Design, Implementation, and Performance Measurement of a Native-Mode ATM Transport Layer (Extended Version)

R. Ahuja, S. Keshav, and H. Saran

April 30, 1996

Abstract

We describe the design, implementation, and performance measurement 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, provide per-virtual circuit Quality of Service guarantees, and take advantage of ATM Adaptation Layer 5 functionality. It provides reliable and unreliable data delivery with a choice of feedback and leaky-bucket flow control. These services can be combined to create per-virtual-circuit customized transport services. Our work is novel in that it provides high-performance, reliable, flow-controlled transport service using cheap Personal Computers (PCs).

We describe the mechanisms and the operating system support needed to provide these services in detail. An extensive performance measurement allows us to pinpoint and eliminate inefficiencies in 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 for small messages is around 720 μs. These figures compare favorably with the performance from far more expensive workstations and validate the correctness of our design choices.[1]

Keywords: ATM, Transport Layer, Personal Computer, AAL 5, Native-mode ATM.

1 Introduction

Most current ATM networks use TCP as the transport layer, with IP-over-ATM providing the network layer [5, 4]. This approach, though necessary in the short-

[1]An earlier version of this paper was presented at the IEEE Infocom'96 Conference. The paper was selected by the conference as one of its top papers and referred to the Transactions for possible publication after the Transactions' own independent review.

term, is likely to soon prove inadequate for three reasons. First, ATM networks will provide end-to-end Quality of Service (QoS) guarantees to individual virtual circuits [12]. These guarantees are lost by IP, since it multiplexes multiple transport connections with disparate QoS requirements onto a single VC [9, 31]. Moreover, TCP cannot directly use the QoS guarantees provided by an ATM network because it neither obeys a leaky-bucket behavior envelope, nor responds 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, 16]. However, ATM Adaptation Layer 5 (AAL5) already does data checksumming. Thus, this TCP functionality is redundant and costly. A transport layer that turns off data checksumming will deliver higher throughput than TCP/IP-over-ATM, since can eliminate this overhead. Third, TCP has inherited the patches and fixes of two decades of protocol tuning [30]. Despite this, or, perhaps, due to it, TCP performance is unpredictable and heavily dependant on particulars of the operating environment [33]. Even small changes, such as the in the loss rate or the buffer size in intermediate routers, can dramatically affect performance [7].

We believe there is a need for a transport layer that (a) can provide guaranteed service quality relatively independent of the operating environment, (b) exploits the services of an underlying AAL5 layer, and (c) that has been designed afresh to provide clean semantics. We describe the design, implementation, and performance measurement of such a transport layer, that is targeted specifically for Asynchronous Transfer Mode (ATM) networks i.e. a *native-mode* ATM transport layer. The layer embodies much of our past work in flow and congestion control [19].

In this paper, we not only sketch out the design of our transport layer, but also give insights into its implementation. [2] We had to redesign many aspects of our implementation to achieve our dual goals of high performance and per-connection service quality. We describe how we measured and tuned our system step-by-step to achieve workstation-quality performance on low-end Personal Computers.

Perhaps the main insight from our work is that it is hard to achieve per-virtual circuit quality of service guarantees with current operating systems and host-adaptor cards. Most Unix-like operating systems have no support for real-time processes, hard process priorities, or fine-grained timers. In the absence of these facilities, it is impossible to guarantee strict performance bounds to real-time applications. Second, we did not have access to the microcode on the host-adaptor card. Thus, we had to work around its deficiencies, such as supporting only a single receive queue shared by all VCs, which allows low-priority connections to adversely affect higher-priority connections. Third, to

[2] The transport layer described in this paper was originally designed and implemented as part of the IDLInet project at the Indian Institute of Technology, Delhi. The implementation described in this paper, though similar in spirit to the original, is almost totally rewritten, reflecting substantial improvements in the design.

achieve high performance, we had to tune *every* aspect of the protocol stack. Even small changes in the implementation can dramatically affect performance. Other workers in this field have also come to the same disheartening conclusion [3, 16]. Fourth, it is hard to implement locking in the kernel. Choosing how large a section to lock and when to release a lock are often matters of good taste rather than hard science. Unfortunately, poor choices can lead to degraded performance, or worse, deadlock. Finally, we found that Personal Computers are not as low-end a platform as one might imagine. With careful design and implementation, it is possible to extract workstation-quality performance from a platform that is cheap, easily available, and costs a fifth as much!

2 Service Description

We believe that the set of services provided by the transport layer should match the anticipated application workload. Specifically, we anticipate a demand for two types of service qualities from future applications [11]. Continuous media applications 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 contract. For these applications, the transport layer should reserve resources in the kernel and host-adaptor card, and should ensure that the application stays within its specified traffic envelope. We would also like to support data applications, which demand reliability (that is, the abstraction of error-free, in-sequence packet delivery), and as high a throughput as possible. To provide these applications with their desired QoS, the transport layer must provide error-control, flow-control, and high performance. Finally, some applications may require a raw bit-stream abstraction upon which they can build custom flow and error control mechanisms. For these applications, the transport layer should allow raw access to the AAL5 virtual circuit, simultaneously ensuring that data transfer from this class of connections does not adversely affect the performance guarantees of the other two classes. Note that the second and third service classes correspond roughly to the Internet's TCP and UDP protocols.

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. These are: 1) simplex data transfer, 2) error control, 3) open-loop and feedback flow control, 4) unlimited application message size, 5) a choice of blocking and non-blocking application interface, and 6) a choice of byte stream and message transfer semantics.

Currently, we support three combinations of the above services. These are *guaranteed-performance service*, *reliable service* and *best-effort service*. All three services support options 4, 5, and 6. Guaranteed-performance service ensures that an application conforms to pre-specified leaky-bucket parameters. Moreover, an application's QoS specification (currently, only the bandwidth requirement) is made available to the network and every layer of the protocol stack, so that adequate resources may be reserved. Reliable service provides error-detection, timeouts and retransmissions (for error control) and feedback flow control. With best-effort service, there is no flow control, error control (other than that provided by AAL5), or provision of QoS guarantees. No resources are reserved for reliable and best-effort service. [3] Our transport layer, as currently implemented, can support multicast for unreliable and guaranteed-performance services if multicast-VCs are available at the ATM level. It does not provide reliable multicast service. We will develop other services derived from the basic services above, as the need arises.

3 Design Principles

We use five basic principles in designing our stack. First, we want to eliminate multiplexing of virtual circuits [9, 31] other than at the physical layer. Some network 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 many virtual circuits per-endpoint, we need not multiplex transport connections, allowing us to provide per-VC QoS all the way from the ATM layer to the application. [4] Of course, this does not preclude an application from multiplexing multiple media streams on to a single transport connection. Our aim is to give applications the option of opening a separate transport connection per media stream.

Second, we want a clean separation of 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, so that losses in the network automatically affect the flow rate. This severely degrades performance in high-loss environments, such as wireless links.

Third, we want to provide 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. Indeed, our performance results validate their insightful conclusions.

Fourth, we do not want to replicate functionality provided by AAL5 or ATM

[3] Unfortunately, in the implementation described in this paper, we did not have access to the host-adaptor card's microcode. So, we were neither able to reserve resources for guaranteed-service connections on the card, nor give certain connections priority over others.

[4] Unfortunately, the host-adaptor used in our implementation, a FORE Systems HPA-200 card, uses a single receive queue for queueing packets from all receivers. Since we do not have access to the microcode on the card, we cannot avoid this level of multiplexing. However, on a card interrupt, we immediately move packets to per-VC buffers, so that the effect of sharing the queue is minimized.

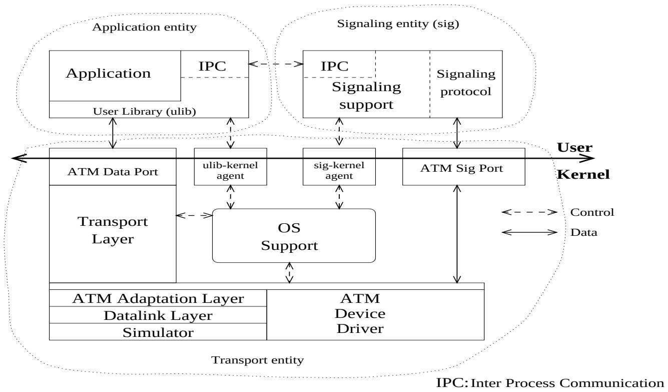

Figure 1: Top level view of the ATM stack. The stack consists of three entities: the application entity, the signaling entity, and the transport entity.

signaling. For example, connection management, traditionally a transport layer function, is relegated to signaling. Similarly, data checksumming is done only by the ATM adaptation layer.

Finally, we want our implementation to be highly portable. Our transport layer is implemented in three different testbeds, each with its own hardware, operating system (OS), and signaling protocol. To minimize work in porting our implementation, we have isolated all OS and hardware dependant code with general, clean and well-defined interfaces (Figure 1. We also wrote our own memory management, task scheduling, timers, locking, and signaling code. The only support the protocol stack needs 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 modern operating systems.

We believe we have achieved our goal of portability since the stack (including the transport layer) was first implemented and tested on the REAL packet-level simulator [18], and then ported to the DOS, IRIX, Solaris, FreeBSD, and Brazil[5] operating systems on PC, SPARC, and Silicon Graphics hardware. For clarity, this paper discusses only the Brazil implementation on PCs. The FreeBSD implementation is described in Reference [14].

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.

An end-system implementing our stack is shown in Figure 1. The ATM stack

[5]Brazil is a research version of the Plan 9 operating system from Bell Labs [25]

consists of three main entities: the application entity and the signaling entity in user space, and the transport entity inside the kernel. Note that 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.

The application (user program) is linked to an OS-specific user library (`ulib`) that provides network access, session layer services (such as duplex channels) and isolates the application from the underlying OS and hardware platform. The services provided by `ulib` are similar to the Berkeley socket interface [21], except that applications can specify QoS parameters during connection set-up. The question of what QoS parameters an application should specify is a matter of much debate. For the moment, an application may only ask for a bandwidth, which is assumed to be its peak rate requirement, and reserved as such. We will make this specification richer as we gain more experience with QoS-sensitive applications.

The similarity of the user library interface to the socket interface allows us to easily port applications written for Berkeley sockets. For example, we successfully ported a video-on-demand client and server written using TCP/IP and Berkeley sockets to our stack in about an hour: only the socket-related system calls had to be replaced with equivalent calls to user library. More details on the `ulib` interface can be found in Reference [29].

The signaling entity establishes connections on behalf of user applications and tears down connections either when requested by an application or in the event the application crashes [29]. Since the signaling entity must survive application crashes, it cannot be part of the user library. All applications on an end-system share a single signaling entity. The signaling entity is partitioned into the signaling support and signaling protocol components to allow us to change the signaling protocol without affecting the rest of the code. For example, the FORE Systems' SPANS protocol that we implement now could be replaced with Q.2931 without affecting the part of the signaling entity that cleans up after an application crash. The signaling entity is in user space so that it is easy to modify and to port to other platforms. On a production platform, for efficiency and security, this should probably reside in the kernel.

The transport entity has components both in the data plane and the control plane. In the data plane, it moves data between the application and the host adaptor. (We assume that the host-adaptor provides AAL5 frame transport, as is the case with all modern host-adaptors.) In the control plane, it is responsible for call admission and allocating resources to VCIs for guaranteeing bandwidth.

The transport entity consists of the transport layer, the device driver, and the OS support module (Figure 2), which all reside in the kernel. The main reason for placing the transport entity in the kernel is to achieve high performance. Second, we felt that a user-space implementation would have a harder time accessing OS resources such as clocks, timers, and interrupt handlers. Our experience with Brazil's IP implementation, which is in user space, convinced

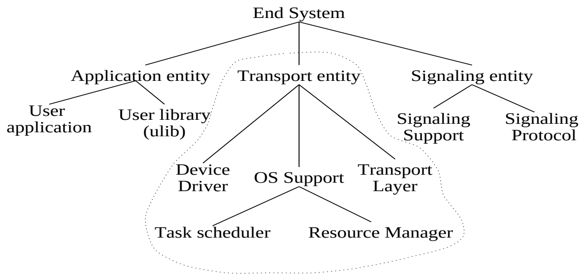

Figure 2: Components of the ATM stack. This paper deals mostly with the transport entity.

us that a user-space implementation of the transport layer would have poor performance because of the extra data copies and context switches for every message read and write. Recent work by Thekkath et al [32] and Edwards and Muir [8] suggest that with careful design, it might be possible to implement the transport layer in user space and still achieve good performance. We intend to explore this in future work.

We now focus our attention on the transport entity.

5 The Transport Entity

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 task scheduler and a resource manager for managing local resources. We now describe the functionality of the transport entity in detail, starting with the OS support module.

5.1 OS Support: 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 supposed 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 its work.

In the Brazil kernel, the task scheduler is a kernel thread that periodically (in our case, every 50 ms[6]) handles any expired timers, and then calls any scheduled

[6] In Brazil, the smallest possible scheduling period is 10ms. The smaller the polling period, the better the granularity of handling timers, but the greater the CPU scheduler overhead. A scheduling period of 50 ms represents a compromise between overloading the CPU scheduler

tasks 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 its 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) is defined as one unit of work.

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 VCIs, and schedules tasks round-robin within the same priority. Hence we can allocate different QoS to different connections. Specifically, we give the highest priority to guaranteed-performance connections, with a weight corresponding to their bandwidth allocation, so that they are able to meet their delay and bandwidth bounds. Reliable connections get an intermediate priority and a unit weight. Finally, best-effort connections get the lowest priority. Best-effort tasks are handled only when no higher-priority tasks await service. Within each service class, tasks are serviced in round-robin order.

Our design has several advantages. First, interface procedures 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 packets. 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 very cheap. This allows us to easily exploit 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 a set of efficient timer routines for this purpose.

5.2 OS Support: Resource Manager

The resource manager is responsible mainly for admission control at the time of call setup. The admission test requires the manager to know the amount of CPU and network resources available to the transport layer, and the fraction of these resources that are already consumed. While performance measurements allow us to determine exactly how much CPU processing time each Transport Protocol Data Unit (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. 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. We intend to explore the use of *resource reserves* [22] for this purpose.

The resource manager is also responsible for cleaning up after an abnor-

---

and having poor granularity in setting timers.

mally terminated application. Specifically, if an application dies, the resource manager requests the signaling entity to tear down any associated connections. Symmetrically, if the signaling entity receives a teardown message, it requests the resource manager to mark the connection as unavailable. The details of this interaction are described in Reference [29].

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 *simplex* virtual circuits, error control, and flow control (duplex service is provided by `ulib`, which opens and manages two simplex transport connections). In addition, it segments application layer buffers into TPDUs and reassembles them on the receive side. Here, we outline 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. A user message that is larger than this size must, therefore, 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 should be chosen for each environment to trade off per-fragment overhead, the connection's error characteristics and the available timer resolution. Indeed, this is the choice of 'Multiplexing Block' in Reference [9].

In our implementation, we experimented with TPDU sizes of 4 Kbytes and 8 Kbytes, which matched the virtual memory page size of 4 Kbytes. Performance measurements, not presented here, showed that for reliable service (where we perform timeouts and retransmissions), we could achieve higher throughput with 4 Kbyte TPDUs than with 8 Kbyte TPDUs, because the retransmission overhead is larger for 8 Kbyte TPDUs. For convenience, we use 4 Kbyte TPDUs for all service classes. Adaptively matching the TPDU size to the operating environment is clearly an area for future work.

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 called SMART that is outlined below and

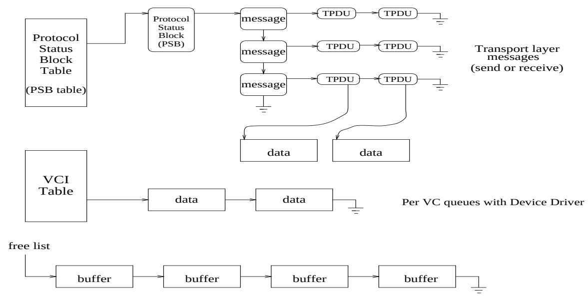

Figure 3: Data structures used for storing messages. Each protocol status block (PSB) has pointers to send and receive message lists. A message consists of a number of transport protocol data units (TPDUs), and each TPDU corresponds to a single AAL 5 frame.

described in more detail in Reference [20]. (Note that the transport layer does not do checksumming, since this is already taken care of by the AAL layer. Since checksumming requires the transport layer to touch every data byte, by avoiding it, we gain a substantial improvement in performance.)

The transport layer uses per-TPDU cumulative acknowledgments. and the acknowledgment also carries the sequence number of the TPDU that generated the acknowledgment. This allows sources to determine which sequence numbers have been correctly received. For example, if the receiver receives packets 1,2, and 4, it sends acks (1,1), (2, 2), and (2,4), where the first element of the tuple is the cumulative acknowledgment, and the second is the packet causing the acknowledgment to be sent. A source receiving these tuples can decide that packets 1, 2, and 4 have been correctly received, and that 3 has been lost (since ATM networks do not reorder packets). In SMART, a retransmission is triggered either by a repeated cumulative acknowledgment (fast retransmission) or by a retransmission timeout. During a fast retransmission, the sender scans the sequence numbers in the range from the cumulative acknowledgment to the packet being acknowledged, and retransmits packets not correctly received and not already 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 increased. If not, the transport layer retransmits the packet at the head of the error-control window. This scheme combines the efficiency of selective retransmission with the robustness of Go-back-N retransmission. It allows a sender to quickly fill a gap in the error-control window without stalling while waiting for a timeout,[7] or paying the overhead and com-

[7] The scheme is so robust that a timeout is a rare event. In fact, a bug in the code that handles timeouts in the protocol went undetected for several months because timeout seldom

plexity of a selective acknowledgment scheme. A detailed performance analysis of SMART can be found in Reference [20].

To reorder out-of-order TPDUs, the receiver must have a buffer at least as large as the error-control window. The error-control window size is negotiated by the peer transport layers 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 feedback flow control and open-loop flow control (see Section 5.3.4).

If the scheduling discipline at all the switches along the path is round-robin like, feedback flow control is based on Packet-pair flow control [19]. 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, that is, the capacity of the slowest server on the path from the source to the destination (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 [19]

Our transport layer tries to cleanly separate flow control and error control. Windows are used for error control and to size buffers at the transmitter and receiver. Rate-based 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, as in TCP, packet losses trigger a slowdown in the sending rate [26, 13], which may not be warranted by the current congestion level. Thus, for example, on a lossy wireless link, a TCP transmitter is unable to use the link capacity, since TCP assumes every loss is dues to congestion, and shuts down the flow control window [7].

If the network does not support round-robin scheduling, the transport layer is forced to use a dynamic-window flow control scheme similar to TCP flow control [13]. This has the same problem as TCP, that is, error control and flow control are linked. However, unlike TCP, we use the SMART retransmission strategy, instead of Go-back-N. This allows us to recover much faster from packet loss, improving flow control behavior as a side effect.

5.3.4 Open-loop Flow Control

The transport layer shapes guaranteed-performance traffic with a leaky bucket to provide open-loop flow control. We believe that the traffic shaping function

occurs.

should be located close to the application, so that it can quickly provide feedback to the application about its allowed flow rate. Specifically, an application sending data faster than its leaky bucket rate fills its input buffer and is put to sleep by the shaper. This control is much easier to implement at the transport layer than at the host adaptor or a remote Network Interface Unit (NIU), as is usually the case, because the host-adaptor or NIU cannot turn off user processes as easily as the transport layer.

Implementing leaky bucket shaping 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 determines how many TPDUs can be sent immediately and the earliest time that the next TPDU can be sent. The shaper sends as many TPDUs as it can and also sets a timer for the earliest time the next TPDU can be sent.

5.4 Data Flow

The transport layer provides four interface procedures visible to the outside world (see Figure 4). These functions provide an *asynchronous* interface to the outside world, and hence are designed to have minimal functionality, returning as soon as possible. All other transport layer processing is done by task which are executed synchronously under the control of the task scheduler, to take care of the *heavy-weight* protocol processing. Since tasks can be scheduled from asynchronous events, we need to lock queueing and dequeuing of tasks in the task queue with a global task lock. We refer the reader to Section 5.5 for a more detailed description of the locks in our implementation. Here, we will note only that all tasks are procedure calls from the task scheduler. Therefore, we do not need to lock data structure shared among tasks. We only need to lock access the task scheduler itself, and data structure shared between tasks and interface procedures (such as send and receive buffers).

The overall data flow from the source to the destination (Figure 4), involves the following steps.

1. The user application calls the ulib function `atm_write`, which acts as the session layer, selecting the appropriate data-descriptor in case of a duplex connection, and making a `write` system call on it. This, in turn, calls the `t_send` interface procedure in the kernel, which fragments the user message into TPDUs while simultaneously copying the data from user space to kernel space. The procedure schedules the `t_schedule_send` task and returns.

2. The task scheduler calls `t_schedule_send` which attaches a transport header to TPDUs marked as eligible by the flow control protocol, and calls the AAL layer's `a_send` routine. `a_send` hands the packet to the

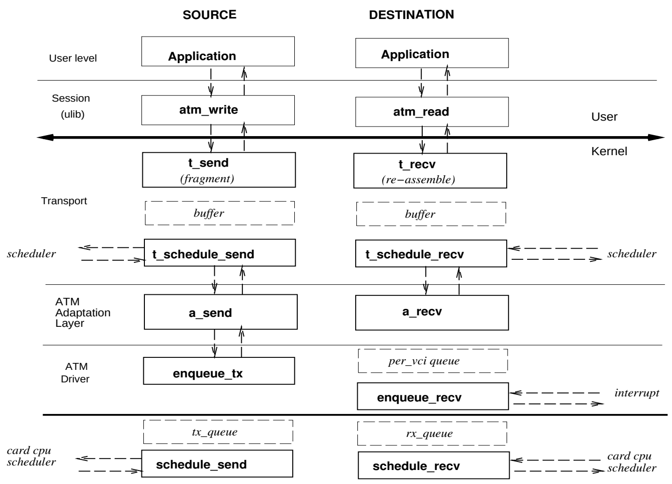

Figure 4: Data flow from source to destination. Asynchronous events are handled by interface procedures that spawn tasks which are handled by a task scheduler. A detailed explanation is in Section 5.4

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.

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 if it doesn't need to be retransmitted. Note that there are two copies on the send side: from user to kernel space, during the `write` system call, and from the kernel to the physical medium by the host-adaptor card.

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 `t_schedule_recv` task calls `a_recv` which retrieves the packet from the per-VCI AAL queue. The task checks the packet for validity and enqueues the packet at the right place in the received message queue. 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. The application reads a message by calling ulib's `atm_read` which selects an appropriate 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 awakened by the `t_schedule_recv` function when the next message is complete. Note that there are two data copies on the receive side: the DMA from the card to kernel space, and a copy from kernel space to user space during the `read` system call.

5.5 Implementation Experience

In this section, we will discuss some problems we ran into during the Brazil implementation.

5.5.1 Data copy in interface procedure

As described in Section 5.1, we wanted our interface procedures to quickly return after handling an asynchronous event, such as a read or a write from a user process, to minimize the response time. The task scheduler can then prioritize access to the CPU among more CPU-intensive tasks, allowing us to provide VCs with differential qualities of service. Unfortunately, we were unable to place the time-consuming data-copy operation in a task, because a user process can delete a buffer after writing it to the transport layer. Therefore, unless the interface procedure immediately copies the data into kernel space, it is possible that by the time a task is scheduled, the data has already been overwritten. While we could have blocked the user process in the write system call, awaiting the task scheduler to schedule a data copy task, in the interests of minimizing a user's response time, we decided to copy data into the kernel during the write interface procedure. This makes it a heavier weight operation that we would like. As we gain experience with the protocol stack, we may reconsider our decision. Our implementation structure is flexible enough to allow us to easily switch between the two alternatives.

5.5.2 Send and Receive Locks

Locking proved to be a major problem in doing an in-kernel implementation of our design. If locks are not carefully set and removed, the kernel can deadlock, a situation that is very hard to replicate and debug. Locking too large a region makes locking easier to implement, but has a performance overhead, because it forces tasks that are locked out to either sleep or busy-wait, both of which are expensive operations. Locking too small a region also has a performance overhead, because we need to frequently acquire and release locks. Thus, we had to balance these two objective, using good taste and common sense to decide where to place locks. We describe our choices in this and the next few sections.

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 transport layer tasks, we need to lock these queues with a per-VC Send lock and Receive lock.

5.5.3 Readers-Writers lock

In addition to the Send and Receive locks, 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. An application is put to sleep if it tries to do a read when 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 the readers-writers lock (`RWlock`), since this will prevent the signaling entity from doing any communication with the kernel. Hence, the readers-writers lock has to be *released* when the application is put to sleep, and *reacquired* on wakeup.

5.5.4 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 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

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 are in three main parts - latency in the protocol processing, throughput achievable with various services, and the costs of each component of the transport layer as a function of message size.

We took measurements on a testbed consisting of two IBM PC-clones with Intel 80486 processors running at 66 MHz. (As of early 1996, this would be considered bottom-of-the-line PC hardware, but they were nearly top-of-the-line when we started our work in mid-1994!) Each endpoint has 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 100 Mbps nominal bandwidth. The systems were otherwise unloaded, except for the fact that they were also connected to Ethernet, and were running a standard IP stack in user space.

6.1 Latency

Since the resolution of the Brazil CPU clock (in milliseconds) is not sufficient to measure processing delays, which are in microseconds, we had to measure these quantities indirectly. The general methodology was to create a bottleneck in the portion of the stack to be measured, then invert its throughput to determine its processing delay. We measured the throughput by sending 10,000 64-byte messages [8]. For example, to measure the cost of the write system call, we generate messages as fast as possible from a user process, and drop the message just before it would be handed to the transport layer. By measuring the transmission rate 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.

[8] Even though an application sends the same message (that resides in the same memory area) every time, it is copied into a *new* TPDU inside the kernel on each `write` system call. So our measurements are not affected by the fact that data is always in the processor's cache.

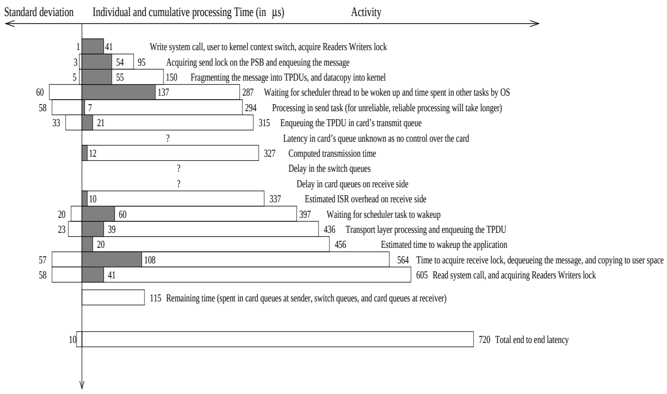

Figure 5: Individual (shaded) and cumulative delay and standard deviation at various points of the protocol stack when sending 64 byte packets between two otherwise unloaded machines. We have estimated times that we could not measure because we did not have access to the host adaptor microcode

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 (32 bytes of user data and a 32 byte transport header) was ≈120ms if we did not explicitly wake up the task scheduler after scheduling a task on asynchronous send and receive events. This latency was mainly due to the delay in scheduling, and is unacceptably high. On an average, the processing of a packet was delayed by 25ms at each end, while waiting for the scheduler to wakeup after its normal 50ms sleep period. To reduce end to end latency, we modified the Interrupt Service Routine (ISR) and the `t_send` routine to wake up the scheduler on receiving a packet from the device, and on receiving a packet from the user application, respectively. These changes reduced the round trip time to a mean of 1.44ms with a standard deviation of 0.01ms as shown in Figure 5.

The total time taken by the transmitter before the data is given to the card for transmission is roughly 315 μs. Even after explicitly waking up the task scheduler on an asynchronous event, the bulk of this time is still in waiting for the task scheduler to become active (the waiting time decreases from 25 ms to around 140 μs). While we could have further cut down this latency by eliminating the task scheduler and making a direct call into the `t_schedule_send` task from the `t_send` routine, this would have prevented us from prioritizing among transport connections, which is necessary for providing per-transport connec-

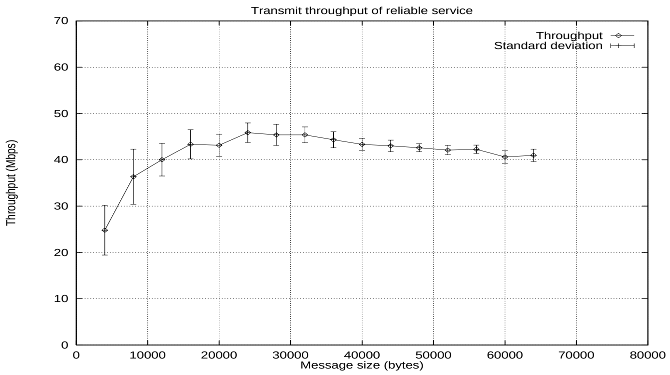

Figure 6: Transmission rate measured at the sender for a reliable connection. Note that the transmission rate increases, then decreases with increase in message size.

tion QoS. Nevertheless, the user-to-user latency of about 720 μs is quite small and limited only by the speed of the receiver's CPU. Recently, Keeton et al have measured a user-to-user latency of 700 μs between two SparcStation 20's with identical host adaptors and switches [17]. We are able to obtain identical latency with an endpoint that costs a fifth as much. Indeed, we hope to achieve correspondingly better performance with Pentium 133MHz processors and the faster PCI bus.

Note that 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. We also could not adequately measure the receive side performance because we could not source packets from the device driver to create a bottleneck at different layers of the receiving stack. However, we expect the latency on the receiving host to be more than that on the sending host because we have two context switches on the receive side (from interrupt-mode to kernel-mode, and from kernel-mode to user-mode) before the user application gets its message.

6.2 Throughput

6.2.1 Throughput for Reliable service

We measured the throughput of our system by sending 10,000 messages as fast as possible from one PC to another via the switch, and measuring either the transmission rate in user space (for reliable service), or reception rate in user space (for guaranteed and best-effort service). We repeated the measurement 50 times to obtain confidence bounds. We did not 'warm up' the system before collecting measurements.

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 6 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, reaching its maximum of around 47Mbps, and then falls with further increase in message size. We found that this is the result of two opposing forces that come into play with increasing message sizes. Per-message overheads (read and write system calls and at the transport layer) get 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. Moreover, for longer messages, it takes longer to locate 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.

Figure 6 shows a somewhat non-intuitive decrease in the standard deviation of the transfer rate with an increase in the message size. For small (4-20 Kbyte) messages, the time to transfer a message is dominated by the scheduling and system call overhead, and for large message sizes (30-64 Kbyte), by the data copy time. Since the data copy time is relatively constant for a given message size, as the message size increases, the standard deviation in the transfer rate decreases.

We found it very important to reduce 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. Moreover, on a loss, the receiver must wait at least one round-trip-time for receiving the lost packet, and must block buffers for reassembly while awaiting this retransmission. So an increase in the latency for reliable transfer can be expensive, especially at high bandwidths. We discovered that if we woke up the scheduler once every 50ms, instead of on every interrupt (in order to amortize the cost of waking up the kernel process) the throughput for reliable connections reduces to as low as 15Mbps. By waking up the scheduler on every interrupt, the throughput increases to 47Mbps.

Note that the peak achievable throughput with reliable service is quite impressive for a PC that would, in 1996, barely qualify as bottom-of-the-line. Keeton et al have measured the performance of a SparcStation 20 running Solaris with TCP/IP over ATM, and report a peak achievable throughput of 82 Mbps [17]. We are able to achieve nearly 60% of their performance at roughly 15% of their cost!

6.2.2 Throughput for Guaranteed and Unreliable Service

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 re-

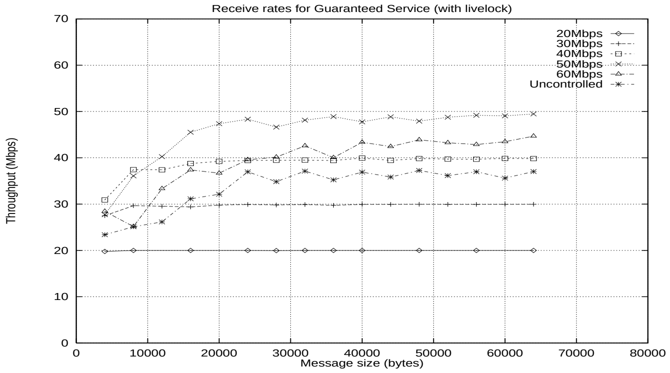

Figure 7: Throughput measured at the receiver as a function of message size for different transmission rates (ISR drops excess packets). Note that due to receive livelock, the throughput actually decreases if the sender sends faster than the receiver can process data. We do not show confidence intervals in this figure because it would make it very hard to read. The confidence intervals here are comparable to those in Figure 6.

ceiving rate is smaller than the sending rate because of losses in the receiver (Figure 7) and actually *decreases* with increase in sending rate above around 50Mbps. 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. On an interrupt, the driver places a newly filled buffer in a per-VC queue. We set a limit on the number of buffers a connection can have in its per-VC queue. This threshold, though currently fixed, can be dynamically varied depending on the number of active connections [6]. This provides a modicum of per-VC fairness, since no single connection can hog all the buffers in the receiver.

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 [27].

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 perfor-

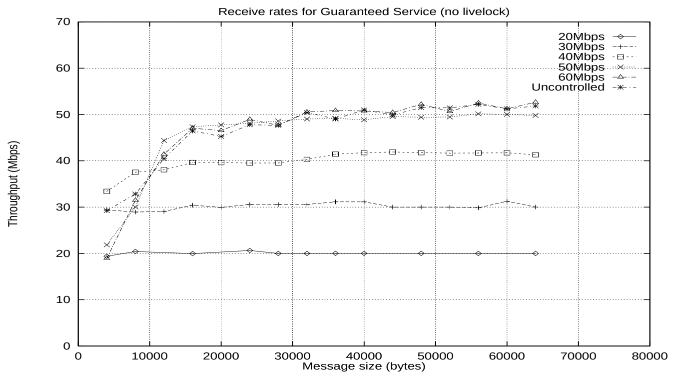

Figure 8: Throughput measured at the receiver for different transmission rates (host-adaptor drops excess packets). Note that there is no receive livelock, the throughput does not drop as the sender sends faster.

mance 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.

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, and stops interrupting the CPU. Once we go below a low watermark, we start supplying buffers to the card again. Once we made these changes, the throughput improved dramatically (Figure 8). 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

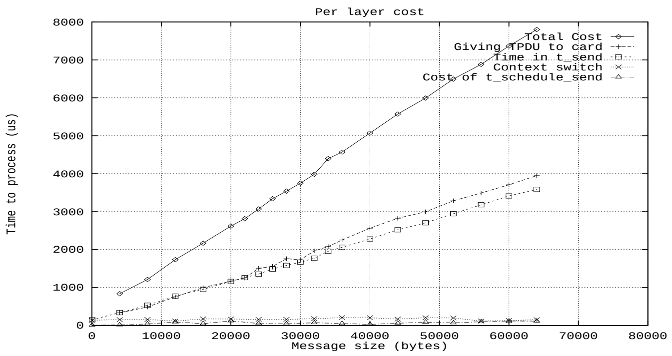

Figure 9: Processing cost for different parts of the stack as a function of message size. `t_send` is the interface routine, and `t_schedule_send` is the task that implements transport-layer processing. Note that the per-byte costs dominate for large messages.

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.

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 stops 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.

Note, however, that our scheme has a fairness problem. The current version of the software on the FORE card has only one queue of received buffers (instead of a per-VC queue). So, if the card drops frames, it cannot do that fairly - it just drops 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-adaptor if we want per-VC fairness.

6.3 CPU Cost as a Function of the Message Size

Conventional wisdom decrees that transport protocol implementers should try to minimize per-byte costs, since these dominate the total CPU cost of an implementation. Recent work by Kay and Pasquale claims that most application messages are small, and for small messages, the per-message cost dominates [16]. To verify these claims, we measured the per-byte, per-TPDU, and per-message costs on the transmit side, assuming unreliable service. The results of these measurements are presented in this section (Figure 9).

We expected the major costs on the transmit side to be:

1. The cost of a context switch from the user context to the kernel context.

2. The cost of copying data from user-space to kernel space and other operations in `t_send`.

3. The cost of processing in the transport layer (in `t_schedule_send`).

4. The time spent in the device driver, which issues the transmit command to the card. This time also includes the bus transfer time, and the cost of housekeeping operations like recovering memory from TPDUs which have been transmitted by the card.

We find that the context-switching cost is independent of the message size, and forms a major component of the total cost only for messages smaller than 1K (Figure 9). The cost of `t_send` and the time spent in the device driver increase linearly with message size. Thus, for large messages, these per-byte costs dominate. Surprisingly, the schedulable portion of the transport layer (the `t_schedule_send` task), has very little cost, and this cost is nearly independent of the message size. On reflection, we realized that this task only adds an appropriate header to each TPDU in message before handing it to the device driver. Thus, it contributes rather little to the overall CPU cost. However, we still would like to schedule this task, since it decides the order in which TPDUs are served by the device driver. We can eliminate the task scheduler on the transmit side if and only if the host-adaptor supported per-VCI queueing on the transmit side, and provided per-VCI differential qualities of service.

We conclude that for large messages, the cost of the `t_send` procedure and the time spent in the device driver dominate, while for small messages (¡ 1 Kbyte), other costs, such as context switching time, also become significant. Because of its high cost, we now focus on the cost of each component of the transport layer's `t_send` function, which are as follows:

1. Time spent in the write system call, and in enqueueing the incoming message in the PSB, not counting the cost of copying and fragmenting the message. This is a per-message cost.

2. Time spent in fragmenting the message into TPDUs and enqueueing the TPDUs in the transmission queue. This is a per-TPDU cost.

3. Time spent in copying data from user space to kernel space. This is a per-byte cost.

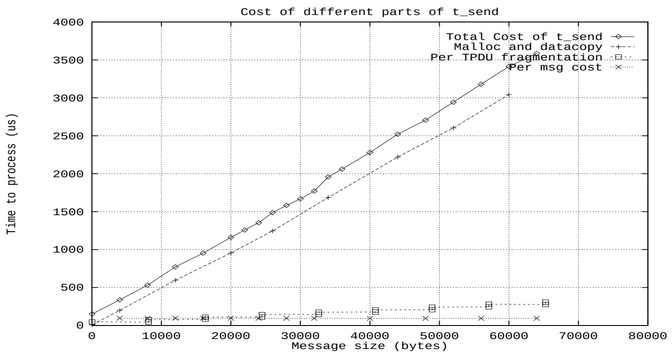

Figure 10: Processing cost for different components of the t_send interface procedure as a function of message size. Note that the per-byte overheads dominate for large messages.

Figure 10 plots the cost of each component as a function of the message size. We see that the per-message cost, which is the time spent in making the write system call and enqueueing the message, is constant. This is around 95 μs, of which 41 μs is spent in making the write system call and acquiring the Readers Writers lock (Section 5.5.3), and the remaining time is spent in other per-message overhead.

The time spent in fragmenting the message into TPDUs and enqueueing them (the per-TPDU cost) 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 1000 bytes), the major component of the cost of `t_send` is the per-message and per-TPDU cost [16]. For larger messages, the cost of copying data dominates and the per-message and per-TPDU costs together make up only a small part of the total time spent in `t_send`. Thus, our measurements are consistent with those in Reference [16]. (We note in passing that the raw costs of operations such as data copy on a 66 MhZ 486 PC are consistently around 2/3 of the costs on a DECstation 5000/200.)

7 Discussion

From the viewpoint of protocol performance, the main aims of our transport layer are to provide high performance and differential qualities of service. Performance measurements allow us to quantify the degree to which we achieved both goals.

In terms of high performance, because of careful measurements and performance tuning, as well as the absence of a data checksum in software, we are able to achieve the same latency and nearly 60% of the throughput of a SparcStation 20 at a small fraction of its cost [17]. The throughput bottleneck is the CPU on the receiving side. We hope to achieve much higher throughput by upgrading from a 66 MHz 80486 machine with an EISA bus to a 133 MHz Pentium processor with a PCI bus. Similarly, much of the latency for small messages is in waiting for the scheduler to become active. This latency is likely to be smaller on a higher-speed CPU, where the CPU load of other processes would be proportionately smaller. (The switch is not a performance bottleneck in our environment, and is unlikely to be one in most local-area ATM networks.)

In terms of differential service quality, we are able to successfully regulate an application to its stated leaky-bucket values. When combined with a network that provides endpoint-to-endpoint guarantees of service quality to 'well-behaved' sources, we can, therefore, guarantee application-to-application quality of service. Moreover, the task scheduler allows us to give higher priority to guaranteed-service connections. (As we remarked earlier, the major open problems with our implementation are the lack of coordination with the CPU and disk scheduler and a lack of control over the host-adaptor card. We hope to address this lacunae in future work.)

We now briefly retrace the steps we took to tune our implementation. First, the task scheduler is awakened both by the device driver and by `t_send` on packet receipt. This reduced the round-trip latency for small packets from 120 ms to 1.44 ms. This decrease in latency also considerably improved the performance of the reliable service, which is sensitive to the round-trip delay. Second, we detected and corrected the livelock condition in the interrupt service routine. This improved the performance of guaranteed service and best-effort service connections. Third, we introduced hysteresis in supplying buffers to the card by using high- and low-watermarks, further improving their performance. The point is that we were able to *systematically* improve protocol performance by measuring and tuning our implementation.

One surprising conclusion we came to was that for large messages, the only costs that matter are the copy cost and the time spent in the device driver. For these messages, it does not matter how much time we spend in the rest of the transport layer, as long as we keep these two major costs under control. The lesson here is that for large message sizes we should make transport layer processing as complex as necessary to optimally use the network. However, for small messages, every instruction counts. We cannot gain improvement in performance by tuning any single aspect of our implementation.

8 Related Work

It is useful to contrast the semantics of our transport layer with TCP. We use a different retransmission scheme from TCP, and propose a rate-based flow control scheme that does not shut down on packet losses. Further, unlike TCP, we do not do data checksumming, relying, instead, on the AAL5 checksum.

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 [15]. 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. Reference [10] provides a an excellent survey of issues in high-speed transport protocol design as well references to several other transport protocols.

9 Status and Future Work

The transport layer we have described in this paper is complete, and has been operational on our testbed for the last year. We have released source code to universities at no cost. A port to the FreeBSD operating system is also operational.

An 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. We refer the reader to References [28, 22, 23, 24] as representative of work in this area.

10 Conclusion

We have described the design, implementation, and performance tuning of a native-mode ATM protocol stack. The transport layer provides three classes of service: reliable, guaranteed-service, and unreliable data transfer. An unusual feature is leaky-bucket policing at the transport layer for open-loop flow control. Our design is novel in that it is targeted to AAL5 and inexpensive Personal Computers. We have also tried to provide Quality of Service guarantees for reliable and guaranteed-performance connections by eliminating multiplexing, QoS-aware task-scheduling, and providing error control, flow control, and leaky-bucket shaping. 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 of more than 50 Mbps Mbps user-to-user for unreliable data transfer. End-to-end delays in a local-area network 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.

Acknowledgments

The first version of the transport layer was written by R.P. Rustagi and R.N. Moorthy of the Indian Institute of Technology, Delhi, as part of the IDLInet project. Sandeep Gupta of the University of Maryland, College Park, added mbuf code and removed extra data copies. Puneet Sharma from the University of Southern California reverse-engineered the microcode download program and wrote the first version of the ATM device driver. Our thanks to them all.

We would also like to thank two anonymous referees for their perceptive comments, which have considerably improved the presentation in the paper.

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] G.J. Armitage, "Multicast and Multiprotocol Support for ATM Based Internets," *ACM SIGCOMM Computer Communication Review*, Vol. 15, No. 2, April 1995.

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

[6] A.K. Choudhury and E.L. Hahne, "Dynamic Queue Length Thresholds in a Shared Memory ATM Switch," *Proc IEEE INFOCOM'96*, March 1996.

[7] A. DeSimone and S. Nanda, "Wireless Data: Systems, Standards, and Services," *Journal of Wireless Networks*, Vol 1, February 1996.

[8] A. Edwards and S. Muir, "Experiences Implementing a High-Performance TCP in User-Space," *Proc. ACM SIGCOMM '95*, Cambridge, September 1995, pp. 196-205.

[9] D.C. Feldmeier, "Multiplexing Issues in Communication System Design," *Proc. ACM SIGCOMM'90*, October 1990, pp. 209–219.

[10] D.C. Feldmeier, "A Framework of Architectural Concepts for High Speed Communications Systems, " *IEEE Journal of Selected Areas in Communications*, Vol.11 No.4, May 1993

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

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

[13] V. Jacobson, "Congestion Avoidance and Control," *Proc. ACM SIGCOMM'88*, pp 314-329, 1988.

[14] A. Jain and S. Keshav, "Native-mode ATM in FreeBSD: Experience and Performance," *Proc. NOSSDAV '96*, April 1996.

[15] H. Kanakia, P.P. Mishra and A. Reibman, "An Adaptive Congestion Control Scheme for Real-Time Packet Video Transport, " *Proc. ACM SIGCOMM'93*, August 1993.

[16] J. Kay and J. Pasquale, "The Importance of Non-Data Touching Processing Overheads in TCP/IP," *Proc. ACM SIGCOMM'93*, August 1993.

[17] K. Keeton, T.E. Anderson, and D.A. Patterson, "LogP Quantified: The Case for Low-Overhead Local Area Networks," *Proc. Hot Interconnects III: A Symposium on High Performance Interconnects*, Stanford University, Stanford, CA, August 10-12 1995.

[18] S. Keshav, "REAL: A Network Simulator," *CSD TR 88/472*, University of California Technical Report, December 1988.

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

[20] S. Keshav and S.P. Morgan, "SMART Retransmission: Performance with Random Losses and Overload," *Preprint*, January 1996.

[21] 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.

[22] C.W. Mercer, "Operating System Resource Reservation for Real-Time and Multimedia Applications," *PhD. Dissertation, Carnegie Mellon University*, May 1996.

[23] Klara Nahrstedt, Jonathan M. Smith, "The QoS Broker," *IEEE Multimedia,* Spring 1995, Vol.2, No.1, pp. 53-67.

[24] Klara Nahrstedt and Ralf Steinmetz, "Resource Management in Multimedia Networked Systems," *IEEE Computer*, May 1995, pp. 52-64.

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

[26] 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.

[27] K.K. Ramakrishnan, "Performance Considerations in Designing Network Interfaces, " IEEE Journal on Special Areas in Communications: Special Issue on High Speed Computer/Network Interfaces, February 1993.

[28] K.K. Ramakrishnan, L. Vaitzblit, C. Gray, U. Vahalia, D. Ting, P. Tzelnic, S. Glaserand W. Duso "Operating System Support for a Video-On-Demand File Service," *Proc. 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, November 1993.

[29] R. Sharma and S. Keshav, "Signalling and Operating System Support for Native-Mode ATM Applications," *Proc ACM SIGCOMM'94*, September 1994.

[30] W.R. Stevens, "TCP/IP Illustrated: Volume 1," *Addison-Wesley*, 1994.

[31] D. Tennenhouse, "Layered Multiplexing Considered Harmful," *IFIP Proc. Protocols for High Speed Networks*, Elsevier Science Publishers, May 1989.

[32] C.A. Thekkath, T.D. Nguyen, E. Moy, and E.D. Lazowska, "Implementing Network Protocols at User Level," *Proc. ACM SIGCOMM'93*, San Francisco, September 1993.

[33] L. Zhang and S. Shenker and D.D. Clark, "Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic," *Proc. ACM SIGCOMM'91*, September 1991.