Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: WUCS-96-11

1996-01-01

Efficient User space Protocol Implementations with QoS Guarantees using Real-time Upcalls

R. Gopalakrishnan and Guru M. Parulkar

Real-time upcalls (RTUs) are an operating systems mechanism to provide quality-of-service (QoS) guarantees to network applications, and to efficiently implement protocols in user space with (QoS) guarantees. Traditionally, threads (and real-time extensions to threads) have been used to structure concurrent activities in user space protocol implementations. However, preemptive scheduling required for real-time threads leads to excessive context switching, and introduces the need for expensive concurrency control mechanisms such as locking. The RTU mechanism exploits the iterative nature of protocol processing to eliminate the need for locking, and reduce asynchronous preemption, while ensuring real-time operation. In addition to efficiency, eliminating... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation
Gopalakrishnan, R. and Parulkar, Guru M., "Efficient User space Protocol Implementations with QoS Guarantees using Real-time Upcalls" Report Number: WUCS-96-11 (1996). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/403

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/403

Efficient User space Protocol Implementations with QoS Guarantees using Real-time Upcalls

R. Gopalakrishnan and Guru M. Parulkar

Complete Abstract:

Real-time upcalls (RTUs) are an operating systems mechanism to provide quality-of-service (QoS) guarantees to network applications, and to efficiently implement protocols in user space with (QoS) guarantees. Traditionally, threads (and real-time extensions to threads) have been used to structure concurrent activities in user space protocol implementations. However, preemptive scheduling required for real-time threads leads to excessive context switching, and introduces the need for expensive concurrency control mechanisms such as locking. The RTU mechanism exploits the iterative nature of protocol processing to eliminate the need for locking, and reduce asynchronous preemption, while ensuring real-time operation. In addition to efficiency, eliminating the need for concurrency control considerably simplifies protocol code. RTUs have been implemented in the NetBSD OS on the Sparc and Pentium platforms. We used the RTU mechanism to implement a sender and a receiver program that communicate using UDP sockets over 155Mbps ATM, and compared the throughput with that obtained when RTUs are not used. Our results show that the RTU based programs maintain the same throughput regardless of other system activities. Without the RTU mechanism, background load reduces throughput by as much as 80%. Using the RTU mechanism, total network bandwidth can be partitioned among different UDP streams, and delivered to user programs even with background system load. We have also implemented the TCP protocol in user space using the RTU mechanism. For each TCP connection, RTUs are setup for performing TCP output, input, and timer processing functions. The use of RTUs, in conjunction with shared memory between kernel and user processes for data movement, and the ability of the ATM adaptor driver to separate headers and data, makes our user level TCP implementations the most efficient one that we know of for providing QoS guarantees within the endsystem.

Efficient User space Protocol Implementations with QoS Guarantees using Real-time Upcalls

R. Gopalakrishnan

Guru M. Parulkar

WUCS-96-11

March 4, 1996

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

Efficient User space Protocol Implementations with QoS Guarantees using Real-time Upcalls

R. Gopalakrishnan
gopal@dworkin.wustl.edu

Gurudatta M. Parulkar
guru@flora.wustl.edu

Department of Computer Science
Washington University in St.Louis

Abstract

Real-time upcalls (RTUs) are an operating systems mechanism to provide quality-of-service (QoS) guarantees to network applications, and to efficiently implement protocols in user space with (QoS) guarantees. Traditionally, threads (and real-time extensions to threads) have been used to structure concurrent activities in user space protocol implementations. However, preemptive scheduling required for real-time threads leads to excessive context switching, and introduces the need for expensive concurrency control mechanisms such as locking. The RTU mechanism exploits the iterative nature of protocol processing to eliminate the need for locking, and reduce asynchronous preemption, while ensuring real-time operation. In addition to efficiency, eliminating the need for concurrency control considerably simplifies protocol code.

RTUs have been implemented in the NetBSD OS on the Sparc and Pentium platforms. We used the RTU mechanism to implement a sender and a receiver program that communicate using UDP sockets over 155Mbps ATM, and compared the throughput with that obtained when RTUs are not used. Our results show that the RTU based programs maintain the same throughput regardless of other system activities. Without the RTU mechanism, background load reduces throughput by as much as 80%. Using the RTU mechanism, total network bandwidth can be partitioned among different UDP streams, and delivered to user programs even with background system load.

We have also implemented the TCP protocol in user space using the RTU mechanism. For each TCP connection, RTUs are setup for performing TCP output, input, and timer processing functions. The use of RTUs, in conjunction with shared memory between kernel and user processes for data movement, and the ability of the ATM adaptor driver to separate headers and data, makes our user level TCP implementation the most efficient one that we know of for providing QoS guarantees within the endsystem.

1 Introduction

There is a growing need to provide support for multimedia processing within computer operating systems. This will enable a variety of exciting applications, such as interactive video, customized news services, virtual shopping malls and many others. A large fraction of data handled by these applications will be of the continuous media (CM) type. The transfer of CM data over the network and its processing at the endsystem must be in such a way that its periodic nature is preserved. This requirement is also referred to as the need to provide "real-time" guarantees for data transfer and processing. Emerging networks such as ATM and the proposed integrated services Internet [8] with reservation protocols such as RSVP [23] can provide guarantees for data transfer by managing network resources appropriately. Similarly, the operating system (OS) must manage endsystem resources so that processing needs implied by the bandwidth and delay requirements of each connection are satisfied. This will complement the guarantees provided by the network for data transfer and build upon the increasing processing power of the endsystem hardware.

Providing processing guarantees for multimedia data at the endsystem is important. Without these guarantees,

one can never be sure that two or more applications will not interact in an adverse manner. The situation would be somewhat similar to going to a ballgame in which there is no guarantee that only as many tickets as there are seats have been sold. In addition to the aspect of providing guarantees, the scheduling mechanism must ensure *timeliness*, which means that the data units of each connection must be presented at the right instant to the application after the necessary processing. For multimedia applications to become ubiquitous, these two features must be provided so that users are assured that the operating system will preserve the periodic and continuous nature of the data streams over arbitrarily long durations, no matter what other processing activity occurs during this time. In the networking domain, this is referred to as the need to provide quality-of-service (QoS) guarantees. Until now researchers have been concentrating on how to provide QoS guarantees within the network. We have developed a QoS framework for the endsystem. Important aspects of this framework are-

- **QoS Specification:** Ways to allow an application to specify the QoS it requires for each data stream at a high level
- **QoS Mapping:** Deriving various low level resource requirements from the QoS specification
- **QoS Enforcement:** Scheduling mechanisms within the endsystem to ensure that processing requirements of an application are satisfied
- **Protocol Implementation:** Implementing protocols using the components provided in the framework

This paper reports on a mechanism called the real-time upcall (RTU) that can be used as a QoS enforcement mechanism in the endsystem, and to efficiently implement protocols in user space with QoS guarantees. RTUs are an alternative to real-time threads [22] for protocol processing. The most important benefits of the RTU mechanism are an improvement in runtime efficiency and simplification of protocol code. and claims–

Arguments for the RTU Approach and Claims. The RTU approach is based on the following arguments.

- It is a well recognized fact that the upcall mechanism allows efficient protocol implementations [7]. Our experience strongly suggests that upcalls are the right abstraction to implement all concurrent activities in protocol processing. Our real-time extension to the upcall mechanism maintains all these advantages, and in addition provides QoS guarantees to network applications at lesser run-time cost than other mechanisms such as real-time threads. We claim that to provide end-to-end guarantees, both protocol processing, as well as application processing must be scheduled in real-time. By implementing protocols in user space as part of application processes, a single real-time mechanism can be used to schedule protocol as well as application processing.
- The RTU mechanism is the most efficient one that we know of, for implementing protocols in user space with QoS guarantees. We make the important observation that protocol processing is repetitive and iterative in nature specially for bulk data transfer and continuous media applications. Given a large chunk of data, the protocol stack processes it in terms of (usually fixed size) protocol data units (PDUs). Protocol processing involves iterating over each PDU, and repeatedly performing the same operations. Based on this observation, we realize two optimizations for efficiency.
    1. We express the processing requirements for each connection in terms of the number of PDUs to be processed within a given time period, and the procedure (or upcall handler) that iterates over the PDUs to perform protocol processing. Our claim is that our simple upcall scheduler that tracks processing time in terms of number of PDUs processed, can outperform a general purpose real-time thread scheduler. Thus the repetitive nature of protocol processing can be exploited to provide *scheduling efficiency*.
    2. In a fully preemptive scheduling scheme used by real-time schedulers, threads must lock shared variables before access. Locking mechanisms for real-time threads must prevent unbounded priority inversions, and have to be implemented in the kernel. Therefore real-time threads have to make system calls to manipulate locks. For a Pentium machine running NetBSD the system call overhead is about 770 processor cycles (or 280 instructions) [5]. Since PDU processing itself takes only a few hundred instructions, getting and releasing a lock per PDU increases PDU processing time manyfold. The RTU mechanism exploits the iterative nature of protocol processing to implement preemption by allowing a

RTU to complete its current iteration (i.e processing the current PDU) and then switching to the higher priority RTU. This scheme eliminates the locking operations (system calls) during protocol processing. It also avoids situations in which a thread is switched in, only to yield immediately because a lock is held by a lower priority thread [11]. Eliminating concurrency control also simplifies protocol code and speeds up development and testing.

- Although user space protocols have been shown to deliver good performance, we believe that further innovations are required to handle the additional dimension of QoS guarantees. Accordingly, we have implemented several system support mechanisms for user space protocols. For example, we use shared user-kernel memory to avoid data copying and virtual memory (VM) operations. Our scheduling scheme reduces context switching, and eliminates system calls for data movement. With these features, we can get the same efficiency as, if not better than, the best kernel resident protocol implementations.

Important Experimental Results. We briefly state our important experimental results. RTU scheduling ensures that each handler invocation completes before its deadline (i.e the next period). RTUs can provide bandwidth guarantees for UDP to applications even in the presence of background load on the endsystem. For comparison, note that without RTUs, the same background load causes an 80% reduction in throughput. The RTU mechanism provides bandwidth sharing and bandwidth partitioning among different streams on the endsystem, even with background system load. The RTU scheduling scheme also allows us to control the number of preemptions and therefore reduces context switching. This results in upto a 9% increase in useful utilization for certain workloads. By eliminating the need for locking, thereby avoiding context switches caused by lock unavailability, we expect even higher effective utilization as well as more efficiency gains.

Paper Outline. The outline of the paper is as follows. Section 2 gives the QoS framework that we have developed for the endsystem. Sections 3 to-5 present the basic RTU mechanism and its ability to provide bandwidth guarantees. In particular, Section 3 discusses limitations of existing scheduling mechanisms, Section 4 describes the RTU mechanism and the scheduling scheme, and Section 5 presents experimental results. Sections 6 -8 describe user space protocol implementations with RTUs. In particular, Section 6 discusses issues in user space protocol implementation, Section 7 describes how protocols can be efficiently implemented with RTUs, and Section 8 describes highlights of our user space TCP. Finally we present our conclusions and plan of future work.

2 The QoS Framework

We have developed a QoS framework [1] from the point of view of application processes that run on the endsystem. We identify four dimensions to the problem of providing QoS guarantees. These are–

QoS Specification Specification of QoS requirements is essential to provide performance guarantees. Since applications can have widely varying requirements, a structured and general way to specify QoS is necessary. A typical approach is to identify a few application classes that can capture the needs of most applications, and to define parameters within each class using which users can specify their requirements. We have identified four application classes that encompass continuous media, bulk data, low bandwidth transaction messages, and high bandwidth message streams. It is important to note that applications can only specify high level parameters, and other low level parameters must be derived automatically.

QoS Mapping The QoS specifications mentioned above are at the application level. Since several resources (such as CPU, memory, and network connections) are involved in communication, the specifications must be mapped to resource requirements. For example, network connection bandwidth must be determined from the specified QoS parameters while setting up the network connection. Likewise, the amount of processing required must be determined so that the CPU capacity can be allocated to ensure that protocol data units (PDUs) are processed at the desired rate. The operation of deriving resource requirements from QoS specifications is referred to as QoS mapping, and is illustrated in Figure 1. From the specification for a data stream, the mapping operation derives network connection attributes such as bandwidth and cell delay, pacing parameters to determine the rate at which the network interface must transmit cells, processing attributes for the protocol code, and memory requirements. We have worked out details of the mapping operation for the resources mentioned above [1].

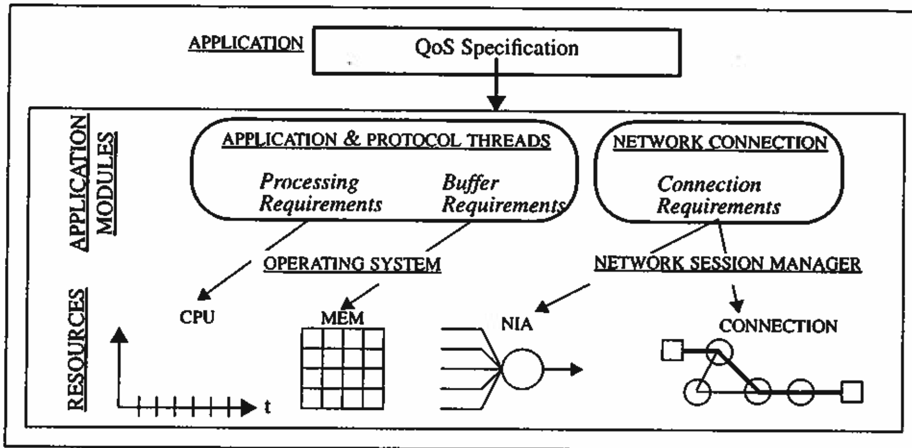

Figure 1: QoS Specification and Mapping

QoS Enforcement The mapping operation as mentioned above derives resource requirements, that are allocated by the operating system to each application during the setup phase. During the data transfer phase, the operating system implements the QoS enforcement function, which involves scheduling various shared resources to satisfy these allocations. In particular, the CPU scheduling policy of the operating system largely determines how the aggregate processing capacity of the endsystem is shared between different network sessions and is therefore crucial in determining the QoS provided. We therefore focus on CPU scheduling mechanisms that can be implemented efficiently, and integrated into protocol code.

Protocol Implementation Model Given these solutions for QoS specification, mapping and enforcement, protocol code has to be structured to take advantage of these facilities. The protocol implementation model facilitates the mapping of protocol services to appropriate implementation components provided in the framework. An important objective of the model is to provide efficient OS mechanisms to improve the efficiency of protocol implementations by reducing the overhead associated with data movement and context-switching operations that have been shown to dominate protocol processing costs [7].

2.1 CPU Requirements for Protocol Processing

QoS specification and mapping steps are explained in greater detail in [4]. This paper mainly deals with QoS enforcement and protocol implementation. However, for the purpose of this paper, it is necessary to understand how processing requirements are expressed for each connection. Processing requirements depend upon parameters such as the network bandwidth allocated to the connection, PDU size, and time to process a PDU. The bandwidth of the connection and the size of a PDU determine the duration of time required by the host interface adaptor to drain a PDU. In order to keep up with the adaptor, protocol processing at the sender must occur at a rate so that at least one PDU is generated and enqueued at the adaptor in every such duration. Equivalently, we could require that a batch of B PDUs be generated in every interval of length T, that is B times the original duration. *Batching* is useful because, if protocol sessions are active in different processes, it is expensive to switch between processes after each processed PDU [21]. Other trade-offs involved in the choice of batch size are discussed in [1]. Depending on the host platform, and the amount of processing involved, the computation time C required to process B PDUs can be determined. Thus, we arrive at a periodic processing model as shown in Figure 2. The period is T, and the requested computation time is C. Note that such a periodic model is widely used in the real-time scheduling domain to express processing requirements [22]. In addition, it closely mirrors the periodic nature of continuous media data. The main difference is that the RTU mechanism uses the number of PDUs processed per period, rather than time, as a measure of computation requirement. This is based on the assumption that protocol processing time is constant for each PDU since PDU sizes do not change for most protocols during the life of the connection.

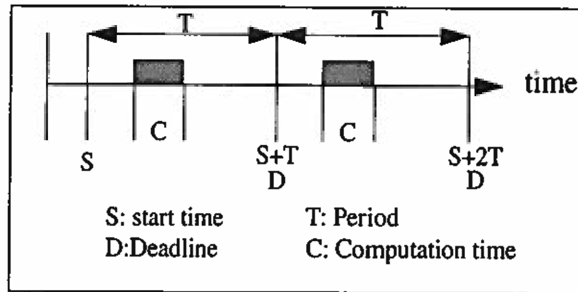

Figure 2: Periodic Processing Model

3 Existing QoS Enforcement Mechanisms and their Limitations

The existing UNIX scheduling mechanism is inadequate to support the periodic processing model. This is because the UNIX scheduler is designed for time sharing rather than real-time operation. We therefore need the functionality of a real-time thread mechanism provided by operating systems such as RT-Mach [22] or Solaris [14]. However, there are several drawbacks to using existing real-time thread mechanisms for protocol processing.

- The real-time nature of threads, coupled with the preemptive scheduling mechanism used in most systems, introduces additional complexity to support locking of shared variables. When a real-time thread locks a shared resource, *priority inheritance* [20] is required to prevent unbounded priority inversions. Both RT-Mach [9] and Solaris [14] address this problem, but it comes at a high cost. Each lock operation requires a system call to setup state in the kernel, so that priority can subsequently be inherited (if required) [14]. Locking is inherently pessimistic, since it protects against the unlikely eventuality that access to shared resources overlap in time. It therefore runs counter to the "success oriented" strategies used by protocol implementations. In addition, locking combined with preemptive priority scheduling causes unnecessary context switches due to lock contention. This occurs when a thread preempts a running (low priority) thread, only to find that a lock that it requires is unavailable [11].

  Locking is used in most protocol implementations. For example in the BSD TCP implementation, the output processing, input processing, and timer processing routines lock the protocol control block (PCB) for a TCP connection before they process each segment. This is done, among other reasons, to prevent the state from changing while a segment is being processed. Since the PCB is updated for each segment, a pair of lock operations may be needed for each segment. This would make scheduling mechanisms as implemented in RT-Mach and Solaris too expensive to use. Although a completely non-preemptive scheduling scheme would also eliminate the need for locking, it is not a viable option since it can provide processing guarantees only at very low utilizations.

- Context switching costs are known to dominate PDU processing costs [7]. Therefore, the number of context switches must be kept to a minimum. However, a fully preemptive real-time scheduling policy can lead to excessive context switching compared to quantum based time sharing schemes, thereby reducing effective utilization. We note as before that although non-preemptive scheduling keeps context switching to a minimum, it does not allow high utilization.

- Only few operating systems directly support the periodic model for real-time processing. Solaris only provides a fixed number of real-time priorities. In addition real-time facilities are restricted to the superuser.

Thus there was a real need to explore alternative abstractions and scheduling schemes, that could meet the special needs of protocol processing such as providing high utilization and low context switching overheads.

4 The Real-time Upcall (RTU) Approach

We present the RTU mechanism as an alternative to real-time threads for implementing protocol processing. An upcall is a well known mechanism [6] to structure layered protocol code. Protocol code in a user process can register an upcall with the kernel and associate a handler function with each upcall. The kernel can then arrange to have the handler invoked when an event occurs for the upcall. Each RTU is associated with a period and the number of PDUs to be processed in each period (derived by the QoS mapping operation). An event for an RTU is the start of a new period[1] and occurs in the clock interrupt. There is an admission control operation (schedulability test) to ensure that each RTU gets its requested time to run in each period [3]. If admitted, each RTU handler is invoked periodically in real-time.

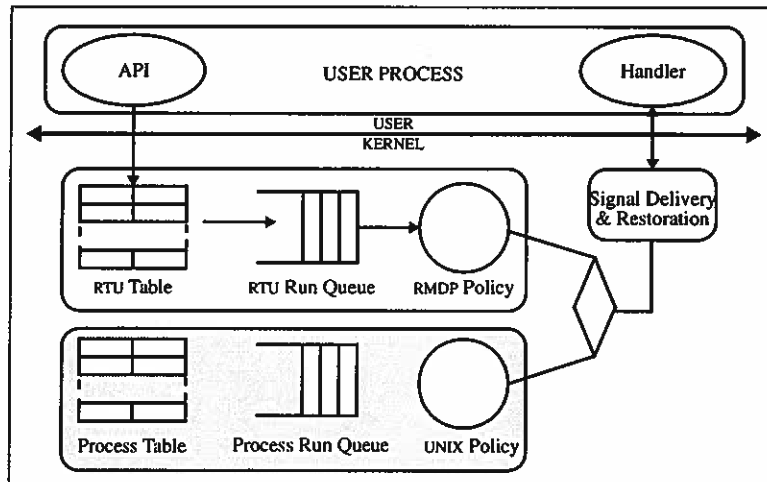

Figure 3: RTU Organization

The overall organization of the RTU facility is shown in Figure 3. The RTU facility is layered on top of the normal UNIX process scheduling mechanism. No modifications to the existing UNIX scheduler, or its policies are needed. A process creates an RTU using the system call API. A process can create multiple RTUs. The create operation returns a *file descriptor* that can be used to perform subsequent operations on the RTU. A memory address can also be associated with each RTU during creation. This is typically the address of the protocol control block (PCB) which can be used to isolate different connections in the same process. Once created, an RTU can be run using an API call. A running RTU is inserted into the RTU run queue in each period. When the RTU reaches the head of the run queue, its handler is upcalled, and the registered memory address associated with it is passed as an argument. RTUs in the run queue are scheduled according to a modified rate-monotonic scheme called RMDP (RM with Delayed Preemption). The RMDP policy picks out the process that contains the highest priority RTU, and makes it the currently running process. The RTU delivery and restoration routines ensure that the handler in the process is invoked with the correct arguments, and the system state is restored after the handler returns.

The *signal* mechanism is a somewhat similar mechanism that is provided by UNIX. For example, the combination of the *alarm* and *signal* facilities in UNIX can be used to setup a handler to be called periodically. One limitation with the existing facility is that only one such alarm can be setup per process. Another drawback is that there could be arbitrary delays before the alarm signal actually gets delivered and so real-time behavior cannot be ensured. The third major limitation is that the scheduler cannot guarantee that all signal handlers will get their requested amount of processing time. Finally, signals have traditionally been used only for exception handling and are inappropriate as a processing abstraction.

Benefits of RTUs for Protocol Processing. RTUs are less expensive to implement than real-time threads. An RTU runs on the program stack and the kernel needs to maintain very little state information per RTU. RTUs are local to a process and typically synchronization between RTUS in different processes is not necessary

[1] For low latency applications the event could be packet arrival although we have not implemented this.

(nor supported) because they would be associated with different protocol sessions. Another feature of the RTU abstraction is that a running handler will not be preempted by a higher priority handler in an uncontrolled manner. Since protocol processing is iterative, we have implemented preemption as completion of the current iteration (which might be processing the current PDU) and returning from the handler routine. The obvious advantage of this scheme is that there is no need to save the activation record and register context of the handler across preemptions. The other advantage is that two (or more) RTUs in a process that share common variables need not lock them before access. Since simplicity often translates to greater efficiency, we believe RTUs are an optimal mechanism to organize protocol processing in user space with QoS guarantees.

We make a final observation relating various processing abstractions. If we were to classify processing mechanisms provided by an OS in terms of increasing functionality, the order would be procedures, upcalls, threads, and processes. The cost of using each mechanism also increases in the same order. Our discussion about RTUs indicates that a similar hierarchy exists for real-time processing. Generality and functionality increase as we go from RTUs to RT-threads, with a corresponding increase in the cost of usage. Viewed in this light, the upcall (RTU) mechanism comes across as a first class OS mechanism that is specially suitable for iterative (real-time) processing. We believe that in the real-time domain, the RTU mechanism fulfills the need to do efficient concurrent iterative processing, just as the thread mechanism fulfills the need to do efficient (general purpose) concurrent processing. In the next section, we describe how the RMDP policy takes advantage of the iterative nature of protocol processing to eliminate the need for concurrency control and reduce number of preemptions.

4.1 The RMDP Scheduling Policy

The RMDP policy is a modification of the rate monotonic scheduling policy [16]. In the basic RM policy, the priority of a thread is inversely proportional to its period (or directly proportional to its rate). The RM policy requires the highest priority thread in the run queue to be running at any instant. The RMDP scheme also uses RM priorities, but it uses a different preemption scheme. In this scheme, an RTU handler is allowed to do some minimum amount of work every time it is run. This work is measured in terms of number of PDUs processed. Therefore, in addition to the batch size B which is the maximum number of PDUs that the RTU can process each period, we associate a value b which is the minimum number of PDUs that the RTU is allowed to process (i.e., one iteration of the handler) without interruption from higher priority RTUs. Thus higher priority runnable RTUs have to wait until the running handler completes its current iteration. Since a handler completes at least one iteration every time it runs, the number of context switches is reduced. In addition, if variables shared with other RTUs are accessed only inside an iteration, there is no need to lock them. We now describe how RMDP is implemented using shared memory between each RTU and the scheduler.

The RMDP algorithm is shown in Figure 4. The data structure shown on the right is placed in a memory region that is shared by an RTU and the scheduler. Each time a running RTU handler generates/consumes a PDU, it increments the *pdu_count* field. It then checks if the number of PDUs processed so far (the value of *pdu_count*) equals the batch size B. If so, it yields the CPU since it has obtained its processing requirement for the current period. This gives control to the RMDP scheduler which schedules the next RTU (if any). If *pdu_count* is less than B, then the RTU handler goes on to process the next b PDUs (i.e the next iteration). Before doing so, it checks to see if the *yield_request* field in the shared memory structure is set. The *yield_request* field gets set in the kernel clock interrupt routine when a higher priority RTU becomes runnable. If the handler notices this field to be set, it yields the processor, and it is put back in the run queue. The count of PDUs that remain to be processed for the current period is stored in the shared memory (not shown), and when the handler is resumed, it needs to process only these many PDUs. Other systems [13, 18] have used shared memory to allow the user level scheduler and the kernel level scheduler to communicate, but their objectives and the algorithms used are different. Although the RMDP algorithm uses a cooperative scheduling scheme, it can protect against misbehaving RTUs that do not yield the CPU when they are requested. This is because the RTU handlers run at the lowest priority level just as any other user process, and therefore cannot block interrupts. More details are given in [2].

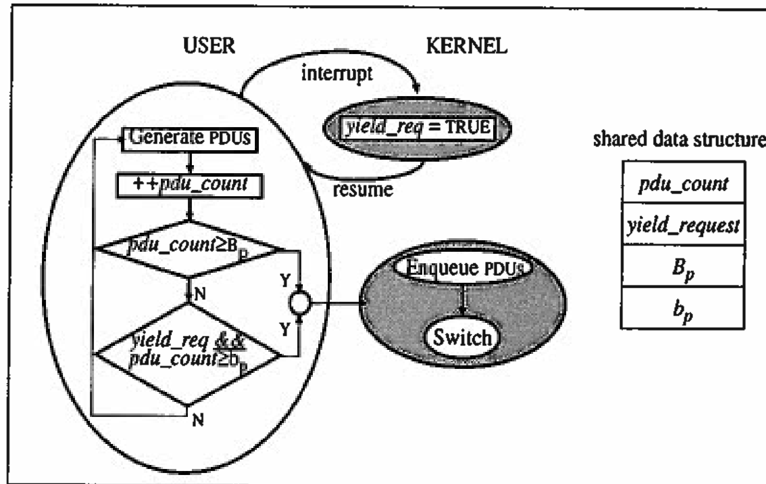

Figure 4: The RMDP Scheme

5 Experiments Using RTUs

Our initial implementation of the RTU facility was in the NetBSD kernel on a Sun 4/360 Sparc-1 machine. We subsequently ported it to the Pentium platform. The Pentium machines are equipped with 155 Mbps ATM interfaces on the PCI bus. We report two kinds of experiments. We first show that the RTU facility meets its design goal of *timeliness* and *guarantees* even with other system loads (background load). We then show how the RTU mechanism can be used to program sender and receiver programs that can provide accurate throughput guarantees and bandwidth sharing/partitioning across connections even in the presence of background system loads. All experiments reported are on 100 Mhz Pentium with clock interrupts occurring every 1 millisecond (msec).

5.1 Ability to Meet Deadlines

We have performed several experiments to verify if all RTUs meet their deadlines and to quantify the reduction in context switching. We report one of these experiments to show that RTUs meet deadlines and provide CPU utilization that is guaranteed by theoretical results [16]. We created 6 RTUs in each of two processes with periods ranging from 40 msecs to 90 msecs in increments of 10 msec. We varied the total CPU utilization consumed by the 12 RTU handlers and measured how close each invocation came to its deadline. The utilization for a handler was obtained by measuring the time it took to complete under idealized conditions when there were no page faults or preemptions, and dividing this time by its period. For each invocation of an RTU, we noted its completion time and subtracted it from its deadline to obtain the time remaining for its deadline. We then took the minimum of these remainders among all the invocations (which represents the worst case) and plotted it against each utilization value as shown in Figure 5. In this experiment, no preemption delay was introduced by the handlers, which means that they yielded as soon as a higher priority RTU became ready. We repeated this experiment with programs such as *primes, netscape*, and *make* running concurrently. We notice that upto a utilization of 0.8 all the ordinates are positive which means that all handlers complete before their deadlines. When utilization increases further, the lowest priority RTUs start missing deadlines. We also see that the completion time for RTUs increases slightly late when there is other system load.

This experiment clearly demonstrates that all deadlines are met, and that the RTU mechanism guarantees timeliness even when there is background system load. The experiment also shows that OS scheduling overheads do not reduce the effective CPU utilization. We would like to point out that the NetBSD kernel is non-preemptible, and so when other processes are running, a RTU handler may have to wait for the worst case time spent in a system call, before it can run. Thus background load can affect an RTU only for the duration of the longest system call and our results indicate that this worst case does not affect performance.

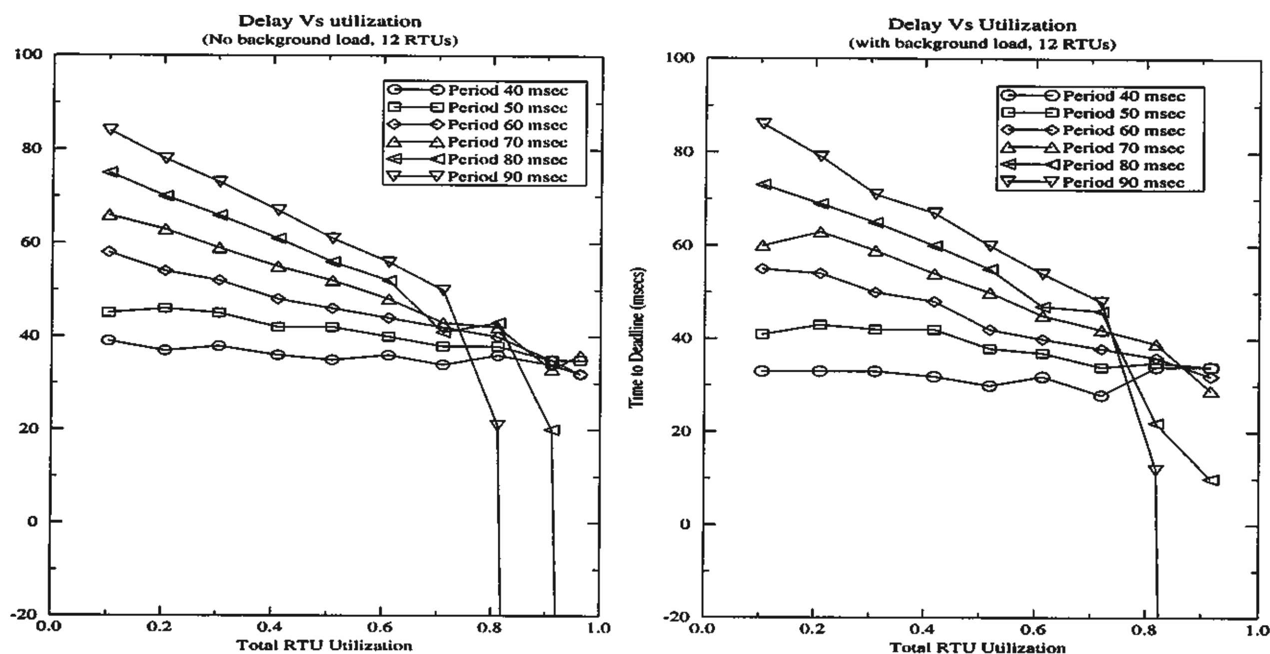

Figure 5: Real-time Behavior of RTUs

5.2 Throughput Guarantees with RTUs

We report another experiment to show that RTUs can provide throughput guarantees even with other processes running in the endsystem. We implemented a sender and a receiver program that communicate over a UDP socket. The UDP that we used is part of the standard NetBSD implementation in the kernel. The sender (receiver) is implemented using an RTU that is invoked every period, and writes (reads) a batch of UDP packets in each period. The programs are also implemented in the standard manner without any RTU support where the sender (receiver) program writes (reads) the UDP socket in a loop. The number of UDP packets sent (N), and the packet size was varied for both cases, and the throughput seen at the receiver was measured. The experiments were performed both without any other processes sharing the system, and with other system load.

Figure 6 shows the throughput for the two cases when there is no system load on the receiver's system, and when system load is present. When there is no load we see that the performance of the RTU based receiver is the same as the standard receiver program. In both cases, the throughput increases with packet sizes. When background load is introduced, the RTU based receiver throughput remains the same. However for the non-RTU receiver, throughput drops by 80% for 8K packets. The main reason for the reduction is that the socket buffers at the receiver overflow since the receiver process is not being scheduled often enough to be able to drain its buffers. Thus we conclude that RTU mechanism can provide processing guarantees that cannot be provided by standard UNIX scheduling.

The next experiment shows that RTU based programs can provide bandwidth sharing and partitioning among different connections. We created 6 streams (sender-receiver pairs) on the two machines. The sender RTU was invoked every 20 msec, and sent a batch of 4 packets of size 8KB each per period. The receiver RTU period was 10 msec and its batch size is 2. Each stream was also implemented without RTU mechanism. The 6 RTU based streams were run at the same time and the throughput for each stream was measured at the receiver. The same measurements were obtained for the non-RTU based streams. The results are shown in Figure 7. As can be seen, the 6 RTU streams obtain an equal share of the total bandwidth of 78.6 Mbps. The non-RTU streams interfere with each other causing their socket buffers to overflow. As a result, all of them have very low throughput. When the above experiment is performed with system load due to other processes, the bandwidth sharing provided by the RTU mechanism is preserved whereas the throughput for the non-RTU case drops even lower relative to its value in the no load case.

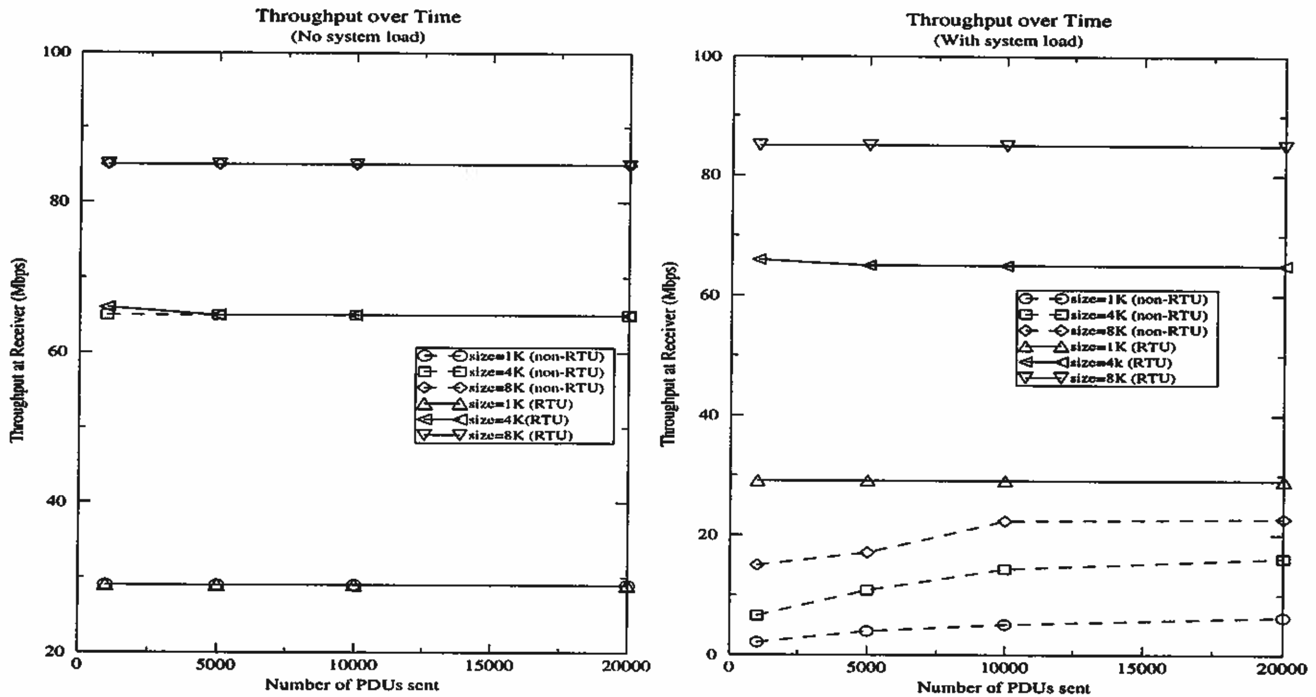

Figure 6: Bandwidth Guarantees for UDP

6 User Space Protocol Implementation–Issues and Related Work

Most of the important arguments for user space protocols have been brought out in [21, 17]. These include ease of prototyping and maintenance, co-existence of multiple protocols, and ability to exploit application specific knowledge to improve performance. The most compelling reason is that user space implementations can improve efficiency of protocol processing by taking advantage of the ability to move data directly between user processes and the network interface bypassing the OS buffers. Mechanisms such as packet filters [19], BQIs [21], and the ATM VCI [11] are used to achieve this. Since RTU handlers run in user space, they facilitate application level (or user space) protocol (ALP) implementations. The RTU mechanism allows application processing to be also scheduled in real-time, and therefore has the potential to provide end-to-end guarantees.

In an ALP, boundary operations such as connection setup and release are implemented by a single trusted entity because they involve setting up of resources that are shared among all application processes. The data transfer protocol processing that largely determines QoS for a session executes as part of the application process. This processing is structured using OS mechanisms such as threads [21], processes [11] or in our case RTUs. The ALP model is well suited to native mode ATM implementations where PDUs of each transport connection are carried on a separate network connection with appropriate QoS parameters that meet the requirements of the session. Another benefit of using ATM is the simple demultiplexing scheme that is based on identifying each connection by its VCI. Thus, both incoming and outgoing data can move directly between user space and the VCI without having to be moved into OS buffers [12]. Thus the OS is involved mainly in scheduling the processing of network data in the different processes. As an aside, we note that the RTU approach for QoS guarantees is not restricted to user space protocols, and in fact it can be easily extended to protocols implemented in the kernel.

Important efficiency issues in user space protocols are to avoid data copying, reduce context switches, and eliminate system calls during packet processing as outlined in the following paragraphs.

- *System Calls for Network I/O*: In the ALP model the buffers that move across the user-kernel boundary are PDUs. With kernel resident protocols, these buffers are much larger, and are passed as parameters in system calls. If the same mechanism is used to initiate PDU movement in user space protocols, the overhead due to system calls will be very high.

- *Data Movement*: Data movement across the user-kernel boundary is another important efficiency issue. Data can be moved either by copying or by page remapping. Data copying is expensive because of the limited memory bandwidth in most systems. Page remapping costs can also be high if performed on a per

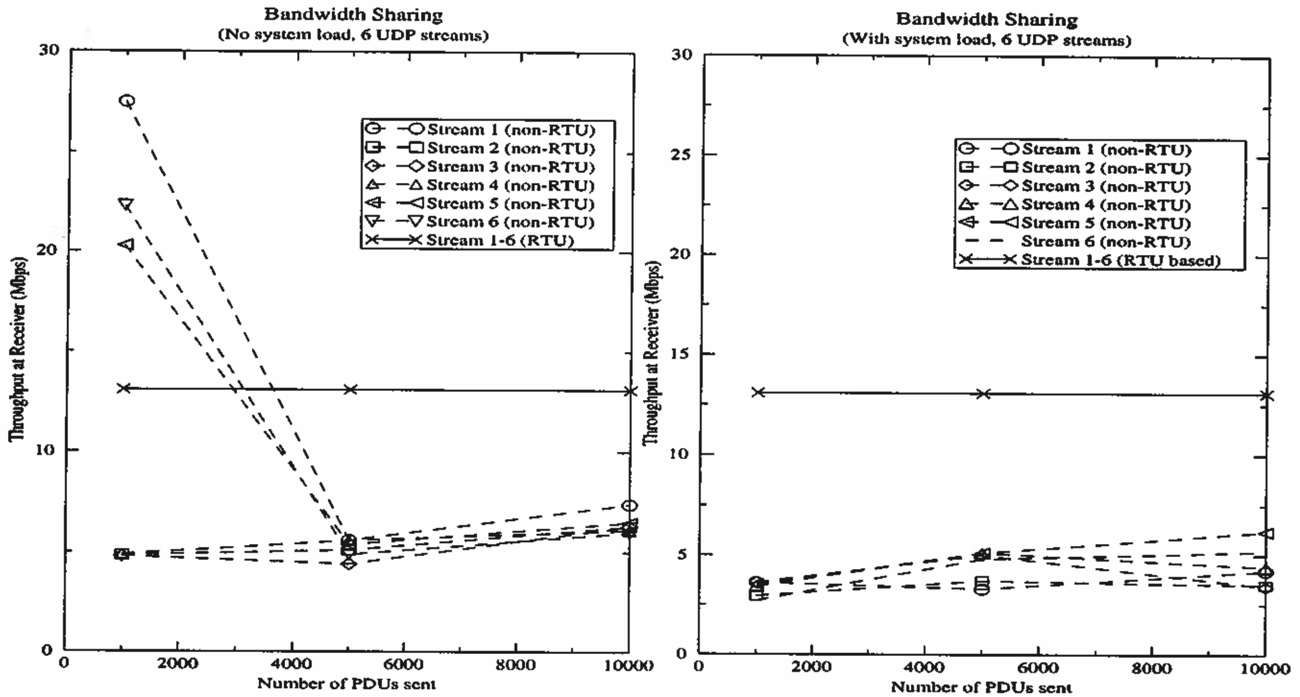

Figure 7: Bandwidth Sharing among UDP Streams

PDU basis. This problem gets exacerbated in user space protocols because system calls are needed for cross domain copying and page remapping operations.

- *Asynchronous Event Processing:* Processing of asynchronous network and protocol events such as packet arrival and timer expirations are usually triggered by interrupts in current protocol implementations. Using this mechanism in user space protocols leads to excessive context switching because successive events may be destined for different processes. In addition, event driven scheduling disrupts priority based scheduling schemes leading to priority inversions.
- *Scheduling Concurrent Protocol Activities:* Protocol processing for each connection is usually structured as "upper" and "lower" halves. The "upper" half is driven by application calls such as writing or reading data. The "lower" half is driven by network events such as packet arrival or timer expirations. These halves must run concurrently since they cannot wait for each other. Thus, some mechanism for concurrency is required to implement protocols. With concurrency, arises the need for a concurrency control mechanism for synchronization between the upper and lower halves.

6.1 Related Work

There have been several user space protocol implementations [17, 21, 11, 12]. Most of them share common features such as using a connection server for connection setup and teardown, avoiding copying of data by using memory "pools" to locate network data, and demultiplexing packets at the adaptor level. Most previous efforts have focused on the TCP/IP protocol stack. The first implementation was for the Mach OS using Mach threads for concurrency and Mach IPC for data movement [21]. This implementation was shown to perform better than the standard TCP implementation in Mach. However, it did not perform as well as a monolithic kernel implementation. Moreover, we expect that if RT-threads in RT-Mach are used in place of the original C-threads, in order to provide QoS guarantees, the cost of concurrency control will be very high and make the implementation perform even worse. Implementations of TCP in HP-UX [11] and of U-Net in SunOS [12] are representative user space implementations in UNIX. The HP implementation used real-time priorities to speedup protocol processing, although it was not intended to provide bandwidth guarantees. U-Net does not deal with concurrency issues in protocol processing and is at present a "proof of concept" implementation. We will focus on the HP implementation [11] to illustrate the deficiencies with current user space protocol implementations. We argue that these shortcomings are the

inevitable result of the lack of efficient concurrency mechanisms, and the lack of OS mechanisms to integrate scheduling with protocol processing.

- *Lack of an efficient concurrency mechanism:* Since there is no thread facility in HP-UX, the upper and lower halves of a TCP connection were implemented as separate processes. The main drawback to having the two halves in different domains is that connection state information must be placed in shared memory and pointer data must be handled correctly. Furthermore, there is context switching between upper and lower halves. The design also makes it difficult to support more than one TCP connection per process.

- *Costs of concurrency control:* The use of shared memory between upper and lower halves introduces the need for concurrency control since processes are scheduled in a preemptive manner. Concurrency control using semaphores was found to be expensive since this involved making a system call for each semaphore operation. In addition, locking leads to situations in which one half is scheduled to run and discovers that a lock that it needs is held by the other half. This results in two unnecessary context switches.

- *Handling Process Termination:* The lifetime of a TCP connection is longer than that of its endpoint processes. If a process on one end terminates while a connection is active, then the TCP code must linger long enough to be able to initiate connection termination procedures with its peer. In a kernel implementation of TCP, this is not a problem since the TCP state is in the kernel. The HP implementation uses a separate process to achieve this [11]. Other implementations transfer state to the kernel when the process exits [17]. The first solution is awkward, and the second solution requires the kernel to know the termination procedure for each user space protocol. Handling process termination at the user level in a general manner requires integration of process scheduling with protocol processing and is described in Section 7.3.

7 User Space Protocol Implementation with RTUs

Our user space protocol design provides the following features to protocols. To get the maximum benefit of these features, protocols must be implemented using RTUs, although non-RTU implementations can also take advantage of these features.

- It eliminates all system calls in the packet processing path. Shared memory is used in place of system calls to initiate network I/O. The buffer management facility uses "lock free" [10] data structures to eliminate the need for synchronization between the user and the kernel during I/O operations. In addition, RTUs in the same process that share a common buffer pool need not synchronize among themselves due to fact that the RMDP scheduling policy does not preempt a handler asynchronously.

- It provides for movement of network data between a user process and the ATM adaptor without having to go via kernel buffers. *Zero copy* operation is available for applications that can work with non-contiguous buffers. "Wired down" shared memory between each user process and the kernel is the basic mechanism used to achieve this. Furthermore, no VM operations are needed during packet processing.

- The number of scheduling context switches can be controlled by tuning the minimum number of PDUs processed in each RTU invocation. In addition, eliminating locking avoids unnecessary context switches due to lock denials.

In the rest of this section, we present details of these features and show how they relate to each of the issues raised in Section 6.

7.1 Concurrency and Asynchronous Event Processing using RTUs

The first step is to determine the concurrent activities in the protocol and create an RTU for each such activity as part of connection setup. Consider for example the sending side of a unidirectional bulk data transfer protocol. One RTU could be used to packetize and send data, and another RTU could be used to process control information (such as acknowledgments) sent by the receiver. The attributes of the sending RTU (period and the the batch size)

would depend on the bandwidth requirement specified by the application. The attributes of the RTU that handles control information would have to be decided based on the nature of the protocol. Some simple protocols (such as UDP) may require only one RTU at each endpoint. In general, the connection setup procedure would create the RTUs on both the sending and the receiving endsystem. A protocol control block (PCB) is also allocated for each connection and registered with each of its RTUs. The PCB should be used to store the protocol state for the connection, so that multiple active connections within a process can be isolated from one another.

RTU handlers must be written so that they cooperate with the scheduler. An RTU that processes PDUs iteratively must check the *yield_request* field (Figure 4) after each PDU. In addition, an RTU handler must never block waiting for another RTU since this will lead to a deadlock. This however is never a problem since, traditionally, protocol functions in the kernel are written so they can be called from the interrupt handler, and therefore do not block.

The main benefit of the RTU mechanism is that asynchronous events need not be handled as they arrive. For example, events such as packet arrivals are buffered and handled in a batch when the RTU runs. Timer expirations can also be handled using RTUs. In fact, the TCP timers in the kernel are handled by a timer routine that runs periodically and can be easily implemented in user space as an RTU. Since these events occur only in 200 msec and 500 msec intervals, there is no need to batch them.

7.2 Shared Memory Mbuf Facility and Data Movement

We have implemented a "wired down" shared memory mechanism between each process and the kernel to store network data. The shared memory region is called the communication area (CAR) of a process. The CAR is very similar to the *pool* model used in [11] and to the *communication segments* in U-Net [12]. However it does not need any support from the network adaptor hardware. The CAR is of fixed size and is allocated in a contiguous portion of the process address space. The CAR is also mapped to a region of kernel memory so that the kernel can setup DMA between the adaptor into the physical pages of the CAR. Since the pages in the CAR are wired down, the kernel can setup DMA in the adaptor interrupt service routine. All connections within the process share the pages in the CAR. The pages in the CAR are partitioned into transmit and receive portions. The kernel manages free buffers of the CAR for receive, and the user process manages the free buffers for transmit. This eliminates the need for synchronization between the kernel and the user to manage the free list of buffers. Figure 8 shows two processes with their CARs that are also mapped to kernel address space. The *management page* of the CAR contains information about transmit and receive queues for each VCI in the process.

Most of the TCP/IP protocol implementations use the *Mbuf* facility [15]. To use existing TCP/IP code with minimal modifications, we ported the *mbuf* facility to run in user space, and use the memory in the CAR to allocate the buffers used by *mbufs*. Accordingly, pages in the CAR are allocated among normal *mbufs* and *cluster mbufs* [15] where our cluster is equal to a page size (4KB).

Since RTUs are guaranteed to run every period, the size of the CAR required to avoid overflows can be kept small. This advantage is not present in other user space protocol implementations.

Trade-offs with wired pages Wiring down pages is necessary to move data directly between adaptor and user space but it comes at the cost of potentially lowering performance of other application processes. We do not consider this to be a problem in our 32 MB machines. The number of wired pages in a CAR is proportional to the connection bandwidth, and inversely proportional to the period of the RTU that processes the data. Since the total incoming bandwidth is limited to 155 Mbps, and RTU periods can be as low as 10 msec, the upper bound on the number of of wired pages in the system is within 1 MB.

7.2.1 Lock Free Buffer Operations

All *mbuf* network I/O and memory management operations that require synchronization between the user protocol and the kernel are implemented using lock free data structures [10]. The basic data structure is a one-reader-one-writer FIFO queue to pass buffers between the user protocol and the kernel. The *management page* has two such queues—one used by the ATM driver to return transmit *mbufs* to the CAR after they have been transmitted(*txdone_q*), and the other for the RTUs to return receive *mbufs* to the driver after they have been consumed by the application (*rxdone_q*). For each VCI, the *management page* also has a *tx_q* that contains PDUs for transmission, and an *rx_q* that contains the received PDUs. A queue consists of an array, with a *read* pointer and a *write* pointer that are indices into this array. The operation of the queue is explained in [10]. One important feature

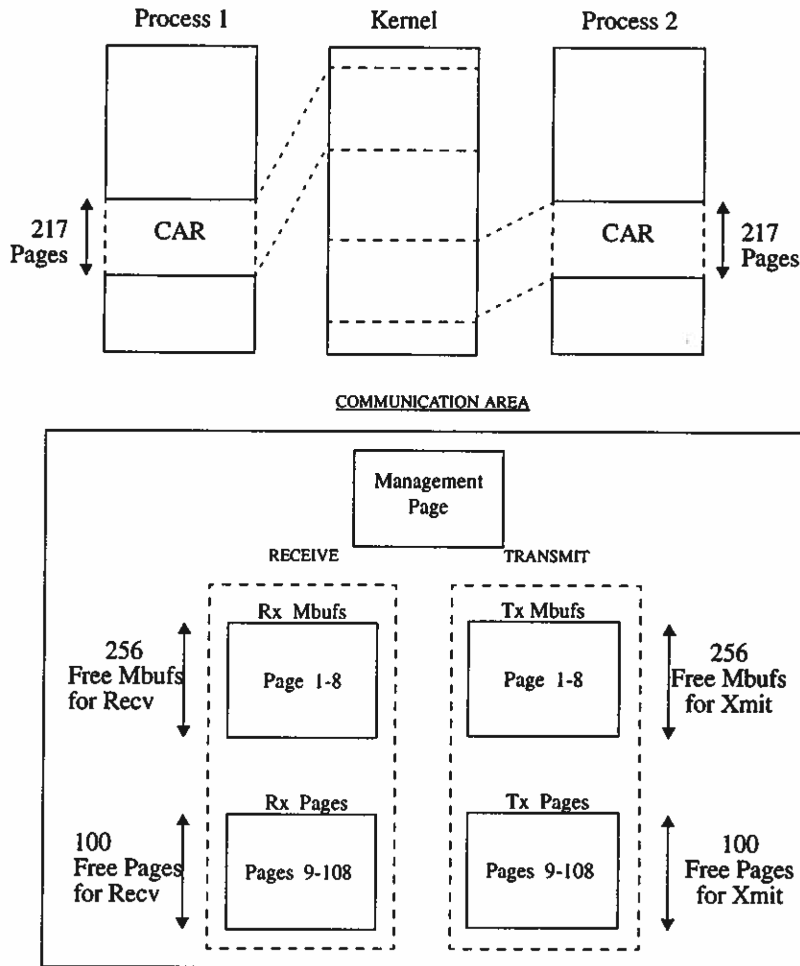

Figure 8: Communication Area

of our implementation is that we store indices of MBUFS rather than their addresses in the array. Since the CAR is contiguous in memory, these indices can be converted into either user addresses or kernel addresses without the need for any VM operations.

An important advantage of RTU scheduling is that there is no need for synchronization among RTUs in a process for updating queues that are shared among all connections. These include the *txdone_q*, *rxdone_q*, and the free list of transmit *mbufs* and cluster *mbuf* pages. Thus, the use of lock free queues eliminates locking between user and kernel, and the cooperative scheduling implemented by RMDP eliminates locking between RTUs that share CAR resources.

7.2.2 Data Movement Without Copying and System Calls

Using the user level *mbuf* facility, data movement can occur without any system calls, cross domain copying, or VM operations. In the send direction, the protocol creates an *mbuf* chain containing the packet. It then places the index of the head of the chain in the send queue corresponding to its VCI. Once a batch of packets have been placed in the queue, the sending RTU yields the CPU by making a system call. For each *mbuf* index in the queue, the kernel converts it into the corresponding kernel virtual address using simple pointer arithmetic. The resulting *mbuf* chain is then enqueued at the adaptor for transmission. The transmission completion interrupt returns the mbuf chain in the *txdone_q*.

In the receive direction, essentially a similar operation is performed except that the ATM driver has the ability

to separate headers from data. The header size (typically 52 for TCP/IP with the TCP timestamp option) is looked up based on the incoming VCI. If the data portion is a multiple of the page size, it is placed in pages using DMA and is associated with a *cluster mbuf*.

From the above description, we can see that the use of shared memory along with batching eliminates system calls to initiate data movement in both the transmit and receive directions. Data movement and scheduling occur in a single context switch. The use of indices rather than addresses obviates the need for VM operations in the critical path.

Finally, we make an observation regarding *zero copy* operation. Since the ATM adaptor supports cache coherent DMA into *mbufs*, *zero copy* operation comes for free if the application can accept data in the *mbuf* format. If there is a need to place data in contiguous memory, then we must either incur one extra copy or remap data pages. If page size segment sizes and header-data separation are used, remapping can be done without any further copying.

7.3 Handling Process Termination

By integrating scheduling and protocol processing, the RTU mechanism extends process lifetimes to connection lifetimes. For example, a process with a user space TCP connection should be prevented from terminating before it closes its connection. In [11] a separate process is required to do this for each connection and in [17] connection state is copied to the kernel on process exit.

In our framework, the kernel uses its knowledge of active RTUs to decide when to deallocate a process. When a process terminates, the kernel sets a *abort* field in the shared memory (section 4) of each RTU in the process. At this point, the process is still not deallocated and RTU invocations are allowed to continue. Each RTU handler is expected to check the *abort* field, and if it is set, it initiates connection abort procedures with its peer. Once the connection is freed, the RTUs terminate. The kernel waits until all RTUs terminate, and then deallocates the process. This procedure is protocol independent and has been implemented for TCP.

8 Example Implementation—TCP

We have implemented the TCP/IP protocol stack using the general mechanisms described above. We are using Pentium based platforms with NetBSD and 155 Mbps ATM interfaces in an ATM testbed. The relevant features of our TCP/IP implementation are the following.

- Input, output, and timer processing for a connection is done using separate RTUs that are created during connection setup. A protocol control block that contains a pointer to an *inpcb* structure is created and associated with each RTU. The PCB has pointers to the send and receive queues for its VCI. Upto 6 TCP connections can be allocated in a process. More VCIs can be supported by increasing the size of the CAR *management page*.

- The RTU handler for input processing dequeues IP packets from its VCI queue and does IP and TCP processing. The *mbufs* are then enqueued at the user level *socket* queue to be consumed by the application. The output processing RTU handler looks at the send queue in the socket structure and calls the *tcp_output* function with the number of segments to send in the batch[2]. The timer processing RTU is upcalled every 100 msec. It then calls the functions required to process the 6 TCP timers. All handlers are written to check for yield requests and abort indications from the kernel after each TCP segment is processed.

- Connection setup in user space TCP implementations are usually done by a central server. We use the existing TCP implementation in the kernel to do connection setup. We use a variant of the standard *accept* and *connect* system calls to setup the TCP connection on behalf of the user process. The connection state is then *upcopied* to the user space *inpcb* structure that is allocated by the user level TCP routines. Important state information include the port number, initial sequence numbers for send and receive, the window sizes, TCP options, and window scale factors. We currently allocate VCIs for each connection from a pool of already created VCIs. We plan to use a more general and robust scheme when ATM signaling becomes available.

[2]The number of packets actually sent depends on the state of the connection and is decided by the *tcp_output* function

8.1 Experiments and Expected Results

We had hoped to include experimental results in this version itself. However we missed this by a couple of weeks. The objective of our experiments are to compare our TCP implementation with the existing kernel implementation. We are confident that we will have experimental results that will demonstrate that

- Our user space TCP implementation will outperform existing user space implementations on similar platforms.
- The RTU based TCP will deliver higher throughput than the kernel implementation. Since we have eliminated one copy operation, this goal will likely be achieved.
- The RTU based TCP implementation will provide throughput guarantees even in the presence of background system load, and allow bandwidth sharing, and partitioning among TCP connections, in the same way as our earlier experiments with UDP suggests.

9 Conclusions

We have presented a new mechanism to implement protocols in user space with QoS guarantees. We observe that current process scheduling is inappropriate for providing QoS guarantees. We have also given several reasons why existing mechanisms such as RT-threads will lead to inefficiencies. We have argued that the RTU mechanism is both necessary (from an efficiency standpoint) and sufficient (as an OS abstraction) for providing QoS guarantees for applications as well as protocol processing. We have implemented the RTU mechanism and the RMDP scheduling scheme in the NetBSD OS and demonstrated its efficient performance. Our experiments show clearly that RTU mechanism provides throughput guarantees, bandwidth sharing and partitioning among connections, even in the presence of background system load. We also highlight important performance issues in user space protocol implementations. We show that system support is required to reduce the high cost of data movement and context switching in user space protocol implementations. Using our implementation framework, we can eliminate all system calls, VM operations and data copying operations in the packet processing path while providing QoS guarantees. We also discuss the benefits of integrating protocol processing and scheduling. We strongly believe that our methodology for user space protocol implementation is the most efficient that we know of for providing QoS guarantees within the endsystem.

References

[1] Author(s), "Efficient Quality of Service Support in Multimedia Computer Operating Systems," Technical Report XXCS-94-26, Dept. of Computer Science, XXX University, September 1994.

[2] Author(s), "Real-time Upcalls: A Mechanism to Provide Real-time Processing Guarantees," Technical Report XXCS-95-06, Dept. of Computer Science, XXX University, September 1995.

[3] Author(s), "Bringing Real-time Scheduling Theory and Practice Closer for Multimedia Computing," Accepted at ACM SIGMETRICS, May 1996.

[4] Author(s), "Quality of Service Support for Protocol Processing Within Endsystems," High Speed Networking for Multimedia Applications, Kluwer Academic Publishers, 1995.

[5] Chen, J.B., et. al., "The Measured Performance of Personal Computer Operating Systems," 15th *ACM Symposium on Operating Systems Principles*, Dec. 1995.

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

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

[8] Clark, D.D., Shenker, S., Zhang, L., "Supporting Real-time Applications in an Integrated Services Packet Network: Architecture and Mechanism," *ACM SIGCOMM*, Aug, 1992, pp.14-26.

[9] Dannenberg, R.B., et. al., "Performance Measurements of the Multimedia Testbed in RT-Mach" Tech. Rep. CMU-CS-94-141, School of Comp. Sc., Carnegie mellon University.

[10] Druschel, P., Peterson, L.L., Davie, B.S., "Experiences with a High-Speed Network Adaptor: A Software Perspective," *ACM SIGCOMM*, Aug. 1994, pp. 2-13.

[11] Edwards, A., Muir. S., "Experiences Implementing a High Performance TCP in User Space" *Proc. ACM SIGCOMM*, 1995, pp.196-205.

[12] Eicken, T.V., et. al., "U-Net: A user Level Network Interface for Parallel and Distributed Computing," 15th *ACM Symposium on Operating Systems Principles*, Dec. 1995.

[13] Govindan, R., Anderson, D.P., "Scheduling and IPC Mechanisms for Continuous Media," 13th *ACM Symp. on Operating Systems Principles*, 1991.

[14] Khanna, S., et. al., "Realtime Scheduling in SunOS5.0," *USENIX*, Winter 1992, pp.375-390.

[15] Leffler, S.J., et. al., "The Design and Implementation of the 4.3BSD UNIX Operating System," Addison Wesley Publishers, 1989.

[16] Liu, C.L. and Layland, J.W., "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *JACM*, Vol. 20,No. 1, January 1973, pp. 46-61.

[17] Maeda, C., Bershad, B.N., "Protocol Service Decomposition for High Performance Networking," 14th *ACM Symp. on Operating Systems Principles*, Dec 1993, pp. 244-55.

[18] Marsh, B.D., et. al., "First Class User Level Threads," *ACM Symposium on Operating Systems Principles*, Oct. 1991, pp.110-21.

[19] Mogul, J.C., et. al., "The Packet Filter: An Efficient Mechanism for User Level Network Code," 11th *ACM Symposium on Operating System Principles*, Nov. 87, pp.39-51,

[20] Sha, L. et. al., "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, Vol.39, No.9, Sep 1990, pp.1175-1185.

[21] Thekkath, C.A., et.al., "Implementing Network Protocols at the User Level," *ACM SIGCOMM*, Sep. 1993, pp. 64–72.

[22] Tokuda, H., Nakajima, T., Rao, P., "Real-Time Mach: Towards Predictable Real-time Systems," *USENIX Mach Workshop*, Oct 1990.

[23] Zhang, L., et. al., "RSVP: A New Resource Reservation Protocol," *IEEE Network*, Sep. 1993, pp.8-18.