Simple Protocol Processing for High-Bandwidth Low-Latency Networking

José C. Brustoloni Brian N. Bershad

March 1992
CMU-CS-93-132

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was sponsored by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. government.

Keywords: ATM, high-speed networks, ILP, protocol bypass, protocol processing, buffer management, flow control, Mach, TCP/IP

Abstract

Protocol and operating system overheads have become the limiting factor for communications performance on fast networks such as ATM. A large component of these overheads stems from the *protocol redundancy* that arises when layering higher level protocols on top of lower level ones. ATM, for example, requires specific mechanisms for connection management, flow control, congestion avoidance, and segmentation and reassembly. With these mechanisms in place, it is relatively simple and inexpensive to provide for reliable sequenced delivery at the network interface level, making similar functionality in higher-level protocols such as TCP/IP redundant. In this paper we present a protocol architecture specifically tailored for communication over high-bandwidth low-latency local or metropolitan ATM networks. Our architecture yields high performance by eliminating protocol redundancy and by exploiting common-case communication behavior. With this approach, we can combine the functionality typically found in four separate layers of the ISO model — data link through session — in a single pass over the data, delivering high throughput and low latency. Our protocol architecture requires minimal hardware support from the network interface and switch fabric, yet efficiently provides services such as segmentation and reassembly, flow control, congestion avoidance, and error recovery. We have implemented our protocol architecture on a switch-based ATM network consisting of DECstation 5000/200 workstations running the Mach 3.0 operating system. Our implementation achieves latencies and bandwidths close to the physical limitations imposed by the hardware, yet offers applications a high-level reliable transport interface.

1. Introduction

Networks such as ATM [22] and HiPPI [13] are becoming increasingly important due to their high bandwidths and low latency. High bandwidth is important for applications such as those found in supercomputing, visualization and multimedia. Low latency is key to efficient remote procedure call support and fine-grain distributed computation. Bandwidth and latency gains in the physical medium, however, do not necessarily result in correspondingly better communication performance for end applications [27]. The work described in this paper is targeted at providing to applications network latencies and bandwidths close to inherent hardware limits for an ATM network. To do this, we bypass higher-level protocols because the functionality found in those protocols, such as TCP/IP [24], is made redundant by services required in the lower-level network interface. ATM, for example, requires specific support for connection management, flow control, congestion avoidance, and segmentation and reassembly, making similar TCP/IP functionality redundant.

Our protocols and interfaces are designed from scratch to avoid biases detrimental to ATM that favor obsolete or extraneous architectures or protocols. This approach is not new: most of the work on fast RPC, for example, is layered on top of assumptions about the architecture or network used [15, 32]. Our design requires only minimal architectural support from the host interface and switches, favoring solutions that work on hardware inexpensive enough to come pre-configured on every workstation, much as Ethernet [21] currently does. Specifically, in our implementation:

- *we optimize our protocols for the common case* [18] of low-latency, high-bandwidth communication over an ATM network scaling up to metropolitan distances — roughly 40 km. Assuming that the communicating hosts are connected directly through ATM rather than an internetwork reduces the complexity of the network protocol software. Metropolitan ranges impose only moderate propagation delays, making it possible to use simple protocols for flow control, congestion avoidance and error recovery. Internetworking and routing are left to higher-level software that is invoked on an exception basis.
- *we exploit integrated layer processing* [7] because each separate protocol layer can take time comparable to that required to transport the data over a high-speed network. For the host to keep up with the network, layers must be combined or eliminated. Our network interface is organized in a single layer, and requires that each data byte be accessed only once on the path between application and the physical medium.
- *we integrate buffer management* for applications and protocol processing. Using the same buffers for both, each data byte needs to be moved only once between the network and the application's address space, minimizing data copying.

Although our discussion focuses on ATM networks, many of the techniques we present are also applicable to other high-speed networks, including those found in tightly coupled multicomputers. For example, our buffer management and segmentation and reassembly approaches could be applied to the fat-trees used in the CM-5, and our flow control and congestion avoidance algorithms could be used in the Paragon mesh, where receiver overrun is becoming an increasingly important problem [5].

The rest of this paper

The rest of this paper is organized as follows. In Section 2 we provide some background on ATM networking and protocols, and describe the hardware we used in our work. In Section 3 we discuss the difficulties associated with the traditional layering of higher-level protocols on top of ATM. In Section 4, we describe the overall organization and implementation of our protocol architecture, in particular our approach to buffer management, segmentation and reassembly, flow control and error recovery, signaling, multiplexing, and the integration with Internet protocols. In Section 5 we discuss performance. In Section 6 we survey related work. Finally, in Section 7 we present our conclusions.

2. Background

ATM is the international standard chosen for BISDN, the proposed global fiber-optic network that integrates and replaces the current separate networks for data, voice and video communication [22]. ATM packets (*cells*) have a fixed length of 53 bytes. Of these, 5 bytes are the ATM *header* with the virtual path identifier and virtual connection identifier (VPI/VCI) used to route the cell to the destination. The other 48 bytes of the cell are the *payload*, which carries user data.

Messages are broken up into cells according to one of several ATM adaptation layer options (AAL). Our hardware supports AAL 3/4. In this option, the ATM payload (*segment*) is divided into a 2 byte segmentation and reassembly (SAR) header, 44 bytes of data, and a 2 byte SAR trailer. The SAR header contains two bits to indicate segment type (beginning, continuation, end of message, or single segment message), a modulo-16 segment sequence number, and a 10 bit message identifier (MID). The SAR trailer contains the segment checksum and length of the data in the segment. A convergence sublayer (CS) runs on top of SAR and inserts and removes a four byte header and a four byte trailer in messages. The CS header contains the message type, the lower 8 bits of the MID, and the size of the buffer required for the message. The CS trailer carries a byte for protocol use, the lower 8 bits of the MID, and the total message length.

To transmit a message, the CS header and trailer are inserted, and the resulting message is broken up into segments each 44 bytes long, except possibly for the last one, which may need to be padded with zeros to complete a segment. Segments are sent in individual cells to the destination, where they are reassembled into the original message.

Every host in an ATM network is connected to one or more switches that route packets between hosts and other switches. In contrast to networks such as Ethernet, a connection must be established between any pair of hosts through intermediate switches before they can communicate. A connection is established or torn down using specific signaling protocols running on top of CS, and is identified by the VPI/VCI fields of the ATM header. Cells from a given connection are routed over a path determined at connection establishment time, and cannot be misordered or duplicated by the network. A connectionless service has also been proposed for ATM, but its specifications and exact characteristics are still in progress. As with other fiber-optic networks, bit error rates are extremely low (on the order of 10^{-14}). Cells may, however, be lost due to corruption, congestion, and buffer depletion at intermediate switches and endpoint nodes. Neither the ATM hardware nor the adaptation layers provide for automatic acknowledgements and retransmission.

2.1. Experimental hardware

Our protocol architecture is designed to function well using simple network interfaces and switches. Such configurations are likely to become ubiquitous because of the low-cost promised by their simplicity. The system described in this paper uses a Fore Systems' ASX-111 ATM switch [10], and 140 Mbps TCA-100 TURBOchannel ATM computer interfaces [11]. The interfaces[1] link DECstation 5000/200 workstations [8] that use a 25 MHz MIPS R3000 CPU [16] running over a 25 MHz TURBOchannel I/O bus [9].

The Fore Systems' interface does not have its own CPU, and does not support DMA. The interface performs only cell delineation, FIFO buffering, checksumming, and interrupt management. We believe that this is the minimum support required from any ATM host interface. Even at 140 Mbps, it is not feasible to interrupt the host CPU with each cell's arrival as the time between cells is only about 3 microseconds.

The host CPU is presented with a FIFO interface through which it sends or receives ATM cells using programmed I/O. This requires that each word in the cell be individually written or read by the host CPU. There is not a clear case for or against DMA-support in low-latency networks. The use of programmed I/O can reduce latency since no DMA set-up is required, and can eliminate cache consistency problems since all messages go through the CPU. However, programmed I/O also limits the available bandwidth across the I/O bus. With DMA the TURBOchannel can transfer one 32-bit word per cycle, giving a peak rate of 800 Mbps. In contrast, programmed I/O requires 8 cycles to read a word[2]. This yields a raw input bandwidth of 80 Mbps for the CPU to transfer data from the receive FIFO to memory, which is only about half of the network transmission rate. On the transmit path, the CPU needs 3 cycles to write words across the I/O bus, giving a peak of 160 Mbps for the host to read from memory and write to the transmit FIFO — enough to saturate the link. As a result, the receive path will tend to overrun when driven by a similarly equipped host, so flow control between the sender and receiver is needed to avoid loss of cells. The receive FIFO has capacity for 292 cells, while the transmit FIFO can hold up to 36 cells.

Each interface is connected to a switch, which has capacity for up to 16 input ports and 16 output ports (switches with 64 ports are rapidly becoming available). Each output port has a FIFO with capacity for 256 cells, and shares with three other output ports two additional FIFOs with each a capacity of 512 cells. A given output port has, on average, sufficient buffering for 512 cells, but is capable of holding up to 1280 cells during peak traffic periods. Switch delay is less than 10 microseconds if the output FIFO is empty.

3. The problem with layered protocols over ATM

A simple way to incorporate a new network technology such as ATM into an existing operating system is to layer it beneath an existing protocol stack such as TCP/IP or UDP/IP. In this way, ATM provides the physical transport only, analogous to an Ethernet, a token ring, or even a serial line. A layered implementation would have the ATM layer move cells through to the hardware

[1] The Fore interface delivers slightly less bandwidth than the 155 Mbps capacity of the network.

[2] The number of cycles required depends on the particular TURBOchannel implementation; the minimum is 4 cycles [9].

FIFOs. The next layer would perform SAR, possibly using linked buffers since the SAR layer does not know the total message length and cannot allocate a contiguous buffer of the total size necessary on reception. A CS layer on top of SAR would insert or remove the CS header and trailer, in the latter case verifying that the reassembled length equals that in the CS trailer. IP, which provides routing, fragmentation and reassembly, would be layered on top of CS. UDP, which provides connectionless service and demultiplexing of packets to user endpoints, and TCP, which provides connection-oriented, reliable stream communication with automatic flow control and error recovery, would in turn be layered on top of IP. Finally, sockets, which provide application-level endpoints and an additional buffering layer [19], would be placed on top of UDP and TCP.

Several existing implementations follow this approach [29, 3]. While a layered implementation reuses existing protocol software, it also introduces redundancies that affect end-to-end latencies and bandwidth [33, 17].

- Special connection management and routing mechanisms are required because ATM uses its own signaling protocols to establish and tear down connections. These protocols are different from the ones used by higher-level protocols such as TCP.

- The ATM SAR protocol makes IP fragmentation and reassembly redundant. Multiple segmentation and reassembly layers increase latency because a message must be broken up and reconstructed multiple times, and decrease usable bandwidth because SAR structural information must be carried in the message. IP typically enforces a maximum packet length (MTU) of 576 bytes[3]. Any packets larger than that are fragmented and reassembled, as in ATM SAR. To prevent IP fragmentation, TCP typically uses a maximum segment size (MSS) of 512 or 536 bytes (the slack is for the TCP/IP header). As IP headers alone are at least 20 bytes long, it is impractical for IP to use an MTU of 44 bytes (the maximum AAL 3/4 segment length). The TCP header adds another 20 bytes, so it is impossible for TCP to avoid ATM SAR because even the TCP/IP header alone cannot be transmitted in a single AAL 3/4 cell (a single-segment AAL 3/4 message can hold only 36 bytes). The ATM interface would thus need to inform IP of a larger, fake MTU, for example 576 bytes, and then do its own SAR to break the large IP fragments into 44-byte segments. Consequently, the message may be fragmented multiple times, once by IP (or preventively by TCP), and then again by the ATM SAR. While an outboard processor can perform ATM SAR processing to reduce CPU overhead, it does not eliminate the latency and bandwidth costs of redundant fragmentation.

- The small ATM packet size leads to poor network efficiency when used to carry IP traffic because IP headers are relatively large compared to the size of application data. The distribution of TCP packet size for application data is sharply bi-modal, with a major peak at 0 bytes and a secondary peak at the MSS [2]. The same measurements have shown that 77% of all TCP packets carry between 0 and 36 bytes of application data; 6% between 37 and 80 bytes; 2% between 81 and 124 bytes, 10% carry 512 bytes, and 4% carry 536 bytes. That data could be transmitted without TCP/IP headers in 1, 2, 3, 12 or 13 AAL 3/4 segments, respectively, requiring an average of 2.7 cells per message. The TCP/IP header adds at least 40 bytes to the application data and raises the average number of cells per packet to 3.7. This decreases the transmission efficiency by 37%. Similar observations about UDP behavior reveal that headers raise the average number of cells from 2.1 to 2.5, which is a 19% loss in efficiency.

[3]Larger MTUs can be used in specific LANs

- TCP's flow control scheme reacts to congestion by reducing window sizes and by retransmitting lost segments [24], but also relies on implicit lower-level limits that do not exist naturally in ATM networks. For example, on Ethernet or FDDI [26], only one host or gateway can transmit at a time, so any attached node can never receive data above the fixed network rate. In an ATM network, though, switches can receive data at multiple input ports simultaneously, and at each port at full link bandwidth. If all data is destined to a single output port and available buffering is finite, data loss will result. Moreover, TCP's *slow start* mechanism [6] *causes* data loss, even in the absence of cross traffic (traffic from other connections traveling over the same path or subpath). Slow start increases the window size each time an acknowledgement is received. Eventually, the buffering capacity of the destination or intermediate nodes is exceeded, causing packet loss, and reducing the window back to one segment. As a result, window sizes oscillate between extremes [28].

Finally, in a layered implementation such as the one outlined above, a message's data must be manipulated and copied multiple times: first from the hardware FIFOs to cells in host memory, then from cells into linked buffers (SAR), then into processor registers for checksumming (TCP or UDP), and finally into user buffers (socket layer). The transmit path is symmetric. This behavior wastes CPU cycles and interacts badly with the limited memory and I/O bus bandwidth commonly found in workstations [23]. For example, memory-to-memory copy bandwidth on a DECstation 5000/200 is limited to about 160 Mbps, which is only slightly greater than the 155 Mbps ATM transmission rate, and substantially less than the 622 Mbps transmission rate in future ATM networks. Any protocol layer or sublayer that references the data in the packet will take time comparable to that necessary to transfer the data over the physical network. With multiple protocol layers and passes over the same data, protocol processing quickly becomes the bottleneck in the pipeline between communicating applications on different hosts.

4. An integrated protocol architecture

To address the problems of redundant processing outlined in the previous section, we have designed and implemented a protocol architecture specifically tailored to the requirements and capabilities of ATM. As previously mentioned, we are focusing on the case of local and metropolitan area networking, specifically over distances limited to about 40 km. Additionally, in the work presented in this paper we are concerned exclusively with data communication, supporting only statistical multiplexing of network bandwidth.

Our network architecture supports communication between *endpoints*, which are the entities through which user tasks send or receive messages. An endpoint is analogous to a BSD Unix socket [19]. The task that creates an endpoint is its *owner*, and is the only one allowed to send or receive messages through it.

Endpoints have associated *buffer areas* that are co-mapped into the address spaces of the kernel and the endpoint owner task. Buffers for received messages are allocated from the buffer area of the receiving endpoint. An endpoint's protocol and buffer area size are defined when it is created by a task. The protocol determines the transport discipline applied to messages sent or received through the endpoint, and can be one of datagrams, sequenced packets and raw datagrams.

- Datagrams are sent over a multiplexed connection between the communicating hosts, and

are automatically flow controlled. A datagram endpoint can simultaneously receive messages from multiple hosts.

- Sequenced packets are transported over an exclusive connection between the respective endpoints, and are automatically numbered, acknowledged, and retransmitted in case of error or time-out.

- Raw datagrams are provided for internetworking and wide-area accesses. Raw datagrams undergo only ATM, SAR and CS layer processing. They are neither reliable nor flow-controlled. These functions are left to higher-level software. Like datagrams, raw datagrams are sent over multiplexed connections.

Datagrams and sequenced packets can be up to 64 KB long, and are intended for the common case of hosts connected by ATM in the same local or metropolitan area network. We limit the size of raw datagrams to 9244 bytes because they are not flow-controlled (this limit was chosen for compatibility with IEEE 802.6 and with the size of the receive FIFOs of the hardware we use). The protocol of communicating endpoints must be compatible.

Our protocols are implemented within a kernel-level device driver that provides the ATM, SAR, CS and signaling layers, as well as the flow control and error recovery required for datagrams and sequenced packets. The driver is organized in a single layer and runs entirely in kernel mode for protection. Other protocols, such as TCP/IP, or application programming interfaces, such as sockets, can be layered on top of the native interface for compatibility. These higher layers can be implemented inside the kernel or at user level [20].

In the rest of this section we discuss in more detail our approach to buffer management, segmentation and reassembly, flow control, error control and recovery, signaling, multiplexing, and integration with Internet protocols.

4.1. Buffer management

Protocol implementations often use their own buffers, distinct from the buffers used by the end applications. In BSD Unix, for example, protocols use *mbufs* [19], while applications use buffers allocated from the heap or stack. This imposes at least one extra data copy through mbufs on the path between the application and the network.

We integrate buffer management for protocols and applications, and therefore eliminate this copy. Endpoints have exclusive buffer areas that are statically mapped between the kernel-level protocol driver and the endpoint's associated address space. When a message arrives from the network, the network interface allocates the buffer for it from the buffer area of the receiving endpoint. Since this buffer area is already mapped to the endpoint's owner address space, the application can receive the message with no copies or virtual memory manipulations. When sending data, the sender blocks until the message has been sent (and, in the case of sequenced packets, acknowledged). This avoids having to copy an outgoing message into an intermediate buffer. Outgoing messages should be allocated from the mapped buffers to ensure that the driver always has rapid access to message data, even when the currently running address space does not own the data. This is necessary during the transmission of a long message that may span address space context switches. The buffering structure is *exposed* and applications are co-responsible for how buffers are utilized. If multiple threads run in the address space from which a message is sent, it it the

application's responsibility to synchronize buffer accesses as necessary for integrity. Buffers can be deallocated either immediately after being sent, or later by an explicit call.

Integrated buffer management does not cause memory protection problems because the network interface runs inside the kernel and is trusted, and each application can only access the buffers of the endpoints it owns. Misbehavior by an application, e.g. not freeing message buffers, only affects that application, since the buffer areas of each endpoint are separate.

4.2. Segmentation and reassembly

Because of ATM's extremely small cell size, segmentation and reassembly performance are critical to overall network performance. To minimize the impact of SAR, we reduce the number of cell manipulation steps by combining the ATM, SAR and CS functionality into a single pass over the data.

The ATM, SAR, and CS headers, trailers, and pads are generated or verified on the fly as data is transferred into or out of the FIFOs. They are never stored in memory during either send or receive. When the beginning segment of a message is received, the CS header is read to determine the size of the incoming message, and a contiguous buffer of the appropriate size is allocated from the receiving endpoint's buffer area. The receiving endpoint is determined by the ATM header's VPI/VCI field, which we use to index into a table that maintains a list of free buffers for the respective endpoint. Once the buffer has been allocated in the receiver's address space, the network interface reads data words from the receive FIFO and places them directly in the correct location in the buffer. When an end of message is received, the message data is in the buffer ready for use by the application. The result is that ATM, SAR and CS processing can be performed at a rate close to that at which the host CPU can transfer words across the I/O bus.

4.3. Flow control and congestion avoidance

Network flow control is required to ensure that a sending host does not overrun the resources available at the receiver. Overrun is possible because a receiver must react to and process data as it arrives, whereas a sender may carefully stage the transfer for maximum bandwidth. Our basic approach to flow control is to allow the receiver a similar degree of control over data staging so that it can predict and reserve resources on a message by message basis.

Because of the nature of network traffic, which consists of many small messages and a few large messages, our flow control strategy separates messages into two classes, *short* and *long*. Short messages, which can be transported in one or two ATM cells, and which account for almost 80% of Internet traffic, are the common case for which we optimize. We transport short messages without any explicit flow control. Instead, we dedicate a small fraction of a receiver's resources to short messages. Since the bandwidth and buffering requirements of short messages are minimal, the dedicated allotment is more than enough in most practical situations.

Long messages, greater than two segments, require explicit flow control and congestion avoidance mechanisms. We provide flow control for long messages with a *sliding window* protocol [31]. Before a long message is transmitted, it is broken into *windows*, which are the unit of synchronization between sender and receiver. After sending a window, the sender must receive a corresponding

synchronization cell before sending the next window in the message. We use a window size of one segment for the initial window in a message, 71 segments (explained below) for all but the last window, and between 1 and 71 segments for the last window. The initial window, being small, is unlikely to overflow the receiver's FIFO. If a host receives a new initial window while it is already receiving two long messages, then the receiving host queues the corresponding synchronization message. The synchronization is sent when there is only one long message outstanding. Using this *buffer scheduling* technique, the receiver substantially reduces the chances of FIFO overrun.

The choice of window size is a compromise between the conflicting requirements of bandwidth efficiency and buffering limitations. In our windowing scheme, only one window of each message can be in transit at any given time, and the receiver sends a synchronization cell when it processes the initial segment in a window. Small windows are therefore less likely to cause congestion; only a large number of concurrent messages can cause buffer overrun. On the other hand, large windows allow better utilization of the available bandwidth because a large window allows the synchronization cell to overlap with the transmission of the window. When the current window has been completely sent, its synchronization has already arrived at the sender, and the sender can continue transmitting without delay. Overlap occurs if $WS \geq RTT \times BW$, where WS is the window size, RTT is the round-trip time, and BW is the available bandwidth. The maximum reception bandwidth of the network interface is about 50 Mbps, and the maximum transmission bandwidth is around 110 Mbps. The round-trip time, assuming that the propagation delay is 5 microseconds/km, that hosts are separated by d km and s switches, and that every queue encountered on the path is empty, is $RTT = 10d + 20s$. Switches with 16 ports can handle up to 3600 hosts such that no two hosts are more than 5 switches away from each other. A network of switches with 64 ports each can handle over 250,000 hosts with a maximum distance of 5 switches. Assuming hosts separated by up to 40 km and 5 switches, a window size of 71 segments enables transmission at the maximum reception bandwidth.

The FIFO space reserved for reception of a long message will depend on the actual RTT, and on the relative send and receive bandwidths, BW_t and BW_r. If $BW_t = BW_r$, in principle, no FIFO space is required, but as mentioned, the transmit and receive bandwidths can be unequal (in our case, $BW_r < BW_t$). We can conservatively bound the necessary FIFO space by assuming that $RTT = 0$, since cells of the next window would start arriving as soon as processing on the current window begins. Assuming $RTT = 0$, the maximum FIFO space is $QS_{max} = 2WS - BW_r/BW_t$. Assuming conservatively that $BW_r = 40$ Mbps, $BW_t = 110$ Mbps, and $WS = 71$ cells, then $QS_{max} = 116$ cells. In allowing simultaneous reception of two long messages, the required queue size doubles to 232 cells. The remaining 60 cells in the receive FIFO are left as a slack for short messages or initial (single-segment) windows. The slack will in practice be higher because long messages are not the common case, FIFO usage by each long message is expected to peak at different instants, RTT is not zero, and final windows are on average only half full.

Cell loss resulting from receiver overflow can still occur if a large number of short messages or initial windows are sent to a host that is already receiving long messages. The receive FIFO, though, can buffer at least 30 short messages (60 cells) or 60 initial windows, and can be drained in less than 2.5 milliseconds. This enables more than 10000 short messages per second to be consumed in the worst case of two long messages outstanding.

Congestion avoidance

Network congestion can occur whenever the aggregate bandwidth destined for a particular switch output port exceeds that output port's capacity. For example, congestion can occur in a single-switch network if two or more hosts continuously flood a single receiver, even though neither sender exceeds the maximum bandwidth of its link to the switch. Congestion often results in cell lossage because the switch has finite buffering capacity.

We use four simple techniques to avoid congestion: dynamic window sizing, buffer scheduling at the receiving host, randomized time-outs with exponential back-off, and static traffic analysis with topology optimization. As previously described, the first two techniques, dynamic window sizing and buffer scheduling, reduce the possibility of congestion between hosts connected to the same switch.

Buffer scheduling at the receiving hosts is performed with local information only. Consequently, we cannot detect traffic between intermediate switches. Congestion is possible in the switches if many large messages are transmitted between hosts connected to different switches. In this case, congestion must either be handled with intermediate resource reservation, link-by-link flow control, or timeouts and retransmission. Because the first strategy increases connection setup time and can waste network resources during a connection's idle periods, and the second strategy requires sophisticated switch and interface hardware, we rely on the third strategy.

The receiver uses timeout to reclaim its reserved FIFO space in cases such as sender failure, or continuous loss of synchronization cells, allowing other long messages to be received. The sender uses timeout to detect a lost initial window, lost synchronization cell, or lost segments at the end of a window (which the receiver cannot detect, since it keeps waiting for these segments). If the sender did not timeout on these events, remaining windows in affected messages would remain queued for transmission indefinitely. On timeout, the sender retries several times before aborting the transmission request. The timeout used by different hosts is randomized and increases exponentially according to the number of retries. This tends to spread retransmissions in time, and allows forward progress in spite of congestion.

The effectiveness of timeouts is enhanced by the strict FIFO buffering in the hosts and switches. A cell cannot linger indefinitely in a FIFO, so any cell not relayed within a certain bounded time has been dropped. Since we bound the distance and number of switches, we can also bound the round trip time, which makes it possible to avoid spurious timeouts. The timeout value is based on the worst-case assumptions that the window and its acknowledgements travel through 5 switches across 40 km, that queues are nearly full at each intermediate switch, and that end-host interrupt latency is extremely high (30 msecs).

Lastly, if congestion remains a frequent problem despite the protocol's best efforts, the problem remains with the network topology or link capacities. At this point, the only solution is to analyze the traffic and reconfigure the network so that hosts that communicate the most with each other are connected to the same switch. Other possible rearrangements include using larger switches, or increasing the number or transmission rate of links between congested switches. This is analogous to the routine management of Ethernet subnetworks, routers and gateways.

4.4. Low-overhead error control and recovery

As with segmentation and reassembly, ATM's small cell size imposes a high demand on any error control and recovery mechanism. ATM and AAL 3/4 require that each cell be checked for errors in the ATM header, payload CRC, segment type, and data lengths. The AAL 3/4 specification prescribes that an entire message should be dropped in case an error is encountered in one of its received segments.

ATM and AAL 3/4 error control plus the flow control timeouts and retransmissions described in the previous section are nearly sufficient to provide a reliable transport protocol. A reliable transport can be provided with small additional overhead *if* it can be integrated with the error and flow control mechanisms. In the common case of messages being delivered without error, the only additional required service is message sequencing, and end-of-message synchronization (acknowledgment) to indicate that an entire message has been correctly received. We include these services in our protocol implementation for the case of sequenced, reliable messages.

Any error detected by the receiver causes the receiver to discard the entire message in which the error occurred. Consequently, a sender must retransmit every message from the beginning. The arrival of a message that is already being received is not considered an error, but rather the result of a timeout on the sender (for example, due to a lost acknowledgment). Previously received data in the message is simply dropped by the receiver.

Timeouts are sufficient for error control and recovery, but recovery can be accelerated by immediately returning to the sender an indication of the error during events that are unrelated to congestion. A *Negative Acknowledgement* is returned when a message has been corrupted. The sender retries the message several times but eventually gives up and returns an error to the sending process.

An *Overflow Error* indicates a high-level resource reservation problem that occurs whenever the receiver does not have sufficient buffer space to absorb the incoming message. The protocol does not attempt to recover from an overflow error, as it reflects a problem with resource reservation at or above the session layer. Instead, the protocol returns the error indication in response to the initial window request. The sender relays the error indication up to the sending process.

Errors resulting from network congestion where a cell has been dropped by an intermediate switch are not acknowledged by the receiver. These include errors in the segment type, segment sequence number, or an inconsistency in the reassembled message length. In such cases, the offending message is simply dropped. The sender detects the error through a timeout after a randomized delay as described in Section 4.3. The sender retries several more times, each with an exponentially increasing timeout. We do not indicate the error immediately with a negative acknowledgement from the receiver to the sender because, in the case of congestion, this would tend to synchronize the transmission of different senders, preserving the congestion.

4.5. Signaling and multiplexing

The hardware we used employs the SPANS signaling protocol. This protocol allows hosts to send *requests* to switches to open or close connections. The switches in turn send corresponding *indications* to the destination of the connection. The destination determines whether or not the

indication should be accepted and returns a *response* that is translated into a *confirmation* to the source host. Signaling messages are sent over a well-known meta-connection that is maintained by hosts and switch software. Nevertheless, signaling can incur significant latency on the order of tens of milliseconds per switch. This latency is not a concern when connections remain open for a long time (e.g., for sequenced packets), but becomes prohibitive if connections are to be set up on demand to send specific datagrams.

To provide efficient datagram support, we transport datagrams over multiplexed connections (*lines*) between hosts. If a datagram is to be sent to a host for which no line exists, a line is dynamically opened. Subsequently, any datagrams sent between the two hosts use the same connection, so signaling costs are paid only once. The endpoints connected by the line are also dynamically created. Since the line initiator does not know the endpoint that will be assigned to the line at the peer host, it names instead a *virtual endpoint* as the destination endpoint. The signaling implementation treats this case specially, dynamically creating the line's endpoint at the peer host. The initiator learns the identity of the line endpoint at the peer at the time it receives the corresponding line open indication in the reverse direction. The protocol software reclaims lines when the number of free endpoints on a host drops below a threshold.

The buffer for a received message is allocated directly from the buffer area of the destination endpoint, rather than the line endpoint. Multiple messages may be received simultaneously over different line endpoints for the same destination endpoint. Each line endpoint reassembles a message on a different buffer, although all buffers are allocated from the same buffer area.

4.6. Integration with Internet protocols

Internet protocols may be layered on top of our network interface to allow communication over heterogeneous or wide-area networks. IP communicates through a special, well-known endpoint pre-allocated on each node. The IP endpoint uses the raw datagram protocol; flow and error control must be provided by higher level software (e.g. TCP) rather than the network interface. The TCP/IP stack can be implemented inside or outside the kernel; in the latter case a privileged call is offered for the IP task to register itself as the owner of the IP endpoint.

We provide two features to improve the performance of integration of Internet protocols with our network interface: fast address lookup, and fast demultiplexing. An address lookup service maps IP addresses or host names into physical ATM addresses. The session layer software can use this service to bypass TCP/IP layers when the ATM address lookup succeeds. Fast packet demultiplexing is made possible by *port registration*. Buffers for incoming IP messages are normally allocated from the IP endpoint buffer area, and delivered to threads waiting on the IP endpoint. However, the IP task can register specific ATM endpoints for given TCP or UDP ports. If an IP message is received and there is a specific endpoint registered for the TCP or UDP port indicated in the header, the buffer for the message is allocated from the buffer area of the specific endpoint, and not of the IP endpoint. If the IP task arranges for the specific endpoint to be co-owned by the applications having access to the given TCP or UDP port, a copy is avoided, since the buffer is allocated from an area already mapped to the correct application address spaces. Furthermore, such messages are delivered to any threads that are waiting on the specific endpoint. These threads may be running on the end application address space, further reducing the demultiplexing latency [20].

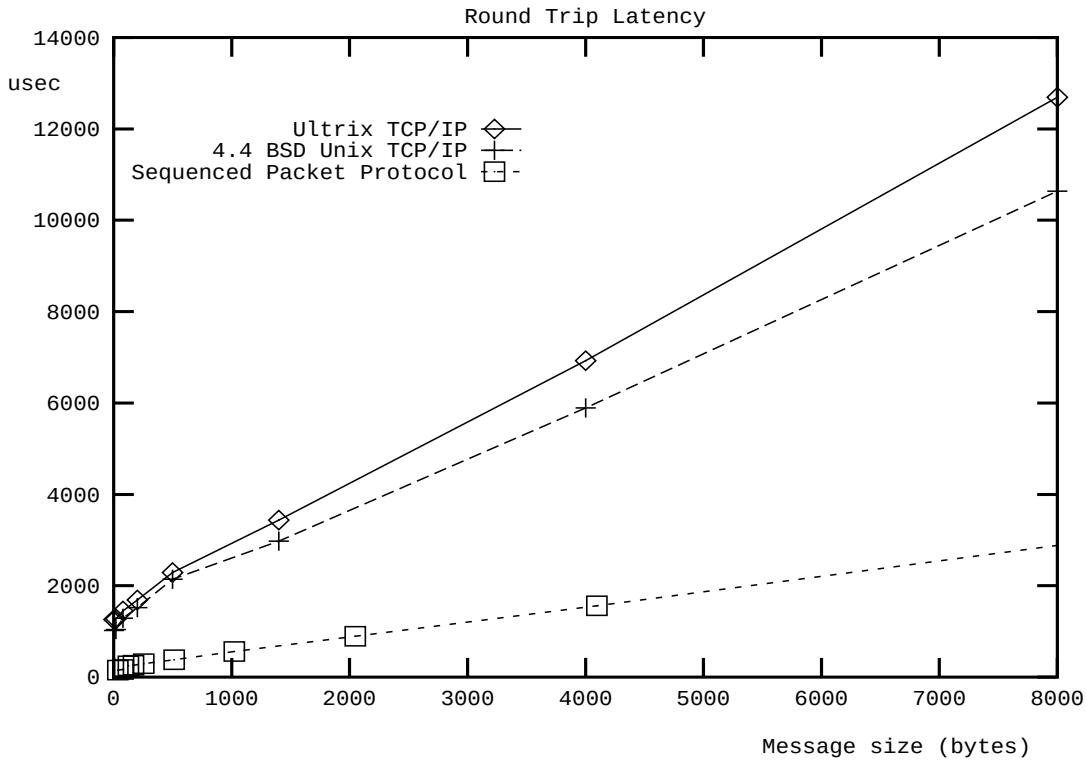

Figure 1: Process-to-process round-trip latencies using TCP/IP and using our integrated sequenced packet protocol.

5. Performance

In this section we present the performance characteristics of our protocol implementation in terms of throughput and latency. All measurements are between two DECstation 5000/200 workstations connected directly by Fore Systems TCA 100 ATM interfaces.[4]

Our protocol suite is implemented with about 4100 lines of C code (not including signaling support). Determining which part of our implementation corresponds to which conceptual protocol layer is difficult because the implementation is integrated. Roughly, 3300 lines are responsible for device management, and the ATM, SAR and CS "layers." Our higher-level extensions to the base ATM services constitute slightly less than 15% of the code. Fewer than 3% of the code deals with reliable messaging. This confirms that higher-level functionality can be integrated into the network interface level without adding undue complexity.

Process-to-process (user-level) round-trip times comparing our sequenced packet protocol and two TCP/IP implementations, one from Ultrix and one from 4.4 BSD, are shown in Figure 1. (The TCP measurements are taken from [33].) The round-trip time for 4-byte messages is 152 microseconds for our sequenced packet protocol, 1261 microseconds using Ultrix TCP/IP, and 1021 microseconds using BSD 4.4 TCP/IP. The sequenced packet protocol requires two messages (sent by the user) and two acknowledgments (sent by the driver, but mostly overlapped with

[4]We do not include switch delay in our throughput and latencies to facilitate comparisons to other published numbers on the same hardware.

user processing). In addition to latency much lower than that of TCP/IP, our interface achieves considerably higher throughput, as indicated by the different inclinations of the curves in Figure 1. We believe our better performance can be attributed primarily to the integrated layer processing organization of our implementation, and to our integrated buffer management. The large latency and bandwidth difference between our protocol and both versions of TCP/IP argues strongly in favor of bypassing higher-level protocols whenever possible. The performance disparity supports our decision to build protocols and operating system interfaces from scratch, and confirms earlier predictions that good protocol performance can be obtained with relatively simple hardware [4].

Figure 2 shows the throughput obtained with different message sizes for our three ATM protocol options. Throughput for sequenced packets and datagrams converge rapidly to their maximum sustained throughput of 48 Mbps. Both curves have their knee at about 4 KB. For messages at or above 4 KB, the throughput overhead of flow control, error and sequence control and connection multiplexing amounts to less than 5%. The maximum throughput that could theoretically be achieved at the receive side, considering only the I/O bus bandwidth and loss due to headers and trailers while neglecting possibly significant cache effects, is 62 Mbps. Our end-to-end throughput achieves 76% of this upper bound. The throughput curve for the raw protocol stops at a relatively small message size because it is not flow-controlled, causing receiver FIFO overrun at larger message sizes.

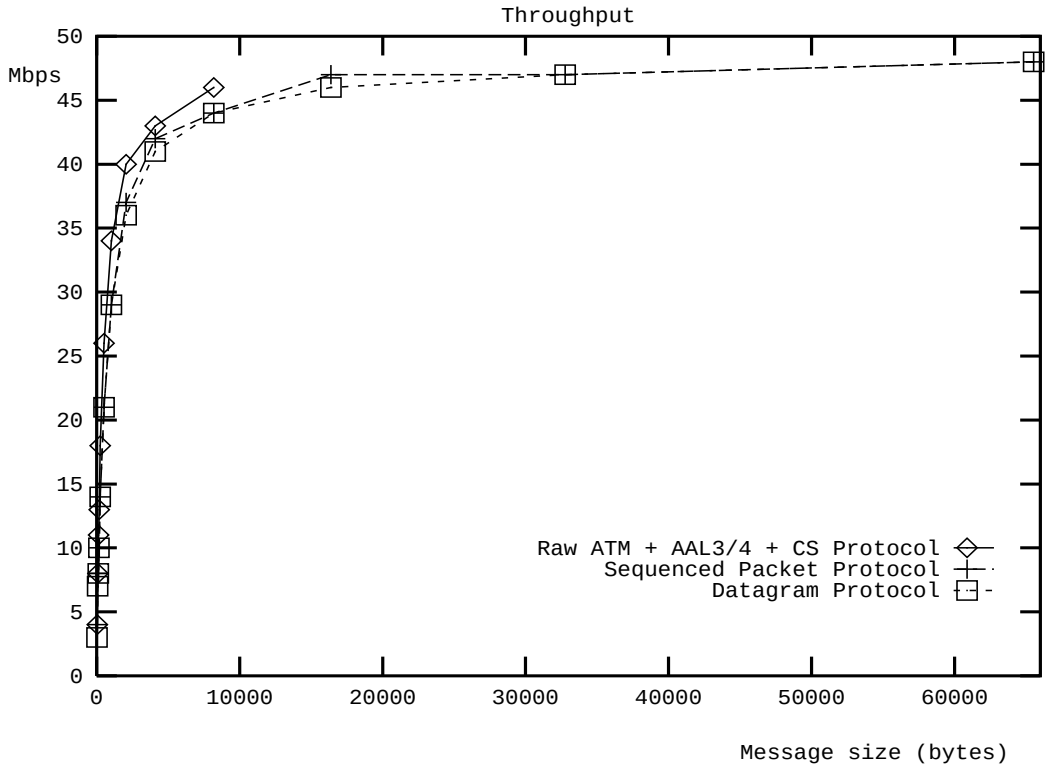

Figure 2: Process-to-process throughput using our integrated protocols.

To determine the relative overheads of flow control, error and sequence control and connection multiplexing, we measured the round-trip times and throughput when sending messages using sequenced packets (reliable, flow-controlled), datagrams (flow-controlled), and a raw protocol implementing only the ATM, SAR and CS layers over exclusive connections. Figure 3 shows the round-trip times for messages smaller than 800 bytes. Up to 80 bytes, the round-trip time for se-

quenced packets is only marginally higher than that for the raw protocol. This is because reliability is integrated at the network interface level, and the only non-overlapped cost of the acknowledgement is that of sending one synchronization cell. The round-trip time for datagrams is slightly higher because of multiplexing/demultiplexing overheads. Messages larger than 80 bytes sent both using sequenced packets and datagrams are flow-controlled, causing the step seen in the figure. This step is due to the non-overlapped synchronization at the initial window. Other synchronizations, however, are overlapped, and the step remains constant for higher message sizes, allowing the flow control overhead to be amortized with larger transmission units.

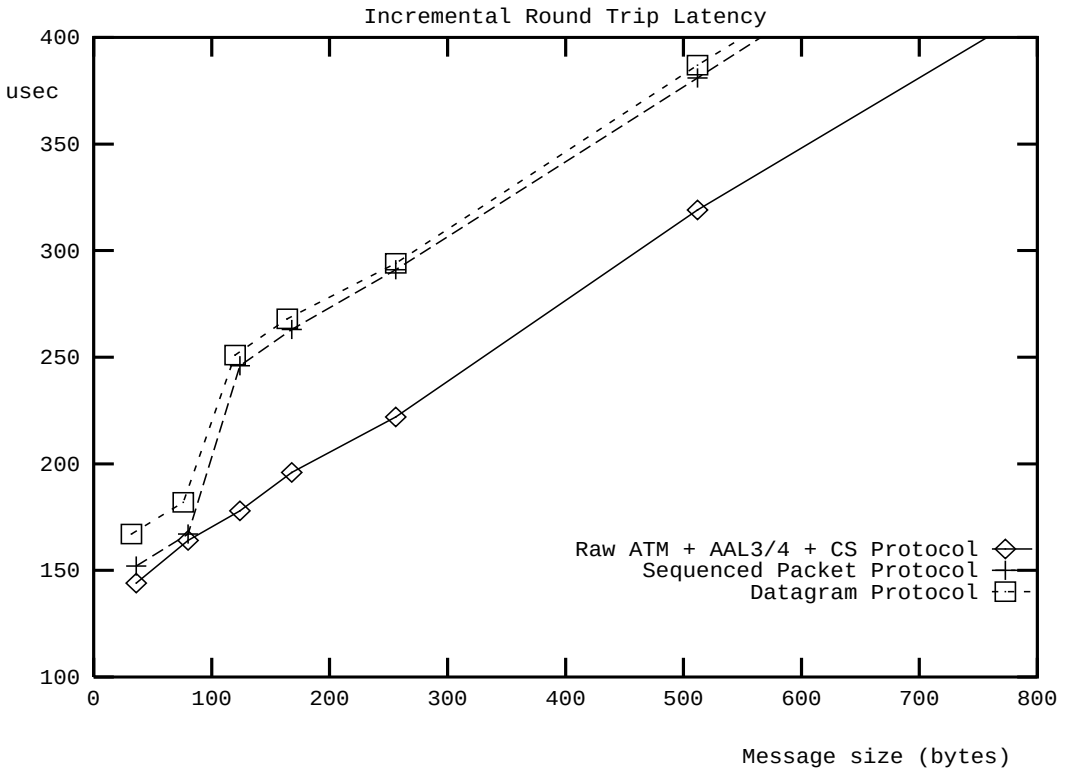

Figure 3: Comparative process-to-process round-trip latencies using our integrated protocols.

6. Related work

Similarities between our work and that of others exists along two dimensions: protocol structure and protocol strategy. In terms of protocol structure, integrated layer processing was first proposed by Clark and Tennenhouse as a necessary technique to achieve good protocol performance by eliminating the number of times that data in a message is manipulated [7]. Our integrated buffer management strategies are similar to those used in DEC SRC RPC implementation [27]. Our overall approach of integrating ATM, SAR, CS, buffering, flow control, and error recovery demonstrates that additional functionality (for example, error recovery) can be integrated into a single layer with little overhead. Protocol bypass has been proposed before by the *x*-kernel [14] developers, with their emphasis on the *factorization* of common layers in different protocol stacks. Factorization implies a layered implementation, though, which conflicts with the goals of integrated layer processing.

In terms of protocol strategy, our flow control and congestion avoidance scheme is similar to

TCP's in that we use sliding windows, an initial window size of one segment, increase the window size after acknowledgement of the initial window, and drop the window size back to one segment in case of timeout and message retransmission.[5] There are, however, important differences. First, our initial window size is much smaller, reducing the probability of cell loss in case other messages are being received concurrently at the destination. Second, on successful transmission of the initial window, we switch immediately to a window size statically determined to support transmission at full bandwidth with low probability of buffer overrun instead of hunting for the right window size using slow start. Third, we send synchronizations at each window rather than each segment, reducing synchronization bandwidth overhead. Fourth, our retransmission with exponential back-off effectively integrates the mechanism that TCP implicitly uses when managing congestion over broadcast-based networks such as Ethernet and FDDI. Fifth, our buffer scheduling coordinates reception of messages on different sessions so as to reduce the probability of FIFO overrun, while TCP control is per-connection only. Finally, our flow and congestion control algorithms support both connection-oriented and connectionless communication, while TCP is connection-oriented only.

Our flow control scheme can be contrasted to those being investigated in DEC SRC's Autonet-2 [29] and CMU's VC-Nectar [30] projects. In AN-2, each cell received is acknowledged individually on a link-by-link, per-connection basis, using a non-standard piggy-back field in the ATM header of cells flowing in the reverse direction. In contrast, VC Nectar uses explicit *credit cells* to indicate the existence of reserved buffers for a specified number of cells over a connection. The sender only attempts transmission if it has obtained the required credit from the next node on the path. Our synchronization cells have function similar to AN-2's acknowledgements or VC Nectar's credit cells, but on an end-to-end basis. Link-by-link control allows more precise tracking of widely varying traffic conditions, but at the expense of considerably more sophisticated host interfaces and switches.

7. Conclusions

The high-bandwidth, low-latency performance of the coming generation of networks can be easily sacrificed through redundant protocol processing. These networks, such as ATM, require specific low-level support for services such as connection management, flow control, congestion avoidance, and segmentation and reassembly. Slight, but well-integrated extensions to these base services are sufficient to ensure efficient reliable delivery. Our work demonstrates that it is possible to integrate multiple protocol layers into a relatively simple single-layered implementation that delivers both low latency and high throughput.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. F. Rashid, A. Tevanian Jr. and M. W. Young. "Mach: A new kernel foundation for UNIX development", in *Proceedings of the Summer 1986 USENIX Conference*, July 1986, pp. 93-113.

[2] Ramón Cáceres. "Efficiency of ATM Networks in Transporting Wide-Area Data Traffic", Technical Report TR-91-043, International Computer Science Institute, Berkeley, CA, July, 1991.

[5] The drop-back to one segment occurs implicitly since a message is retransmitted from the first window on error.

[3] D. Cohen, G. Finn, R. Felderman, A. DeSchon. "ATOMIC: A Low-Cost, Very High-Speed LAN", University of Southern California, Information Sciences Institute, ISI/RR-92-291, Sept. 1992.

[4] E. Cooper, O. Menzilcioglu, R. Sansom and François Bitz. "Host Interface Design For ATM LANs", in *Proceedings of the 16th Conference on Local Computer Networks"*, IEEE, Oct. 1991, pp. 247-258.

[5] P. Dale. "A Critique of OSF/1", presented at the *1992 DARPA Software Systems Workshop*, March 1993.

[6] David D. Clark, Van Jacobson, John Romkey and Howard Salwen. "An analysis of TCP processing overhead". *IEEE Communications Magazine*, June 1989, pp. 23-29.

[7] David D. Clark and David L. Tennenhouse. "Architectural considerations for a new generation of protocols", in *Proceedings of the 1990 SIGCOMM Symposium on Communications Architectures and Protocols*, ACM, September 1990, pp. 200-208.

[8] Digital Equipment Corporation. "DECstation 5000/200 KN02 System Module Functional Specification", Workstation Systems Engineering, Palo Alto, CA, Aug. 27, 1990.

[9] Digital Equipment Corporation. "TURBOchannel Hardware Specification", DEC, Maynard, MA, May 1990.

[10] Fore Systems. "ASX ATM Switch Architecture Manual", version 2.0. Pittsburgh, PA, 1992.

[11] Fore Systems. "TCA-100 TURBOchannel ATM Computer Interface User's Manual", version 1.0. Pittsburgh, PA, 1992.

[12] Fore Systems."SPANS: Simple Protocol for ATM Network Signaling", version 2.0. Pittsburgh, PA, 1992.

[13] Ken Hardwick. "HIPPI World – The Switch is the Network", in *Proceedings of the 37th IEEE Computer Society International Conference*, February 1992.

[14] N.C. Hutchinson, L.L. Peterson, M.B. Abbott, and S. O'Malley. "RPC in the x-Kernel: Evaluating New Design Techniques", in *Proceedings of the 12th ACM Symposium on Operating System Principles*, Dec. 1989, pp. 91-101.

[15] David B. Johnson. "The Peregrine High-Performance RPC System", in *Software - Practice & Experience*, Feb. 1993.

[16] Gerry Kane. "MIPS Risc Architecture", Prentice Hall, Englewood Cliffs, NJ, 1987.

[17] J. Kay and J. Pasquale. "Measurement, Analysis, and Improvement of UDP/IP Throughput for the DECstation 5000", Tech. Report, Computer Systems Lab, Dept. of Computer Science and Engineering, Univ. California at San Diego, January 1993.

[18] B.W. Lampson. "Hints for Computer System Design", in *9th ACM Symposium on Operating System Principles*, October 1983, pp. 33-48.

[19] Samuel J. Leffler, Marshall K. McKusick, Michael J. Karels and John S. Quaterman. "The Design and Implementation of the 4.3BSD UNIX Operating System", Addison-Wesley Pub. Co., Reading, MA, 1989.

[20] C. Maeda and B. Bershad. "Protocol Service Decomposition for High-Performance Internetworking", Carnegie Mellon University School of Computer Science Technical Report CMU-CS-93-131, March 1993.

[21] R.M. Metcalfe and D.R. Boggs. "Ethernet: Distributed packet switching for local computer networks", in *Communications of the ACM*, July 1976, pp. 395-404.

[22] S.E. Minzer. "Broadband ISDN and asynchronous transfer mode (ATM)", *IEEE Communications Magazine*, Sept. 1989, pp. 17-24.

[23] John K. Ousterhout. "Why aren't operating systems getting faster as fast as hardware?", in *Proceedings of the Summer 1990 USENIX Conference*, June 1990, pp. 247-256.

[24] J.B. Postel, editor. Transmission Control Protocol. Internet Request for Comments RFC 793, September 1981.

[25] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky and J. Chew. "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures", in *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, ACM, October 1987.

[26] F.E. Ross. "FDDI — An Overview", in *Proceedings of the COMPCON Spring 87*, San Francisco, CA, 1987, pp. 434-440.

[27] Michael D. Schroeder and Michael Burrows. "Performance of Firefly RPC", in *ACM Transactions on Computer Systems*, ACM, Feb. 1990, pp. 1-17.

[28] S. Scott, L. Zhang and D. Clark. "Some Observations on the Dynamics of a Congestion Control Algorithm", in *ACM Computer Communication Review*, vol. 20, no. 5, October 1990.

[29] M. Goguen. "AN2: A Self-Configuring Local ATM Network", presentation at Carnegie Mellon University, Nov. 5, 1992.

[30] Peter Steenkiste, personal communication.

[31] A. Tanenbaum. "Network Protocols", in *ACM Computing Surveys*, vol. 13, no. 4, Dec. 1981, pp. 453-489.

[32] C. Thekkath and H. Levy. "Limits to Low-Latency Communication on High-Speed Networks", University of Washington Dept. of Computer Science and Engineering Technical Report UW-TR-91-06-01, June 1991.

[33] A.Wolman, G. Voelker and C. Thekkath. "Latency Analysis of TCP on an ATM Network", University of Washington Dept. of Computer Science and Engineering Technical Report UW-TR-93-03-03, March 1993.