Native-mode ATM in FreeBSD: Experience and Performance

A. Jain and S. Keshav
AT&T Bell Laboratories, 600 Mountain Ave. Murray Hill NJ 07974 USA
`ankur, keshav@research.att.com`

Abstract

We describe our experience in porting a native-mode ATM protocol stack from Plan 9 to the FreeBSD operating system. We discuss problems with implementing: (a) in-kernel locks, (b) a lightweight task scheduler, (c) burst and cell-level smoothing, and (d) user-kernel communication, and present a number of techniques to work around deficiencies in the FreeBSD kernel. Our performance results are comparable to that achieved with high-end workstations, but at a fraction of the cost: a user-to-user latency of 650 microseconds for small messages, and an end-to-end throughput of over 54 Mbps with reliable service.

1 Introduction

We believe that in order to exploit the end-to-end QoS capabilities of future ATM networks, end-systems must not-only provide QoS-sensitive operating systems, but also make per-connection QoS available to applications. Unlike IP-over-ATM, where the IP layer hides and fritters away ATM-level QoS, we have designed and built a Native-mode ATM protocol stack that makes ATM-level QoS visible to applications [7, 1]. The stack includes not only a transport layer that adds reliability and flow control to AAL5, but also signaling and operating system support for connection management, handling abnormal termination, and end-system resource management. We refer the reader to reference [1] for further details on our stack, and to reference [2] for a survey of related work in the area of next-generation transport-layer protocols.

This paper describes our experiences in porting the native-mode ATM stack to the FreeBSD operating system and its performance on that platform. The first implementation of the stack was on a network simulator, which allowed us to debug the protocol stack in a controlled environment. From there we ported it to MS-DOS, and subsequently a Plan 9-variant called Brazil [6]. Each of these platforms has its problems: DOS has minimal support for application development, and Brazil is neither generally available nor widely used outside Bell Labs. We chose to use FreeBSD as our platform because it is cheap, and runs on relatively cheap platforms (x86 PCs), yet has the full functionality of a multitasking operating system. Moreover, the source code for the system is freely available. Though it is not a real-time OS, its other attributes still make in an excellent choice for networking research. Thus, we believe that our experiences in modifying the FreeBSD kernel to add in a native-mode ATM stack would be of interest to the networking community.

The paper is laid out as follows: in Section 2 we present an overview of the stack. Section 3 describes the major sections of the stack that are OS-dependent and required porting effort. In Section 4, we outline the problems we ran into in carrying out the port. Finally, Section 5 gives a detailed performance evaluation.

2 Overview

An end-system implementing our stack is shown in Figure 1. The ATM stack consists of three main entities: the application entity and the signaling entity in user space, and the transport entity inside the kernel.

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 [7, 5], 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.

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. 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 transport entity is responsible for transfer-

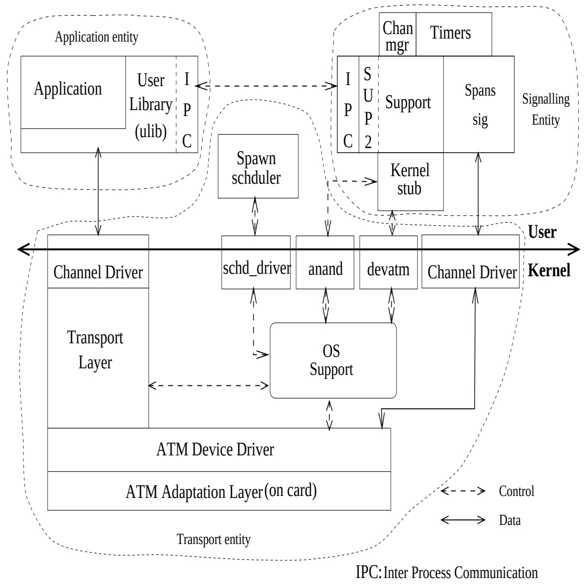

Figure 1: Overview of the ATM stack. The stack consists of three entities: the application entity, the signaling entity, and the transport entity.

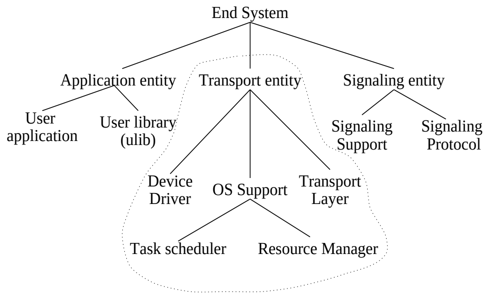

Figure 2: Components of the ATM stack.

ring the data through the kernel down to the host-adaptor. (We will assume that the host-adaptor provides AAL5-frame transport, as is the case with all modern host-adaptors.) It also performs call admission and allocates resource to VCIs for guaranteeing QoS (bandwidth and delay). Since the transport entity must provide high performance and arbitrate between multiple user requests, 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` [5]. 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 modern operating systems.

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.

The transport layer provides *simplex* virtual circuits, error control, and flow control. In addition, it segments application layer buffers into TPDUs and reassembles them on the receive side. The layer is described in more detail in [1]. 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 its work.

In the Brazil kernel, the task scheduler is a kernel thread that periodically (in our case, every 50 ms) handles any expired timers, and then calls any scheduled 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 weighted-round-robin scheduler, that can assign different weights to different VCIs. Hence we can allocate a different QoS to different connections.

The *resource manager* is responsible for admission control at the time of call setup. 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 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.

3 Porting requirements

Our first task was to identify the parts of the code that required major changes. These correspond to features in the Brazil operating system absent in FreeBSD. The modules that need to be changed are briefly listed below, and described in more detail in subsequent sections.

1. IPC module for communication between `ulib` and `sigd`
   In Brazil all inter-process communication takes place with named files. In FreeBSD, we replaced this with Unix-domain sockets.

2. Channel driver
   The channel driver is used by the application to send and receive data to and from the kernel-resident transport layer. In Brazil the user space to kernel space communication is again through named files. We decided to replace these with pseudo-devices (for reasons explained below).

3. Task scheduler
   In Brazil the task scheduler runs as a separate kernel thread. FreeBSD has no such counterpart. Thus, we had to come up with a way to emulate kernel-threads.

4. Locking
   There are many data structures which are accessed and modified both by the application and the transport layer scheduler. This calls for locking, which is built into Brazil but not in FreeBSD. We, therefore, implemented locking in the FreeBSD kernel.

5. Abnormal termination of applications
   Brazil has an error-handling exception stack which allows the kernel to automatically undo commands (such as held locks) in case an exception occurs. In the absence of the exception stack, we had to explicitly handle unlocking due to exceptions in the FreeBSD kernel.

6. Page size issues
   To send an AAL 5 frame, an ATM device driver loads the device with the physical address of a memory buffer containing the frame. Brazil guarantees that all memory allocations occupy adjacent pages in physical memory. Thus, the device driver needs to compute only one physical address per memory buffer. In FreeBSD, memory buffers larger than one page may occupy two separate physical pages. Thus, we had to explicitly fragment large memory buffers into page-sized chunks, and compute a physical address for the start of each chunk.

4 Porting experience

4.1 IPC module

The IPC module hides the OS-specific details of inter-process communication from the user-library (`ulib`) and signaling entities. In Brazil it is based on named pipes and in FreeBSD we used Unix domain sockets. Unfortunately, on closing a socket, the Unix file names are not automatically deleted. We got around this problem by explicitly keeping track of allocated file name, and deleting them on a close.

4.2 Channel driver and channel manager

We had a choice of mechanisms for user-kernel data transfer. One is the standard socket mechanism [5]. A socket allows an application to obtain a file descriptor to which a connection can be bound, and data can be transferred with standard `read` and `write` system calls. The main problem with sockets is that they are shared by all protocol stacks, and cannot be cleanly specialized for the native-ATM stack. In particular, when data is copied out to user space on a read, the socket layer simply frees the associated `mbuf`. We would like, instead, to return the `mbuf` to the device driver's buffer pool for reuse by the host-adaptor card. Thus, we decided to use a special-purpose pseudo-device instead of sockets.

The system is initially populated with a number of pseudo-devices, which are managed by a channel manager. When an application wants a new communication endpoint, the `ulib` routine contacts the channel manager (using IPC) and obtains the file descriptor for a free pseudo-device. User reads and writes to the pseudo-device are directed to the transport layer as usual. Since the user-network interactions are hidden inside `ulib`, this change to the communication mechanism results in no change to Brazil applications.

The channel manager performs bookkeeping on open and available channels. It is informed by the kernel (via another pseudo-device) when an application terminates, and uses this information to keep track of which channels are available. On receiving a request from `ulib` through IPC it allocates a free channel, generates a cookie (capability) and passes the (channel_number, cookie) pair to the user application as well as the kernel (transport layer) so that it can initialize the connection. The cookie is used by the `ulib` during `bind` to authenticate itself. Since no application can read or write from the channel without a bind operation, this effectively secures the channel from unauthorized access.

4.3 Transport layer scheduler

The communication task-scheduler needs to run as a separate kernel thread and should be called periodically. We also need to put the scheduler to sleep when it cannot acquire a lock to a shared data structure. Thus our requirement was a scheduler which would run in kernel space, but could be put to sleep like a user-process when it could not acquire a lock. We solved this problem by implementing the scheduler as a user-space process which writes once to a special pseudo-device. The write initiates the scheduler in the kernel, which is simply an infinite loop that schedules tasks, then puts itself to sleep (using the `tsleep` call) for 50 ms.

The problem with the first implementation of this scheme was that the scheduler process could not be terminated on system shutdown. This happened because of the way signals work in FreeBSD (or similar Unix-like kernel). The action on a signal is taken only when a process enters the kernel due to a trap, a system call, etc. However the scheduler-process enters the kernel only once and then spends its entire life in the kernel (hibernating for long periods so that other processes get their fair share of CPU). As a result the action on a kill signal would never be taken. This resulted in the system hanging when we tried to call a shutdown, because shutdown would endlessly wait for the scheduler to die. The hard part was detecting the reason for such a behaviour but the solution was simple i.e, on every wakeup the scheduler checks for pending signals and in case there are any it takes the actions on those signals before proceeding with its normal tasks.

4.4 Locking

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 a `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.

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. In Brazil, we used a `RWlock` data-structure for proper locking of these reads and writes.

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 the `RWlock`, 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.

Locks are not available in the FreeBSD kernel, so we ported the Brazil locking code to FreeBSD. This involved porting both spin locks (which are trivial, once we had the test-and-set primitive), and `Queue` locks, which are similar to monitors. A process sleeps on the `Queue` lock, if the lock is not available, and is awakened by the holder on an unlock. The sleep is implemented using FreeBSD's `tsleep` primitive. Unfortunately, the kernel automatically wakes up a a process waiting on a `tsleep` if the process receives a signal. If this case is not carefully handled, two processes may simultaneously hold a lock (because the newly awakened process thinks that it was awakened by the lock-holder). We solved this problem by replacing `tsleep` with a new call, `Tsleep`. `Tsleep` looks at `tsleep`'s return value. `tsleep` returns 0, if awakened up , `EWOULDBLOCK`, if timed out, `EINTR` if awakened by a signal and needs to be interrupted, and `ERESTART` if awakened by a signal and needs to be restarted.If `tsleep` returns 0 or `EWOULDBLOCK` `Tsleep` returns the same. If `tsleep` returns `EINTR` then the action on the signal is taken by explicitly calling `postsig`. If the process survives `EINTR` or return value is `ERESTART` then `Tsleep` restarts `tsleep` with a smaller timeout.

4.5 Abnormal termination

If a process terminates abnormally, the kernel must release all the locks it holds, otherwise the system will hang. Each lock descriptor has a 'holder' field that contains the PID of the lock's holder. On abnormal termination, the function `remove_lock`, called during a kernel-initiated close of a communication pseudo-device, checks all locks to see if they are held by the process being killed, and unlocks all held locks. If the process being killed is holding a `Queue` lock, the procedure also wakes one of the processes waiting for it.

4.6 Timeout computation

The reliable service component of the transport layer uses a strategy similar to the one in TCP-Reno for dynamically determining an optimal window size. It maintains a threshold called the slow-start threshold initialized to half the largest allowed window size. The flow-control window size starts at 1 and doubles every round trip time (RTT) if the window size is smaller than the slow-start threshold. The window increases by one every RTT once it increases past the slow-start threshold. In case of a loss, the window size and the slow-start threshold are halved. The window size is also halved on the occurrence of a timeout.

Timeout periods are calculated as a multiple of the smoothed measured RTT value. In FreeBSD the resolution of the clock is 10 ms. So, if the true RTT is smaller than this value, the transport layer computes the smoothed value of RTT, and thus the timeout, as zero. This leads to timeouts being called at every scheduler period, so that the window size never increases beyond 1 or 2 TPDUs and throughput plummets.

We remedied this by making the smallest possible timeout twice the scheduler period (of 50 ms). Since packet are almost always acked within this time, with this change, window sizes grow to their optimal values. While this coarsening of the timeout value might usually affect performance, our transport layer uses the SMART retransmission strategy, which is essentially independent of timeouts [4]. Thus, choosing a coarser timeout value does not affect protocol performance.

5 Page size issues

The Brazil OS guarantees physical contiguity of memory buffers. Thus, when a device driver wants to send an AAL5 frame contained in a buffer, it merely hands the ATM interface card the physical address of the start of the buffer and the card does the rest. FreeBSD does not guarantee physical contiguity for buffers larger than one FreeBSD page. Thus, we had to modify our code to detect non-contiguous buffers and segment them into physically contiguous areas, each of which is passed to the card in a separate descriptor.

6 Send-side buffer pools

Incoming data from the ATM interface must be written into host-memory. While the device driver could do a `malloc` on every receive to obtain memory, this is inefficient. To achieve high throughput, the Fore device driver maintains its own a pool of free buffers. We discovered that a substantial part of the transmit-side latency was in acquiring a buffer using `malloc` on the send side. Moreover, the allocated buffer is not guaranteed to be page-aligned, which means that even buffers smaller than one page may need to be split on page boundaries to guarantee physical contiguity. For these reasons, we decided to introduce our own page pools for send-side buffers. These are allocated at boot time and are managed entirely by our stack. Memory allocation requests from the page pool return page-aligned buffers, so that they need not be further fragmented in the device driver.

Access to the memory pool becomes a point of contention when a host has tens or hundreds of active connections. We need to put an application to sleep when no more pool buffers are available and wake it up when buffers are released. This requires a counting semaphore. Since FreeBSD does not provide one, we implemented a standard PV counting semaphore in the kernel. We had to take great care in using these semaphores to avoid deadlocks when large numbers of applications are simultaneously active, and one or more could be holding other kernel locks.

7 Packet-level and cell-level smoothing

From our work with Brazil, we knew that the bottleneck in the system is the receive side CPU. If the sender sends data in a burst, the receiver cannot keep up with it, and cells get dropped. This leads to AAL5 frame loss. It makes sense for the sender to smooth out its transmission to the extent possible, so that the receiver does not have to deal with packet bursts. We use both packet and cell-level smoothing to achieve this goal. For guaranteed service, the transport layer schedules a task that periodically drains packets from the user's application-level buffer. This provides packet-level smoothing. These packets are then smoothed at the cell level using a facility provided by Fore's ATM adaptor card. Essentially, for each AAL5 frame, we can ask the card to introduce pauses of length X cells for every 16-X cells it transmits at the line speed (of 155 Mbps). We adaptively choose X to maximize smoothing gains without affecting the transmit performance.

Our algorithm involves computing, at each scheduling activation, the number of bytes being transmitted during the current activation. The idea is that the more the data transmitted in the curent activation, the larger the bursts emitted by the card. Thus, we should do more aggressive smoothing (choose a larger X). To do so, we define a number of thresholds and corresponding values of X. Whenever the number of packets transmitted exceeds a particular threshold, we increase the level of smoothing (upto a value of X=6). As a result, even at heavy loads, the receive throughput is stable, and the loss rate is kept low. Our method works only because the transmitter cannot fully load the link's 155 Mbps capacity. If we had a more powerful machine, we would use a more sophisticated cell-level smoothing algorithm.

8 Performance

We measured the performance of our implementation on a small testbed of two 66Mhz Intel 80486-based PCs and a FORE ASX-200 switch. We stress that the PCs represent two-year old technology, and at the current time (April 1996) are considered to be nearly the bottom-of-the-line. One of the PCs acted as a sender, and the other as a receiver. The PCs were unloaded except for our measurement software. Our measurements are in two parts. In the first part, we measured the end-to-end latency in exchanging small messages. The sender sent 5,000 32-byte messages to the receiver, which read the message, then returned the message back to the sender. We measured how long it took to complete 5,000 such round-trip exchanges, thus measuring the user-to-user latency for small messages. We measured a mean round-trip time of around 1.3 milliseconds, corresponding to a one-way latency of around 650 microseconds. This compares favorably with the Brazil implementation, for which the corresponding number is 720 microseconds [1].

We also measured the throughput achievable using the reliable, best-effort, and guaranteed services as a function of the message size. With reliable service, a peak speed of 54 Mbps is achievable with 80 Kbyte messages (Figure 3). Speeds of nearly 50 Mbps are achievable with a message size of 8 Kbytes. We did a simple two-part least-squares fit (shown in the Figure) to express the throughput as:

throughput (Mbps)

$= 0.006239 * pkt\_size(bytes) + 4.25;$

if(pkt_size $<$ 8Kbytes)

$= 4.50399e - 05 * pkt\_size + 49.51;$

if(pkt_size $>$ 8Kbytes)

The half-power point (where the throughput is half the peak, is around 2200 bytes.

With best-effort service, receive speeds of a peak speed of 56 Mbps is achievable with 40 Kbyte messages (Figure 4). With guaranteed service, (Figure 5) the receive rate matches the requested rate for message sizes larger than about 4-8 K (depending on the required rate). This shows that (a) our leaky-bucket regulator works as advertised, and (b) applications requiring high throughputs should use larger message sizes.

All our measurements are shown with the the corresponding measurements with Brazil. In all cases, the FreeBSD port outperforms the Brazil implementation on identical hardware. There are two reasons for this. First, we have had an opportunity to further tune our implementation, which gives some improvement. Second, Brazil supports a client-server distributed computing model, where the client PC must frequently talk to the compute server over the Ethernet. Since we could not turn off the Ethernet when doing our measurements, part of the performance hit with Brazil is the interrupt and CPU time fielding background Brazil traffic. It is interesting that the so-call 'macrokernel' approach of FreeBSD outperforms the Brazil 'microkernel' approach. We certainly did not expect that!

Our performance numbers compare very favorably with speeds reported for much more expensive and higher-performance hardware [3]. For example, with dual-processor SparcStation 20 running Solaris 2.4 and Fore's SBA-200 host adaptor cards (essentially identical to our cards), the report peak throughput for reliable (TCP) service is 81.2 Mbps. The reported round-trip delay for 8-byte messages on the same platform is 1368 microseconds. We are able to achieve 70 percent of the reported throughput and lower round-trip delays on a platform that costs about a tenth as much! We know from previous work that our bottleneck is the receiver's CPU. With Pentium-class CPU's on the send and receive side, we expect nearly 100Mbps peak throughput, for a cost of only a fifth of a dual-processor SparcStation.

9 Conclusion

This paper presents our experience in porting a native-mode ATM stack to the FreeBSD operating system. Our contribution in this paper is in pointing out several subtle problems in porting protocol stacks between mutually-incompatible operating systems. This may be of use to others in the field who are undertaking similar exercises. We have also presented performance measurements that show that we can achieve top-of-the-line workstation-class performance with bottom-of-the-line PC hardware. We can achieve a throughput of 54 Mbps with reliable service, and 56 Mbs with best-effort or guaranteed service.

The source code for the protocol stack is avail-

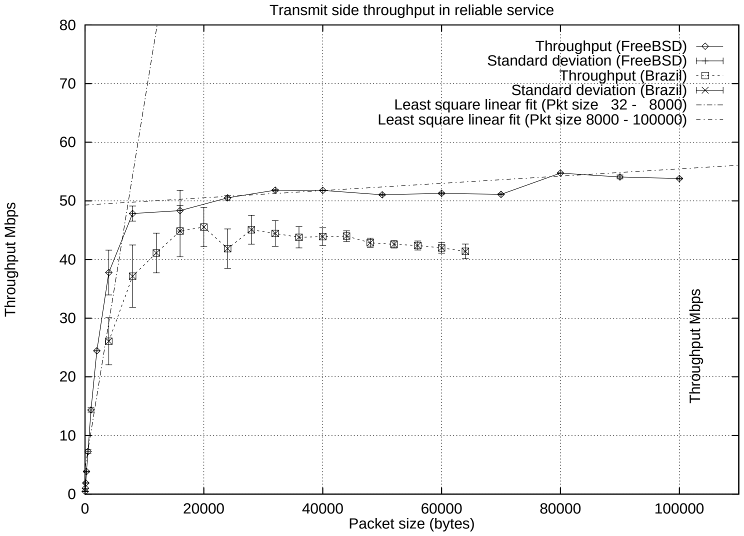

Figure 3: Throughput of reliable service as a function of message size. The throughput increases with increase in message size because of a decrease in per-packet overhead.

able free-of-charge to universities and non-commercial institutions. For details, please send email to keshav@research.att.com.

10 Acknowledgments

Ritesh Ahuja did the Brazil implementation of the native-mode stack and started work on the FreeBSD port. Without his dedication and hard work, we would have had nothing to work with.

References

[1] R. Ahuja, S. Keshav, and H. Saran, "Design, Implementation, and Performance of a Native-Mode ATM Transport Layer," *Proc. INFOCOM '96*, March 1996.

[2] 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

[3] 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.

[4] S. Keshav and S.P. Morgan, "SMART: Retransmission: Performance with Random Losses and Overload," Submitted to ACM Sigcomm '96, January 1996.

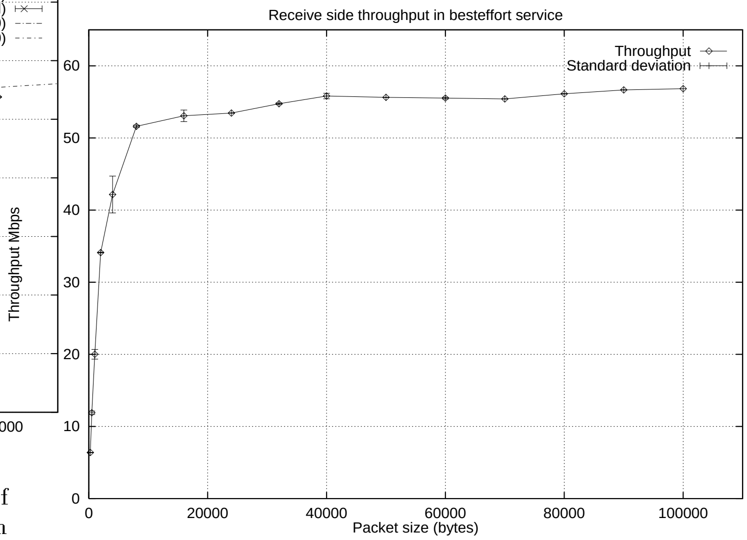

Figure 4: Throughput of best-effort service as a function of message size. The sender can send faster than the receiver can receive data, since the bottleneck is the receiver's CPU.

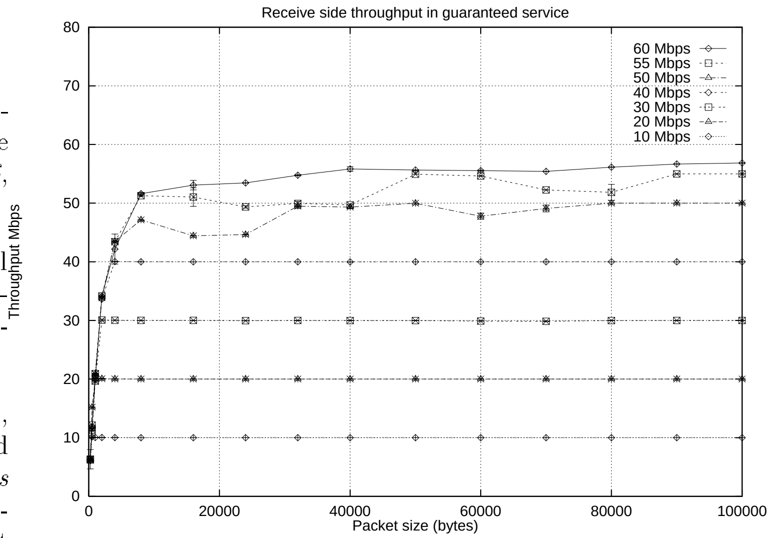

Figure 5: Throughput of guaranteed service as a function of message size. As the message size increases, the sender is better able to achieve the desired rate.

[5] 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.

[6] 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.

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