a_j/FT_i cells/sec. It will have to wait at most FT_i cell times before it is first serviced after it is enqueued for transmission (start-up delay). Since the smallest value of a_j is 1, the smallest value of bandwidth at level i is $1/FT_i$ and defines the minimum "granularity" of bandwidth partitioning for that level.

Justification for choice of HRR. As before, our objective is not to argue the merits of any pacing scheme but to choose a prototypical scheme that can serve as a model. Our main focus is to demonstrate how to derive pacing parameters from QoS specifications for the given model. Since the HRR scheme requires support from NIA hardware, in our implementation we will use whatever pacing support is available from the network adaptor. We are in the process of designing an adaptor [11] that provides a single parameter pacing control. The scheme is equivalent to a HRR scheme with one level and we propose to use it for our experiments.

2.4.3 The Network Connection Model

The network connection represents a "data pipe" that carries fixed size cells from one (or more) source hosts to one or more destination hosts. For our purposes we assume that applications use simplex connections between a single source (host or endpoint) and a single destination. We assume that QoS requirements are specified for the peak bandwidth, the average bandwidth (over some interval), and the maximum delay that a cell can experience. Accordingly, we assume a model proposed in [14] in which these requirements are specified by the following parameters.

- The minimum cell interarrival time x_{min} (peak bandwidth).
- The minimum value of the average cell interarrival time x_{ave} over an interval I (average bandwidth).
- The source to destination delay bound for each cell.

The model guarantees that if the host conforms to the minimum and average bandwidth specified above, the connection will be provided the requested delay bound (and the requested bandwidth).

Justification for Network Model. The network connection model presented above allows applications to control the two important parameters namely bandwidth and delay for a connection. The performance of this model has been studied in [14] and implementations of the model have also been reported. While network connection models that provide control on other parameters such as delay jitter could also have been considered, the described model is sufficient for our needs. As before, our interest is in demonstrating how QoS parameters drive the choice of network connection attributes. We make the simplifying assumption that the network delay bound is fixed and is known to the endsystems.

3. QoS Specification and Mapping

Having looked at the endsystem model, we are now ready to present the first problem namely the specification of QoS parameters and its mapping to resource requirements according to the resource models presented above. We state the problem, and present our solution. We then describe the work in progress and give an overview of the related work.

3.1 Problem Statement

We have seen that QoS requirements for an ATM network connection are specified using parameters as shown in Section 2.4.3. However, at the application level, the choice of which parameters to specify and how to assign values to them needs to be addressed. Furthermore, each module in our application model uses certain resources to carry out its activity. The QoS specified by the application affects the requirements for all these resources and so there is the problem of mapping the QoS parameters to the specific resource requirements for each module. These two problems are further clarified below–

1. *Specification*: There are two problems in specification of QoS. Firstly, there is a wide diversity in the types of applications and their communication requirements, but it would be impractical to allow the same diversity in the types of QoS specifications. Secondly, an application user or programmer may be able to specify QoS parameters at a high level of abstraction, but low level parameters are needed to be able to determine CPU and NIA requirements. An example of the second problem is a DIS application that knows to specify only the rate at which it wants to process messages because it is the most important parameter from its viewpoint. However the application may not know the detailed resource requirements to support its needs and therefore cannot specify it.
2. *Mapping*: As mentioned before, several system components are involved in network communication and the requirements specified above must be projected onto each of them. Because of the need to keep application specifications simple, and of the fact that an application may not even be aware of the resources involved in supporting its communication, the application QoS parameters must be *mapped* by the communication subsystem to the resource requirements according to the resource model. For example, in the network connection model (as per Section 2.4.3), QoS parameters such as delay, bandwidth and loss are specified at the level of cells. However, at the application level the delay is measured end-to-end, and is over larger units of data. Similarly, the bandwidth refers to net throughput (without header overheads), and the unit of error control is usually an ADU. The mapping operation is needed to convert application layer parameters to lower layer parameters. Another example of the mapping operation is the case when the transport layer may use control connections (to exchange control messages) that are invisible to the application. The QoS parameters for these connections must be chosen by the transport subsystem.

   In our case, the mapping problem involves deriving the network connection attributes (to be requested from the network session manager), the NIA scheduling attributes (to be requested from the network driver), and the CPU scheduling attributes for the threads involved in supporting network communication (to be requested from the system scheduler).

The specification and mapping operations are illustrated in Figure 5.

3.2 Proposed Solutions

The main aspect of our solutions to the specification and mapping problem are—

- We choose a canonical set of application types that have substantially different communication requirements and identify QoS attributes for each type. We claim that most of the data streams used by applications will fall into one of these types differing only in the values of their QoS attributes. Therefore supporting these basic types would serve the communication needs of all application types.
- For each type, we map the QoS attributes specified by the application to parameters of the resource models used for the CPU, NIA, and the network connection resources. The innovative aspect of our solution is that the communication subsystem automatically determines resource parameters after which existing resource management mechanisms of the OS ensure that QoS is maintained. This means that protocol code need not be burdened with maintaining the "timeliness" of protocol operations during run-time. This makes user-level protocol implementations (see Section 5) simple and efficient because no kernel interaction is needed to schedule protocol operations.

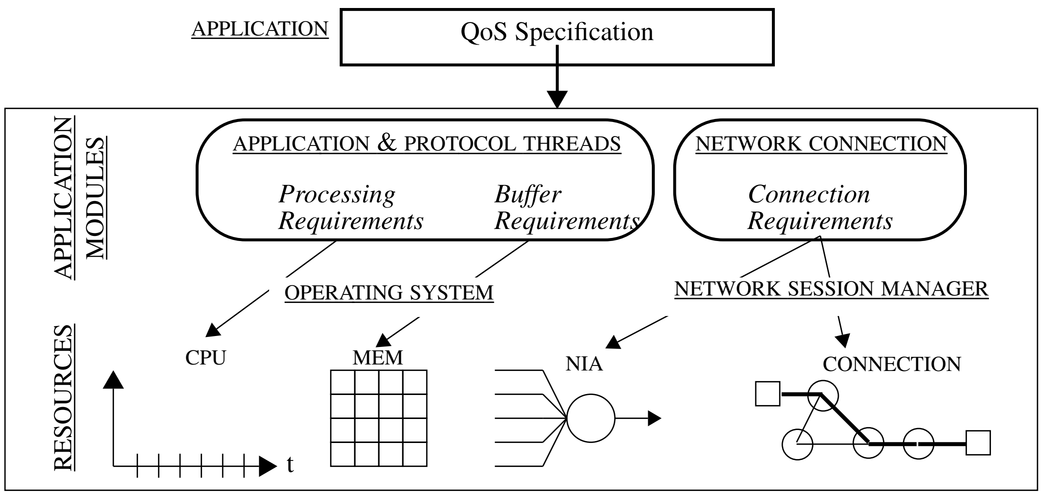

FIGURE 5. QoS Mapping

3.3 QoS Specification

We first consider three significantly different classes of QoS requirements for network communication that applications could have. For each class, we list the QoS attributes that must be specified. We then argue that these classes are necessary and sufficient to support many target multimedia applications.

Application Classes. We consider three types of application communication requirements. These are

1. *Isochronous*: This includes the domain of continuous media both compressed and uncompressed (video, audio etc.) in which data is generated and/or consumed at an endpoint at a fixed rate such as from a camera/microphone. The traffic is characterized by three parameters—*frame rate*, *maximum* frame size and *average* frame size. In addition, a maximum *delay* between generation of a frame and its display at the receiver is specified.

   The motivation for specifying the maximum frame size is to determine the maximum application buffer requirements. The average frame size allows us to reserve less than the maximum bandwidth that would be dictated by the maximum frame size. The delay refers to the time shift in the CM stream at the destination with respect to the source and is measured as the time interval between the beginning of a frame at the source (typically the generation time) and its beginning on the destination (when it gets displayed).

2. *Burst*: This includes most cases where bulk data transfer is required. Unlike isochronous traffic, a bulk data source does not have a natural period and generates data blocks of varying size. QoS parameters are therefore a maximum *blocksize* that determines buffer requirements, and either a *minimum bandwidth* or a *maximum delay* for each block. Delay here refers to the duration from the time the block is generated till the time it is available in the

receiver's buffer. Delay and bandwidth are equivalent, and given the bandwidth and the size of the block, the maximum tolerable delay can be calculated. This type of transport service would be useful in an interactive multimedia explorer application for transferring images, documents etc. where each burst is assured guaranteed bandwidth (or delay) bounds. We can allow these parameters to be specified on a per burst basis because the assumption is that the burst size is large enough so that the time to allocate local resources is much smaller than the duration of the burst.

3. *Low delay*: This class includes the applications that require low response time such as RPC requests/responses, and for moving control information (such as for ACKs). As in the burst case, the QoS parameters can be specified by a maximum *message size* and a maximum tolerable *latency*. Low latency traffic differs from the *burst* traffic, in that the entire message is moved across the IPC interface (as a single ADU), and the ADU size is small enough that the protocol does not wait for any feedback from the receiver (such as ACKs, flow credits etc.). We refer to this kind of message traffic as *message delay bounded*. Bandwidth requirements for this type are very low compared to the *burst* type and delay is the dominant parameter.

   Another kind of low latency requirement is one that specifies a message rate (*message rate bounded*) rather than a per-message latency value. Although these two forms may appear to be equivalent, the former implies that the message traffic is expected to extend for some duration at the specified rate. This allows messages to be delivered in *batches* from/to the user process as long as the desired rate is maintained. This is motivated by applications such as DIS that have large number of messages coming into an endpoint. Because these two types have latency as their critical requirement, they are grouped under a single class. The mapping of these two types of requirements to CPU and NIA resources is quite different as we shall see.

The three classes and their parameters are shown in Table 1. We have separately represented the two types of the low-

TYPE	MAX_SIZE	FRAME RATE	AVG SIZE	DELAY	BANDWIDTH
Isochronous	✓	✓	✓	✓	
Burst	✓				✓
Message delay bounded	✓			✓	
Message rate bounded	✓			✓	

TABLE 1. QoS Specification

delay application class.

Justification of Classes. We argue that the four types represent prototypical communication requirements. The isochronous type covers the class of continuous media that will constitute a significant proportion of multimedia traffic. Applications such as video distribution and retrieval are well known examples in this class. The *burst* type is the most natural characterization of communication requirements of applications such as multimedia mail and remote navigation through hypertext because such applications need to transfer a large block of data (e.g., pictures, images, text, video fragments) to the destination application within a fixed interval after it has been requested. The low-delay applications are representative of distributed computing where delay bounded messages are sporadically generated and have very low delay requirements. The bounded rate messages are required to efficiently support sustained incoming message traffic where the message rate needs to be guaranteed. Examples of these applications are large scale distributed interactive simulations (DIS).

We can view the choice of classes from another angle. If we consider bandwidth, delay and ADU sizes as three independent variables, and have a "high" and "low" requirement for each, then we have eight possible combinations. For large ADU sizes, both low bandwidth and low delay is clearly not simultaneously possible. Similarly high bandwidth and high delay is not possible for low ADU sizes. We also eliminate the two cases with high delay and low bandwidth because they are not demanding in terms of performance. Of the remaining four combinations, two of them have large ADU sizes and high bandwidth requirements, and the other two have small ADU sizes and low bandwidth requirements. We refer to these two categories as *bandwidth critical* and *delay critical* respectively. The bandwidth critical category can in general tolerate larger delays than the delay critical category. However, within each category we differentiate between applications that can predict the traffic over longer durations and those that cannot. For example in the bandwidth critical category, the *isochronous* class can specify the interval between successive frames whereas the *burst* class cannot. Similarly in the delay critical category, the *message delay bounded* class cannot predict future messages whereas the *message rate bounded* class can do so. This difference in predictability affects the way thread attributes are chosen and connection parameters are chosen. In general, for traffic that is expected over a longer duration, processing of successive data units can be grouped together. In addition, the QoS parameters can be enforced for a group of data units and not necessarily for each data unit.

Choice of QoS Parameters. The choice of parameters in each class have primarily been driven by whether the application can give meaningful values to these parameters and whether the parameters significantly affect resource requirements. We argue that applications do not have accurate estimates about long term (i.e average) values for their bandwidth requirements. This is true particularly for the *burst* and *message delay bounded* case where future traffic pattern depends on user behavior. For the *burst* case, specifying QoS requirements per burst suffices, because reserving host resources (such as CPU) is purely a local decision and can be done on a per burst basis. However, the time for setting up the network connection for each burst could be significant and so we may be forced to reserve a peak rate connection. Ideally we would like burst reservations (such as fast buffer reservation) also at the network connection level.

Likewise for delay bounded messages, applications typically cannot specify interarrival times using statistical measures or by other means. Therefore only a per message delay requirement can be specified. It must be mentioned that the delay bound is not strictly adhered to and the system tries to provide the lowest possible delay for each message.

Message rate guarantees are likely to be specified only at a receiver. We have in mind multicast connections in which each endpoint receives large number of messages but sends only a few. The receiver does not need to derive connection attributes nor pacing control parameters. Only thread attributes need to be derived from the QoS parameters.

In the case of *isochronous* type the traffic is more predictable because of the nature of the source. The parameter choices made here reflect the characteristics of compressed frames of CM data. For compressed CM data, the average frame size is computed over a fixed interval and we assume there is one frame of maximum size in this interval (The I frame in MPEG for example). Where this is not the case, an extra parameter could specify the length of the interval over which the average size is computed. This allows us (in some cases) to keep the connection bandwidth closer to the average bandwidth requirement than to the peak.

3.4 QoS Mapping

We have considered four types of QoS requirements and their parameters. The values specified for these parameters must hold at the boundary between the communication subsystem and the rest of the application that generates/consumes the data. We have seen in the IPC model that several modules operate in a pipeline fashion in the communication data path. As mentioned in Section 3.1, we must map the values of the QoS parameters to resource requirements for each module. We describe the mapping process for a connection in the sequence in which it would actually be done during operation. Accordingly, the mapping steps are the following.

1. We scale the application bandwidth required across each module, and partition the application specified delay among the modules. The scaling operation gives a multiplying factor to account for the increase in the required bandwidth due to protocol header and retransmission overheads if any. The delay partitioning gives the delay budget for each module and is used to derive the requirements for the resources used by the module. We show details of this operation in Section 3.4.1.
2. We use the delay budget and scaling factor derived earlier to determine the parameters for the network connection (shown in Section 3.4.2.).
3. From the network connection peak bandwidth derived in the previous step, and the delay budget, we can determine the pacing attributes that must be enforced at the NIA for the given connection. This mapping is shown for the HRR pacing model in Section 3.4.3.
4. From the network connection peak bandwidth, and the delay budget derived in Step 1, we can derive scheduling attributes for the protocol and application threads. This is shown in Section 3.4.4.

3.4.1 QoS Parameter Scaling

Scaling Bandwidth Requirements. Bandwidth critical applications generate *frames* or *blocks* of data that are presented to the communication subsystem for transmission. The splitting up of a frame or block into smaller segments such as ADUs, PDUs and cells, requires control information in the form of headers to be appended to each such data unit. Due to the header overhead, the actual number of bytes sent are more, which translates to an increase in bandwidth requirement at the network connection level. Appendix A shows the derivation of a multiplying factor that accounts for this increase.

Partitioning Delay Requirements. Unlike the bandwidth that must be scaled as we go along the data path, delay has to be *partitioned* across the modules in the datapath. Figure 6 shows the end-to-end delay which is the time that is available for an end-to-end data unit that is handed over to the communication subsystem at the sending host, to be delivered to the receiving application. This data unit could be either a video frame within a stream, an image file, or a message. For our discussion, we assume that data units are processed in terms of fixed size PDUs.

From Figure 6 we see that after a PDU leaves the sending host some time elapses before it reaches the receiving application. We call this the *transit delay* and is given by the sum of items 5+6+7. For a data unit to meet its deadline at the receiver, the time at which its last PDU must leave the sender is the end-to-end delay minus the transit delay. It can be seen that all data units that are generated during this time must leave the sending host by the end of this time in order to meet the end-to-end delay bound. Therefore this time is called the "*averaging*" or "*smoothing*" interval because the NIA smooths out the transmission of data units that are presented to the communication subsystem over this interval. In the case of a CM stream, if this time is two frame periods for example, then two consecutive frames must leave the sender at the end of every two frame periods. In the case of a burst that is presented to the communication subsystem, all of it must leave the sender within this time.

We now consider the sending host. The smoothing interval obtained above is the deadline for the PDUs generated within it to leave the host. We have assumed that the protocol thread generates PDUs periodically (period P) and enqueues it at the NIA. The NIA then paces out each set of PDUs before the next set is enqueued. In Figure 6 the sum of items 1+2 is the thread period P, and item 3 is the time taken by the NIA to begin pacing out a PDU after it is enqueued and depends on the pacing interval parameter. As before, the time at which the last batch of PDUs must be enqueued at the NIA is the smoothing interval minus the sum of items 1+2+3. This sum is also called the *startup delay* because the NIA must wait for this time before it can start sending the first PDU. The smoothing interval minus the startup delay is the time that is available for completing the protocol processing at the sender. It can be seen that larger the thread period and the pacing interval, less is the concurrency in operation of the CPU and the NIA.

Finally, the smoothing interval minus the startup delay is also the time available for data generated in the smoothing interval to be transmitted on the "wire" and thus determines the bandwidth chosen for the network connection. This transmission time is the sum of items 3+4 in Figure 6.

The delay partitioning described above reveals a lot of trade-offs. A larger value of startup delay leaves less time for the NIA to finish sending the data, and so a higher bandwidth for the network connection is required. In addition, it means that protocol processing must be completed in a smaller time. However, keeping the startup delay small would lead to incurring higher scheduling cost at the endsystems. Similarly, choosing a larger value for the transit delay reduces the smoothing interval and increases connection bandwidth and processing cost on the sender. However it allows the network connection to be setup with a larger delay and allows the thread period at the receiver to be larger.

The way to think about these trade-offs is that, given an end-to-end delay requirement, the communication subsystem must partition it so that the lowest possible "cost" is incurred. The cost captures the notion of optimality because several valid partitionings may be able to meet the specified delay bound. The values obtained for each component of the delay after the partitioning step above, are used to determine the network connection parameters, the pacing parameters, and the protocol and application thread attributes as shown in the next three sections.

An important thing to note about the delay partitioning is that it does not take into account the cell delay variation (delay jitter). This is because for large data sizes, the jitter is a small component of the total end delay. Therefore its effect can be taken into account by using an upper bound for the cell delay.

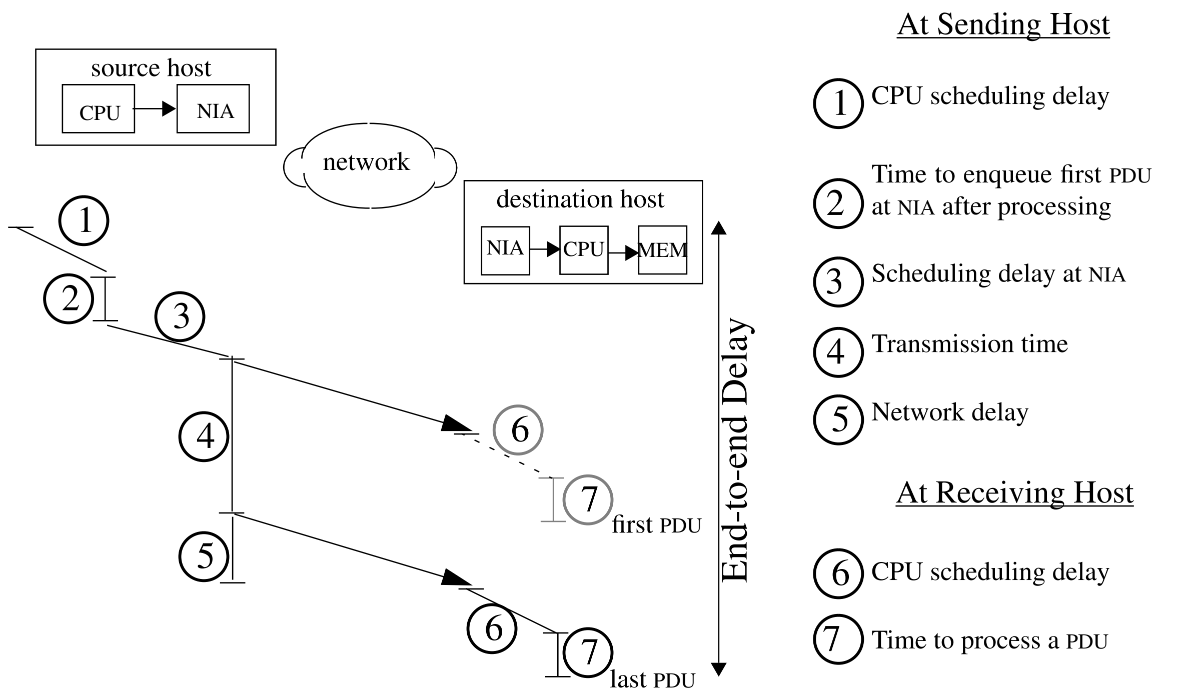

FIGURE 6. Partitioning Delays

We also do not explicitly mention buffer requirements for jitter removal at the receiver while describing the delay partitioning process. However it can be easily seen that the smoothing interval at the sender also corresponds to the smoothing interval at the receiver and can be used for jitter removal.

3.4.2 Mapping to Network Connection Parameters

In this section the derivation of the peak and average bandwidths of the network connections for bandwidth critical applications is discussed. In the case of the delay critical connection types, only the delay parameter is relevant. Since we have assumed connection delay to be constant, we will not discuss the delay critical case.

Deriving Bandwidth Parameters. Given the smoothing interval, we need to estimate the maximum amount of data that can be generated in this time because this data must leave the sending host by the end of this interval. The size estimate, along with the smoothing interval and startup delay values derived earlier, determine the peak bandwidth of the connection as shown below for the two types.

1. *Burst*: We consider the case when bandwidth can be reserved on a per burst basis. In this case, given the requested throughput and the size of the burst we can estimate the end-to-end delay. We then partition the delay as shown above and obtain the smoothing interval and the startup delay. Therefore given the burst size, and the time that remains after subtracting the startup delay from the smoothing interval, we can calculate the network connection bandwidth as shown in Appendix A.2. If no burst reservation scheme is available, we could use the specified maximum buffer size to calculate the end-to-end delay and obtain the network connection bandwidth as before. If the actual burst sizes are large enough that the startup and transit delays do not dominate the end-to-end delay, then the actual throughput will be close to the requested value.

2. *Isochronous*: There are two cases depending on whether the frames have constant size or variable size (due to some compression schemes). We use the application specified delay QoS parameter and partition it to obtain the smoothing interval and the startup delay. For constant size frames, the amount of data in this interval is the frame size multiplied by the number of periods in the interval. As before, we use this size and the value of the smoothing interval minus the startup delay, to estimate the connection bandwidth as shown in Appendix A.2.

   For variable sized frames we get the benefit of having a larger smoothing interval. This is because the maximum frame size may occur only once in several frames (MPEG video). Therefore we may assume that only one frame in the smoothing interval has the maximum size specified in the QoS, and for the other frames we can use the value specified by the average frame size parameter. This allows us to choose the connection bandwidth to be closer to the average rate at which data is generated rather than the peak. This is shown in Figure 7 in which the transmission of two successive frames is smoothed out over two time periods.

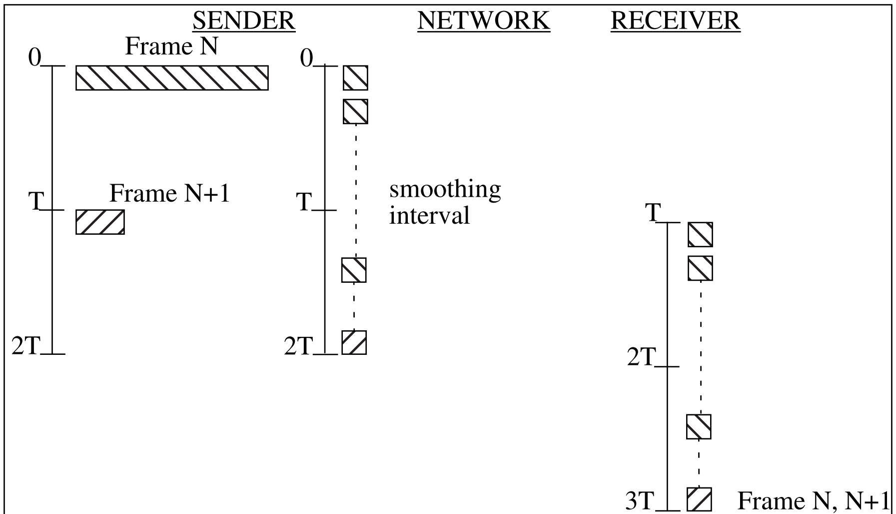

FIGURE 7. Smoothing Interval

   The total delay is three frame periods and after subtracting propagation delays we have two frame periods for the smoothing interval. The maximum amount of data that can be generated in two consecutive frame times is the sum of the maximum size frame and the average frame size. The transmission of the maximum size frame can "spill over" to the next frame time without violating the delay bound. At time '3T' the receiver has two frames that can be played back at times '3T' and '4T'.

3.4.3 Mapping to Pacing Parameters

The pacing parameters must ensure that outgoing cells on the connection are paced out at the peak rate (derived above), and the delay at the NIA is within the permissible value calculated in the delay partitioning step. For bandwidth critical connection types we show how pacing parameters are chosen so that the network connection bandwidth derived in Section 3.4.2 is maintained. For delay critical connection types we show how pacing parameters must be chosen to ensure low delay. We assume the HRR model introduced in Section 2.4.2.

Maintaining Peak Bandwidth. In the HRR scheme, for each connection we must choose a level (that determines the frame time) and some number of slots in this level (that determines the bandwidth). Starting at the lowest level (highest bandwidth granularity), we check if the requested connection bandwidth is higher than the minimum bandwidth granularity at that level. If this is not the case, we consider levels in increasing order and choose the first level that satisfies the condition. Suppose the level chosen has n_i slots, a bandwidth granularity of B_i, and the requested bandwidth is B_{req}. Then the number of slots 'k' allotted to this connection must satisfy $k = \lceil B_{req}/B_i \rceil$. If these many slots are available then they can be assigned to the connection. Section 6 shows a complete example of how these choices are made.

However if not enough slots are available at a level some reassignment of the hierarchy must be done. We do not discuss the details of this procedure here but mention a few options and their trade-offs. One option is to simply assign the connection to a lower level that has unused slots. The disadvantage of this is that some link capacity may go unused. The other option is to move to a higher level that has enough free slots. However this increases the required number of slots, and it is less likely that they will all be available. Another problem with this is that it increases the frame time which contributes to the startup delay. Finally, slots from lower level may be moved to higher levels. This operation entails changing the hierarchy structure and there are run-time overheads associated with doing this.

Maintaining the Delay Bound at the NIA. The main feature of the HRR scheme is that it can provide a range of levels (i.e frame times) and control the bandwidth assigned to each level. Since delay critical connections cannot estimate bandwidth requirements accurately, the only pacing parameter that we try to control is the delay at the NIA. In particular, we would like to provide each message the lowest delay. We therefore assign these connections to the level that has the smallest frame time. This is because the maximum delay for a connection at the NIA is the frame time of its level. The number of slots assigned to a connection must be enough to accommodate its maximum message size. The NIA fills in the level 1 slots with cells belonging to low-delay connections queued in some order (possibly FIFO). The advantage of this is that the percentage of link bandwidth allocated to delay critical connections is bounded. If messages are generated fairly infrequently then they will be guaranteed the lowest delay in the system.

3.4.4 Mapping to Thread Attributes

We now consider the choice of scheduling attributes for the protocol and application threads. We assume the periodic thread model introduced in Section 2.4.1. The discussion here assumes that each of them execute as separate kernel threads. For efficiency reasons however, we may implement them as user level threads and have a single KT that runs the user threads. The discussion below does not apply to the processing of delay bounded message connections because they are not periodic in nature. We first look at the factors affecting choice of thread attributes and show how to choose the attributes of these threads.

Choice of Thread Periods . The rate at which cells are paced out on a connection, and the PDU size together determine the time period required to send one PDU. In order to keep up with the NIA, the protocol thread at the sender must process a PDU once every such period. Similarly the application thread period depends on the time taken to transmit an ADU. However if we setup a thread to generate one PDU every period then a context switch is required for every PDU. Therefore if the thread period is comparable to the context switch time then a large fraction of time is spent for scheduling, and utilization will be adversely affected.

However since the upper limit on the period can be as much as the startup delay, we can reduce the context switch overhead by choosing a larger value for the thread period and process a *batch* of PDUs in each period. The use of batching is further described below.

Batching. Batching is the technique used to make an appropriate choice of thread period. If a batch of B_p PDUs are processed per invocation of a protocol thread, the thread period obtained above, gets multiplied by the factor B_p. Similarly the application thread uses a batch factor B_A. The batch size must be chosen so that the period is a large multiple (say 100) of the context switch time. However the period cannot be made arbitrarily long because in the worst case one period must elapse before the first batch of PDUs are generated and handed over to the NIA for transmission. This reduces the concurrency in the operation of the CPU and the NIA. Moreover the period cannot be set to a value that would violate the startup delay bound determined in the delay partitioning step. Therefore the choice of the batching factor must be made after evaluating the tradeoffs mentioned above. Expressions for the periods of application and protocol threads are derived in Appendix A.3.

3.5 Work in Progress

We are in the process of defining an API to the CS that incorporates our choice of classes and parameters for QoS specification. This API will be similar to the `'socket'` interface but the `connect` and `accept` calls will have parameters to specify QoS. We highlight issues relating to two main aspects in QoS mapping that are being worked out. The first relates to the methodology for mapping and the second relates to implementation issues that arise in the host communication subsystem.

***Mapping Methodology*:** Having developed the mapping methods described earlier we want to evaluate how well they perform in practice. In particular, we want to check if the values derived for the network connection attributes, pacing parameters, and thread attributes give good performance in a particular implementation environment.

We first consider the choice of network connection attributes. We have mentioned the trade-off between bandwidth and delay while describing the mapping method. We are specifically interested in knowing whether bursty CM data (such as MPEG video) can take advantage of higher tolerance to delay by reducing the connection bandwidth requirement. We plan to study this issue by experimenting with the endsystems and network testbed available to us. In addition, we want to consider alternate network models (e.g the ATM forum standards) that require attributes such as delay jitter and various kinds of losses to be specified and study how these attributes must be chosen.

An important question we will like to settle concerning the NIA model based on the HRR scheme, is whether the number of levels, their bandwidth granularities, and frame times can be chosen statically. We currently do not have any implementations of this scheme to experiment with. Therefore we will use an NIA implementation under development that provides just two levels of pacing control [11] and study its effectiveness.

There are several issues that we are considering in order to make the choice of thread attributes for protocol processing. Each protocol requires an analysis of the communication subsystem components to identify the application and protocol threads, and the time they take to process a unit of data. Depending on the complexity of the protocol code, these times may be either constant or vary with the protocol state and runtime options. The estimation of these times can either be done by static analysis of protocol code instructions or by inserting probes during runtime. We are investigating how to estimate these times automatically. We will also measure system parameters such as context switching time and timer overheads in order to determine lower bounds on thread periods. These times may not be constant and may vary with system load. We will use these measurements to pick a value of B_p because it determines the thread period and thereby affects the startup delay. We have identified an implementation platform and we are deciding the set of target protocols for the four classes of applications. Some of the measurements that we expect to take are the following.

- *Guarantees*: The most obvious measurement is whether the QoS parameters specified at the communication subsystem boundary are being maintained. For example, in the *isochronous* case, we could measure end to end delay and see if it meets specified bounds. Similarly for the *burst* case, we could measure the application level throughput delivered. Through these measurements, we will study the effects of different loads, traffic mixes and QoS parameter values on the guarantees provided.
- *Variability*: One of the capabilities that we expect to have is the ability to change QoS requirements during runtime and see if the system tracks the changes. This can be measured for example by having a "dial" that the application can use to change its bandwidth requirement and measure the delivered throughput as before.
- *Loading*: Most measurements of network performance today are on unloaded systems to determine peak performance. However we would like to see steady performance under various load conditions. We expect to be able to measure as a function of total load the integrity of requested QoS parameters per application.

***Implementation*:** One of our goals is to incorporate the automatic mapping procedures into a communication subsystem and make it *protocol independent*. This suggests that a general purpose library be provided that can be used by protocol code to derive resource attributes from the QoS parameters provided to it. We expect the library routines to be able to derive scheduling attributes for the protocol and application threads, pacing attributes, and network connection attributes. The library will have a resource dependent set of routines that depend on the particular resource models available, and a set of resource independent routines that will be called by the protocol code during the setup phase. Using this we expect to validate our claim that protocol code need not be aware of platform dependent resource management issues.

3.6 Related Work

- *Specification* – QoS specification comprises two aspects namely, choice of parameters and choice of application classes.

  QoS Parameters: The specification of QoS has mostly been a datalink and network layer issue, and dealt with from the point of view of network service design. Because the basic service provided by the network is transmission and routing, the QoS parameters pertain to these services. Only recently has the need for supporting continuous media (CM), led to transport layer QoS specifications for individual applications [28, 6, 30]. However these parameters are mainly intended for specifying network requirements and not endsystem requirements, and in most cases directly map to network QoS parameters. As a result, the number of parameters are large and at a low level, and application users are typically unaware of what values to provide for each parameter. For example in [28] traffic source characteristics are specified with upto 11 quantities (*maximum data unit size*, *period, number of data units in a period* etc.), and QoS is separately specified with upto 5 quantities (*max delay*, *granularity of data error* etc.). Some traffic models such as *Linear bounded arrival processes*(LBAP) that are meant to describe application traffic have been used in several efforts [2, 3, 29, 42]. However, this model does not capture traffic types such as CM for example, and is not sufficient in itself.

  Application Types: Several application types have been considered to facilitate QoS specification. In [34] two types namely, *stream* and *message* are provided in order to support CM data traffic and message traffic. In [46] four classes of applications are considered — *unreliable real-time (video, audio), reliable real-time (process control), unreliable nonreal time (datagram) and reliable nonreal time* and suitable parameters are chosen to quantify requirements in each class. In general most work has considered guaranteed service to only predictable traffic such as CM data. Also there is a proliferation of types that are special cases of a general class. Therefore these choices are either not comprehensive, or they are too redundant. Almost none of them deal with a mix of requirements that includes both low latency and high bandwidth types.

- *Mapping* – The two main issues that have been addressed are the mapping methodology and its implementation.
  Methodology: Most of the work deals with mapping from QoS specifications to the network connection parameters. For example, translation from transport parameters to network connection bandwidth and delay parameters is shown in [28]. In [2] an economic cost model is used to partition application specified delay across the data path. However there no single solution for mapping QoS specifications for each class, to both local and network resources. Mapping to CPU scheduling parameters is shown in [34] for the *stream* and *message* application types. However, no specific thread model is used, and the parameters chosen do not ensure that QoS guarantees will be met. Mapping to the special case of a token ring network adaptor parameters is shown in [42].

  Implementation: In most cases, the mapping operations are done by a local server and is not part of the OS. The server also does an admissibility test to determine if the requested QoS an be provided. This is described in [29] where a QoS brokerage service is used to translate between application QoS and network QoS and an admission service is used to map between application and network parameters to local system parameters. The problem with this approach is that the server must have information about processing requirements of all supported protocols, application types and local resources. A separate server is unnecessary if mapping functions are part of each process and the process directly makes resource requests from the OS.

4. QoS Enforcement

We use the term QoS enforcement to refer to the policies and the mechanisms used by the communication subsystem for scheduling resources. The schedulable resources have been identified as the CPU, the NIA and the network connection. For the network and the NIA resources, the host communication subsystem is only involved in deriving and allocating the connection and pacing parameters. It does not enforce these requirements during operation because scheduling cell transmissions is done in the NIA and switch hardware and does not require OS intervention. The CPU is the only resource that is scheduled among threads by the OS. We view the scheduling of threads on the CPU as scheduling of batches or bursts of PDUs. We state the problems that arise in the scheduling of protocol threads on the CPU, mention key features of our solution, and then describe our implementation of the scheduling mechanism. The work in progress and a review of related work are also presented.

4.1 The CPU Scheduling Problem

The goal of scheduling is to ensure that the CPU is shared among threads in a manner that satisfies their requirements (expressed as thread attributes) while keeping the utilization high. Furthermore, to provide performance guarantees, It is necessary to have a *schedulability criterion* to decide if a new thread can be admitted into the system without violating requirements of existing threads. The problems that we face in achieving this are as follows.

1. *Pre-Emption Control*: We have adopted a real-time scheduling policy based on RM priorities in order to provide performance guarantees. The RM policy is pre-emptive 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 are 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. Therefore controlling pre-emption is important to attain efficiency.
2. *Improving Predictability*: For the RM scheme, there exist simple schedulability tests to decide if a set of threads are schedulable. Some tests also take into account the overheads due to timer handling and context switching. Since we have modified the basic RM scheme to reduce pre-emption, these tests cannot be used as they are. We therefore need to come up with sufficiency conditions that can be used to check if a set of threads are schedulable under our modified scheme.
3. *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) and forcibly terminate it after its time quota is over. Since a protocol thread must enqueue the PDUs at the NIA before terminating, the kernel must ensure that it does not terminate the thread before this happens. 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.

4.2 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. This means that every time it yields the processor, it has some PDUs that can be passed on to the NIA. The operations of enqueueing/dequeueing of PDUs at the driver as well as yielding the processor can be done in a single context switch.
- For the RM threads, we bound the number of pre-emptions that the thread can experience during each period. We then derive sufficiency conditions for the threads to be schedulable.
- 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. The protocol thread then yields the CPU by itself after it enqueues/dequeues PDUs at the driver.

4.3 CPU Scheduling Solutions in Detail

Delayed Pre-emption. The idea behind our proposed solution is that whenever a process (consisting of application and protocol threads) is scheduled, it runs until it generates/consumes an integral number of PDUs even though a higher priority process may become ready while it is running. The rationale behind this is that, if a thread at a sending host for instance 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. Similarly 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 specifies the number of PDUs in a batch 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.

The choice of b_p for a thread is determined by two conflicting criteria. In order to meet deadlines of higher rate threads, the time to process b_p PDUs should be a small percentage of the minimum thread period in the system. However, for efficient operation, it should be significantly larger than the time it takes to process a context switch and the associated network driver operations to enqueue/dequeue PDUs. If $b_p = B_p/f$ then during a single invocation (i.e thread period) a thread can be pre-empted at most f times.

Schedulability Testing. 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 RM policy. Therefore an admission control mechanism can be built into the scheduler that allows predictable performance.

Tracking CPU Usage . Our solution to limit the number of scheduling context switches is to process some minimum number of PDUs every time a thread is run. In order for a thread to keep track of the work done so far it is more appropriate to use as a measure of usage the number of PDUs processed rather the time for which it has been running. The reasons why this is useful are

- The processing requirements of the protocol thread are already expressed in terms of the number of PDUs that must be generated or consumed per invocation (the batch count derived in the mapping operation). The same holds for the application thread that processes ADUs. If the measure of CPU usage is the number of PDUs processed so far, it is trivial for the protocol threads to keep track of their usage. The processing time estimated by the mapping process is therefore used only for the purposes of evaluating the schedulability criterion.
- The fact that the protocol thread can keep track of its usage, allows it to combine in a single system call, the operation of moving PDUs across the user kernel boundary as well as the operation of yielding the CPU after it has processed a batch of PDUs. This eliminates the need for separate system calls needed for passing PDUs between the user process and the driver. It also makes it unnecessary for the scheduler to intervene to terminate the thread once its requested time is consumed.

The three techniques described above complement each other and can be implemented using a single integrated mechanism. An efficient implementation of this scheme that keeps kernel intervention minimum is described below.

4.4 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 (network driver). We view the network driver as also implementing a thread scheduling policy because thread scheduling and burst scheduling are closely related. For example, the scheduling component of the network driver must know if any PDUs have been received on a connection in order to make scheduling decisions. More importantly, the scheduling component must be able to inform a running thread whether a higher priority thread is waiting for the CPU and it must be able to determine how many PDUs have been processed by the currently running thread.

Our implementation uses a data structure in *shared memory* between each user protocol thread and kernel to maintain control information about scheduling. The kernel writes into shared memory in order to inform a running thread if a higher priority thread is ready. The protocol thread uses the shared memory to inform the kernel about how many PDUs it has processed since it was last scheduled on the CPU. If a higher priority thread becomes ready before the thread can finish processing one batch (B_p), then it yields the CPU provided it has at least completed processing of b_p PDUs in its current run.

The description above makes some assumptions about the way the application and protocol threads schedule packet processing. Our description suggests that at a sending host for example, the application and protocol threads make “vertical traces” through the code starting at the application and protocol layers, and ending at the network driver interface, before going back to the application layer. In each trace, the application and protocol threads process user data corresponding to one PDU before going back to prepare the next one. After completing each PDU, the protocol thread updates the count field in the data structure, and checks if the kernel has indicated if a higher priority thread is ready. If it detects this condition, then it continues until it has generated b_p PDUs. It then calls into the driver to yield the CPU as well as enqueue the b_p PDUs. However if no higher priority thread is waiting, it increments the count field and goes back for the next PDU. The receiving protocol thread is similarly structured. The pseudo code for a sending protocol thread is given in Appendix B.4.

4.5 Work in Progress

We are working on the following problems. The first is the analysis of the modified RM priority scheme and simulating it with a realistic task set to quantify the expected reduction in context switching overhead. The second is an implementation on a target platform, and to measure the gain in performance.

Analysis and Simulation. We have analyzed the modified RM scheme and derived conditions under which a set of tasks are schedulable under this scheme. The result gives an upper bound on the amount by which the CPU must be “over reserved” in order to guarantee schedulability. We have also obtained an upper bound on the number of context switches in this scheme. We expect average case performance to be much better. This can be demonstrated only by simulation of various task sets. Simulation will also reveal the amount of concurrency in the operation of the NIA and the CPU that can be obtained by enqueueing PDUs generated at the NIA whenever a context switch is made. Expected results are

- A tighter analytical bound on the amount by which the processor must be over-reserved.
- Comparison of the number of context switches in the two schemes as a function of different realistic task sets.
- Comparison of the benefits of the increase in concurrency in operation of the CPU and the NIA for different task sets.

Implementation. Our ideas for delayed preemption and usage tracking based on PDU count have to be evaluated in a practical setting. These features must therefor be incorporated into a standard RM scheduler implementation. We also would like to compare performance with existing RM and EDF schemes currently in use. Usage tracking based on PDU count has performance and fairness issues that can be addressed only in an implementation. A performance issue is whether protocols that generate a high percentage of control PDUs or variable length PDUs will have wide variability in the obtained QoS. We expect to also have a solution for the potential fairness problem that could arise if a thread takes more than its share of CPU time by keeping its PDU count to be less than what it is. The different steps in our implementation are outlined below.

- Shared Memory Implementation: We will implement the proposed shared memory mechanism between user process and the kernel in order to store the data structures needed for scheduling.
- Scheduling Optimizations: We will implement the changes to the RM scheme as an external *scheduling policy*. This feature is supported by an OS such as Mach 3.0 which also supports the RM scheduling discipline.
- Evaluation: We expect an improvement in performance compared to a simple RM scheme specially at higher loads. We will measure this improvement by running both schemes under differing load conditions and measuring the QoS parameters as before.

4.6 Related Work

CPU Enforcement. QoS enforcement issues specially CPU scheduling has been dealt with extensively in the context of multimedia. We describe the related work according to the choice of priority, the implementation mechanisms, and the ability to provide guarantees.

- *Priority Scheme*: The priority scheme commonly used is either RM, EDF or a mix of the two. The use of EDF is described in [17, 34,42] in which a logical deadline is associated with each message and the process that has the earliest deadline is run. There are several problems with these approaches. Firstly, a context switch could potentially be associated with each message. Also, messages in different processes are often processed serially and this is unsuitable for use with an ATM like network in which it is preferable to cycle through each connection sending small amounts of data from each message. The EDF scheme requires that user processes must keep track of time and calculate deadlines for each message which could be a cumbersome procedure and could involve context switching. The choice of RM is made when periodic threads are used [26]. However the need to reduce the context switching costs in RM is not addressed. Policing of usage in each period is done using the *reserve* mechanism. However, the relationship between the reserve, and the amount of protocol processing time required is not easy to estimate and could vary for each period. Therefore the merits of tracking usage using reserves is not clear. In addition, the need for integrating the thread scheduling policy and the scheduling of transmission of bursts is ignored in all these works.
- *Implementation Mechanism*: [17] proposes an implementation policy called *split-level* scheduling. To ensure that the thread with the globally highest priority is run, the user level scheduler function has access to the priority of all threads in memory shared with the kernel. If the user scheduler detects that the running process does not have the earliest deadline, it yields the CPU. If this condition is detected by the kernel, the running process is pre-empted. However, the use of asynchronous pre-emption is inefficient if data has to passed on to the NIA for transmission. The implementation also ignores the context switches needed to pass data to the network interface. In addition, there is no suggestions for reducing the number of context switches between different processes. Implementation issues are largely ignored in the other solutions that we have described.
- *Providing Guarantees*: None of the enforcement solutions above take into account context switching time in the schedulability tests. Potential overload conditions could result specially if there is no control over pre-emptions. In [29] there is need for two tests–a *local* schedulability test, and a *global* schedulability test of the transport processing in the kernel. Our solution makes this unnecessary by placing all protocol processing in a single domain.

Our solution rectifies many of the above problems in scheduling protocol processing operations. It is specifically designed for integrating protocol processing with NIA operation. Accordingly, the scheduling policy, the usage tracking mechanism, and the cooperation between user threads and the kernel scheduler are integrated with network interface operations such as data movement and event handling. We expect that this integration will result in significant performance gains over other implementations.

5. Protocol Implementation Model

In this section we examine the issues relating to our choice of the protocol implementation model (PIM). The PIM has a significant effect on performance because it determines the costs due to data movement across address spaces and between devices in the communication path, and context switches associated with movement of data. We first describe the main features of the protocol implementation model we have adopted and the rationale behind our choice of the model. We state the problems that we shall address, especially those that arise due to the mechanisms we plan to use in order to provide performance guarantees in the endsystems. We will then describe our proposed solutions and how they will be implemented. We conclude with a discussion of trade-offs and a review of other solutions that have been reported.

5.1 Application Level Protocol Implementation Model

In current state-of-the-art implementations, the communication subsystem is implemented in the kernel and is accessed by user processes through a high level IPC interface. This model is referred to as the kernel resident protocol (KRP) model. In this model, the kernel maintains state for all protocol sessions in protocol control blocks (PCBs), schedules protocol operations for all sessions as well as handles network events for all sessions. The kernel coordinates network I/O for all processes which involves moving data across protection domains and process scheduling.

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 the need for the kernel to demultiplex data has been removed. Furthermore, protocol processing can be done 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. 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 and it has the flexibility that comes with user level implementations.

The main implication of the ALP model on providing QoS guarantees for protocol processing is that by controlling the resource allocation to each process (or thread), we can control the QoS offered to each protocol session. This cannot be done quite as easily in the KRP model because there is no readily available accounting mechanism for resources that the kernel uses.

5.1.1 Key Features of ALP Implementations

We list some of the common architectural features found in ALP implementations. The architectural components may be realized with different implementation techniques depending on the system. Relevant features are –

1. *Separation of Protocol Services*: A communication session consists of a setup phase, a data transfer phase and a release phase. These operations are supported by separate protocol services. The setup and release phases occur at the “boundary” of a session, and involve operations such as resource reservations, name resolution and negotiation with service provider. Therefore these control operations need to be implemented by a trusted server. The server also maintains state information that is shared by all sessions such as routing tables, name to address mappings and so forth. Other than the setup and release phases, the protocol services related to data transfer are part of each application process and mainly determine performance. This separation of protocol services is also called protocol decomposition.
2. *Logical Protocol Layers*: As a result of protocol decomposition, a protocol layer is not confined to a single address space such as the kernel, but exists in multiple processes. A transport protocol layer that implements the data transfer service could be active in many processes simultaneously. Therefore such a layer is referred to as a logical protocol layer. This is in contrast with the KRP model in which a layer can be readily identified as a set of code modules and its data structures (PCBs) that is part of the kernel.

Since we are concerned with providing QoS for data transfer, we only discuss the protocol processing issues relating to this. However in order to evaluate our solutions we will need to implement other aspects of ALP model as well.

5.2 Problem Statement

It is recognized that a major portion of protocol processing time is spent in moving data. For example, in the KRP model, data has to be moved between the user process and the kernel and between the host memory and the NIA [22]. In addition to memory access cost, moving data across protection boundaries involves context switches. Similarly asynchronous interrupts cause context switches and affect the protocol performance. As mentioned before, the cost of these operations depends on the implementation model.

For the ALP model, we discuss when these operations occur and the problems they pose for performance. We will draw comparisons with the KRP model because this is the state-of-the-art, and our objective is to achieve at least as good performance as existing implementations. The operations are the following.

1. *Data Passing*: In the ALP model the data units that move across the user–kernel boundary are PDUs, whereas in the KRP model the units of data are ADUs. Because an ADU is broken up into several PDUs, moving PDUs one at a time between user and kernel using system calls will be very inefficient because of the context switching involved. Therefore, the existing mechanism used to pass PDUs across the user–kernel boundary will be inefficient.
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 NIA can access it. On the receive side, data received into kernel buffers is mapped to user buffers. Because the cost of page mapping operations can be high depending upon the system mapping data pages for every send/receive operation for a PDU is not efficient.
3. *Asynchronous Event Processing*: Processing of asynchronous protocol events such as packet arrival and timer expiry are usually triggered by interrupts. However event driven processing conflicts with the processing order dictated by the priority based scheduling policy that we have adopted for maintaining QoS. In addition, interrupts cause context switches and so the problem of how to handle asynchronous protocol events with minimal interrupt overhead needs to be addressed.
4. *Control Operations*: In the ALP model all protocol processing is done in user space and therefore protocol operations that require kernel capability have to be done using system calls. Because system calls are expensive, ALP implementations are put at a disadvantage compared to KRP 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 avoided in the KRP model because the kernel maintains the buffers. This extra data movement could lead to poorer performance for ALPs.
   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 have to be made to do 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. Doing timer operations and handling timer interrupts efficiently in the ALP model is thus an important problem.
   c) Page Remapping: A protocol that receives misordered data needs to do page remapping to make the data contiguous. Misordering happens when a PDU is lost and its successor is written in its place by the NIA. If the protocol is in the user space, system calls will be necessary to do this. Because system calls are expensive, system support for page remapping is necessary.

A solution to these problems is necessary to make ALP implementations compare favorably to KRP implementations. These solutions are outlined in the next section.

5.3 Solution Outline

- We use data structures in shared memory between each process and the driver to share information about outgoing and incoming PDUs. As mentioned in Section 4.3, a single system call is made by a sending protocol thread to combine the operations of yielding the CPU as well as enqueueing PDUs for transmission. Control information about these PDUs are written in fields in the shared data structures and organized as a linked list. Similarly, the driver creates a linked list of such data structures that describe incoming PDUs which are examined by the protocol thread when it runs. This drastically reduces the use of system calls to pass PDUs between a protocol thread and kernel.
- 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. This eliminates the need to remap pages between user and kernel on send

1. The data has to be page aligned for mapping

and receive operations. With DMA capability of the NIA, this mechanism can potentially achieve *zero-copy* semantics.

- We avoid generating interrupts for asynchronous events and instead rely on the thread scheduler to dispatch the thread that processes events. The scheduling of threads is done according to the attributes derived from QoS specifications by the mapping operation. We therefore describe our style as *specification driven* rather than *event driven*. This approach has the added 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.

We discuss important aspects of these solutions below.

5.3.1 PDU Movement without System Calls

As mentioned earlier, a sending protocol thread prepares each PDU in buffers in shared memory. Similarly, the NIA writes (using DMA) incoming PDUs into free pages in this shared memory. Since a shared memory mechanism is already being used for scheduling (as described in Section 4.3) the same mechanism can be 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* and the details are given in Appendix B.2. Using shared memory for handshaking 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. Similarly, 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 to be processed when the control is given to the network driver and the protocol thread respectively. 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.

5.3.2 Data Buffering

As we have seen, shared memory is used to implement a command and response mechanism that reduces system call overhead. We use the same shared memory to implement a buffering mechanism so as to avoid copying and/or page remapping. We describe the buffering mechanism and its effect on send and receive operations.

Send Side. The protocol thread allocates pages in shared memory that it will use to store the headers and the data of packets. Data units such as ADUs and PDUs can be represented using pairs of {*virtual_address*, *length*} values because the application and protocol are in the same address space. Because the memory is also mapped to the network driver, the buffers need not be remapped or copied when pointers to PDUs are passed through the descriptors. If the kernel needs to access the data, it can convert the virtual addresses specified by the process to virtual addresses in its address space because it has access to the page tables. Typically, the driver does not need to touch user data and merely translates PDU buffer addresses to physical addresses that must be supplied to the NIA. The buffering system as described above, allows *zero-copy* semantics.

Receive Side. The receive buffers are also pre-allocated and enqueued at the NIA as a linked list of page frames for each connection. The NIA writes data into these free pages and fills in the PDU descriptors as described before. When the receive side protocol thread is run, the driver translates these addresses to virtual addresses in the process address space. As before PDUs and ADUs can be represented using {*pointer*, *length*} pairs. If the NIA can separate headers and data into different buffers, then contiguity of user data can also be ensured. Thus no further remapping or copying operations are necessary in most cases where network errors do not occur.

5.3.3 Interrupt Free Receive Operation

Currently interrupts are used to trigger processing of incoming packets. However the bandwidth guarantees provided by the connection and the periodic nature of processing at the sending host ensures that data arrives at the receiver in a predictable manner. Therefore the processing of this data at the receiver can also be made predictable and periodic. Interrupt free operation can be achieved by having the periodic thread mechanism of the scheduler to schedule the thread that processes the PDUs enqueued in memory by the NIA. Of course an interrupt may be needed at the beginning of a burst in order to activate the receiving thread. The important difference is that 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. The checking is done to avoid running a thread that has no data to process.

5.3.4 Support for Control Operations in Protocol Processing

We describe the support that we provide for retransmission, timer operations and page remapping. The common idea behind all three solutions is that a protocol thread delays operations that need kernel support until it reaches the point where it has done protocol processing and must make a system call to yield the CPU. Similarly the driver delays notifying the protocol thread about asynchronous events until the thread gets scheduled. Because most operations and events occur on a per PDU basis, the PDU descriptor can be used to share the control information associated with the system call or the event. These solutions are akin to the batching idea that we use to reduce context switching overhead but applied to protocol control operations. The details of the solutions are in Appendix B.3

5.4 Work in Progress

Our first objective is to implement our prototype protocol suite according to the ALP implementation model. The second is to measure improvement in performance obtained over current implementations using the optimizations described earlier. We will need to study the memory management and timer mechanism provided by the target OS in order to make the proposed changes in data buffering and event handling. The implementation plan is fairly clear from earlier descriptions and will include the following.

- The buffer management mechanism to achieve zero-copy send and receive operations. We have to integrate this with the shared memory mechanism that we plan to implement for scheduling protocol threads.
- Implement the PDU descriptor operations in the driver and in the protocol code
- Implement support for protocol control operations such as retransmission and page remapping.

Evaluation will be done by measuring best case performance of the protocol implementation without any other load on the system. For example, raw throughput on a connection will be measured at each layer such as at the NIA, the driver, the user to driver boundary, and at the protocol thread level. These measurements will be made as a function of data unit size. Similarly delay introduced by each layer is also measured as a function of data unit size.

The other implementation idea that we will evaluate is the use of specification driven scheduling of protocol events as opposed to interrupt driven scheduling. We will quantify benefits of our approach by measuring the delay in handling events and comparing it with the interrupt driven case.

The previous set of measurements indicate raw aggregate performance and can be used to compare our implementation with existing kernel resident implementations. However to see if QoS guarantees are met on a per application basis we will measure performance of each application by varying the job mix as mentioned in Section 3.5.

5.5 Related Work

Most of the work in reducing data movement and context switching overheads has been done for in kernel protocol implementations and targeted towards improving TCP/IP performance. We will describe these solutions in order to bring out when and where data movement and context switching occur, and how they are handled. This will help understand the rationale behind many of the techniques we use in our solution. We look at various operations and the costs associated with them

1. *Data movement across protection domains*: Integrated solutions to reduce data movement that make use specialized hardware have been proposed in [37]. For general purpose hardware and generic OS (such as UNIX) software solutions have been proposed. The *x-kernel* [20, 32] is an architecture that addresses the issue of moving data between domains among others. It uses a globally unique data structure [13] to avoid converting data from one type to another when it crosses a domain. It also uses efficient cross domain data transfer [12] with minimal context switch overhead. An implementation that reduces the number of times data is "touched" as it moves between process buffers and the network is described in [4, 8]. Here buffers are located in the network adaptor with checksum support in hardware. This reduces the number of data touches involved in data transfer from five to two. Direct movement of data between user buffers and the network adaptor is described in [36].
2. *Avoiding System Calls*: The segment streaming approach [15] has been proposed to reduce the system call overhead involved in stream IPC. Using shared circular buffers between the user process and the kernel to pass buffer location information, and to minimize physical copying in data transfer is proposed in [28]. System calls to schedule transmission are avoided altogether by having the transport system poll control fields in the shared structure. The memory mapped streams mechanism achieves a similar objective [17]. Upcalls are used in [34] for initiating data transfer in an interrupt driven manner.
3. *Interrupt Handling*: A related issue is minimizing interrupt overhead. Clocked interrupts [36] is a technique that uses a mixture of polling and interrupt driven operation. Similar solutions such as interrupting on a burst basis rather than for each packet have been proposed in [11].

User level protocol implementations are fairly recent and most implementations are more expensive in terms of data movement and context switching. They however have the potential for performing well in a connection oriented network environment such as ATM because data can be demultiplexed directly to application space. In [39] a user level implementation of TCP/IP is described that achieves performance close to kernel level implementation. Some important techniques (which our solution shares) are the use of shared memory between user and kernel to avoid copying data, as well as batching PDUs to reduce context switching cost. However, our design includes several performance enhancements over this. A similar implementation is described in [24] where the user level protocol implementation maintains complete UNIX semantics by using a combination of user level and kernel level protocols.

6. Example

In this section we take the example of a CM application for video. We show all the steps involved and illustrate the mechanisms. For the example, we assume that network connection attributes are specified as in [14]. The OS environment is assumed to be MACH with support for real-time threads. We assume an NIA that implements HRR with parameters described in Section 3.4.

Application QoS Specification. We assume a video application where each frame is 225 KByte and the compression is such that S_{max} = 11.2 KB and S_{avg} = 5.2 KB. Therefore the average bandwidth B_{req} = 1.26 Mbps. We assume delay tolerance D_{req} = 100 msec (3 frame periods). This example corresponds to a full motion video window of size 320x240, with 24 bits per pixel, with intraframe compression ratio of 20:1 and overall compression ratio of 50:1.

Delay Partitioning. Of the end-to-end delay of three periods let us assume 1 period (33 msec) has been assigned to latency, and 2 periods (66 msec) assigned to smoothing interval. We divide the latency equally between the sending host, the network and the receiving host. We describe next how to derive the characteristics of the network connection.

Network Connection Characteristics. To setup a connection the average bandwidth and peak bandwidth must be specified in terms of the average and minimum interval between successive cells x_{ave} and x_{min} respectively. Since the processing delay bound (66 msec) is equal to two periods, we can assume that two frames N and N+1 generated at the sender need to be made available at the receiver after 2 periods have elapsed since the beginning of frame N at the receiver.

We assume that maximum size frames do not occur more than once within the smoothing interval. In this case we assume that in two consecutive periods the maximum amount of data that can be generated in a smoothing interval is $S_{max} + S_{avg} = 11.2 + 5.2 = 16.4$ KB. To keep the smoothing delay equal to two periods, the maximum data generated in the smoothing interval must be sent out by the end of the interval. This means that 16.4 KB must be sent in 67 msec, and since each ATM cell has 48 bytes, the minimum inter cell gap x_{min} is 0.196 msec (giving a peak bandwidth of close to 2 Mbps). The average inter-cell gap x_{ave} is simply $S_{avg}/48 = 0.3$ msec. The cell delay bound can be specified as 10 msec as determined by the delay partitioning.

Pacing Parameters. We assume the NIA has a HRR scheme and so we need to specify the level and the number of slots. We assume a link rate of 155Mbps and choose 3 levels. We choose level 1 to have n_1=35, b_1=5 and a frame duration of 100 microseconds. For level 2, we choose n_2=127, b_2=36 which gives a frame time of 415 microseconds and a minimum bandwidth of 1.024 Mbps. The remaining 25% of the bandwidth is assigned to level 3 with a frame time of 42.4 milliseconds and a minimum bandwidth of 10Kbps.

The bandwidth required in our example is twice the level 2 bandwidth. We therefore assign it to level 2 and choose the number of slots (a_j) as 2 cells. The delay bound of 415 microseconds is also within the limits of the startup delay.

Sender Thread Attributes. We assume that there is a single real-time kernel thread over which we run the application and protocol thread as user level threads. If we choose an ADU size of 2 KB and a PDU size of 0.5 KB, then an average size frame has 3 ADUs. If we process one ADU (i.e 4 PDUs) per batch, the period of the protocol thread is $P_p = (4 \cdot 512/48) \cdot 0.196 = 8.36$ milliseconds.

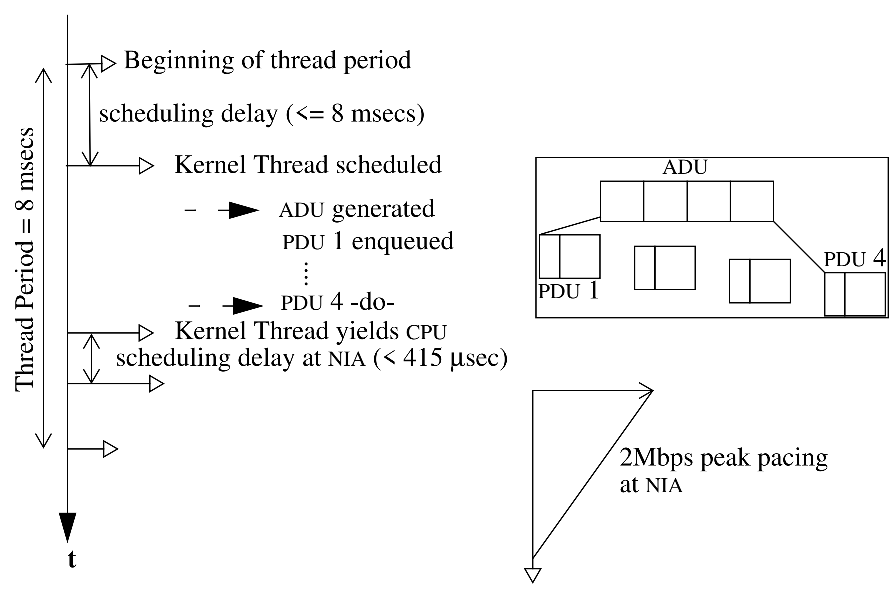

FIGURE 8. Send Side Timing

If this is chosen as the period of the real-time kernel thread, then every 8 msecs the kernel thread is run, the application thread does the processing associated with preparing one ADU for transmission, and passes it through the IPC interface to the protocol thread. The protocol thread prepares 4 PDUs corresponding to this ADU and queues them at the NIA. The protocol and application thread use the user thread scheduling primitives to exercise control over how they run over the single kernel thread. The maximum scheduling delay is the period (8 msecs) and this is below the tolerable bound of 10 msecs at the sender.

7. Conclusions

We have presented a design for providing efficient quality of service support for applications. We have also described the work in progress in order to implement the various solutions proposed. We expect that the work we have outlined here will improve upon existing solutions to provide end-to-end QoS support. We also expect to improve the state-of-the-art in the area of application level protocols. We hope to gain valuable experience and insight on the performance of real applications by implementing them over our prototype network.

References.

[1] Abbot, M.B., Peterson, L.L., "Increasing Network Throughput by Integrating Protocol Layers," *IEEE/ACM Transactions on Networking*, Vol.1, No.5, October 1993, pp. 600-610.

[2] Anderson, D.P., "Metascheduling for Continuous Media," *ACM Transactions on Computer Systems,* Vol. 11, No. 3, August 1993, pp. 226-252.

[3] Anderson, D. P., Govindan, R., Homsy, G., Wahbe, R., "Integrated digital continuous media: A framework based on MACH, X11 and TCP/IP,"

[4] 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.

[5] Blair, G.S. et al., "The role of operating systems in object oriented distributed multimedia platforms," Technical Report MPG-92-51, 1991, Distributed Multimedia Research Group, Lancaster university, Lancaster U.K.

[6] 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.

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

[8] Dalton, C. et. al., "Afterburner," *IEEE Network*, July 1993, pp. 36-43.

[9] Davie, B.S., "The Architecture and Implementation of a High-Speed Host Interface," *IEEE Journal on Selected Areas in Communications,* Vol. 11, No.2, Feb 1993.

[10] "The DIS Vision – A Map to the Future of Distributed Simulation," prepared by the DIS Steering Committee, October 1993.

[11] Dittia, Z.D., Cox, J.R., Parulkar, G.M., "Catching up with Networks: Host I/O at Gigabit Rates," Technical Report WUCS-TR-94-11, Dept. of Computer Science, Washington University in St.Louis.

[12] Druschel, P., Peterson, L.L., "High-performance cross-domain data transfer," Technical Report TR 92-11, Dept. of Computer Science, The University of Arizona, Tucson. March 1992.

[13] Druschel, P., Abbot, M.B., Pagels, M.A., Peterson, L.L., "Network subsystem design," *IEEE Network*, July 1993, pp. 8-17.

[14] 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.*

[15] Gong, F., Parulkar, G.M., "Segment Streaming for Efficient Pipelined Televisualization," *Proceedings of the IEEE Military Communications Conference,* MILCOMM 1992.

[16] Gong, F., Parulkar, G.M., "An Application Oriented Error Control Scheme for a High Speed Transport Protocol," Technical Report WUCS-92-37, Dept. of Computer Science, Washington university in St.Louis. Submitted for publication.

[17] Govindan, R., Anderson, D. P., "Scheduling and IPC mechanisms for continuous media," *13th ACM Symposium on Operating Systems Principles,* 1991.

[18] Hayter, M., McAuley, D., "The desk area network," *ACM Operating Systems Review,* Oct 1990.

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

[20] Hutchinson, N. C., Peterson, L. L., "The *x*-kernel: An architecture for implementing network protocols," *IEEE Transactions on Software Engineering,* Vol. 17, Jan 1991, pp.64-76.

[21] Kalmanek, C.R., Kanakia, H., Keshav, S., "Rate Controlled Servers for Very High-Speed Networks," *IEEE Global Telecommunications Conference GLOBECOMM,* San Diego, December 1990.

[22] Leffler, S.J. et. al., "The design and implementation of the 4.3BSD Unix operating system," Addison-Wesley publication.

[23] Liu, C.L., Layland, J.W., "Scheduling Algorithms for multiprogramming in a Hard-Real-time Environment," *Journal of the Association for Computing Machinery*, Vol. 20, No. 1, January 1973, pp.46-61.

[24] Maeda, C., Bershad, B. N., "Protocol service decomposition for high performance networking," *14th ACM Symposium on Operating Systems Principles,* Dec 1993. pp.244-255.

[25] Massachusetts Institute of Technology, "*Telemedia, Networks and Systems group Annual Report,"* July 1991 - June 1992*,* AR-001, Massachusetts Institute of Technology.

[26] 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.

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

[28] 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, Apr 1992.

[29] Nahrstedt, K., "Network Service Customization: End-Point Perspective (Proposal)," Technical report MS-CIS-93-100, University of Pennsylvania, Computer and Information Sciences Department, December 1993.

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

[31] Neufeld, G.W. et al. Parallel host interface for an ATM network. *IEEE Network,* July 1993, pp. 24-34.

[32] O'Malley, S. W., Peterson, L. L. A Dynamic Network Architecture. *ACM Transactions on Computer Systems.* Vol. 10, No. 2, May 1992, pg. 110-143.

[33] 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.

[34] 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.

[35] Schmidt, D. C., Suda, T., “Transport system architecture services for high-performance communications systems,” *IEEE Journal on Selected Areas in Communications*, Vol. 11, No.4, May 1993, pg. 489-506.

[36] Smith, J.M., Traw, C.Brendan S., “Giving Applications Access to Gb/s Networking,” *IEEE Network,* July 1993.

[37] 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.

[38] Thekkath, C. A., Levy, H. M., “Limits to low latency communication on high-speed networks,” *ACM Transactions on Computer Systems*, Vol. 11, No. 2, May 1993, pg. 179-203.

[39] 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.

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

[41] Turner, J.S., “New Directions in Communications,” *IEEE Communications Magazine*, Vol. 24, No. 10, Oct 1986, pp. 8-15.

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

[43] Wolman, A., Voelker, G., Thekkath, C.A., “Latency Analysis of TCP on an ATM network,” Technical Report 93-03-03, Department of Computer Science & Engineering, University of Washington, Seattle, March 1993.

[44] Wright, D.J., To, M., “Telecommunication Applications of the 1990s and their Transport Requirements,” *IEEE Network Magazine,* March 1990, pp. 34-40.

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

[46] Zitterbart, M., Stiller, B., Tantawy, A. N., “A model for flexible high-performance communication subsystems,” *IEEE Journal on Selected Areas in Communications*, Vol. 11, No.4, May 1993, pg. 507-518.

APPENDIX A: QoS Mapping Details

A.1 Deriving the Bandwidth Scaling Factor

In Section 3.4.2 we derive from the QoS parameters of an application, the number of bytes of user data that must leave the sending host during the smoothing (or averaging) interval. Due to header overheads, the actual number of bytes sent at the data link layer is higher. To estimate this we consider the simple example of a sender with an application thread and a protocol thread for a connection as shown in Figure 9. We assume that S_{in}^A is the number of bytes of user data that need to be sent in a fixed interval.

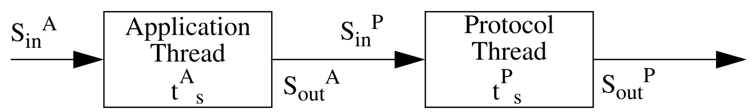

FIGURE 9. Module Requirements

Let S^A, H^A, S^P, H^P be the sizes of the ADU and its header, and the PDU and its header respectively. Because of the header that must be attached to each ADU, the number of bytes the application thread generates in one period has to be scaled up and is given by.

$$S_{out}^A = \left(1 + H^A / S^A\right) \cdot S_{in}^A$$

The data generated by the application layer must be sent out by the protocol layer in the same amount of time. Therefore we have a multiplying factor due to protocol header overhead, and the number of bytes is given by.

$$S_{out}^P = \left(1 + H^P / S^P\right) \times S_{in}^P$$

Therefore the final multiplying factor S_f is a product of the factors derived above and is given by

$$S_f = \left(1 + H^A / S^A\right) \cdot \left(1 + H^P / S^P\right)$$

A.2 Deriving Network Connection Bandwidth Parameters

In Section 3.4.1 we derive from the delay QoS parameter specified by an application, the time allotted for sending data generated during the smoothing interval t_{sm}. In order to determine the peak bandwidth we must also determine the maximum amount of data S_{trans} that can be generated by the sender during the smoothing interval. In order to meet the specified delay bound, S_{trans} bytes of data must be sent out in time t_{sm}. These two quantities allow us to calculate the peak bandwidth requirement. After scaling S_{trans} by the scaling factor derived above, we get the number of cells that must be sent out per second as,

$$(S_{trans} \cdot S_f / 48) / t_{sm}$$

Here the cell payload is assumed to be 48 bytes. The peak bandwidth expressed as the minimum spacing between cells x_{min} is given by,

$$x_{min} = \frac{t_{sm}}{(S_{trans} \cdot S_f) / 48}$$

For calculating the value of x_{avg}, we need to know the average bandwidth. For the *burst* application type, this is not known and so the value of x_{avg} is set to x_{min}. For the *isochronous* case, we use the value S_{avg} specified for average frame size in the QoS parameter. This means that S_{avg} number of bytes of frame data must be sent out in each period T. Therefore we have

$$x_{avg} = \frac{T}{(S_{avg} \cdot S_f)/48}$$

A.3 Deriving Thread Attributes

In Section 3.4.4 we present the tradeoffs involved in the choice of the thread attributes. The period of a thread depends on how much time it has to process a data unit (such as a PDU). This time depends on the peak rate at which PDUs are sent (or received) on the network connection. If a batch of PDUs need to be processed per period, then the period gets multiplied by the batch factor. Given these quantities, we show how thread attributes are derived.

1. *Isochronous*: We begin with the protocol thread. If B_p is the batch factor or number of PDUs prepared by the sending protocol thread per period, each of size S_p, then the period P_p is the time taken by the NIA to send these PDUs. This time is given by the following expression.

   $$P_p = (B_p \times S_p) \cdot x_{min}/48$$

   and is the time to transmit a batch with 48 being the cell size. The execution time required by the thread is the processing time required to prepare B_p PDUs. Let t_p be the processing time per PDU. The execution time C_p for the protocol thread within each period is given by $C_p = B_p \times t_p$.

   The same calculations can be done for deriving the application thread attributes. Because typical ADU size is larger than the PDU, the batching factor B_a for the application thread is less than for the protocol thread. Therefore if C_a is the execution time for the application thread and t_a is the time to process an ADU we have,

   $$P_a = (B_a \times S_a) \cdot x_{min}/48, \quad C_a = B_a \times t_a$$

   Typically P_a is greater than or equal to P_p and the batching factors could be chosen so that B_p is some multiple of the number of PDUs in a ADU.

2. *Burst*: The analysis is exactly same as above

3. *Message Rate Bounded*: The only difference between this case and the previous ones is that the batching factor is same for both the application and protocol threads. This is because each ADU corresponds to a single PDU. If we represent the number of messages per batch as B, and the requested message rate as R_{msg} then the period P is given by $P = B/R_{msg}$. The value of C is the time needed to do both application and protocol processing of a single batch of ADU*s* and PDUs by the application and protocol threads respectively.

APPENDIX B: Scheduling and Protocol Support

B.1 Data Structure for Scheduling

In Section 4.4 the use of a shared data structure between user and kernel was proposed to control the scheduling of periodic threads. The idea behind the scheme is that the protocol thread polls the data structure to see if a higher priority thread is ready and waiting. Similarly the kernel looks at the structure when a thread with priority higher than the running thread becomes ready, to see the number of PDUs generated so far. The data structure is explained in Figure 10 The protocol thread uses a *pdu_count* field in the structure to indicate how many PDUs have been processed so far and the kernel uses a *yield_request* field to indicate if a higher priority thread has become ready. These fields are used as follows—

pdu_count
yield_request
B_p
b_p

FIGURE 10. Shared Structure for Scheduling

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 set to 0 when the thread is run. Whenever a protocol thread, at a sending host for example, generates a PDU it increments this count. When a higher priority thread becomes ready, the scheduler checks this field value. If the value is greater than b_p it takes the set of PDUs processed so far and enqueues them at the NIA before it 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. It then resumes the thread. The thread checks this field every time it completes processing a PDU and yields the CPU by calling into the driver function that enqueues/dequeues PDUs when it has completed b_p PDUs.

B.2 The PDU Descriptor Structure

In Section 5.3.1 a mechanism to initiate sending/receiving of PDUs without explicit system calls is proposed. The idea

```
struct PDU_descriptor {
   struct buf_list {
      char *buffer;
      struct buf_list *next;
   };
   int command_flags;
   int command_words[NCMD];
   int status_flags;
   int status_words[NSTS];
   struct PDU_descriptor *next;
};
```

FIGURE 11. PDU Descriptor

is to use a data structure called the PDU descriptor in shared memory that contains information such as buffer addresses, commands and arguments that request the driver to perform various operations on a PDU, and fields to store the results of these operations. Figure 11 shows the fields in the PDU descriptor data structure. The descriptor is a structure in shared memory that is used to pass control information about PDUs between the protocol threads and the network driver. It can be viewed as parameters in a system call that passes data to the kernel. The `buf_list` field is a linked list of buffer addresses that contain the PDU. `Command_flags` are used by the protocol thread 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 placed in the `command_words` field. Similarly the driver can set the `status_flags` to indicate the status of various operations that it performed on the PDU. The `next` field is used to link PDU descriptors. 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. The `buf_list` field is a linked list of virtual addresses of buffers that hold the PDU. When the driver gets control (see Section 4.3), it converts these addresses to physical addresses and enqueues the descriptor at the NIA. If more than one PDU is generated per period, the protocol descriptors for these are formed into a linked list and enqueued by the driver. The NIA traverses the descriptor list, and for each PDU paces out data from its buffers without involving the CPU. After sending out a PDU completely, it sets a bit in the `status_flags` field 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. The `command_flags` field is used to implement control functions relating to sending PDUs. Figure 12 shows the organization of the data structure in memory.

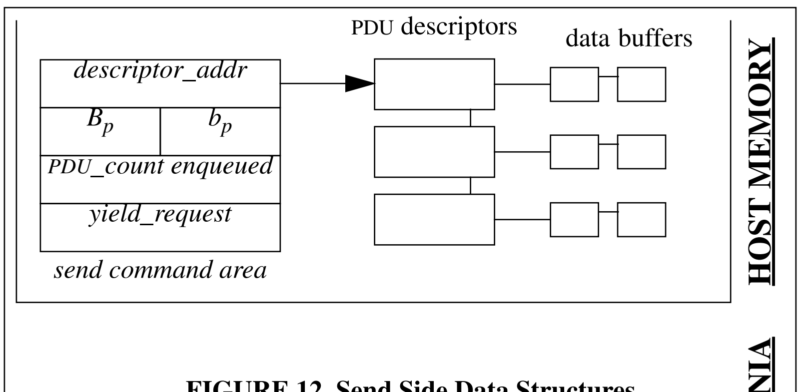

FIGURE 12. Send Side Data Structures

Receive Side. The protocol thread prepares a list of PDU descriptors that point to free buffers as before and

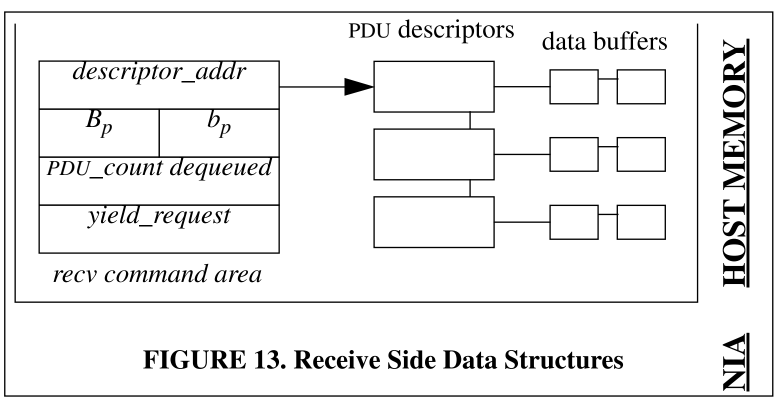

FIGURE 13. Receive Side Data Structures

enqueues it at the NIA. We assume that data buffers are pre-allocated from the shared memory area and put on a free list for the connection and that the NIA can traverse this free list. As PDUs are received, the data is written into the buffers, and control information associated with it is written into a available free descriptor. After the PDU has been received, the NIA writes the address of its buffers in the PDU descriptor, sets a bit in `status_flags` field. Subsequent PDU descriptors are formed into a linked list as in the sending case. When the receiving protocol thread is scheduled by the driver (see Section 4.3) it goes through the list of descriptors that have been received and processes them. The `command_flags` field is used to implement control functions. Figure 13 shows the organization of the memory.

B.3 Support For Control Operations in Protocol Processing

As seen in Section 5.3.4 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 NIA 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 flag in the `status_flags` field of 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 bit in the `command_flags` field. 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 in the `status_flags` 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 NIA (in the case of send) and to the receiving thread (in the receive case).

3. Remapping Support: As mentioned earlier, if the NIA 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 NIA 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 it to its right position. When the protocol needs to remap, 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.

B.4 Pseudo Code for a Sending Protocol Thread

We present the high level description of how a protocol developer would have to organize the protocol code following the techniques presented earlier. We take an example of a sending transport protocol thread that uses an acknowl-

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

edgment based scheme to do retransmission. The code segment illustrates how PDUs are moved between user and kernel, how the thread interacts with the scheduling mechanism, and the scheduling protocol control operations.

```
send_thread() {
  /* Check status of PDUs sent in earlier periods */
  for each pdu_descriptor that is marked as "SENT" by NIA
     if retransmission is not required
        free both PDU and its descriptor;
     else
         move descriptor to ACK_wait_list;
  for each descriptor on ACK_wait_list
     if retransmit-timer has expired, add descriptor to to_send_list;

  /* Now generate the next batch of PDUs */
  while there is more data to send do
     prepare PDU; Fill in descriptor;
     if retransmission is required
        set timer request bit in command_flags field;
        set timer value in command_words field;
     fi
     update pdu_count field in shared memory;
     if yield_request is set transfer to_send_list to driver;
  end
  /* Enqueue the batch at the driver and yield CPU */
  transfer to_send_list to driver and yield CPU.
}
```