Design, Implementation, and Performance of a Native Mode ATM Transport Layer

R. Ahuja
AT&T Bell Labs
Murray Hill, NJ 07974, USA
(908) 582-3581 fax: (908) 582-5857
ritesh@research.att.com

S. Keshav*
AT&T Bell Labs
Murray Hill, NJ 07974, USA
(908) 582-3384 fax: (908) 582-5857
keshav@research.att.com

H. Saran
Department of CSE
Indian Institute of Technology, Delhi
Hauz Khas, New-Delhi 110016, India
hsaran@cse.iitd.ernet.in

Abstract

We describe the design, implementation, and performance tuning of a transport layer targeted specifically for Asynchronous Transfer Mode (ATM) networks. The layer has been built from scratch to minimize overhead in the critical path and take advantage of ATM Adaptation Layer 5 functionality. It provides reliable or unreliable data delivery with feedback or leaky-bucket flow control. These services can be combined to create a customized transport service. Our work is novel in that it is the first end-to-end ATM transport service that provides reliable, flow controlled data transfer.

We describe the mechanisms and the operating system support needed to provide these services. A detailed performance measurement allows us to determine the bottlenecks in our system and to tune our implementation. With this tuning, we are able to achieve a user-to-user throughput of 55 Mbps between two 66 MHz Intel 80486 Personal Computers with Fore Systems' HPA-200 EISA-bus host adaptors. The user-to-user latency is around 720 μs. These figures compare favorably with the performance from far more expensive workstations and validate the correctness of our design choices.

Keywords: ATM, transport layer, Personal Computer, AAL, performance analysis, native-mode ATM

*Corresponding Author

1 Introduction

Most current ATM networks use TCP as the transport layer, with IP-over-ATM providing the network layer [5]. This approach, though useful in the short-term, is soon likely to prove inadequate for several reasons. First, ATM networks will provide end-to-end Quality of Service (QoS) guarantees to individual virtual circuits [11]. These guarantees are lost by IP, since it multiplexes multiple transport connections with disparate QoS requirements onto a single VC. Moreover, current TCP implementations cannot directly use the QoS guarantees provided by the network since TCP does not obey a leaky-bucket behavior envelope, nor respond to ABR resource management cells.

Second, TCP checksums a packet to detect corruption. Since checksumming requires every byte of a packet to be touched, it is a significant overhead [3]. However, ATM Adaptation Layer 5 (AAL5) already does data checksumming. Thus, this TCP functionality is redundant and costly.

Third, TCP has inherited the patches and fixes of two decades of tinkering [4]. The protocol is still poorly understood, and there are many constants that are 'magic'. This slow increase in complexity with time has led to sub-optimal TCP performance in practice, since most users are too scared to touch something, lest they break it.

Thus, we believe there is a need for a transport layer that is aware of an underlying AAL5 layer, and that has been designed afresh to provide clean semantics. We describe the design and implementation of a transport layer, targeted specifically for Asynchronous Transfer Mode (ATM) networks i.e. a *native-mode* ATM transport layer. The layer incorporates much of our past work in flow and congestion control [16].

2 Service Description

We believe that the set of services provided by the transport layer should match the anticipated application workload. We expect ATM networks to support continuous media applications, which need QoS guarantees from the network (expressed in terms of guarantees of minimum bandwidth, priority, maximum end-to-end delay and loss rate), while conforming to some traffic envelope [8]. We would also like to support data applications, which effectively need a zero loss rate. Still other applications may require a raw bit-stream abstraction upon which they can build custom flow and error control mechanisms.

Instead of providing a service corresponding to each anticipated application workload, we provide a set of orthogonal services which can be combined in order to match application requirements. The three major services are: 1) *simplex data transfer*, 2) *error control*, and 3) *open-loop and feedback flow control.* The first service is simply to move data. With error control, the data stream seen by an application will have zero loss rate (possibly after retransmissions) and a corruption rate below some vanishingly small threshold. If the corruption rate is unacceptable, or if retransmissions are too slow, applications have the option of implementing Forward Error Control. With open-loop flow control, application traffic is shaped to conform to some pre-specified envelope (negotiated during call setup). With feedback flow control, the transport layer attempts to match the application's flow rate to the current bottleneck service rate in the network or receiver. By putting together a combination of these services, an application can customize the service interface it receives from the transport layer.

Currently, we support three combinations of the above services. These are *guaranteed-performance service*, *reliable service* and *best-effort service*. Guaranteed-performance service provides open-loop flow control without retransmissions. An application's QoS specifications are made available to the network, allowing it to reserve resources for each VC. Reliable service provides timeouts and retransmissions and feedback flow control. QoS guarantees are not supported. With best-effort service, there is no flow control, error control or provision of QoS guarantees. We will develop other services derived from the three basic services above, as the need arises. (We do not currently support multicast.)

In addition to the three services described above, we also provide: 4) *arbitrary application message sizes*, 5) *a choice of blocking and non-blocking application interface*, and 6) *a choice of byte stream and message transfer semantics*.

3 Design Principles

This section describes the principles we used in designing our transport layer. Our first principle was to eliminate multiplexing of virtual circuits [6]. Many transport layers multiplex multiple transport connections onto a single network layer connection (or, in the case of IP, a single network layer address). This simplifies routing, since there is only one network layer address per machine. However, during multiplexing, application QoS parameters are lost.

Since ATM networks provide multiple virtual circuits per-endpoint, we do not need to multiplex transport connections. This allows us to maintain per-VC QoS all the way from the ATM layer to the application. All protocol information is kept in a per-VC data structure called the protocol status block (PSB). Once the PSB has been located by the AAL layer from the incoming frame, no further search for protocol information is needed. This reduces code complexity.

Second, we wanted a clean separation of the transport layer services, so that they could be mixed and matched. Thus, an application can choose between different error control and flow control options as it desires. In contrast, with TCP, both error and flow control are implemented using windowing. As a consequence, losses in the network automatically affect the flow. While this may be desirable in many cases, it is not necessarily the right thing to do. Our transport layer reduces the dependency by exploiting the Packet-pair flow control protocol [15].

Third, our implementation provides minimal functionality in the critical path, with optimization for the common case. As Clark et al have shown [3], this has the potential to considerably enhance protocol performance.

Fourth, we do not replicate any functionality provided by AAL5 or ATM signaling. For example, connection management, traditionally a transport layer function, is relegated to signaling. Data checksumming is done by the ATM adaptation layer.

Finally, our implementation is designed to be highly portable. After working on protocol stacks for three different ATM testbeds, we realized that we need a transport layer that can easily be tailored to the needs of these various testbeds. The result was the modular design of an ATM stack with general, clean and well-defined interfaces among different parts of the stack. This makes it easily portable. The stack can be tested on the simulator and then ported to different systems, rewriting only those parts of the stack that are system dependent. For clarity, this paper discusses the implementation only in the Brazil OS kernel [1].

4 Implementation Overview

This section gives a brief overview of the protocol stack. The rest of the paper will discuss only the transport layer of the stack.

[1] Brazil is a research version of the Plan 9 operating system from Bell Labs [20]

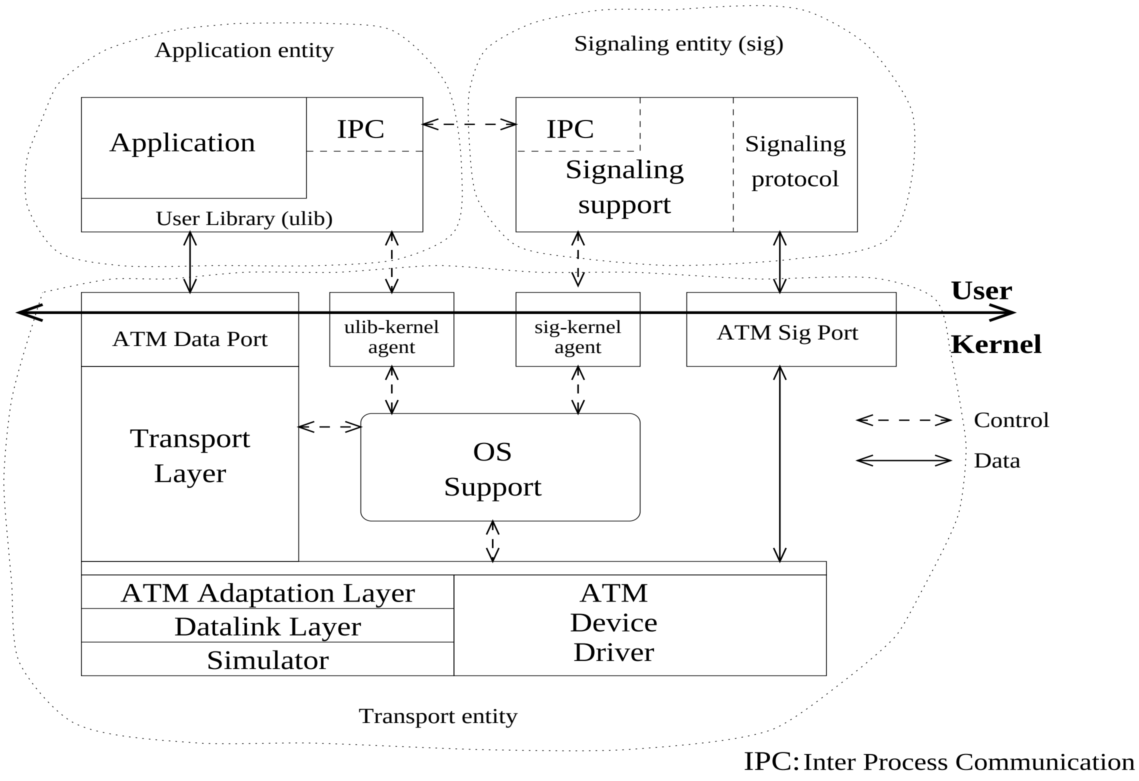

Figure 1: Top level view of the ATM stack. The stack consists of three entities: the application entity, the signaling entity, and the transport entity.

An end-system implementing our stack is shown in Figure 1. The ATM stack consists of three main entities, the application entity, the signaling entity (both in user space), and the transport entity (inside the kernel). To make the stack easily portable, each of these entities is divided into system-dependent and system-independent parts. We now sketch the functions provided by each entity.

1. **The Application entity:** The application is the user program that accesses the ATM network for communicating with its peer. The application is linked to a user library (ulib) which provides network access. By customizing the library for each environment, application code is completely independent of the underlying OS and hardware platform. The services provided by ulib are similar to the Berkeley socket interface [23, 18], except that applications can specify QoS parameters during connection set-up. This allows us to easily port applications written for Berkeley sockets - typical ports take only a few minutes to complete.

2. **The Signaling entity:** The signaling entity is the part of the stack that is responsible for connection management. It establishes ATM connections on behalf of user applications, and tears down connections either when requested by an application or in the event of a mishap like the crashing of an application. Since the signaling entity must survive application crashes, it cannot be part of the user library. Each end-system needs only one signaling entity, which is shared by all applications. The signaling entity is implemented so that the actual signaling protocol can be changed without affecting the rest of the signaling code. For example, the Fore Systems SPANS protocol that we implement now could be replaced with Q.931 without affecting the part of the signaling entity that keeps track of application state. The signaling entity is in user space so that it is easy to modify and to port to other platforms.

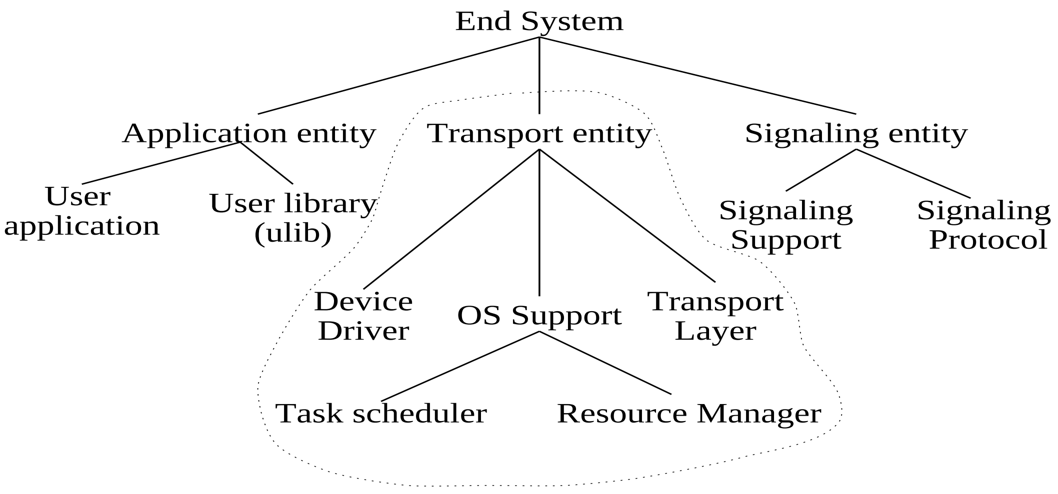

Figure 2: Components of the ATM stack. This paper deals mostly with the transport entity.

3. **The Transport entity:** This is the part of the stack which actually hands the data to the host-adapter. It consists of the transport layer, a task scheduler, and the device driver (Figure 1). Since the transport entity must provide high performance, we decided to put it in the OS kernel. However, in order to make it portable, our design makes minimal assumptions about the OS kernel. For example, we provide our own memory management code to handle operating systems which don't support BSD-style `mbufs` [18]. We also provide our own timers and task management. The only support needed from the OS is a way to handle packet-arrival interrupts, a way to read time, a memory allocation utility, and a way to occasionally call the task scheduler. These functions are available in all current operating systems.

The rest of the paper focuses on the transport entity.

5 The Transport Entity

The transport entity is responsible for transferring the data through the kernel down to the device (host-interface), from where it is picked up by the device and transmitted onto the network. This entity also performs functions like call admission and resource allocation to different VCIs for guaranteeing QoS (bandwidth and delay). In the sequel, we will assume that the host-adaptor provides AAL 5 frame transport, as is the case with all modern host-adaptors, including the Fore Systems HPA-200 adaptor that we used.

The three components of the transport entity are the transport layer, the device driver, and an OS support module (Figure 2). The OS support module in turn consists of a) a task scheduler and b) a resource manager for managing local resources. We will now look at the functionality of the transport entity in detail, starting with the OS support module.

5.1 The Task Scheduler

The transport layer is implemented as a set of *interface procedures* and *tasks*. An interface procedure handles asynchronous events such as packet arrival, user read or write request, or completion of packet transmission. An interface procedure is designed to complete quickly, scheduling a task for handling any CPU-intensive work. A task is a C-language function that is non-preemptively executed by a procedure call from the *task scheduler*. Each task finishes in a known time and can schedule other tasks to complete their work.

Our design has several advantages. First, this division of labor allows us to provide quick response to asynchronous events while CPU-intensive work is prioritized by the task scheduler. This allows high priority packets make their way through the transport layer faster than low priority tasks. Second, since the task scheduler and all the tasks run in the same address space, and each task is just a procedure call, calling a task is a very cheap operation. This allows us to cheaply provide fine-grained multitasking. Of course, no task may block. A task that might block reschedules itself at a future time when it can check on the status of a blocking event. The scheduler provides an efficient set of timer routines for this purpose.

In the Brazil kernel, the task scheduler is a kernel thread which is started by the ATM device driver at boot time. The scheduler is just an infinite loop that periodically (in our case, every 50 ms) handles any expired timers, and then executes any scheduled tasks. Each schedulable task is called by the scheduler with two arguments: the VCI to act on and the maximum amount of work, in number of units, it can do in the call. The function does some processing and returns the amount of work it actually did in that call. In our implementation, processing one transport protocol data unit (AAL 5 frame) accounts for one unit of work. The scheduler can schedule tasks on the basis of their importance and the amount of work that is pending for each task. The scheduler can implement any scheduling discipline in order to allocate the processing resources to different tasks. Currently we have a multi-level weighted-round-robin scheduler, that assigns different priorities to different tasks, and schedules tasks round-robin within the same priority. Hence we can allocate different QoS to different connections, and also allocate different priorities to different best-effort connections.

5.2 Resource Manager

The resource manager is responsible for admission control at the time of call setup. When an application sets up a call using ulib, a call is made to the resource manager to see if sufficient local resources exist to support the new call. This admission test requires the manager to know the amount of CPU resource available to the transport layer, and the fraction of the resource that is already consumed. While our performance measurements allow us to determine exactly how much CPU processing time each TPDU needs, since our kernel is not real-time, we do not as yet have a way to reserve CPU time for the transport layer from the kernel. Thus, the current implementation of the resource manager does not do admission control. Instead, whenever the manager is asked to reserve a resource, it stores the request and responds with a 'yes'. On a request to release resources, it removes the reservation from storage. Further work needs to be done to implement admission control in conjunction with improvements in the task scheduler and the CPU scheduler, so that the task scheduler can schedule tasks on the basis of the resources allocated to a VC, and, in turn, can reserve time from the CPU scheduler.

5.3 Transport Layer

Having looked at the environment in which the transport layer is placed, we now turn our attention to the transport layer itself. The transport layer provides four interface procedures visible to the outside world. The functions `t_send` and `t_recv` provide an interface on the send and receive side respectively, for giving or taking user messages to or from the transport layer. All other transport layer processing is done by the `t_schedule_send` task on the send side,and the `t_schedule_recv` task on the receive side. The `t_schedule_send` task is scheduled by the `t_send`

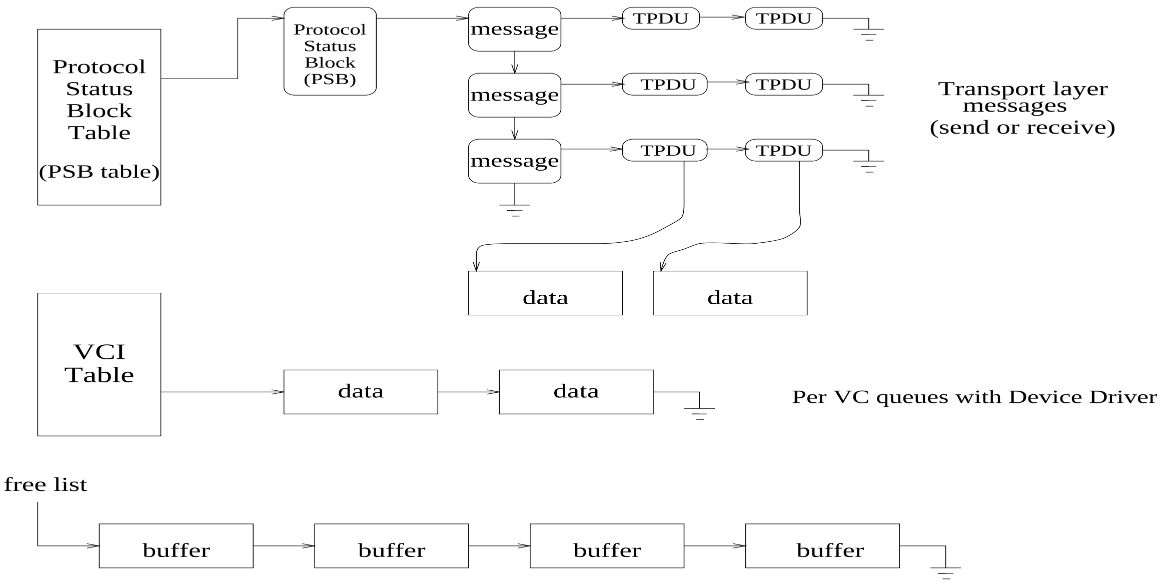

Figure 3: Data structures used for storing messages. Each protocol status block (PSB) has pointers to send and receive message lists. A message consists of a number of transport protocol data units (TPDUs), and each TPDU corresponds to a single AAL 5 frame.

procedure, and the `t_schedule_recv` task is scheduled by the device driver, as described in Section 5.4. Since the `t_schedule_recv` and `t_schedule_send` tasks are scheduled asynchronously, we need to lock queueing and dequeuing of tasks in the task queue. This is done by having a global task lock which is implemented as a spin lock.

The transport layer provides *simplex* virtual circuits, error control, and flow control. In addition, it segments application layer buffers into transport protocol data units (TPDUs) and reassembles them on the receive side. Here, we present the mechanisms required to provide these semantics.

5.3.1 Segmentation and Reassembly

There are two reasons why the transport layer may want to fragment an application message into TPDUs. First, in our implementation, each TPDU corresponds to a single AAL5 frame, which is at most 64 Kbytes long. If the message is larger than this size, it must be fragmented. A more compelling reason has to do with error control. The unit of error detection and retransmission is a TPDU. If this is large, then each loss causes a large retransmission overhead. By keeping TPDUs small, the retransmission efficiency is maximized. Thus, the TPDU size can be chosen to trade off per-fragment overheads, the connection's error characteristics and the available timer resolution. Indeed, this is the choice of 'Multiplexing Block' in reference [6].

We have tried to minimize the overhead for segmentation and reassembly. On the transmit side, the transport layer's `t_send` procedure segments a message while copying an application buffer into a chain of TPDUs. In order to preserve message semantics, the TPDU header has a message number, fragment number, and an end-of-message flag. On the receive side, the `t_schedule_recv` task picks up TPDUs from the network layer and queues them in per-VC message queues (Figure 3). If the VC is reliable and supports message boundaries, fragments are reassembled by the receiving transport layer and the `t_recv` procedure returns only complete messages to an application. Message reassembly is done directly from the per-VC queue into the application buffer during a read system call.

5.3.2 Error Control

While the AAL 5 checksum detects corruption and loss within an AAL frame, this, by itself, is not sufficient for error control. For a reliable connection, lost or corrupted data must be retransmitted. This is done at the transport layer using a novel retransmission scheme described below. Note that the transport layer does not do checksumming, since this is already taken care of by the AAL layer.

To allow a receiver to detect duplicate data from retransmissions (which may be arbitrarily delayed, perhaps extending beyond virtual circuit tear down), sequence numbers are necessary. The transport layer uses a standard three way handshake at startup to choose the initial sequence number correctly [24]. We do not use a two-way handshake for termination, since termination is handled by the signaling entity.

The transport layer uses per-TPDU cumulative acknowledgments for redundancy. Cumulative acknowledgments have the added benefit that if an acknowledgement sequence number is repeated, the source can guess with high probability that the packet with a sequence number one larger than this sequence number was lost. In our scheme, the acknowledgment also carries the sequence number of the TPDU that generated the acknowledgment, allowing sources to additionally determine which sequence numbers have been correctly received [16].

A retransmission is triggered either by a repeated cumulative acknowledgment (fast retransmission) or by a retransmission timeout. In either case, the entire current transmission window is scanned for possible retransmissions (as in Go-back-n). During a fast retransmit, only the packets which are not already retransmitted and not correctly received (for which ack has not arrived yet) are retransmitted. During a timeout, only packets not correctly received are retransmitted - thus packets retransmitted by a fast retransmit but subsequently lost are retransmitted a second time by the timeout. To make the retransmission even more independent of timeouts, we check every two round trip times if the cumulative ack has made any progress. If the cumulative ack remains the same for two successive RTT's, a retransmission is again triggered, but this time only the packet at the head of the window is retransmitted.

This scheme combines the efficiency of selective retransmission with the robustness of Go-back-n retransmission. They allow a sender to quickly fill a gap in the error-control window without stalling while waiting for a timeout, or paying the overhead and complexity of a selective acknowledgment scheme.

To allow retransmissions, a source must keep a copy of the outstanding data, and the size of this buffer is limited by an error-control window. Since a receiver will also need to keep a copy of delivered data to assure in-sequence delivery of data, the error-control window size must be negotiated by the peer transport layers during call setup. This is done during the three way handshake.

5.3.3 Feedback Flow Control

Flow control allows an endpoint to regulate the data transmission rate to match the maximum sustainable flow by that VC in the network. The transport layer provides both open-loop and feedback flow control.

If the scheduling discipline at the switches is round-robin like, feedback flow control is based on the Packet-pair flow control scheme [15]. In this scheme, all TPDUs are sent out in back to back pairs, and the inter-acknowledgment spacing is measured to estimate the current bottleneck capacity (the bottleneck may be in the network or the receiving end-system). This time series of estimates is used to make a prediction of future capacity, and a simple predictive control scheme is used to determine the source sending rate. It has been shown that for a wide variety of scenarios, Packet-pair flow control performs nearly as well as the optimal flow control scheme, that is, a scheme that operates with infinite buffers at all bottlenecks [16]

This scheme cleanly separates flow control and error control. Windows are used for error control and to size buffers at the transmitter and receiver. Flow control is used to match the source transmission rate with the current bottleneck capacity. When windows are used both for flow control and error control, packet losses will trigger a slowdown in the sending rate [21, 12], which may not be warranted by the current congestion level. This becomes important in high-speed networks where the bottleneck service rate is a rapidly changing quantity.

If the network does not support round-robin scheduling, the transport layer uses a dynamic-window flow control scheme similar to TCP flow control [12]. However, while losses do shut down the flow control window, the retransmission uses the strategy described above, instead of Go-back-n.

5.3.4 Open-loop Flow Control

Our transport layer provides open-loop flow control based on leaky bucket semantics. We believe that the traffic shaping function should be as close to the application as possible to allow it to quickly get feedback about its allowed flow rate. An application sending data faster than its leaky bucket rate would fill its input buffer, and when this happens, the application is put to sleep by the transport layer. This control is much easier to implement at the transport layer than at the host adaptor or a remote Network Interface Unit, as is usually the case.

The implementation of leaky bucket is simple - for each virtual circuit, the transport layer keeps track of the time that the last TPDU was sent. On arrival of a message from an application, the transport layer compares the current time with that time to determine how many tokens must have arrived in the interim. This is sufficient to know how many TPDUs can be sent right away, and the earliest time that the next TPDU can be sent. The layer also sets a timer for the earliest time the next TPDU can be sent, based on the leaky bucket arrival rate.

5.4 Data Flow

Having seen how the transport layer works, let us see the overall picture of how data flows from source to the destination (Figure 4), which involves the following steps.

1. The user application calls the ulib function `atm_write`, which acts as session layer for the application. The `atm_write` routine selects the appropriate data-descriptor in the kernel and makes a `write` system call on that data-descriptor. The data given by the application is thus passed into the kernel crossing the user-kernel boundary.

   The write system call in the kernel hands over the data to `t_send`, which fragments the user message into smaller TPDUs while copying the data from user space to kernel space, and buffers the TPDUs to be sent later. It then schedules the task `t_schedule_send` and returns.

2. The network task scheduler eventually calls `t_schedule_send` giving it the VCI to act on. This routine attaches the transport header to each TPDU and calls the AAL layer's `a_send` routine. `a_send` hands the packet to the ATM device driver's `enqueue_tx` routine that enqueues the packet in the device transmission queue along with the DMA address of the TPDU to be sent. The TPDU is now with the card and the responsibility of the host is over.

3. The card picks up the TPDU from the transmission queue and transmits it on the line after adding an AAL5 trailer and segmenting the AAL5 frame into ATM cells. When transmission completes, the card marks the TPDU as sent so that the host can free the memory area being blocked by this TPDU.

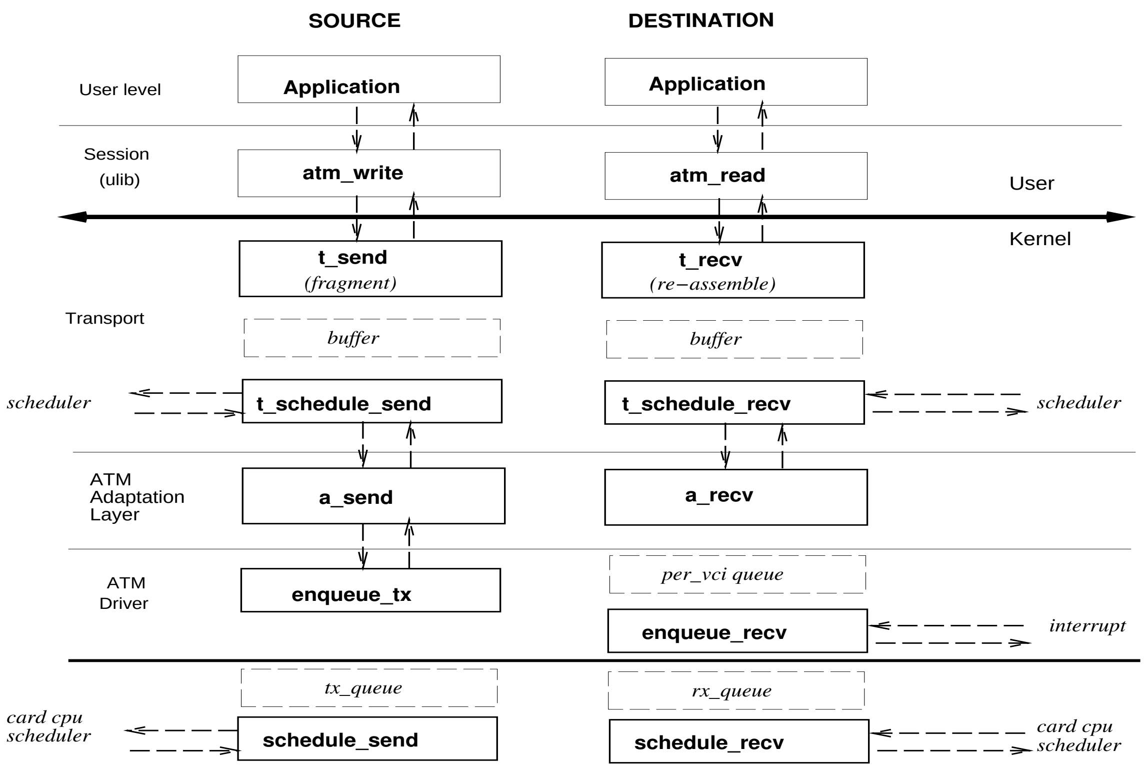

Figure 4: Data flow from source to destination. Asynchronous events are handled by interface procedures that spawn tasks which are handled by a task scheduler. A detailed explanation is in Section 5.4

4. On receiving a packet on the receive side, the card DMA's it into a queue in host memory and interrupts the host CPU. The card checks the AAL5 trailer and drops incomplete or incorrect frames.

5. On receiving an interrupt, the interrupt service routine (ISR) picks up the packet from the card receive queue and puts it into a per VCI queue for the AAL layer. The ISR schedules the transport layer's `t_schedule_recv` routine and returns.

6. The routine `t_schedule_recv` calls `a_recv` which retrieves the packet from the per-VCI AAL queue. After getting the packet, the transport layer checks the packet for validity and enqueues the packet at the right place in the received message queue (Figure 3). It also sends an acknowledgment if the TPDU is for a reliable connection. If the VC supports message semantics and the TPDU received is the last TPDU of a message, the transport layer marks the message as complete so that it can be picked up by the application as a complete message.

7. When the application wants to read a message, it calls ulib's `atm_read` which selects the right kernel data-descriptor and makes a `read` system call. This routine copies the data from the enqueued TPDUs of the next complete message into the user space (thus doing reassembly). If the next message is not yet complete or if there is nothing to receive, the `read` call blocks, and the application is put to sleep. The application is woken up by the `t_schedule_recv` function when the next message is complete.

5.5 Implementation Experience

We gained a lot of experience with the transport layer by implementing it in the Brazil OS and measuring its performance. In this section, we will discuss some interesting problems we ran into while doing the Brazil implementation.

5.5.1 Send and Receive Locks

Locking proved to be a major problem in doing an in-kernel implementation of our design. The user application can do a read or write at any time. A write results in enqueueing a message in the transport layer's send queue, and a read leads to dequeuing of a message from the transport layer's receive queue. Since these reads and writes are asynchronous with respect to the operation of the transport layer, we need to lock these queues with a per-VC Send lock and Receive lock. Each change in the send queue is locked with the corresponding send lock, and every modification in the receive queue is locked using the Receive lock for that connection. This lets us synchronize between the read and write operations of a user application and the transport layer.

5.5.2 Readers-Writers lock

In addition to the Send and Receive locks discussed above, there is another conflict that we needed to resolve. Though we allow only one application to be associated with a connection, which ensures that two applications cannot simultaneously do a read or write on the same connection, the signaling entity can modify the connection state while a read or write is in progress. Thus we have to avoid any reads or writes on data connections when the signaling entity is changing the connection status. This problem is simply a readers-writers problem with a write by the signaling entity corresponding to a writer, and all other accesses being operations by readers. We use a Brazil `RWlock` data-structure for proper locking of these reads and writes. This solves the problem, except for one detail, that we discuss below.

An application is put to sleep when it tries to do a read and the next message for the application is not yet completely received. Similarly, an application is put to sleep if it tries to do a write when the transmission queue of the application is full. However we cannot put the application to sleep while it holds any locks, since this will prevent the signaling entity from doing any communication with the kernel. Hence, the readers-writers lock is released if the application is put to sleep, and reacquired on wakeup.

5.5.3 Resource Release on Connection Teardown

The transport layer creates a Protocol Status Block (PSB) for each connection at the time of connection setup. This PSB has to be freed when a connection is torn down. However, connection teardown is an asynchronous event with respect to other operations of the transport layer. This problem here is similar to a readers-writers problem, with the resource release function acting as a writer and all other parts of the transport layer acting as readers. We could solve this problem with a readers-writers lock as before. However acquiring locks before every access to a PSB can be expensive. Hence we use a more efficient, albeit less elegant, solution.

In our solution, when a PSB is in use, a flag is set in a per-PSB table. The resource release function, which is called only on connection termination, does a busy wait on this flag. Since this function is called only at the time of connection teardown, the inefficiency associated with a busy-wait is still acceptable. As before, the PSB flag is released when an application is put to sleep during the read or write system call and is reacquired at the time of wakeup.

6 Performance Measurements and Tuning

Thus far, we have looked at the design and implementation of the stack. The true test of our design is in the performance it delivers. In this section, we present the results of a detailed performance analysis. Our results help us to understand the bottlenecks in performance, and to tune the layer for better performance. Our results are in three parts - latency in the protocol processing, throughput on the send and receive side, and per-layer costs as a function of message size.

6.1 Latency

We studied latency on a small testbed consisting of two IBM PC-clones with Intel 80486 processors running at 66 MHz. They each had a Fore Systems HPA-200 ATM adaptor card on an EISA bus. The systems were connected via a Fore Systems ASX 100 switch with TAXI 100 links running at 100Mbps nominal bandwidth. The systems were otherwise unloaded.

Since the resolution of the CPU clock (in ms) is not sufficient to measure processing delays, which are in microseconds, we had to measure these quantities indirectly. We measured the throughput obtained for 10,000 64 byte messages when a message is processed up to different stages of the protocol stack. For example, to measure the cost of the write system call, the message is dropped just before it would be handed to the transport layer. By measuring the rate of transmission available to the user application, we can determine the time spent on processing each message. Similarly, to find out the cost of transport layer processing, we drop the message after transport layer has done its processing, and just before it hands the message to the device driver. This gives the total cost up to and including the transport layer processing. Subtracting the cost of making a write system call from this, we get the cost of transport layer processing. Note that in most cases, since standard deviations add when subtracting quantities, the deeper we are in the protocol stack, the more the standard deviation in the cumulative measured delay. However, dependencies between events can lead to a *decrease* in standard deviation of the cumulative delay, since delay variations in an event can be partially offset by an opposing variation in a subsequent correlated event.

The user-to-user round trip time for 64 byte packets (with 32 bytes of user data and 32 bytes of transport header) was ~120ms if we did not wake up the task scheduler after scheduling a task on asynchronous send and receive events. The main part of this latency was due to the delay in scheduling. On an average, the processing of a packet was delayed by 25ms at each end, while it is waiting for the scheduler to wakeup after its normal 50ms sleep period. This delay was unacceptably high. So, to reduce end to end latency, the scheduler is woken up by the ISR on receiving a packet from the device, and by `t_send` routine on receiving a packet from the user application. This change reduced the round trip time to a mean of 1.44ms with a standard deviation of 0.01ms over 50 repetitions. This time can be broken up as shown in Figure 5.

The total time taken by the sending PC before the data is given to the card for transmission is roughly 315 μs. The bulk of this time is waiting for the task scheduler to become active. While we could have cut down this latency by eliminating the task scheduler and making a direct call into the `t_schedule_send` routine from the `t_send` routine, this would have prevented us from prioritizing between transport connections, which is necessary for providing per-transport connection QoS. Some components of the end-to-end latency are missing because we do not have access to the source code of the microcode running on the card. For example, we cannot measure how long it takes for the host adaptor to process a 64 byte packet. This is also a problem, with measuring the receive side performance. However, we expect the latency on the receiving host to be more than that on the sending host because we have

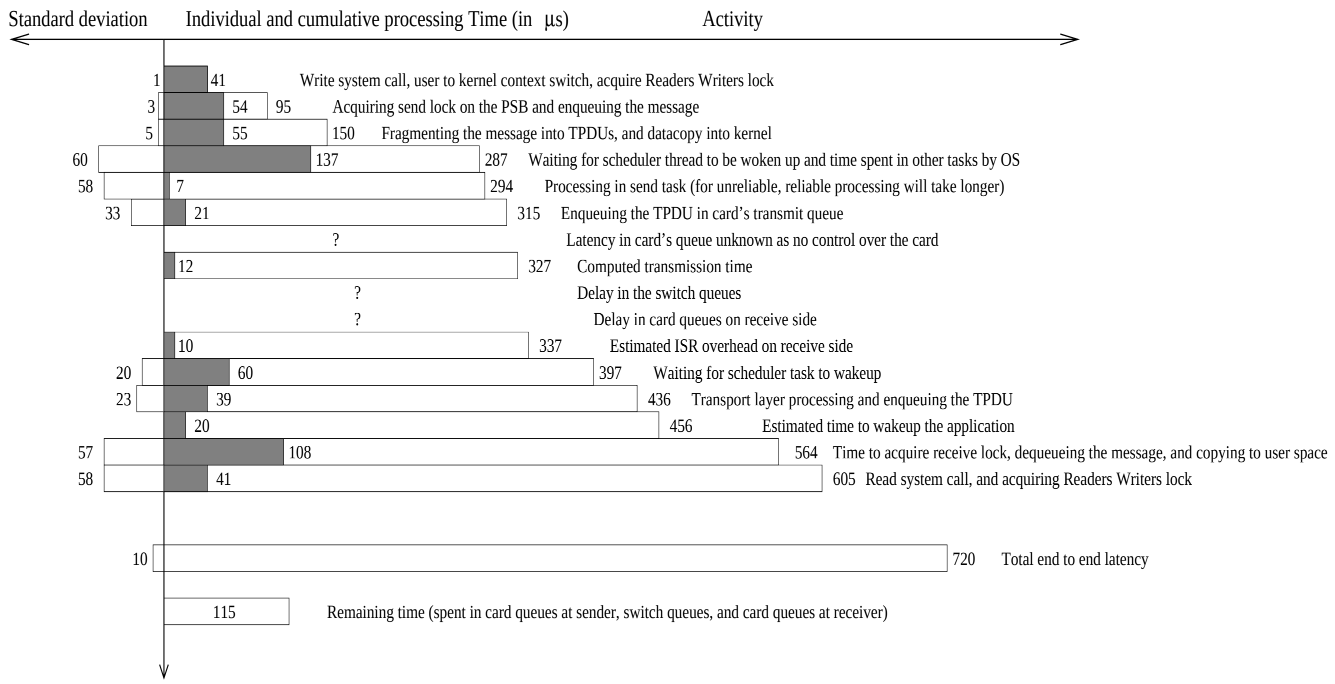

Figure 5: Individual (shaded) and cumulative delay and standard deviation at various points of the protocol stack when sending 64 byte packets between two otherwise unloaded machines. We have estimated times that we could not measure because we did not have access to the host adapter microcode

two context switches on the receive side before the user application gets its message. Nevertheless, the user-to-user latency of about 720 μs is quite small and CPU bound. We hope to achieve correspondingly better performance with Pentium 100MHz processors and the faster PCI bus.

6.2 Throughput

6.2.1 Guaranteed-service

We measured the user-to-user throughput with a TPDU size of 8 Kbytes (32 bytes of transport header and 8160 bytes of user data). Guaranteed-service applications are rate-controlled by a leaky bucket at the transport layer (Section-5.3.4). As Figure 6 shows, for transmission rates less than 40 Mbps, the sending rate is almost perfectly controlled. However, if the application tries to send at higher rates, the host CPU becomes a bottleneck, and the transmission rate starts dropping below the nominal value.

6.2.2 Reliable service

For reliable applications, the rate is controlled by using closed loop window flow control (Section-5.3.3). Since the receiver is slower than the sender, the receive rate determines the transmission rate. Figure 7 shows the rate at which an application is allowed to send data on a reliable connection for different message sizes. Error bars indicate the standard deviation over 50 repetitions. Note that the throughput increases with message size for a while, reaching its maximum of around 47Mbps, and then starts falling with further increase in message size. This can be explained by two opposing forces that come into play with increasing message sizes. The per-message overhead of read and

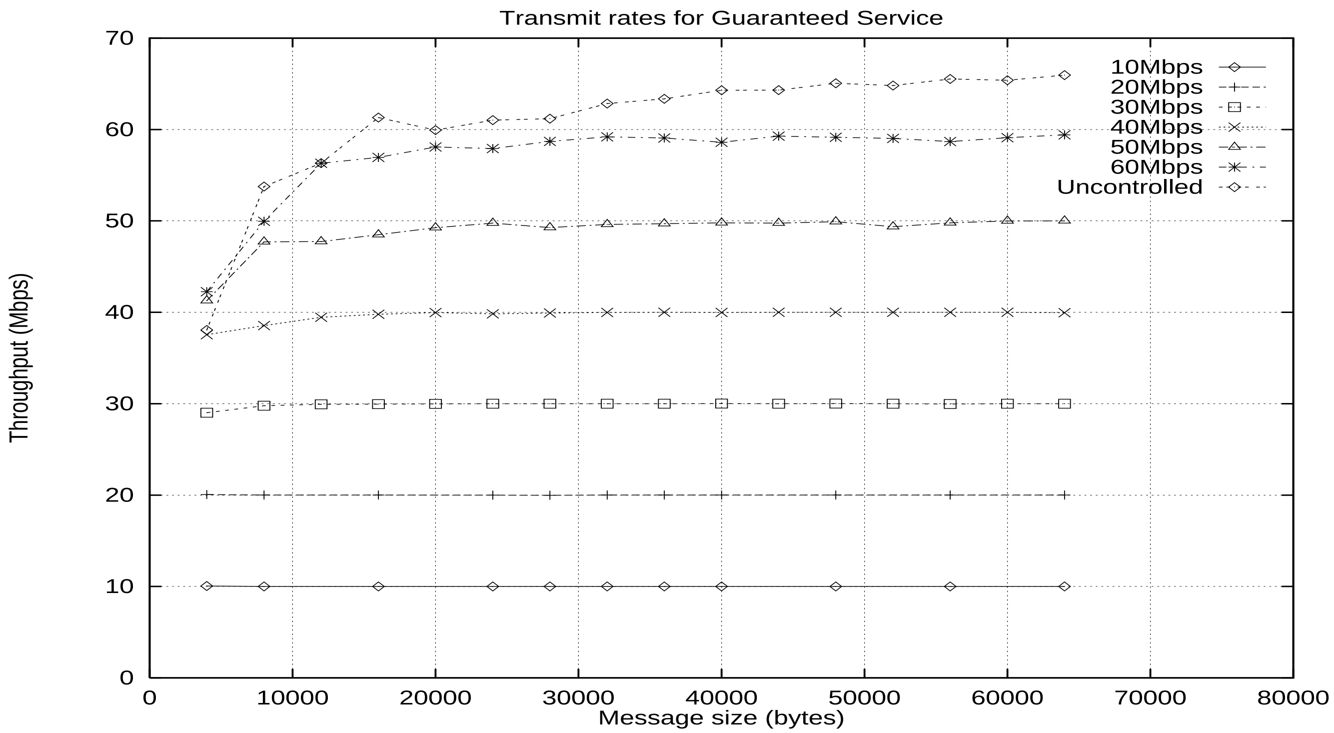

Figure 6: Transmission rate measured at the sender for a guaranteed-service connection. Note that the measured rate matches the requested rate for speeds up to 40 Mbps. After that, the actual sending rate is slightly below the requested rate.

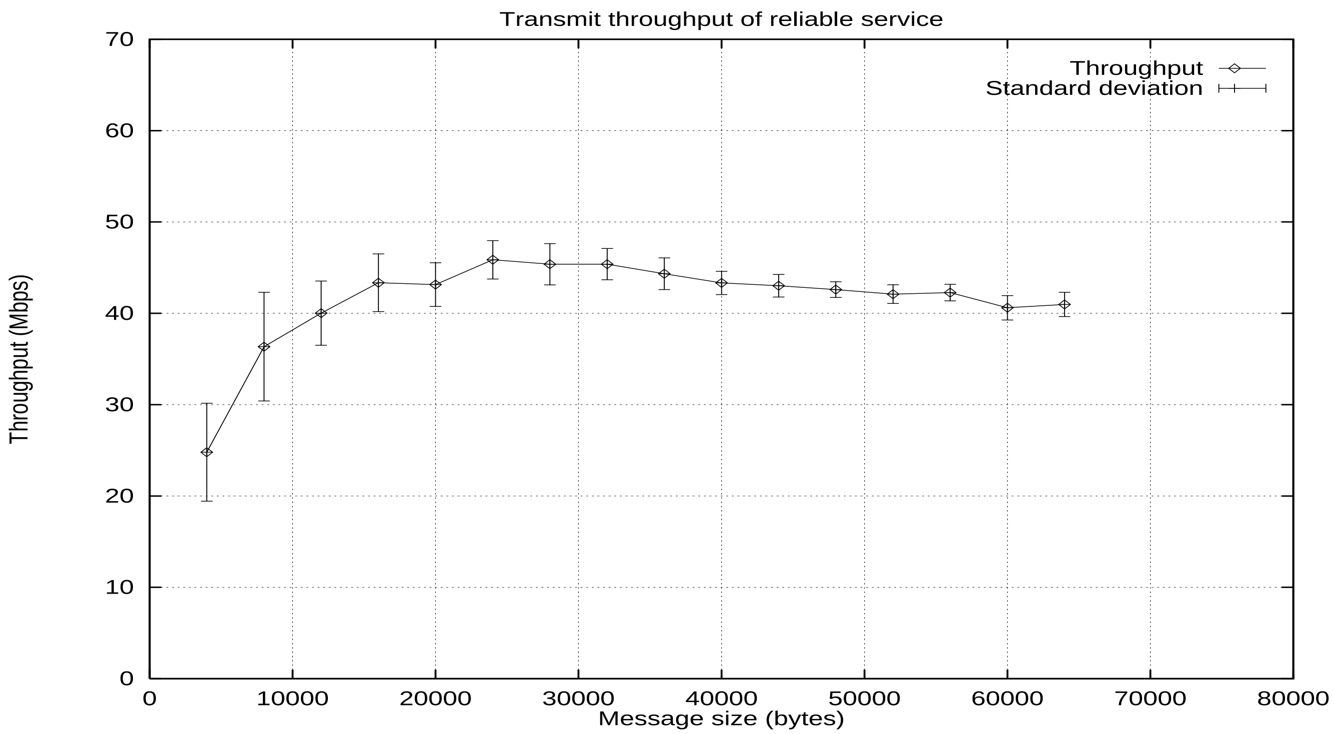

Figure 7: Transmission rate measured at the sender for a reliable connection. This is also the rate seen at the receiver, since the sender and receiver in a reliable connection are in lock-step. Note that the transmission rate increases, then decreases. As the message size increases, the amortized per-message cost decreases, but the per-byte cost increases.

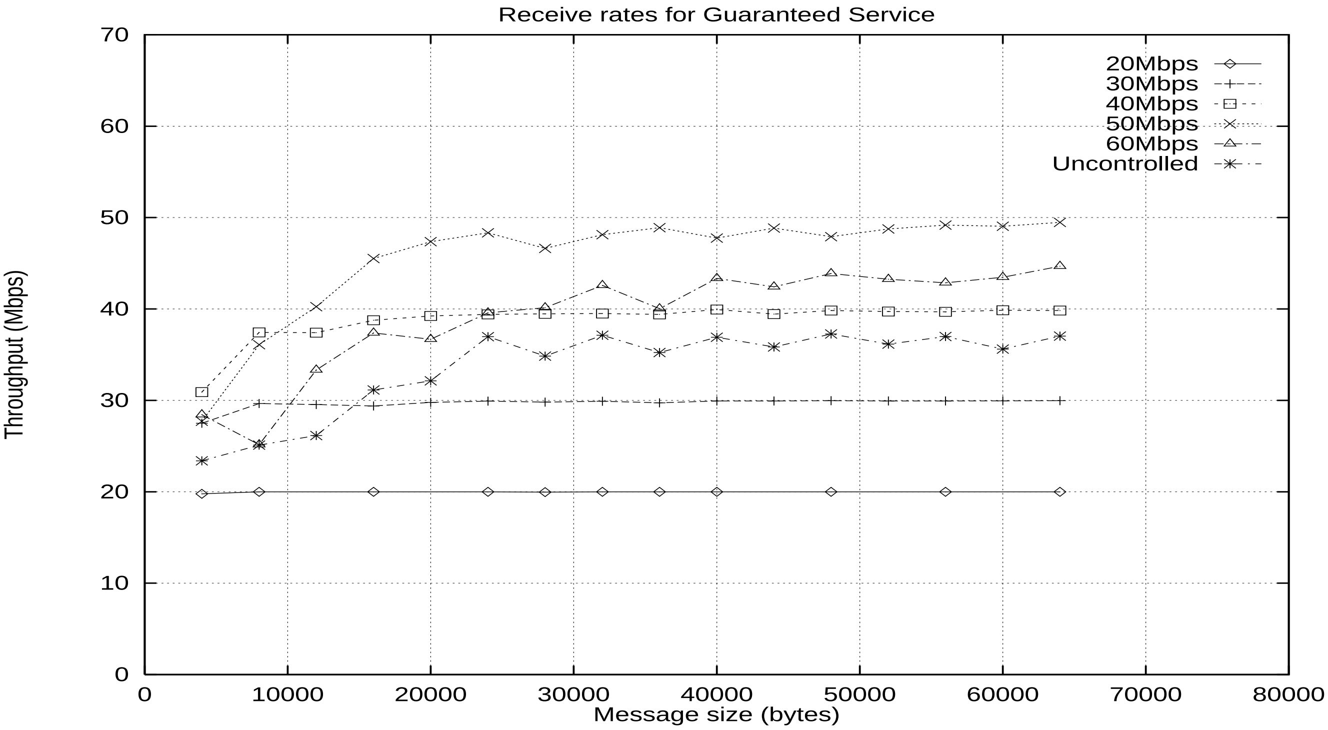

Figure 8: Throughput measured at the receiver for different transmission rates (ISR drops excess packets). Note that due to receive livelock, the throughput actually drops if the sender sends faster than the receiver can process data.

write system calls, and the per-message transport overhead gets amortized with increasing message size. However larger messages block more buffers in the receiver for reassembly (since an application is handed complete messages), increasing the probability of loss due to lack of buffers in the receiver. Further, if the message is many TPDUs long, it takes longer to scan the list of TPDUs to find the correct position for an incoming TPDU. These opposing forces cause the throughput to first increase, then decrease, as a function of the message size.

We found it very important to try to minimize the delay-bandwidth product when using window flow control. With TCP-style flow control, if this product is large, each loss causes a window shutdown, and recovering from the loss takes multiple round-trips. So an increase in the latency for reliable transfer can be expensive, especially at high bandwidths. In fact, if we do not wake up the scheduler on every interrupt, in order to amortize the cost of waking up the kernel process, reliable connections suffer dramatically. The throughput goes down to as low as 15Mbps when the kernel process wakes up periodically (every 50ms), and is not woken up by the ISR at all. Even a small increase in latency on receive side can seriously affect performance because, on a loss, we must wait at least one round-trip-time for receiving the lost packet, and we must block buffers for reassembly awaiting this retransmission.

6.2.3 Throughput and Loss at the Receiver

For a reliable connection, the receiver receives data at exactly the rate at which it is sent. For unreliable and guaranteed-service connections, however, the receiving rate is smaller than the sending rate because of losses in the receiver (Figure 8). The loss rate (Figure 9) increases with increase in sending rate, so much so that the receive rate *decreases* with increase in sending rate. The reason for this is interesting. The host adaptor copies incoming AAL5 frames into buffers provided by the device driver, which maintains a pool of free buffers (Figure 3). On an

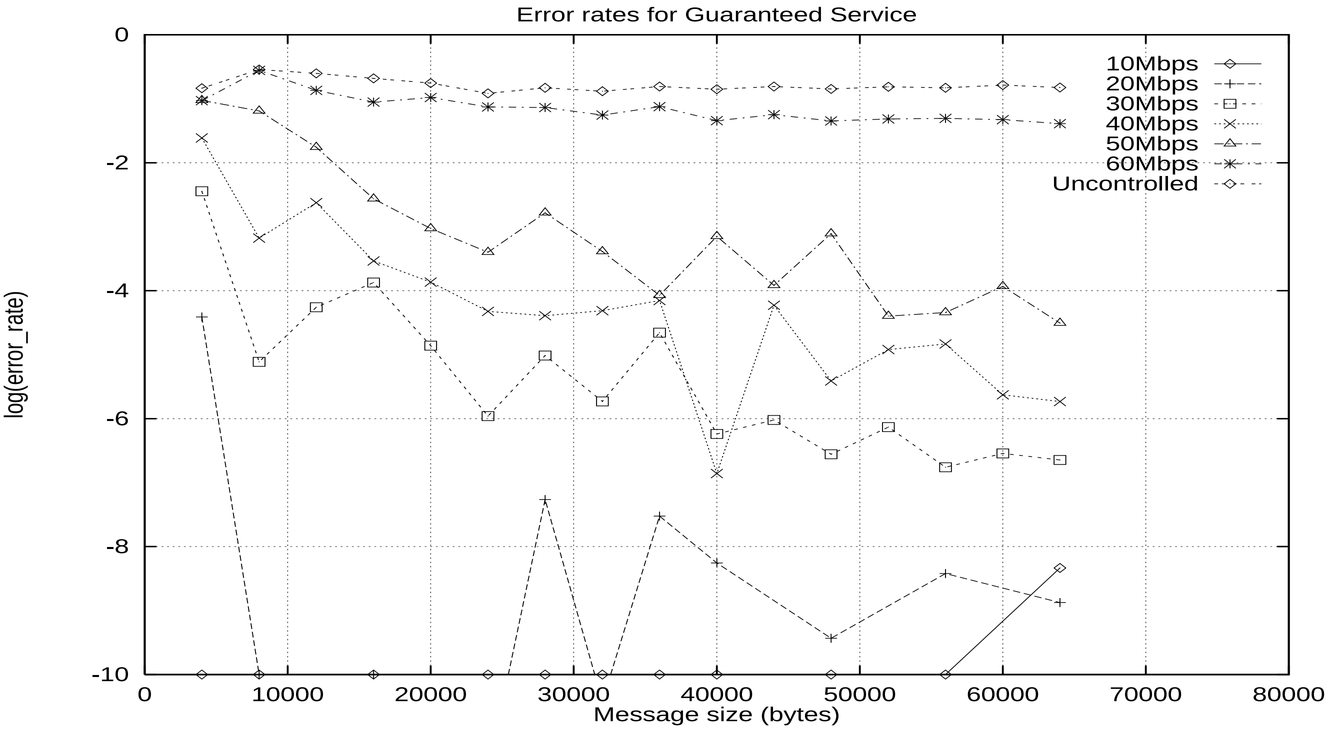

Figure 9: Loss rate at the receiver as a function of transmission rates (ISR drops excess packets). Zero loss is indicated by a value of 10^{-10}. Note that the error rate becomes very large when the sender overloads the receiver.

interrupt, the driver places a newly filled buffer in a per-VC queue, and hands freed buffers to the card for receiving future AAL 5 frames. We set a limit on the number of buffers a connection can have in its per-VC queue. This is to provide limited per-VC fairness so that one connection does not hog all the buffers in the receiver, blocking other, slower connections.

When a packet is received for a connection that has reached its queue limit, the interrupt service routine drops the packet. Thus the work done in reassembling the packet and fielding the interrupt is wasted. As the sending rate increases beyond the receiver's capacity, more and more packets are dropped in the ISR, wasting more and more CPU cycles, which leads to a decrease in receiving rate even though the sending rate is increased. This process is called receive livelock[22].

Another cause of receive livelock is that our message semantics demanded that unreliable connections should not get partial messages. So, on a loss, parts of a message that have been correctly received have to be discarded, which wastes work. We thought we could eliminate some of the wasted effort by changing the message semantics for unreliable connections, and delivering messages even if they have one or more TPDUs missing, letting the application decide what to do with them. Surprisingly enough, this degraded the performance instead of improving it! The reason is that when the transport layer drops entire messages, it frees many buffers for reassembly at once. This makes it less likely that future frames will be dropped in the ISR. Thus, even though we are wasting work in the transport layer, we are avoiding wasted effort in the ISR. This leads us to the conclusion that the loss of work in the ISR is more critical than in the transport layer. We will now describe a scheme that ensures that the ISR does not waste any work on an interrupt - any packet losses are done by the host-adaptor card before an interrupt.

Note first that this scheme has a fairness problem. The current version of the software on the Fore card has

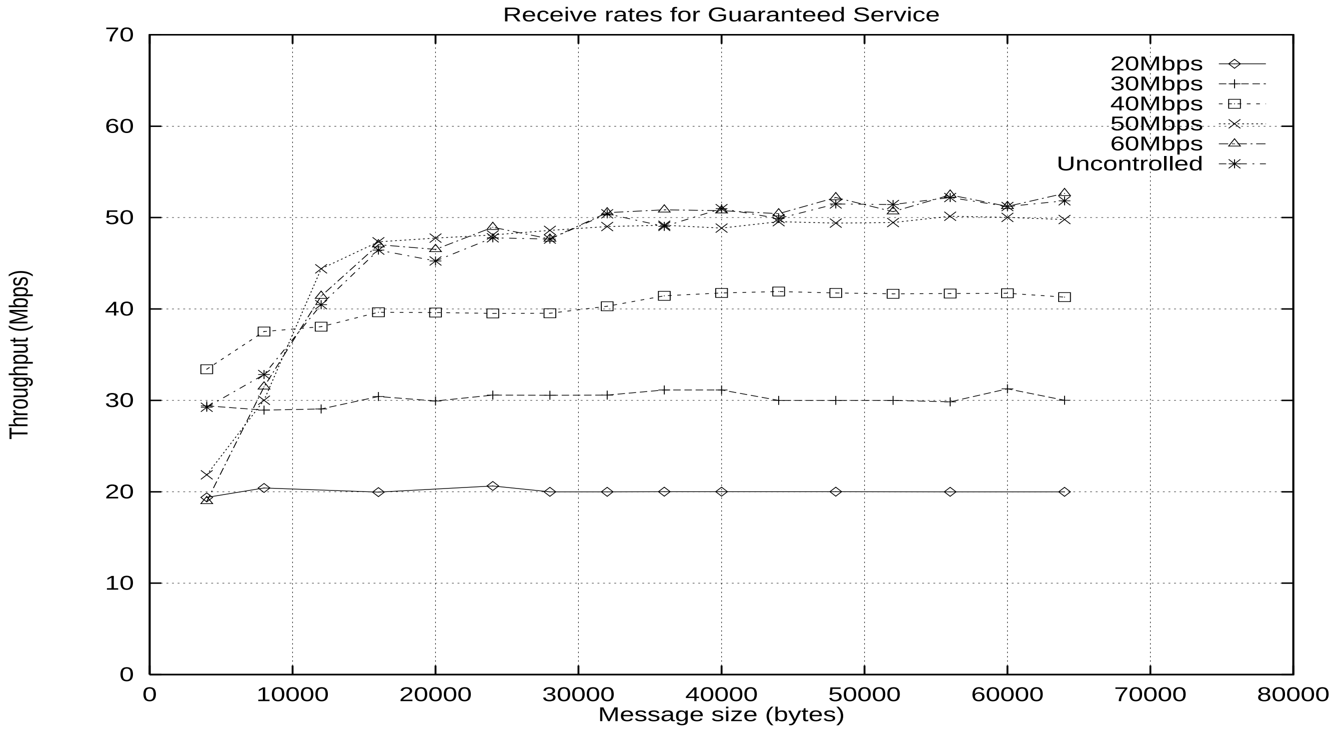

Figure 10: Throughput measured at the receiver for different transmission rates (host-adaptor drops excess packets). Note that there is no receive livelock, the throughput does not drop as the sender sends faster.

only one queue of received buffers (instead of a per-VC queue). So, if the card drops frames, it cannot do that fairly - it will just drop the tail of the queue. Unfortunately, since we cannot control the microcode on the card, we must compromise on per-VC fairness for performance. We conclude that we should do per-VC queuing in the host-adapter if we want per-VC fairness.

The easiest way to move packet losses from the ISR to the host-adaptor is to mask the interrupt from the card during ISR processing, so that the card does not interrupt the CPU when it is overloaded. Once the host processes the pending work, it can again enable the interrupts on the card. Specifically, we can have a high watermark and a low watermark on pending work for disabling and enabling the interrupts. When the pending work goes beyond the high watermark, we disable the interrupt, and when we have done sufficient processing so that the backlog goes below the low watermark, we can enable the interrupts again. However the Fore HPA-200 adaptor does not provide the ability to disable and enable the interrupts while running. The card either always interrupts or never interrupts, depending upon how it is initialized at boot time. So we had to solve this problem indirectly. Instead of masking the interrupt, we stop supplying free buffers to the card on crossing the high watermark. Soon the card runs out of buffers for reassembly, and hence starts dropping packets without interrupting the CPU. Once we go below low watermark, we start supplying buffers to the card again.

Once we made these changes, the throughput improved dramatically (Figure 10). It is clear that we do not have the problem of receive livelock any more since an increase in the transmission rate does not cause a decrease in the reception rate, even though the loss rate increases.This is because we are not wasting any work in dropping packets in the ISR. So the host processor, which is the bottleneck, does not waste CPU cycles in dropping packets and fielding unnecessary interrupts.

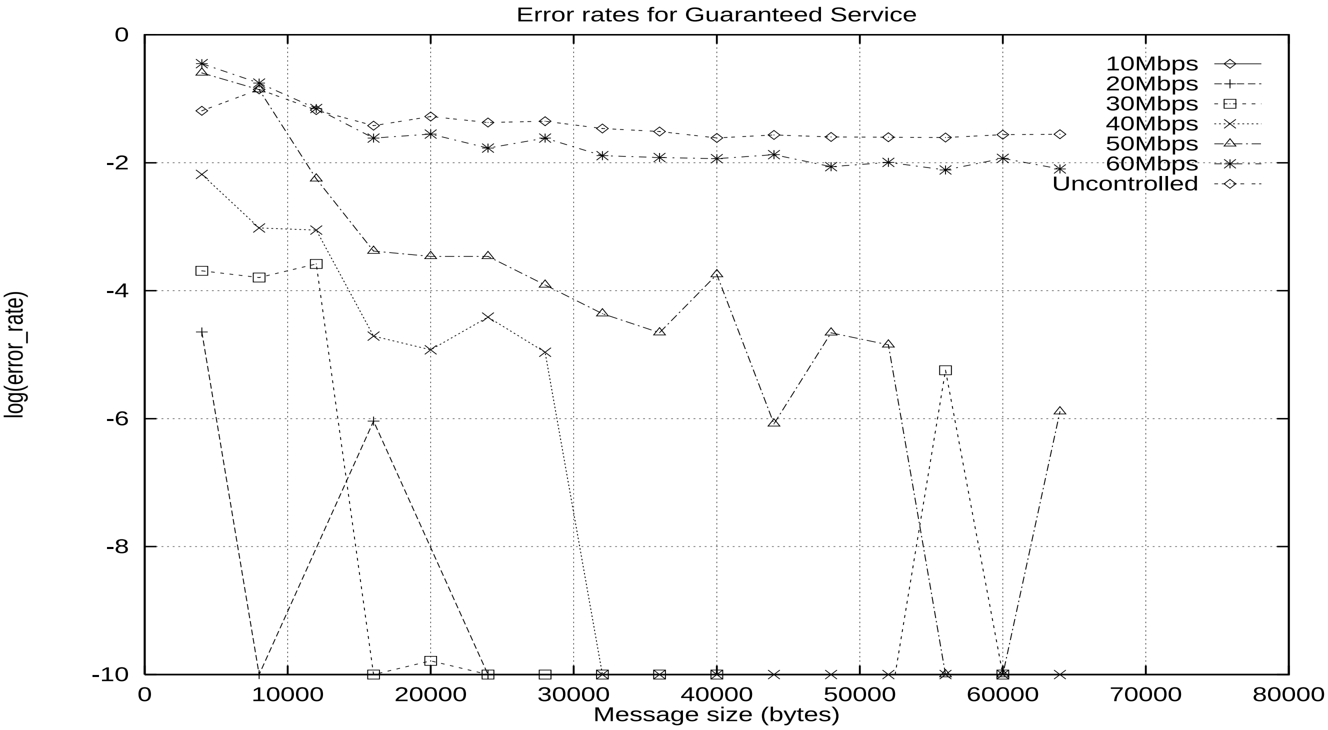

Figure 11: Loss rate at the receiver as a function of transmission rates (host-adaptor drops excess packets). Zero loss is indicated by a value of 10^{-10}. Note that the loss rate is still very large when the sender overloads the receiver. However, since the adaptor drops data, this loss does not result in receive livelock.

In order to gauge the benefit of having a high and low watermark, we measured the performance of a kernel where we did not have these watermarks. Without a high watermark, the host eventually runs out of buffers, and will stop supplying buffers to the card. However, in this case, as soon as even a single buffer becomes free, the driver gives it to the card, and the card uses it up immediately and gives a frame back to the host, again causing it to run out of buffers. Since the interrupt and buffer resupply happen for single buffers, we cannot amortize this overhead. With a high and low watermark, we can amortize the interrupt overhead over many packets. As a result, the received throughput is around 40Mbps when we do not use high and low watermarks, as compared to more than 50Mbps when using these watermarks.

When the ISR drops packets, the receive performance of a reliable connection is better than the receive performance of a best-effort connection, but now the situation is reversed. This is because best effort connections earlier suffered from receive livelock, unlike reliable connections, which is not the case any more.

6.3 Per-Layer Cost

There is considerable variation in the amount of work done in each part of the protocol stack. Some parts of the stack do constant work per message, some parts do constant work per TPDU, and others have an overhead per byte. In this section, we divide the entire work into different components and see where CPU cycles are being consumed. First we consider different components of the transport layer's `t_send` function, which are as follows.

1. Time spent to make a write system call, which is a per-message cost.

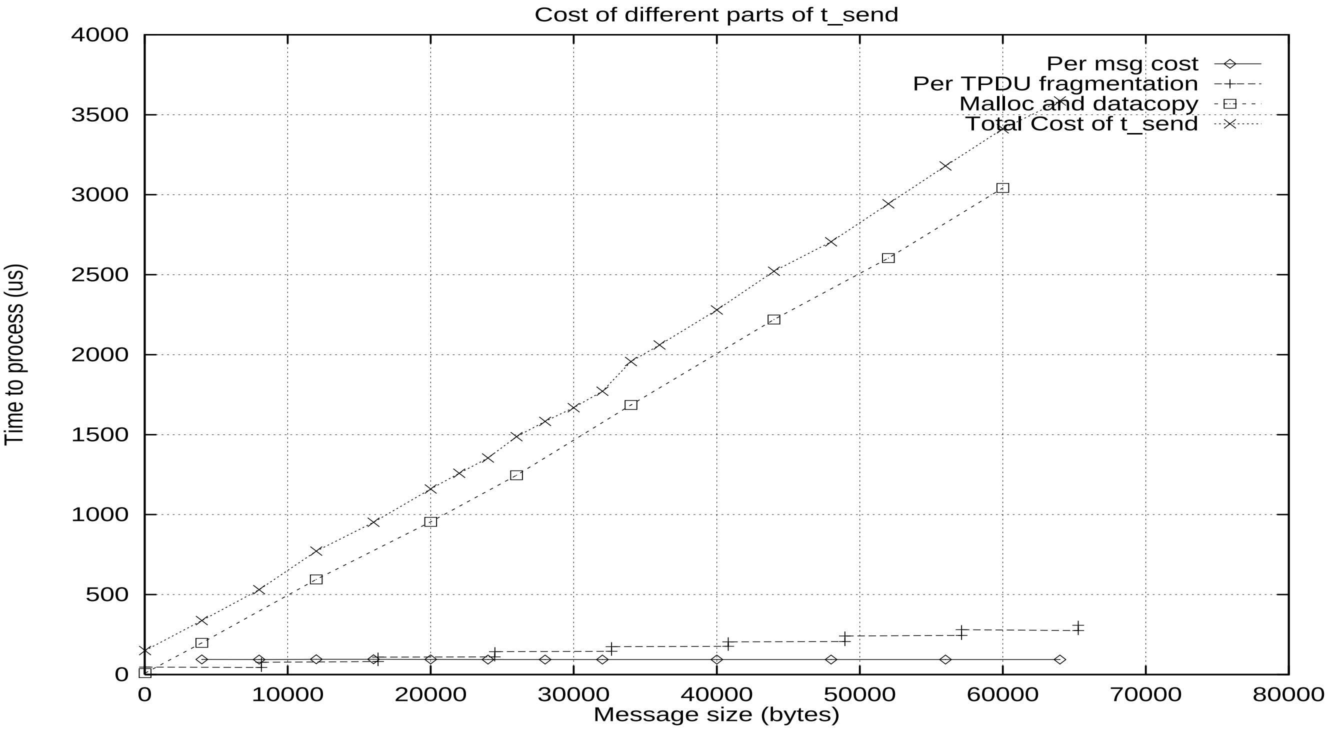

Figure 12: Processing cost for different components of the t_send interface procedure as a function of message size. Note that the per-byte overheads dominate for large messages.

2. Time spent in enqueuing the incoming message in the PSB without fragmenting the message or doing any datacopy. This is a per-message cost.
3. Time spent to fragment the message into TPDUs and enqueueing them in the transmission queue. This gives us the per-TPDU cost.
4. Time spent in copying data from user space to kernel space. This gives us the per-byte cost.

Figure 12 plots the time spent in doing different types of processing in `t_send` as a function of the message size. The per-message cost, which is the time spent in making the write system call and enqueueing the message, is constant. This time is around 95 μs, of which 41 μs is spent in making the write system call and acquiring the Readers Writers lock (Section 5.5.2). The time spent in fragmenting the message into TPDUs and enqueuing them is a step function of message size. It has discontinuities at the multiples of TPDU size. As the number of TPDUs required to carry the message increases by one, this cost jumps by around 35 μs. The third component is the time spent in copying data from user to kernel space. This time also includes the time to allocate memory in the kernel for copying data given by the user application. As seen in the plots, this time is directly proportional to the message size and is about 50 μs per Kilobyte. It is clear from the plots that for small messages (less than around 3000 bytes), the major component of the cost is the per-message and per-TPDU cost [14]. After that, the cost of copying data starts dominating. The per-message and per-TPDU costs together make up only a small part of the total time spent in `t_send` for larger messages.

The remaining processing costs can be divided into the following components (Figure 13).

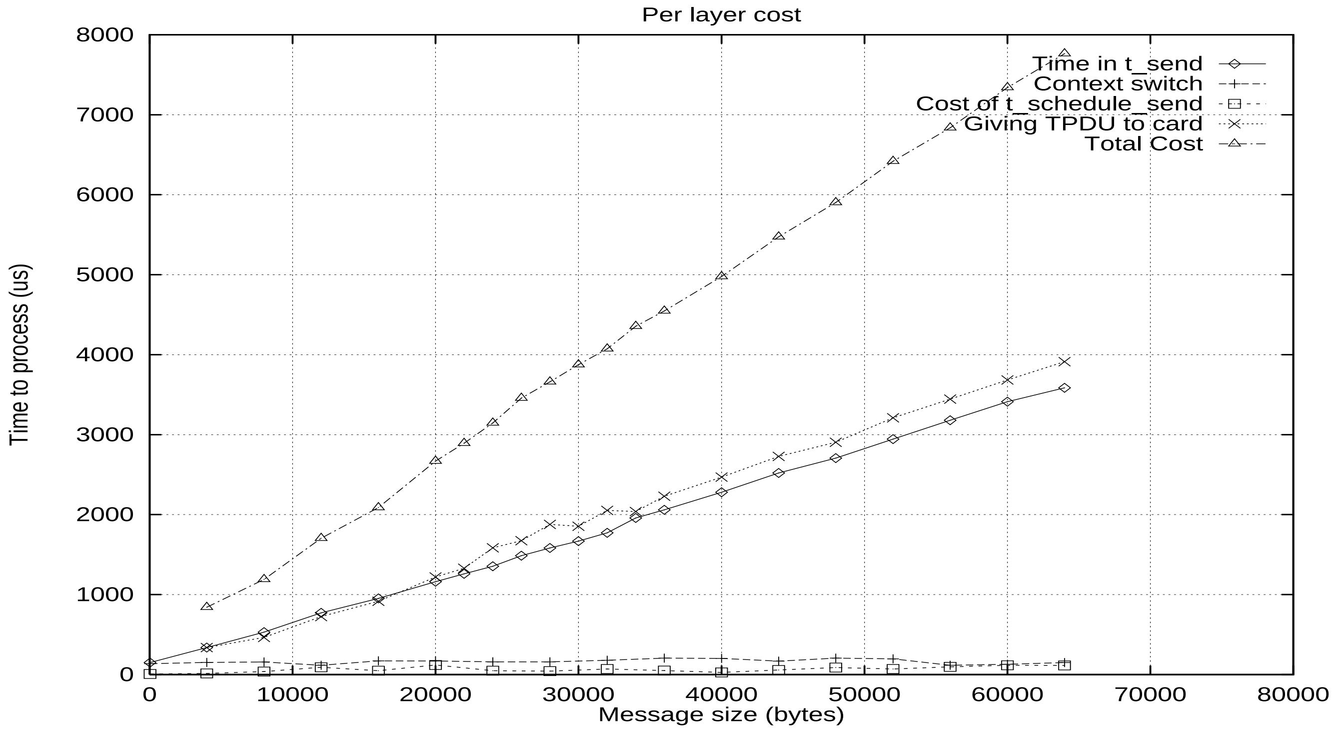

Figure 13: Processing cost for different parts of the stack as a function of message size. Note that the per-byte messages dominate for large messages.

1. The cost of context switch, from the user context in which the write system call is made, to the context of the task scheduler which executes the `t_schedule_send` task.

2. The cost of processing done by `t_schedule_send`, which hands over the TPDU to the device driver after processing.

3. The time spent on the TPDU in the device driver, which issues the transmit command to the card, telling it which VCI and AAL to use for sending this TPDU, and the DMA address and size of the TPDU in the host memory. This also includes the time spent in other house-keeping operations like recovering memory of the TPDUs which have been transmitted by the card, and the bus transfer time.

The context switching cost is more or less constant and remains same irrespective of the message size. It forms a major component of the total cost for very small messages (less than 1K), the other major component being the cost of `t_send`. The task `t_schedule_send` does very small amount of processing for unreliable connections. All it has to do is to take the next TPDU from the queue and hand it over to the device driver, after putting in the right message identifier, and the sequence number and offset within the message. As messages become slightly bigger, the time spent in the device driver and the device itself increases rapidly, and the main parts of the total cost are now the cost of `t_send` and the cost of processing in the device driver. Both these components increase almost linearly with increase in message size, due to which the total processing cost also increases almost linearly.

7 Related Work

It is useful to contrast the semantics of our transport layer with TCP. While TCP's error control uses similar mechanisms, the flow control mechanisms differ considerably since we propose rate based flow control independent of the error control window size. Further, unlike TCP, data checksumming is done at the network layer.

Our work differs from IP-over-ATM in many ways. In the IP-over-ATM approach, the application sees only the IP interface, which does not provide any QoS guarantees. Thus, any guarantees available from the ATM network are hidden. Second, the IP-over-ATM subsystem has to make signaling requests on behalf of the application, which adds considerable complexity to the kernel. In our approach, signaling is called directly from the application library. Third, IP routing assumes a broadcast medium in the local area, which is critical for the ARP protocol. IP-over-ATM has to spend a lot of effort emulating this over the point to point ATM network. By using a native mode ATM stack, all these problems are automatically eliminated.

Our work is closely related to that of Campbell et al [1] who have proposed a multimedia enhanced transport service and a Quality of Service Protocol Architecture [2]. As in our stack, they have placed flow regulation at the transport layer, and have no logical multiplexing of streams. However, they have decomposed the service interface into guaranteed-performance and best-effort flows. This hides the orthogonal aspects of flow control, error control, and QoS specification that our transport layer explicitly presents. As a consequence, their interface does not allow for some combinations that may prove to be useful, such as non-error controlled but feedback flow controlled flows, which is an alternative way to carry Variable Bit Rate video traffic [13]. Second, their stack provides a complex API - for example, applications are expected to provide dummy upcalls for computing the average time taken for a user task. We think that this complexity can result in poor performance. Finally, their choice of a QoS specification seems premature since much is unknown about what is really needed.

There have been numerous attempts to design transport layer protocols in the past. Of these, one of the more complete attempts is the TP++ project [7]. TP++ has several interesting features such as forward error correction, timer-based connections and congestion control based on backpressure. While per-VCI backpressure can require considerable feedback from switches, which is hard to legislate in a multi-vendor framework, several of their ideas are orthogonal to our innovations, and we plan to consider them in future work.

8 Status and Future Work

The transport layer we have described in this paper is essentially complete, and has been operational on our testbed for the last three months. While we have reported several performance tuning experiments, other experiments are also in progress. We hope to release this code into the public domain shortly. A port to the FreeBSD operating system is also in progress.

The one open area for research is in the implementation of the Resource Manager. This component interacts strongly not only with the operating system scheduler, but also with the disk sub-system and the graphics sub-system. We believe that it is an open, but worthwhile challenge, to manage OS resource in the same way we manage network resources in order to provide QoS guarantees in the end-system.

9 Conclusion

We have described the design, implementation, and performance tuning of a native-mode ATM protocol stack. Our design is novel in that it is tuned to AAL5 frame transport, and a design from scratch has allowed us to incorporate new error and flow control protocols. Our transport layer provides reliable or unreliable data transfer. An unusual feature is leaky-bucket policing at the transport layer. We have implemented the transport layer in a research operating system and have extensively measured its performance. This has allowed us to tune our implementation to fit the resource constraints common in current-generation Personal Computers.

Our stack has excellent performance, with throughputs greater than 50 Mbps user-to-user for unreliable data transfer. End-to-end delays are smaller than 750 μs. This performance is possible because of careful design to avoid data-copying overheads, amortizing costs over multiple TPDUs, and minimizing wasted work at the receiver. We believe that these insights are valuable to other protocol stack implementers.

10 Acknowledgments

Sandeep Gupta from University of Maryland, College Park, added mbuf code and removed extra data copies. Puneet Sharma from USC reverse-engineered the microcode download program and wrote the first version of the ATM device driver. Our thanks to them both.

References

[1] *A. Campbell, G. Coulson and D. Hutchison*, A Multimedia Enhanced Transport Service in a Quality of Service Architecture, Proc. 4th International Workshop on Network and Operating System Support for Digital Audio and Video, November 1993.

[2] *A. Campbell, G. Coulson and D. Hutchison*, A Quality of Service Architecture, ACM Computer Communications Review, April 1994.

[3] *D.D. Clark, V. Jacobson, J. Romkey and H. Salwen*, An Analysis of TCP Processing Overhead, IEEE Communications Magazine, June 1989, pp 23-29.

[4] *D. Comer*, Internetworking with TCP/IP Principles, Protocols and Architecture, Prentice Hall, Englewood Cliffs, NJ, 1988

[5] *E. Biagioni, E. Cooper and R. Sansom*, Designing a Practical ATM LAN, IEEE Network Magazine, March 1993.

[6] *D. Feldmeier*, Multiplexing Issues in Communication System Design, Proc. ACM Sigcomm 1990, October 1990, pp. 209–219.

[7] *D.C. Feldmeier*, An Overview of the TP++ Transport Protocol Project, High Performance Networks-Frontiers and Experience, A. Tantawy ed., Kluwer Academic Publishers, Boston, MA, 1993, pp 157–176.

[8] *D. Ferrari*, Client Requirements for Real-Time Communications Services, IEEE Communications Magazine, Vol 28, No. 11, November 1990

[9] *FORE Systems, Inc.*, Programmers's reference manual for FORE HPA-200 EISA ATM card.

[10] *FORE Systems, Inc.*, SPANS: Simple Protocol for ATM Network Signaling.

[11] *ATM FORUM*, ATM: User Network Interface Specification Version 3.0, Prentice Hall, September 1993

[12] *V. Jacobson*, Congestion Avoidance and Control, Proc. ACM SigComm 1988, pp 314-329, 1988.

[13] *H. Kanakia, P.P. Mishra and A. Reibman*, An Adaptive Congestion Control Scheme for Real-Time Packet Video Transport, Proc. ACM SigComm 1993, August 1993.

[14] *J. Kay and J. Pasquale*, The Importance of Non-Data Touching Processing Overheads in TCP/IP, Proc. ACM SigComm 1993, pp 259-269.

[15] *S. Keshav*, Congestion Control in Computer Networks, PhD Thesis, UC Berkeley, published as UCB TR 91/649. Available as ftp://research.att.com/dist/qos/keshav.th.tar.Z

[16] *S. Keshav*, Packet-Pair Flow Control, Submitted to IEEE/ACM Trans. on Networking, preprint available from http://www.cs.att.com/csrc/keshav/papers.html.

[17] *S Keshav and H Saran*, Semantics and Implementation of a native mode ATM protocol stack, AT&T Bell Laboratories Technical Memorandum.

[18] *S.J. Leffler, M.K. McKusick, M.J. Karels and J.S. Quarterman*, The Design and Implementation of the 4.3BSD UNIX Operating System, Addison-Wesley, 1989.

[19] *Puneet Sharma*, Internal report on the device driver for FORE HPA-200 EISA ATM card for Brazil.

[20] *R. Pike, D. Presotto, S. Dorward, R. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom*, Plan 9 from Bell Labs, In Plan 9 - The Documents - Volume Two, Harcourt Brace & Company pp 1-22, July 1995.

[21] *K.K Ramakrishnan and R. Jain*, A Binary Feedback Scheme for Congestion Avoidance in Computer Networks, ACM TOCS, Vol. 8, No. 2, May 1990, pp. 158-181.

[22] *K. K. Ramakrishnan*, Performance Considerations in Designing Network Interfaces, IEEE Journal on Special Areas in Communications Special Issue on High Speed Computer/Network Interfaces, Feb-1993.

[23] *Signalling and Operating System Support for Native-Mode ATM Applications, Proc ACM Sigcomm 1994, September 1994.*

[24] *C.A. Sunshine and Y. Dalal*, Connection Management in Transport Protocols, Computer Networks, 1978, 454-473.