TCP/IP Implementation with Endsystem QoS

Sherlia Y. Shi
Gurudatta M. Parulkar
R. Gopalakrishnan

WUCS-98-10

April 22, 1998

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

TCP/IP Implementation with Endsystem QoS

Sherlia Y. Shi Gurudatta M. Parulkar R. Gopalakrishnan
Department of Computer Science AT&T Research Lab
Washington University in St. Louis

Abstract

This paper presents a *Real-time Upcall* (RTU) [1] based TCP/IP implementation that guarantees throughput for continuous media applications and ensures low latency bounds for interactive applications. RTU is an endsystem rate-based scheduling mechanism that provides *quality of service* (QoS), in terms of CPU cycles, to applications. We restructured the existing NetBSD TCP/IP implementation to exploit the RTU concurrency model and to provide predictable performance.

Our experimental results show that on two 200 MHz NetBSD PCs connected by a 155Mbps ATM link, the RTU based kernel TCP/IP implementation provides excellent throughput guarantees for periodic connections regardless the system or network load. The round trip time (RTT) for low-delay connections with message size of 1 KB is typically as low as 600 micro seconds, and only increases slowly with increasing system load. Another important result is that this performance is preserved even when all three type of connections coexist in our testbed: the periodic connection is guaranteed its share of bandwidth, the low-delay connection achieves low RTT of 1.2 msec, while the best-effort connection still makes steady progress.

1. Introduction

The emergence of high-speed networking provides opportunities for a variety of multimedia applications to communicate across the internet. These applications typically have end-to-end *quality of service* (QoS) requirements to achieve better performance: for example, continuous media (CM) applications need to reserve network and CPU bandwidth during their entire execution time, and interactive applications need to minimize the queueing latency in both the network and endsystems. Providing QoS for multimedia applications is essentially an end-to-end issue but can be decoupled into two parts: QoS provision in networks and in endsystems. Protocols such as RSVP [2] have been proposed to manage network resources efficiently and to support different classes of application specific QoS. Similarly, operating systems must manage the endsystem resources to meet the application QoS requirements in terms of guaranteed CPU bandwidth and minimum queueing delays.

1.1. Endsystem Quality of Service

The key issues in providing QoS in endsystem include: (1) policies that classify application types based on their service requirements; (2) an efficient scheduling mechanism that enforces CPU bandwidth guarantees; (3) separate data paths for protocol processing among connections to minimize interferences and to prevent possible priority inversions; (4) early packet demultiplexing so that resource competitions, such as CPU and memory scarcity, can be resolved promptly.

We classify multimedia applications into three categories:

- **Continuous media applications**, such as audio/video applications, typically exhibit periodic data generation and require constant time to process an application data unit (ADU). The quality of this type of application is mainly dependent upon how their periodic nature can be preserved during the network data transfer and endsystem data processing. To achieve satisfactory performance in terms of predictable throughput and jitter variation, the operating system must then allocate adequate CPU cycles every processing period. In this paper, we refer to this type of connection as the *guaranteed-bandwidth* connection or *periodic* connection for short.
- **Interactive applications**, such as CORBA RPCs, use relatively small messages but are highly delay-sensitive. For this type of application, the OS must minimize the scheduling overheads and queueing delay at network interfaces. We refer to this type of connection as the *low-delay* connection.
- **Bulk data dissemination applications**, such as file transfer, do not have specific bandwidth or latency requirements, and are commonly referred to as *best-effort* connections. The OS must ensure that the existence of best effort connections will not degrade the quality of other higher priority applications, while also avoid CPU starvation of the best effort connection. A typical solution of this problem is to use admission control mechanism to reserve a certain amount of CPU bandwidth for best-effort connections.

1.2. Previous work and our motivations

Real-time Upcall (RTU) is a novel scheduling mechanism that uses a priority-based scheduling policy to allocate CPU cycles to applications in a periodic guaranteed manner [1]. Furthermore, RTU reduces the cost of context switches by adopting a delay preemption policy. The design and implementation details of the RTU mechanism are deferred to Section 3. RTU has been used to implement the TCP/IP protocol suite in the user space [3], and experimental results show that the restructured protocol processing reliably provides QoS guarantees.

However, for reasons such as efficiency and security, protocol suites continue to be kernel-resident. Applications and middleware, such as CORBA, often rely on the standard socket system call interface. It is therefore desirable to adapt the RTU mechanism to the kernel space and to integrate the RTU mechanism with kernel-resident protocols.

The data processing for a network connection consists of kernel protocol processing and application-level data processing. These two parts are coupled by socket data buffering, and their computation time in terms of data units should be balanced to achieve higher performance. Our goals are therefore to provide a monolithic scheduling mechanism across the user and kernel space, to provide QoS on a per connection basis, and to utilize CPU bandwidth efficiently.

Our experimental results show that on two 200 MHz NetBSD PCs connected by a 155Mbps ATM link, the RTU based kernel TCP/IP implementation provides excellent throughput guarantees for periodic connections regardless of the system or network load. The round trip time (RTT) for low-delay connections with a message size of 1 KB is typically as low as 600 micro seconds, and only increases slowly with increasing system load. Another important result is that this performance is preserved even when all three type of connections coexist in our testbed: the periodic connection is guaranteed adequate bandwidth, the low-delay connection achieves low RTT of 1.2 msec, while the best-effort connection still makes steady progress.

1.3. Outline of the paper

The rest of the paper is structured as follows. In section 2, we discuss other work related to providing QoS in endsystems. We then give some necessary background on the RTU approach in section 3. In section 4, we

identify the problems involved in QoS support for protocol processing and the integration of RTUs with the kernel TCP/IP implementation. In section 5, we presents our RTU based TCP/IP architecture. Implementation details and experimental results are given in section 6 and 7. Section 8 concludes the paper.

2. Related Work

Earlier work in [4, 5] identifies the time constraints and computation requirements of continuous media applications, and introduces CPU scheduling mechanisms to support real-time communication and computation services including data computations, I/O services, and communication protocol processing. More recent work in OS scheduling support for multimedia applications addresses a richer set of QoS requirements [6, 7], where the support for real-time applications is integrated with the support for interactive and conventional applications.

Protocol processing in multimedia systems constitutes a relative large part of the total computation time required by multimedia applications. Lots of work has been done in supporting real-time protocol processing. In [8, 9], real-time threads are used for prioritized protocol processing. Each thread handles a different priority class and the priority of the thread matches the priority of the packets it handles. The disadvantage of using multithreading is its context switching overheads due to the preemptive scheduling required for real-time threads. The excessive context switches coupled with data locking contentions lead to poor performance and low system utilization. Furthermore, real-time OS, such as Solaris, only provides a fixed number of priority levels and is not geared towards the periodic scheduling of various multimedia applications.

Another approach of supporting different priorities in protocol processing is to implement the protocol stack as a user library [3, 10, 11, 12, 13]. As a result, communication protocol processing becomes an extension of process threads and can be treated as fully preemptive blocks by the OS scheduler. The advantages of implementing protocol in the user space are implementation flexibility, easiness of debugging and modification, and allowing application specific optimizations. However, user-level protocol implementations impose concerns in security issues and its processing efficiency, since the integration of application-level protocols with the rest of the operating system, which provides low level system abstractions, needs to be properly guarded and introduces extra overheads.

The Scout OS [14] uses the notion of PATHS to associate resource requirements with the flow processing components. In [15], a similar idea of *process-per-channel* is used. QoS for multiple channels are provided via appropriate CPU scheduling of channel handlers. However, work in [15] only provides coarse grained QoS support, in that flows are classified as real-time or best-effort, and FIFO scheduling is used within each class.

Although the theory of providing QoS is well recognized, the repetitive and iterative nature of protocol processing is clearly observed, and the mechanisms of real-time scheduling are maturely developed, we are not aware of any existing work that closely integrates theory and practice to improve performance of standard protocols such as TCP/IP. We believe that our work is the first to support coexisting high-throughput and low-latency TCP/IP connections in an endsystem.

Our work focuses on endsystem QoS guarantees. The iterative nature of networked application implies the data unit boundary to be a well-defined preemption point, where no data locking is required. We restructured the TCP/IP protocol using cooperative scheduling in which the lower priority flow explicitly yields CPU at the message boundary. The constant protocol data unit (PDU) processing time limits the possible priority inversion and results predictable performance. Our implementation is based on the NetBSD operating system and reuses the existing BSD TCP/IP code. This retains the TCP dynamic behavior and allows us to more easily compare with the performance of existing protocol implementation.

3. Background

The solution we presented in this paper of providing endsystem QoS is based on the *Real-time Upcall* (RTU) mechanism. RTU is a real-time concurrency scheduling mechanism that allows applications to explicitly register a code segment (an RTU handler) with associated QoS parameters. The RTU scheduler uses *Rate-monotonic with Delayed Preemption* (RMDP) policy that exploits the iterative nature of protocol processing to reduce context switching overhead and to increase endsystem efficiency.

More details of the RTU mechanism can be found in [1]. In this section, we briefly introduce the necessary background information by first examining how the CPU requirements of protocol processing can be characterized with a periodic processing model. We then explain the RMDP scheduling policy and its implementation with RTU in user space.

3.1. The Periodic Processing Model

A multimedia application has two important properties: the constant time to process a data unit and the maximum delay of a data unit that can be tolerated without noticeable performance degradation. Although QoS requirements vary among applications, they can be realized using the periodic processing model shown in Figure 1.

Numerous proposals have been made for multimedia computing processing models. [16] We choose the periodic model because it is easy to implement and adequate for analysis.

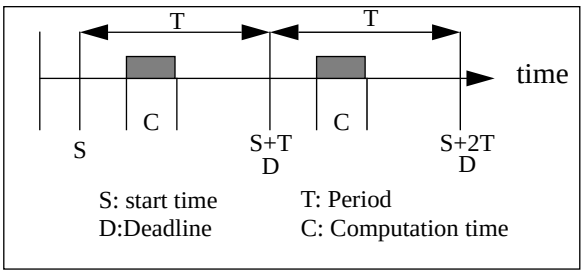

Figure 1: Periodic Processing Model

Using this model for real-time protocol processing, we define the **period** *T* of a real-time data stream to be the **computation time** *C* plus the maximum delay that can be tolerated either due to application buffering or by the user's perception. The **period** *T* can also be viewed as setting a deadline for processing the current data unit. The processing time for a protocol data unit (PDU) is typically constant, therefore the **computation time** *C* is decided by the size of a PDU and can be expressed in number of PDUs. Consequently, the value of $\frac{C}{T}$ represents the throughput required by the application.

To further generalize this model, we can view *C* as the time needed to process multiple PDUs during a single period. This generalization is called "batching" and allows an application to specify a flexible bandwidth requirement. The scheduler ensures that at least one, and up to B_p, PDUs will be processed in a period, depending on the system load.

3.2. The RMDP Scheduling Policy

A complete analysis of *Rate-Monotonic with Delayed Preemption* (RMDP) scheduling can be found in [17]. We illustrate this policy only briefly to aid understanding of some of our experimental results.

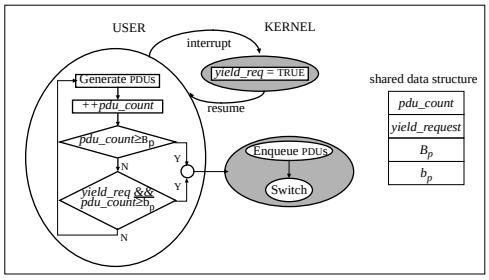

Figure 2: The RMDP Scheduling Algorithm

Figure 2 shows the RMDP scheduling algorithm. It is designed to exploit the iterative nature of multimedia processing by treating each iteration as an atomic operation. Preemptions only occur at the boundaries of data units where no data locking is required. Hence, RMDP avoids the priority inversion problem that commonly happens in a fully preemptive operating system, and reduces the cost of context switching.

In RMDP, the priority of a task is inversely proportional to the task period. Upon the arrival of a higher priority task, the scheduler does not preempt a lower priority task immediately, but rather notifies the lower priority task of this event through a shared communication data structure. Cooperation of all RTUs is expected so that after the notification of an arrival of a higher priority task, the lower priority task explicitly yields the CPU at the preemption point and exits the function.

In our current implementation, we assume cooperative RTUs yield at iteration boundaries, but the QoS enforcement should also consider regulating a misbehaving RTU handler by ensuring that: (1) an RTU handler invocation does not run past its stated computation time; and (2) an RTU yields the CPU when it is requested to do so.

3.3. The Real-Time Upcall Approach

The upcall mechanism is a well recognized operation for efficient protocol implementation [18]. A *Real-Time Upcall* (RTU) is similar to an upcall with the additional feature that an RTU handler function gets a guaranteed share of the CPU over periodic time intervals.

RTU has been implemented in the NetBSD operating system, and its overall organization is shown in Figure 3. The RTU facility is layered on top of the normal UNIX process scheduling mechanism so that a runnable RTU takes precedence over a runnable UNIX process. When no runnable RTUs are present, the system reverts to normal process scheduling.

An application process uses the RTU system call API to register a function as an RTU handler with the QoS parameters: execution period and computation time in terms of data units. RTUs are scheduled using the RMDP policy as we presented in the previous section. At the beginning of an RTU period, the scheduler inserts the RTU into the run queue and orders the run queue by RTU priority with the highest priority RTU at the head of the queue. An RTU handler is upcalled when it reaches the head of the run queue. An RTU handler exits under one of two conditions: it has finished processing all current data units ($pdu_{count} \geq B_p$) or a yield request has been posted and it has finished processing the minimum data units (b_p) in this iteration. In either case, an RTU handler explicitly exits after its critical section, avoiding the otherwise necessary data locking, and reducing the cost of context switching. The system state is restored after the handler returns.

To support low-delay streams, we also implemented *reactive* RTUs. The activation of the reactive RTUs is event-based, either on a packet arrival event or on a system call, so the endsystem scheduling delay is eliminated. However, such unscheduled processing may cause unpredictable behavior and may jeopardize the system normal scheduling, so techniques must be used to prevent reactive RTUs from monopolizing the

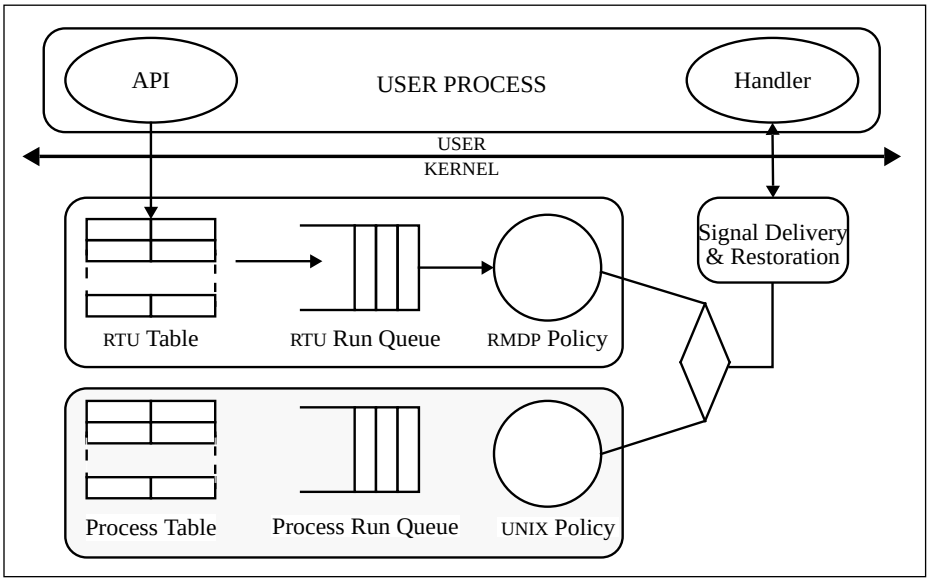

Figure 3: RTU Organization

CPU [19]. In our experiments, reactive RTUs are used only for small messages with sporadic invocations so that the CPU time spent on reactive RTUs is small enough not to disrupt the system scheduling.

4. Problem Statement and Solution Outline

In this section, we investigate the issues in the existing NetBSD operating system and the BSD TCP/IP protocol suite structure that obstruct the provision of QoS for multimedia applications. We also show how the integration of the RTU mechanism with the TCP/IP protocol suite addresses these problems.

4.1. The operation of existing UNIX TCP/IP protocol stack

The asynchronism of protocol processing mostly occurs in the protocol input data path, where CPU controls are transferred upon network events such as packet arrivals and timer expirations. The typical BSD TCP/IP protocol stack is illustrated in Figure 4.

We briefly describe the sequence of operations upon a packet arrival. When packet arrives at the network interface, an hardware interrupt occurs and the CPU jumps to the device interrupt rouinte for link layer packet processing. Then, the adaptor queues the packet according to its link layer header and posts a software interrupt. The interrupt handler orders all software interrupts according to their priorities and schedules the interrupt routine when its priority level is reached. For TCP/IP protocol processing, the software interrupt routine is the IP input function. The packet is then processed through the TCP/IP protocol stack and enqueued into its socket queue. When the process that owns the socket is scheduled by the UNIX process scheduler, it reads the message from the socket and does application specific operations.

4.2. Problems

We identify the problems that need to be solved to achieve efficient real-time protocol processing with QoS guarantees as follows.

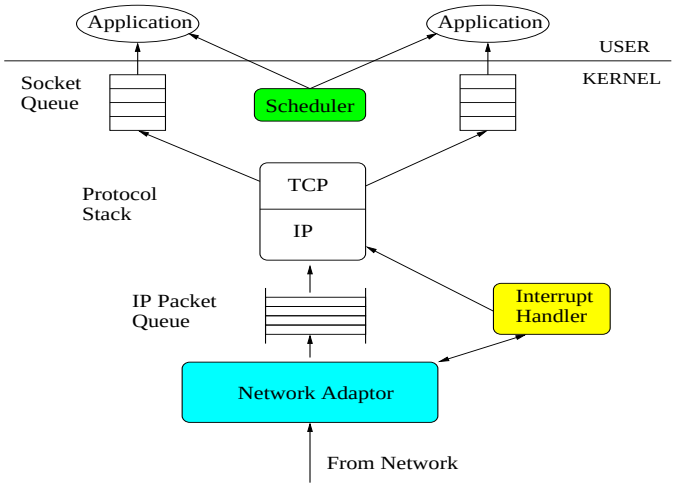

Figure 4: UNIX TCP/IP protocol stack

- **Concurrency control between protocol and application data processing** For security reasons, operating systems usually execute kernel-level code in a protected mode, and enforce kernel- and user-level code boundaries using hardware protection facilities [20]. As the kernel provides basic system facilities, controls user programs to access underlying hardware (e.g. network links, disks) and software constructs (e.g. protocols, filesystems), it executes at a higher priority than the user-level code. Consequently, kernel-resident protocols are privileged over the application programs and can preempt the application upon the packet arrivals. For a TCP connection, however, protocol processing and application level data processing are tightly coupled through the socket layer buffering. If an application does not have enough CPU cycles to promptly read from or write into the socket buffer, the TCP protocol will eventually stall and wait for the application, which results in unpredictable performance.

- **Asynchronous event processing** Processing of network and protocol events such as packet arrivals is usually triggered by interrupts and happens asynchronously to the user's processes. Interrupt driven scheduling disrupts priority based scheduling schemes resulting in priority inversion. For instance, the arrival of a packet at the network interface initiates the protocol processing for a connection, even though the currently running process may not be the process that owns the connection. Therefore, processing of packets belonging to a lower priority connection interferes with the execution of a higher priority process, and leads to QoS violations.

- **Packet processing order** In the current operating system model, resources such as IP packet queue are shared among all connections and served on a first-come-first-served basis. When a large packet of a lower priority connection is queued in the front of the FIFO queue, it delays the processing of all subsequent packets which may belong to higher priority connections and leads to priority inversion.

4.3. Solutions

We respectively address the above problems as below:

Concurrency Control

To ensure that all data manipulations of a multimedia stream take place within the time limit, we need to schedule both the kernel space protocol processing and the user space data operation consistently with a guaranteed share of CPU time. We have described the RTU approach in the user space, in which an RTU handler

is a user specified code segment. The extension of RTU functionality to kernel space is straightforward: to schedule both user and kernel functions as RTU handlers[1], and to allow preemptions between user and kernel RTUs so that the connection priorities are preserved.

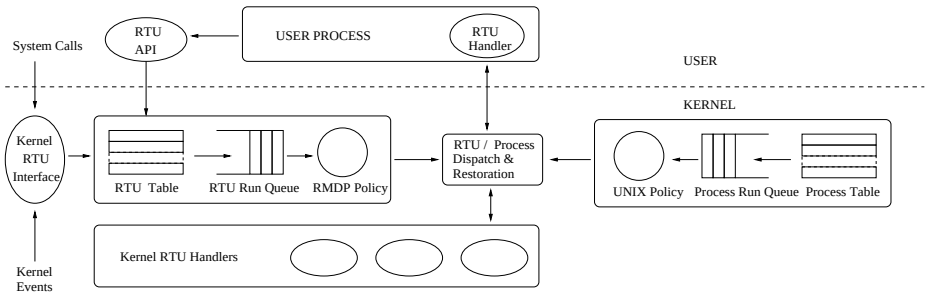

Figure 5: Extended RTU Organization

The extended RTU organization is illustrated in Figure 5. Kernel RTU handlers are protocol functions residing in the kernel's address space. Through the kernel RTU interface, they can be operated either by kernel events or by applications making system calls. For instance, when the application opens an RTU based TCP connection, two kernel RTUs will be created for input/output protocol processing respectively. In the other case, a reactive RTU can be created upon a packet arrival at the network interface. Both kernel- and user-level RTUs share a single RTU run queue and scheduling domain. The unified scheduling of user-level and kernel-level RTUs provides a means for CPU sharing between application data processing and protocol processing without harming the kernel's privileged mode. As we shall see in Section 7, this load balancing over the socket buffers ensures the throughput of periodic connections.

Asynchronous Events

To retain the atomicity of processing a data unit, we disable the software interrupt posted when the network interface delivers a packet to the protocol layer. Instead, the packet is processed only when the RTU associated with the connection is scheduled.

- **Low-delay** connections use high priority reactive RTUs. Upon a packet arrival, a reactive RTU is immediately inserted into the RTU run queue according to its priority and will be scheduled when it moves to the head of the queue.
- **Guaranteed-bandwidth** connections use periodic RTUs and each of them is associated with a timer. A periodic RTU is inserted into the run queue only when its timer expires, so a just arrived packet may need to wait until the beginning of the next period.
- **Best-effort** connections use reactive RTUs with the lowest priority. When a packet arrives, a reactive RTU is inserted into the run queue but due to its low priority, it is usually appended to the run queue.

RTUs in the run queue are scheduled by the RMDP policy. Due to the delayed preemption, we avoid extra context switches and achieve efficiency in protocol implementation.

It must be noted that events, such as a packet arriving at the network interface or other device inputs, are handled by the device interrupt routines at a high system priority. Moreover, traps such as page fault may also occur during an active RTU, causing the RTU to block. Although these high priority kernel activities may disrupt the real-time scheduling of RTUs, they are usually not prolonged and do not cause substantial performance degradation.

[1]We retain the name of RTU for historic reasons, although in the kernel space, a function is simply *called* rather than *upcalled.*

Packet Queueing

To provide per flow QoS, early packet demultiplexing is needed to minimize the interferences among data streams. The connection-oriented *Asynchronous Transfer Mode* (ATM) provides a one-to-one mapping between connections and *virtual circuit identifiers* (VCIs). By providing per-VC packet queues at the network interface, we are able to demultiplex packets at the link layer and schedule CPU bandwidth for protocol processing according to the QoS associated with the flow.

The close integration of RTU scheduling with protocol processing allows us to fully exploit the system resources and network bandwidth. We are able to provide guaranteed bandwidth for periodic media data transfer, and low delay for request-response traffic. In the rest of this paper, we describe the design and implementation details of restructuring the TCP/IP protocol suite to exploit the RTU mechanism and to provide QoS guarantees for multimedia applications.

5. The Architecture of RTU based Protocol Processing

In this section, we first introduce our QoS specifications by applications, then we describe the RTU based protocol processing model and show how QoS is enforced in our system to achieve predicatable performance.

5.1. QoS Specification

In our system, each concurrent task in protocol processing and application data processing is typically implemented as an RTU. Each RTU is associated with application specific QoS parameters which are specified as a two-tuple $< streamtype, priority >$, and are inherited by the protocol processing RTUs of the same connection. The two-tuple QoS parameter indicates the type of the stream, either *low-delay, guaranteed-bandwidth, or best-effort*, and the urgency of the stream within the same stream type. For example, a CORBA RPC server handles requests from all clients as low delay requests, but may grant some clients as more important than others by assigning a higher priority to these clients' connections.

The QoS specification is then mapped to the RTU scheduling parameters during the RTU creation. The *streamtype* field is granted as the more significant part in RTU scheduling with the order of significance as *low-delay, guaranteed-bandwidth* and *best-effort*. The *priority* field differentiates RTUs of the same connection type, additionaly it also indicates the RTU run periods of the periodic RTUs. As part of the QoS specification, a periodic RTU also specifies its needed work load (computation time) in each period. This computation time is specified in terms of number of ADUs and passed to the RTU scheduler during the RTU initiation.

There are more sophisticated QoS flow specifications, which include other QoS parameters such as flow jitter, loss sensitivities, traffic shaping, etc., [21] and QoS adaptation schemes, which allow systems to dynamically raise or reduce QoS depending on resource availability and user requirements [22]. In this paper, we are mainly focusing on how QoS can be enforced in the endsystem based on our integration of protocol processing with real-time scheduling, rather than providing an entirety of QoS transport system. Therefore, we choose the simple two-tuple QoS flow specification for its sufficiency of our experimental demonstration.

5.2. The Protocol Processing Model

As we discussed before, to achieve QoS guarantees on a per connection basis, we need to provide separate data paths among connections such that each of them can be scheduled independently. In addition, both protocol and application data processing must be given an equal share, in terms of data units, of CPU time.

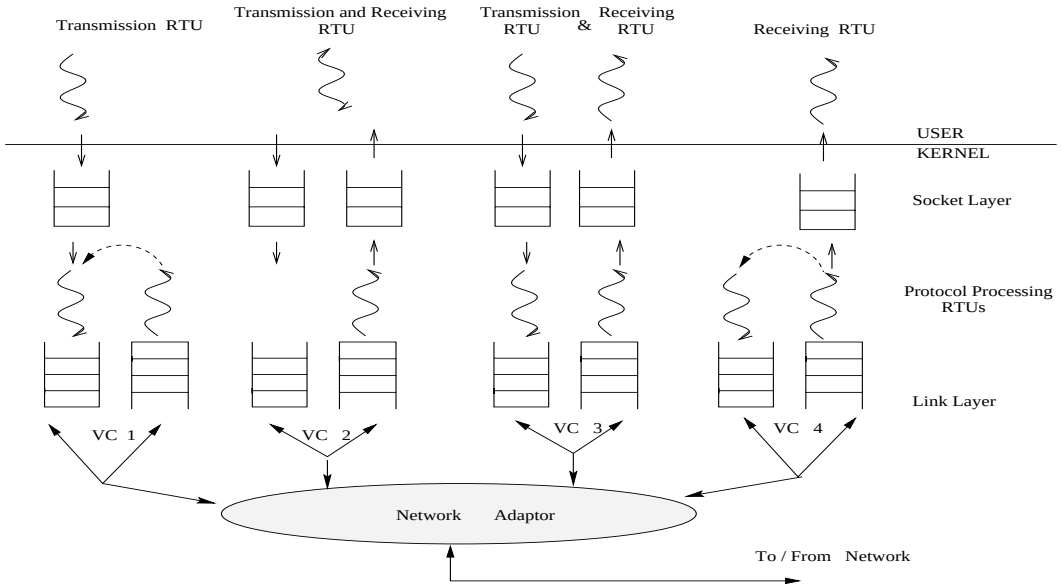

Figure 6: The RTU based protocol processing model

Figure 6 shows the typical RTU based protocol processing model. A user program can open a single RTU for uni-directional data transmission or reception. Or it can have a single RTU to handle both data inputs and outputs, such as in the client-server model, the server handles two-way synchronous calls within a single function. Another approach is to have two RTUs to handle data inputs and outputs separately, such as in a video conference, a host can receive audio data periodically, and transfer video image at the same time.

The data path within the kernel consists of a pair of RTUs, for input/output protocol processing respectively. The QoS specification associated with kernel RTUs are specified by the user program at the beginning of connection establishment. Each pair of RTUs is associated with a socket and a VC queue.

The sending direction When an output RTU is scheduled, it processes the data in the socket buffer through the protocol stack and passes the packet to the network interface for transmission. This sequence of operations is not interrupted (except hardware interrupts) by other system or other RTU activities.

The amount of data processed by an RTU is subject ot the TCP flow and ocngestion contorl mechanism. A periodic RTU sends as many PDUs as specifed by the application only if there is enough space in the TCP congstion window. To eliminate the network congestion effect and focus on endsystem QoS provisions, we use point-to-point link in our experiments so that the TCP window size is mostly affected only by the receiver's socket buffer space. Since both end hosts use same priority RTUs in data processing, the TCP protocol keeps a more consistent window size.

The receiving direction A packet arrival is processed and queued into its VC queue in the device interrupt context. An atomic operation of a receiving RTU includes: it moves data from the VC packet queue, processes it through the protocol stack. If data is only TCP control message, the receiving RTU does the corresponding control operation, e.g. sends an ACK back to the data source. If there is any application data, it enqueues the data into the socket receiving buffer. The application reads from the socket either using an RTU or not. In the former case, the RTU scheduler schedules the corresponding RTU to consume the data, while the UNIX scheduler schedules the process for data consumption in the latter case.

5.3. QoS Enforcement

Typically, periodic RTUs are used for guaranteed-bandwidth connections and reactive RTUs are used for low-delay and best-effort connections. A periodic RTU is created after the connection establishment, and is activated at the beginning of its period. A reactive RTU for input protocol processing is solely triggered by the packet arrival on its own VC, while for output protocol processing, it is activated when the application writes into the socket buffer.

QoS is enforced by two means: (a) the RTU scheduler always inserts the most urgent RTU, according to its two-tuple, at the head of the RTU run queue. Therefore, higher priority RTU runs before lower priority RTUs, regardless whether the higher priority RTU is for kernel protocol processing or for application data operation; (b) a low priority RTU is notified that it must yield after processing the current data unit when a higher priority RTU becomes runnable. These two mechanisms ensure the higher priority connection its requested CPU share, despite the CPU competition from lower priority connections.

The RTU based protocol processing model provides an independent data path for each connection while also allows preemption at the packet processing boundaries. It differs from a multithreaded protocol processing model in that: (a) it allows a broader range of QoS specification than the monotonic priorities of the real-time threads; (b) it provides a unified scheduling domain of user and kernel RTU handlers, eliminating possible priority inversion imposed by the user-kernel boundary protections; (c) it takes advantage of the iterative nature of networked multimedia applications by retaining the atomicity of processing a single data unit and allowing preemption only at data unit boundaries, thus, avoiding expensive operations such as data locking.

6. The RTU based TCP/IP Implementation

We have restructured the kernel TCP/IP protocol suite based on RTUs as a separate protocol domain in the NetBSD operating system. Since our implementation focuses on the TCP/IP data transfer path, the protocol control path including connection establishment, connection tear down, flow and congestion control, and other protocol optimizations are left unchanged [23]. In the rest of this section, we describe the operations of RTUs for the duration of a TCP connection (RTUTCP).

The Connection Establishment: An application creates a socket via the *socket()* system call with RTUTCP as the attached protocol. The application then creates user level RTUs to handle data transmission or receiving via the RTU system call interfaces [1] and associates QoS with the user level RTU. These QoS parameters are also recorded in the RTU control block (RCB) associated with this connection, for the reason we discussed below.

The application then calls *connect()* to establish a connection to the other end host. After the connection is setup successfully, the kernel level RTUs are created for input and output protocol processing and inherit the QoS parameter stored in the RCB. In addition, an ATM VC is resolved for this connection and stored in the RCB.[2]

The Data Transfer State: Figure 7 depicts the data path of our RTU based TCP/IP implementation on a native mode ATM network.

Depending on the type of the connection, an RTU is activated, i.e. inserted into the run queue, in the following two ways: (1) A periodic RTU is invoked every time its timer expires; (2) A reactive RTU is triggered either by a packet arrival event or a *write()* system call on the socket.

[2] Currently, a VC table is pre-established between hosts, and lookup is done based on destination address and connection QoS.

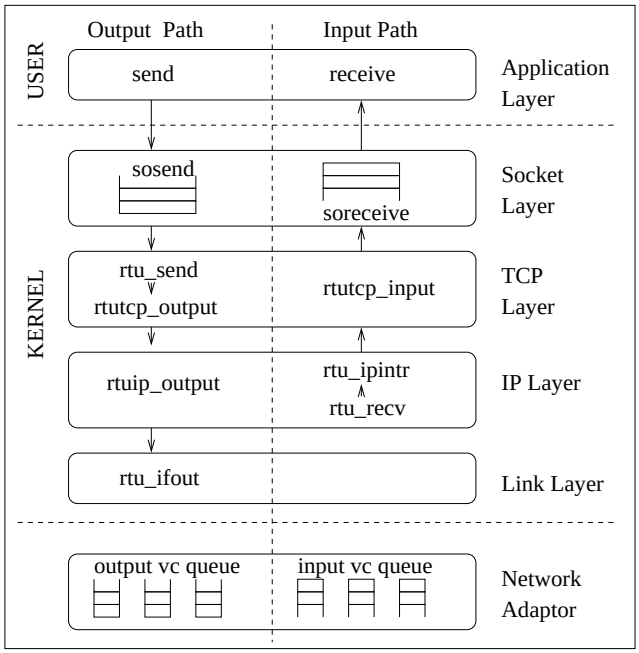

Figure 7: The RTU based TCP/IP Data Path

In Figure 7, the functions *rtu_send* and *rtu_recv* are the two RTU handlers for protocol processing. In the NetBSD OS, protocol processing is protected in the *softnet* system priority level for two purposes: to achieve exclusive access to shared data structures between input and output control threads without data locking operations; and to prevent possible protocol reentrance from other events such as timer expirations. Because *softnet* has a higher priority level than that of the timer event on which the RTU scheduler relies to schedule periodic RTUs, in order to properly update RTU yield requests, we need to lower the protocol processing priority at the RTU preemption points. Thus, we use *rtu_send* and *rtu_recv* as wrappers for the output/input protocol processing functions. The pseudo-code for the common control flow of an RTU handler is as below:

```
if ( stream type is low-delay or best-effort ) {
        do {
                s = splsoftnet();        /* raise the priority */
                rtutcp_output(tp);
                splx(s);                /* release the priority */
        } while ( !yield_request && sendalot )
} else {                /* stream type is periodic */
        do {
                s = splsoftnet();
                rtutcp_output(tp);
                splx(s);
        } while ( !yield_request && pdu_count < Bp && sendalot )
}
if ( yield_request )
        rtu_yield_status = RTU_YIELD;
else
        rtu_yield_status = RTU_EXIT;
```

An RTU handler processes one data unit at a time and then checks whether a *yield_request* has been posted to it. If so, the RTU handler explicitly yields the CPU by setting its *yield_status* flag to RTU_YIELD and exiting the function. The RTU scheduler reinserts the RTU into the run queue so that it can complete its remaining iterations at a later time. If a *yield_request* is not posted at every data unit boundry, then a periodic RTU processes its batch of data units ($pdu\_count < Bp$) and exits the function; while a reactive RTU processes as many data units as enqueued in the buffer (*sendalot*) and exits. These steps are repeated at the next RTU activation.

The output RTU handler *rtu_send* dequeues data from the socket buffer, calls *rtutcp_output* for TCP header processing and checksum computation, then it calls *rtuip_output* for IP packet processing and *rtu_ifout* for the driver to transmit the packet based on the VC number stored in the RCB. The input data path is similar: the driver demultiplexes packets based on their VC numbers, the input RTU handler *rtu_recv* then delivers the packet all the way upto the socket and if the application has previously registered a user level reactive RTU with this socket, *rtu_recv* fires it after enqueueing data in the socket buffer.

The Connection Shutdown: When the data transfer phase is over, a *connection_done* flag is set, and the RTUs should suspend themselves. The resources associated with RTUs such as data structures, timers and VCs are deallocated, and no future invocation of RTUs will be scheduled.

7. Experiments and Results

We have implemented the RTU based TCP/IP protocol suite in the NetBSD operating system. Our experiments are set up between two 200 MHz Pentium Pro PCs connected via a 155 Mbps ATM link with the Efficient Network Inc (ENI) network adaptor.

There are three important goals for our experiments: (1) the RTU scheduling mechanism is effective in that the RTU computation is decoupled from the other system load; (2) the RTU based TCP/IP implementation is efficient in that it fully exploits the CPU and link bandwidth; (3) the RTU based TCP/IP implementation is able to deliver application specific QoS on a per connection basis. We investigate the throughput performance of guaranteed-bandwidth connections and latency performance of low-delay connections with and without the presence of CPU and network competitions, and show how the results verify the efficacy of our RTUTCP implementation. All results discussed below are the measured average through multiple runs.

7.1. Throughput Performance of Periodic Connections

In this experiment, we demonstrate that the RTU based TCP/IP implementation provides user specific bandwidth guarantees for periodic data streams.

We open 3 RTU based TCP connections simultaneously with different RTU periods: 1, 2, 10 msec. Accordingly, with 8 KB size packets, the expected throughput of these three connections are: 62.5 31.25, 6.25 Mbps. In comparison, we use the **ttcp** program to measure the throughput performance of the normal BSD kernel TCP implementation. In both measurements, we set the socket buffer space to 200 KB.

To show the benefit of using RTUs for protocol processing, we repeat the above test with additional CPU competitions. We execute four copies of the **primes** program, which does intensive CPU computation continuously, as the background system load.

Figure 8 left side shows the throughput measured at each of the three RTU based TCP connections. The solid line shows throughput achieved when there is no CPU competition, and the dashed line shows throughput when CPU competition is added. As we can see from the figure, the throughput of all three periodic connections stay almost constant at the application specified value, regardless of whether there is CPU competition or

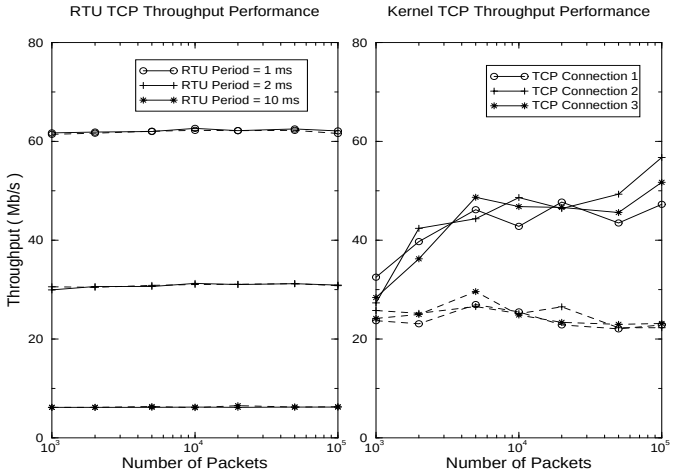

Figure 8: Throughput Performance Comparison
(— no cpu competition · · · with cpu competition)

not. This result clearly demonstrates the effectiveness of RTU based TCP implementation where throughput is guaranteed by preserving the periodic nature of the data stream and decoupling real-time protocol processing from other system load. We also observe that the application level RTU and the kernel level RTU cooperates well through socket buffering so that both application processing and protocol processing are scheduled in a timely manner. In addition, the TCP flow control mechanism does not affect our throughput noticeably, since both ends have similar processing power and transfer packets smoothly without causing congestion at either side. Yet, a large socket buffer is needed to mitigate the effect of TCP slow start.

On the contrary, the kernel TCP performs poorly in guaranteeing throughput. In Figure 8 right side, we show our throughput measurement of three competing TCP connection in the same test environment. The bandwidth of the 155 Mbps ATM link is shared among the three TCP connections, so we would expect a mean of 50 Mbps for each connection. The solid line (no cpu competition) in Figure 8 right side shows approximately the expected mean value but with a large variation in throughput. Additionally, the throughput of all three connections drop to about only 25 Mbps when we started the background competing processes. This reduction of half of the throughput is solely due to endsystem congestion since the OS is unable to provide predictable packet processing, resulting data transfer to stall.

7.2. Latency Performance of Low Delay Connections

As we described before, low delay RTU TCP connections are implemented using *reactive* RTUs. In our experiment model, input/output protocol processing at both end hosts and data processing at the receiver side function as reactive RTU handlers, while the data source transmits packets continuously with a fixed interval. We use 100 ms as the time interval in our experiment.

We open three low delay RTU TCP connections and separately a single kernel TCP connection. Figure 9 shows our measured latency performance for both cases. The left side figure shows the measured round trip time under no CPU competition and the right side shows the performance when CPU competition is presented.

The RTU TCP and kernel TCP perform closely under no CPU competition. The round trip time (RTT) for a 64 bytes packet is about 0.4 msec for RTU TCP connection, and 0.35 msec for kernel TCP. For 8 KB packet, RTT increases to 1.5 msec and 1.7 msec respectively. This result demonstrates that the additional processing of RTU scheduling is very low and does not degrade data path performance. The advantage of RTU TCP for

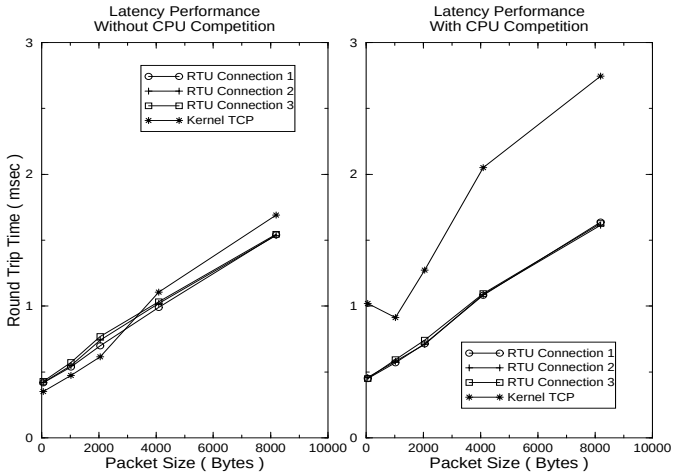

Figure 9: Latency Performance Comparison

packet size (bytes)	**RTT Deviation** (msec) (no cpu competition)	**RTT Deviation** (msec) (with cpu competition)	**Round Trip Delay** (msec) (with cpu competition)
64	0.17836	0.18645	0.45148
1024	0.18645	0.18773	0.58133
2048	0.21145	0.20616	0.71318
4096	0.19244	0.22080	1.08946
8192	0.12217	0.23222	1.62567

Table 1: Average RTT Deviation for Low Delay Connections

its timely scheduling is shown in the right side of figure 9. When the kernel TCP connection suffers much higher delay from other CPU competing processes, RTU TCP retains the same low delay performance. We also show in Table 1 the measured RTT deviation of the RTU TCP connections. We observe that the RTT deviation remains consistent over all cases.

The RTT performance of low-delay connections concludes that the use of reactive RTUs eliminates endsystem scheduling latency in the face of competing processes, and efficiently delivers data packets to the application. Thus, the RTU based TCP implementation provides predictable delay performance on data transfer to the application.

7.3. Coexistence of Multiple Connection Types

An important aspect of RTU TCP is that it is able to provide application specific QoS on a per connection basis. In this section, we experiment with a combination of guaranteed-bandwidth, low-delay and best-effort connections, and show the interactions among them.

Coexistence of guaranteed-bandwidth and best-effort connections We first measure the effect of best-effort connections on periodic connections. We setup three periodic connections with respective RTU period of 1, 5, 10 msec, and a best-effort connection at the same time. The best-effort connection uses RTU TCP as the underlying transport protocol, but does not use RTU at the application level for data transmission

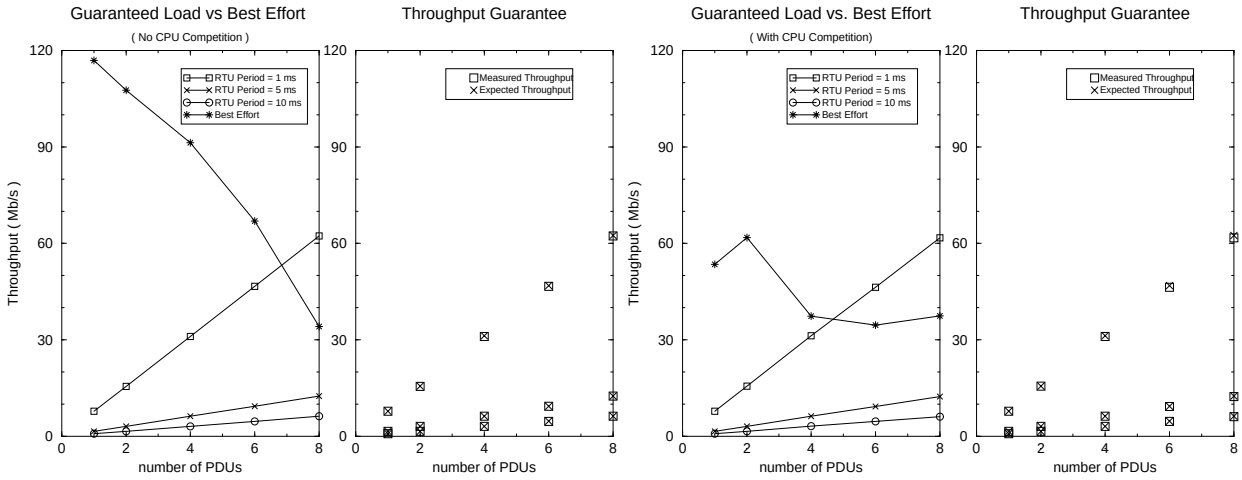

Figure 10: Coexistence of Guaranteed-Bandwidth and Best-Effort Traffic

or reception. The bandwidth specified for each periodic connection is varied as we change the number of data units to transmit in each period. All four connections use 1 KB size packets.

The periodic connections have higher QoS priority than the best-effort connection, so when we increase the specified bandwidth of periodic connections, they achieve their specified throughput while the throughput of best-effort connection drops drastically. In Figure 10(a), the left side shows the throughput of the four connections when we vary the number of packets sent per period (x-axis), and the right side shows the expected throughput versus the measured throughput of the periodic connections. The measured throughput of all three periodic connections increase linearly, and match exactly as the expected value. This clearly shows that the existence of best-effort traffic does not interfere or degrade the performance of guaranteed bandwidth traffic.

We repeat the same experiment with the additional CPU competition processes, as shown in Figure 10(b). Again, the periodic connections acquire their specified throughput, demonstrating that the RTU scheduling mechanism effectively decouples the RTU computation time from other system load. We also observe from this experiment the importance of scheduling application data processing as well as protocol processing in real-time, as the throughput decrease of the best-effort connection is solely due to the fact that the application competes for CPU time with the background processes and is unable to promptly process the data queued in the socket buffer. The data overflow at the socket buffer results the window based flow control mechanism of TCP to detect congestion and to reduce the sender transmission rate.

Coexistence of low-delay and guaranteed-bandwidth connections In this experiment, we show how a low-delay connection interacts with the periodic connections. We setup three guaranteed-bandwidth connections and one low-delay connection simultaneously, each of them uses 1 KB size packets.

From Figure 11, we first observe that no matter there is CPU competition (right side figure) or not (left side figure), the periodic connections always achieve their guaranteed throughput. Because the low-delay connection only transmits message sporadically and consumes a small portion of the link bandwidth, it does not disrupt the periodic scheduling of guaranteed bandwidth connections. On the other hand, the measured RTT time of low-delay connection increases slightly from 0.6 msec to 1.0 msec with the increasing aggregate bandwidth of periodic connections. We account two reasons for this increase: (a) With the increase of aggregate bandwidth of periodic connections, there is a higher probability that a low-delay packet arrives

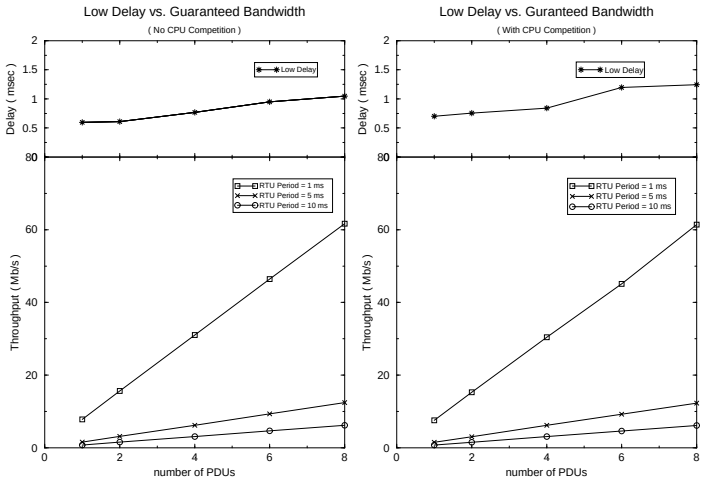

Figure 11: Coexistence of Guaranteed-Bandwidth and Low-Delay Connections

during the execution of a periodic RTU. Because RTU uses the delayed-preemption approach, the processing of a data unit of a periodic connection can not be immediately preempted upon the arrival of a low-delay packet. Thus, preemption delay is introduced to the round trip time of the low-delay connection; (b) Packet arrivals at the network interface are processed in hardware interrupts which cause an immediate transfer of CPU control from the RTU handler, and contribute extra delay in the RTT measurements. Nevertheless, the increase of the RTT, as observed from Figure 11, is not substantial and is bounded by the data unit size of the periodic connections and the link capacity. Another observation is that the low-delay connection retains its performance in the face of added CPU competition. Each sample point in Figure 11 right side appears to be only slightly higher, about 0.1 msec than that in the left side figure, because of the extra context switch from the background processes to the RTU handlers in both end hosts.

In conclusion, the RTU based TCP/IP implementation can concurrently support guaranteed-bandwidth and low-delay connections with QoS guarantees, without degrade the absolute performance.

Coexistence of low-delay, guaranteed-bandwidth and best-effort connections The main advantage of the RTU based TCP/IP implementation is the close integration of scheduling and protocol processing, and consequently QoS is guaranteed on a per-flow basis. In this experiment, we mixed two guaranteed-bandwidth connections, two best-effort connections with a low-delay connection, and show the effectiveness of our implementation.

As we discussed before, the RTT performance of a low-delay connection is affected by the accumulative bandwidth of all coexisting connections, due to the preeemption delay and interrupt overheads. Yet, the increase of the RTT is independent on the number of coexisting connections or their types, rather, it is bounded by the data unit size of the other connections and the overall link capacity. The larger the data unit, the higher the preemption delay, and the higher the aggregate bandwidth of other connections, the higher possibility of an RTU experiencing the preemption delay. As observed in Figure 12 left side, the measured RTT of the low-delay connection is an almost 1 msec. Additionally, the guaranteed bandwidth connections achieve their specified throughput and the best-effort connections share the rest of the link bandwidth. In the right side of Figure 12, the RTT of the low-delay connection shows a steady increase, from 0.7 msec to 1.5 msec, as the increasing of the aggregate bandwidth of the other coexisting connections. The RTT in the right side figure starts at a lower value than that in the left side, because the scheduling of RTU is insensitive to the background CPU competition but is sensitive to the accumulative bandwidth of all coexisting connections.

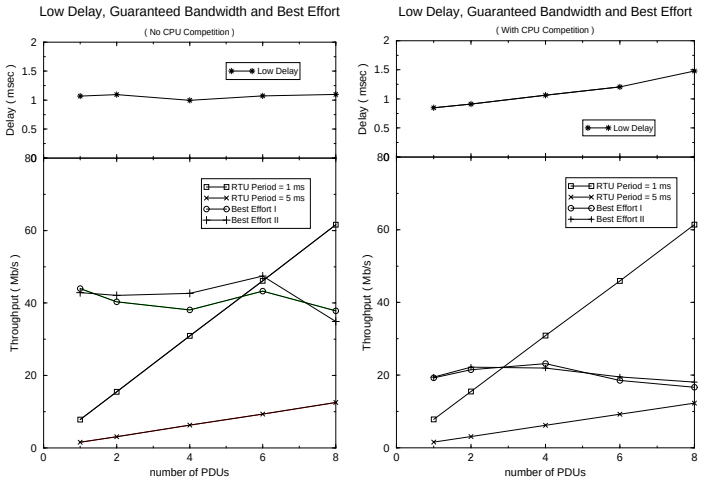

Figure 12: Coexistence of Guaranteed-Bandwidth, Low-Delay and Best-Effort Traffic

Thus, the RTU based TCP/IP implementation is able to provide QoS guarantees of bandwidth sharing and low response time, even with CPU and link bandwidth competition.

8. Conclusion

We have presented a new TCP/IP implementation model based on a real-time scheduling mechanism with QoS guarantees. We describe the insufficiency of real time service guarantees in the normal TCP/IP implementation and show how this insufficiency can be avoided by applying the RTU method. The experiment results show that the RTU based TCP/IP implementation provides throughput guarantees, prioritized bandwidth sharing among multiple connections and low request-response time even in the presence of heavy system background load and network load.

Our exploration of integrating kernel resident protocols with real-time scheduling brings theory and pratice together to achieve efficient and effective protocol implementations. We believe that our work is the first to support coexisting high-throughput and low-latency TCP/IP connections in the endsystem, and provides good performance guarantees. We hope that our experimental results provide a prliminary set of benchmarks for other implementations to reference.

9. Acknowledgment

We thank Woody Zenfell and Fred Kuhns for their helps in proof-reading this paper, making incisive suggestions and correcting all language errors. We also thank Chuck Cranor, for always being available for various questions about NetBSD OS internals and other system utilities. Without them, this paper would never be accomplished.

References

[1] R. Gopalakrishnan and G.M. Parulkar. Design and Implementation of a Real Time Upcall Mechanism. Technical report, Washington University in St. Louis, July 1996.

[2] L. Zhang and et. al. Rsvp: A new resource reservation protocol. *IEEE Network*, pages 8–18, 1993.

[3] R. Gopalakrishnan and G.M. Parulkar. Efficient User Space Protocol Implementations with QoS Guarantees using Real-time Upcalls. Technical report, Washington University in St. Louis, 1996.

[4] K.Jeffey. On kernel support for real-time multimedia applications. In *Proc. Third IEEE Workshop on Workstation Operating Systems*, pages 39–46, April 1992.

[5] R. Govindan and D.P. Anderson. Scheduling and ipc mechanisms for continuous media. In 13^{th} *ACM Symposium on Operating Systems Principles*, 1991.

[6] J. Nieh and M.S. Lan. The design, implementation and evaluation of smart: A scheduler for multimedia applications. In *the* 6^{th} *ACM Symposium on Operating Systems Principles*, Oct 1997.

[7] P. Goyal, X. Guo, and H.M. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *Symposium on Operating Systems Design and Implementation*, 1997.

[8] H. Tokuda and C.W Mercer. Arts: A distributed real-time kernel. *ACM Operating Systems Review*, pages 29–53, July 1989.

[9] N.C Hutchinson and L.L Peterson. The x-kernel: An architecture for implementing network protocols. *IEE Transactions on Software Engineering*, pages 64–76, Jan 1991.

[10] C. Lee, K. Yoshida, C.W. Mercer, and R. Rajkumar. Predictable communication protocol processing in real-time mach. In *Real-time Technology and Applications Symposium RTAS*, June 1996.

[11] C. Maeda and B.N. Bershad. Protocol service decomposition for high performance networking. In 14^{th} *aCM Symposium on Operating Systems Principles*, pages 244–55, Dec 1993.

[12] C.A. Thekkath, T.D. Nguyen, E. Moy, and E.D. Lazowska. Implementing network protocols at user level. In *ACM SIGCOMM*, Sept 93.

[13] D. Yau and S.S. Lam. An architecture towards efficient os support for distributed multimedia. In *Proc. IS&T/SPIE Multimedia Computing and Networking Conference*, San Jose, CA, Jan 1996.

[14] http://www.cs.arizona.edu/scout/.

[15] A. Mehra, A. Indiresan, and K.G. Shin. Structuring communication software for quality-of-service guarantees. In 17^{th} *Real-Time Systems Symposium*, Dec 1996.

[16] K. Jeffay and D. Bennett. A rate-based execution abstraction for multimedia computing. In *NOSSDAV*, 1995.

[17] R. Gopalakrishnan and G.M. Parulkar. Bringing Real-time Scheduling Theory and Practice Closer for Multimedia Computing. *ACM SIGMETRICS*, May 1996.

[18] D. D. Clark. The structuring of systems using upcalls. In 15^{th} *ACM Symposium on Operating Systems Principles*, pages 171–180, dec 1985.

[19] J.C. Brustoloni and P. Steenkiste. Evaluation of data passing and scheduling avoidance. In *NOSSDAV*, 1997.

[20] M.K. Mckusick, K. Bostic, M.J. Karels, and J.S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison Wesley, 1996.

[21] C. Partridge. *A Proposed Flow Specification, rfc-1363*. Internet Request for Comments, Sept. 1992.

[22] A.T. Campbell and G. Coulson. A qos adaptive multimedia transport system: Design, implementation and experiences. *Distributed Systems Engineering Journal Special Issue on Quality of Service*, april 1997.

[23] G.R. Wright and W.R. Stevens. *TCP/IP Illustrated, Volume 2*. Addison Wesley, 1995.