Latency Analysis of TCP on an ATM Network

Alec Wolman, Geoff Voelker,
and Chandramohan A. Thekkath

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Technical Report 93-03-03

Abstract

In this paper, we characterize the latency of TCP on an ATM network. Latency reduction is a difficult task, and careful analysis is the first step towards reduction. We investigate the impact of both the network controller and the protocol implementation on latency. We find that a low latency network controller has a significant impact on the overall latency even for a reliable transport protocol such as TCP, and that replacing the ULTRIX TCP implementation with the BSD 4.4 alpha implementation improves the latency up to 20%. We also characterize the impact on latency of some widely discussed improvements to TCP, such as header prediction and combining the checksum calculation with data copying.

Latency Analysis of TCP on an ATM Network

Alec Wolman, Geoff Voelker, and Chandramohan A. Thekkath

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

In this paper, we characterize the latency of TCP on an ATM network. Latency reduction is a difficult task, and careful analysis is the first step towards reduction. We investigate the impact of both the network controller and the protocol implementation on latency. We find that a low latency network controller has a significant impact on the overall latency even for a reliable transport protocol such as TCP, and that replacing the ULTRIX TCP implementation with the BSD 4.4 alpha implementation improves the latency up to 20%. We also characterize the impact on latency of some widely discussed improvements to TCP, such as header prediction and combining the checksum calculation with data copying.

1 Introduction

In this paper, we investigate the latency characteristics of the TCP transport protocol on an ATM network[4]. The characteristics of LAN technologies have changed a great deal in the last few years. With faster network hardware, the disparity between software and hardware costs is even greater. This increases the importance of efficient protocol implementations and efficient operating system interfaces. The following factors in network communication make measuring TCP performance, especially latency, interesting:

- The existence of a high quality TCP software implementation: the BSD 4.4 alpha TCP code.
- The availability of low latency network interfaces: e.g., the FORE TCA-100 ATM interface[4].

This work was supported in part by the National Science Foundation under Grants No. CCR-8907666, CDA-9123308, and CCR-9200832, by the Washington Technology Center, Apple Computer, Boeing Computer Services, Digital Equipment Corporation, and the Hewlett-Packard Corporation. Chandramohan A. Thekkath was also supported by an Intel Foundation Graduate Fellowship.

- The wide use of applications and subsystems (like RPC) that can benefit from reduced latency.

Prior studies have concentrated on the throughput characteristics of TCP on substantially different hardware or networks than the ones we describe here. We believe that studying the latency characteristics of TCP on ATM networks is particularly interesting for two reasons. First, ATM is an emerging communication standard that is likely to be widely deployed. Second, our study allows us to answer the following questions: Can we provide evidence that TCP is a viable option for a transport layer for RPC? How have the changes in technology affected the results of earlier studies (e.g., [2])? Is latency dominated by the cost of operating system services, such as buffer management? If so, can the use of such services be reduced enough to make latency acceptable for applications that require low latency?

1.1 System Overview

Before we describe our experiments, we briefly describe the software and hardware components that we used.

All of our experiments were run on a pair of DECstation 5000/200 workstations, which use a MIPS R3000 processor running at 25 MHz. Each DECstation was equipped with a FORE TCA-100 ATM network interface on the TurboChannel I/O bus. The ATM network interface uses a memory mapped receive FIFO that stores up to 292 53-byte ATM cells, and a similar transmit FIFO that stores up to 36 cells. The transmit engine starts reading from the transmit FIFO as soon as there is one complete cell in the FIFO. The ATM interface does all segmentation and reassembly (SAR) processing in the device driver, and there is no explicit hardware support for this processing.

We extracted the ULTRIX TCP code and integrated the BSD 4.4 alpha TCP code in its place. We used the ULTRIX 4.2A kernel since it supports the DECstation 5000/200, and has a driver for the FORE ATM adapter. The BSD 4.4 alpha version of TCP has improved header prediction, as well as significantly revised input and output processing.

1.2 Measurement Techniques

This paper concerns accurate measurement and analysis of many hardware and software components. We describe below our measurement techniques so that our results may be interpreted more clearly.

Many of our experiments involved making round-trip measurements of user-level processes running on an otherwise idle machine connected by a switchless private ATM network. These round-trip measurements were generated by a user-level process that ran on one machine as a client, and on the other machine as a server. The client connected to the server using TCP, then it started a timer. It then repeatedly executed the following steps: it sent *size* bytes to the server, and then waited to receive *size* bytes from the server. It then stopped the timer and recorded the value of the timer. For all the round-trip measurements in this paper, we ran 40000 iterations for at least 3 repetitions. We then took the average to get the final result. Unless stated otherwise, all tests were run using the ATM network for communication.

Latency measurements typically involve estimates of small code paths that take on the order of microseconds. To achieve this level of granularity, we used a real time clock that ticked at 40ns. This clock is on a TurboChannel card, the AN-1 controller from DEC SRC[12]. The clock is initialized at boot time, and user-level processes can access it by issuing a system call that maps the clock address into the process's address space. Reading the clock is then just a matter of dereferencing a pointer. Code inside the kernel can read the clock in a similar manner. We also added system calls to extract timings from the kernel, so that we could measure events that started in user space and ended in the kernel, or vice-versa. The use of this clock allows us to avoid instruction counting as a technique for estimating the amount of time a small section of code takes to execute. Note that we did not employ the AN-1 network in this study, only the clock on its controller.

1.3 Paper Outline

The rest of this paper is organized as follows. Section 2 summarizes our measurements of TCP latency on the baseline system. Sections 3 and 4 study the effect of several modifications that we felt were important based on the results in Section 2. Many of these modifications are not new and have been suggested by others as well in the literature [2, 6], however, our focus here is on the effect of these modifications on latency rather than throughput.

2 Measurement of the Baseline System

The baseline system that we are concerned with is the 4.4 BSD alpha release running on an ATM network. Earlier work by Kay and Pasquale[5] on TCP/IP performance of the ULTRIX 4.2A system on DECstations 5000/200s using an FDDI network had concluded that major latency improvements in TCP processing times would be difficult to achieve. We were therefore interested in determining the extent of the latency improvements, if any, achieved by the 4.4 BSD TCP implementation.

To isolate system dependencies, we integrated the BSD implementation into a standard ULTRIX 4.2A kernel and measured the latency of the implementation on the ATM network. Our measurements of the overall latency characteristics of the two TCP implementations are given in Table 1. Surprisingly, the latency of the BSD implementation is 15-20% less than that of the ULTRIX 4.2A implementation.

The overall latency measurements provide a general impression of how these implementations of TCP behave on the ATM network, but they fail to show: (1) how much of the latency is contributed by the network adapter; (2) how expensive TCP protocol processing is; and (3) how much of the latency is contributed by operating system mechanisms that are protocol-independent.

We first study the role of the network adapter on overall latency, and then address the next two items.

Size (bytes)	Round Trip Times (μs)	
	ULTRIX 4.2A	BSD 4.4 TCP
4	1261	1021
20	1277	1039
80	1445	1289
200	1685	1520
500	2290	2140
1400	3434	2976
4000	6927	5891
8000	12695	10636

Table 1: Comparison of ULTRIX 4.2A TCP versus BSD 4.4 alpha TCP on the ATM network.

Size (bytes)	Round Trip Times (μs)	
	ATM	Ethernet
4	1021	1940
20	1039	2337
80	1289	2590
200	1520	2804
500	2140	4101
1400	2976	6554
4000	5891	13168
8000	10636	22141

Table 2: Comparison of ATM versus Ethernet latencies.

2.1 Effect of Network Adapters on Latency

To demonstrate the effects of the network controller on latency, we compared the round-trip times of the BSD 4.4 TCP implementation communicating over the ATM network with the same TCP implementation communicating over Ethernet. The results are listed in Table 2. It is clear from the small byte cases (e.g., a 900 μs difference in the 4 byte case) that controller design has a large effect on overall latency. In the cases where a larger amount of data was being transferred, much of the effect can be attributed to the bandwidth of Ethernet.

2.2 Detailed Measurements of Latency

To obtain detailed latency measurements we instrumented the transmit and receive sides separately. We used the same benchmark program described above to exercise both sides. The results of our measurements of both implementations for the transmit side are shown in Tables 3a and 3b, and the results for the receive side are shown in Tables 4a and 4b.

In characterizing the latency of transmitting data using TCP, we divided the transmit operation into four time spans. The first span, **User**, measures the time from the *write* system call to the beginning of the TCP protocol implementation. This span of time includes the copying of data from

Layer		Latency (μs)							
		Packet Size (bytes)							
		4	20	80	200	500	1400	4000	8000
User		45	45	48	67	121	99	174	400
TCP	**checksum**	13	16	24	43	92	223	620	1292
	mcopy	12	13	15	29	67	34	63	126
	segment	66	65	65	67	73	122	235	461
	Total	91	94	104	139	232	379	918	1879
IP		34	35	34	35	35	61	103	200
ATM		20	21	27	34	58	38	61	58
Total		190	195	213	275	446	577	1256	2537

Table 3a: Breakdown of ULTRIX 4.2A Transmit Side Latency

Layer		Latency (μs)							
		Packet Size (bytes)							
		4	20	80	200	500	1400	4000	8000
User		45	45	48	67	121	99	174	400
TCP	**checksum**	10	12	23	42	90	209	576	1149
	mcopy	5	6	26	41	80	29	30	41
	segment	62	65	63	65	71	63	65	72
	Total	77	81	112	148	241	301	671	1262
IP		35	34	35	35	36	36	38	36
ATM		23	24	39	47	71	96	215	498
Total		180	184	234	297	469	532	1098	2196

Table 3b: Breakdown of 4.4 BSD Alpha Transmit Side Latency

user space into the socket mbufs in kernel space.

The second span, **TCP**, measures the time spent doing the TCP protocol output processing. It consists of three components, **checksum**, **mcopy**, and **segment**. **Checksum** is the time spent calculating the TCP checksum over the data and header. **Mcopy** is the time spent copying data from the socket mbufs into driver mbufs. **Segment** is the remaining TCP protocol processing time.

The third time span, **IP**, measures the time spent in IP output processing, and the last span, **ATM**, measures the time spent in the ATM network driver. To obtain an accurate measurement of latency for the last span, we only measure up to when the ATM adapter is signaled to send the last byte of data. We do not include the time of any operations after that because these operations are effectively overlapped with network transmission, which is separately accounted for.

The rows in Tables 4a and 4b have similar meanings. We use the **User** time span to refer to the time from when the data leaves the TCP layer until the time the user process runs again (except for the scheduling time, described below). **TCP** is the time spent doing the TCP input processing, and

Layer		**Latency (μs)**							
		Packet Size (bytes)							
		4	20	80	200	500	1400	4000	8000
ATM		80	80	99	127	197	398	956	1782
IPQ		22	22	22	23	24	44	45	43
IP		40	40	74	74	74	51	52	54
TCP	**checksum**	10	12	23	40	83	211	575	1142
	segment	174	173	175	181	198	170	176	100
	Total	184	185	198	221	281	381	751	1242
Wakeup		49	49	43	54	54	49	54	61
User		101	102	128	136	144	166	216	502
Total		476	478	564	635	774	1089	2074	3684

Table 4a: Breakdown of ULTRIX 4.2A Receive Side Latency

Layer		**Latency (μs)**							
		Packet Size (bytes)							
		4	20	80	200	500	1400	4000	8000
ATM		46	46	70	99	164	363	920	1783
IPQ		22	22	22	22	23	45	46	50
IP		40	40	62	62	62	53	54	43
TCP	**checksum**	10	12	23	40	82	211	578	1172
	segment	135	135	138	141	158	142	143	59
	Total	145	147	161	181	240	353	721	1231
Wakeup		46	47	47	50	49	51	58	67
User		64	65	89	81	102	124	199	468
Total		363	367	451	495	640	989	1998	3642

Table 4b: Breakdown of 4.4 BSD Alpha Receive Side Latency.

has a similar breakdown as on the transmit side. Note, however, that the TCP input processing does not have a **mcopy** row because the extra copy operation is only used on the transmit side to support retransmissions. **IP** is the time spent doing IP input processing, and **ATM** is the time spent processing and reassembling incoming ATM cells.

We also introduced two more time spans. The first, **IPQ**, measures the IP queue scheduling time, i.e., the time from when the ATM driver places received data on the IP queue and signals a software interrupt until the time the data is removed from the IP queue. The second, **Wakeup**, is the user process scheduling time, i.e., the time from when the user process is placed on the run queue until the time the it runs.

For receiving data, the time that contributes to the overall latency is more difficult to measure. The problem is that a certain amount of the cost of the receive processing is hidden by processing done by the sender. For example, if the sending ATM adapter is sending a large number of cells, then the receiving ATM adapter can be processing the first cells as the sending adapter sends the later cells.

We measure the portion of the receive processing that actually contributes to the overall latency as the time of the arrival of the last group of ATM cells comprising the last TCP segment of a data transfer to the time when the *read* system call returns to the user-level process. The difference between the second and first times is the latency of receiving that data. We use the arrival of the last group of ATM cells comprising the last TCP segment to initiate our timings because we know at that point that the sending adapter has finished sending all of the data for that transmission.

We analyzed the data in these tables. The following subsections present our major findings.

2.2.1 TCP Segments

Data transferred using TCP is done in fixed size chunks called TCP segments, and the protocol processing time is noticeably dependent upon the number of segments required to transmit a given amount of data. The BSD implementation of TCP uses a TCP segment size of 4KB with the ATM network interface, whereas the ULTRIX implementation of TCP uses a TCP segment size of 1KB.

The time spent in TCP protocol processing has two components: one that depends only on the number of segments, and another that grows with the size of the data being transferred within a given segment. The latter component is composed of the checksum calculation and a data copy, and the former is composed of protocol control block manipulation, header processing, timer management, and acknowledgment processing. The segment component is roughly constant and independent of the amount of data being transferred.

For example, for all our data transfer sizes except 8KB on the receive side, the data fits into one TCP segment in the BSD TCP implementation. However, once the size of the data being transferred grows above 1KB, the ULTRIX implementation fragments the data into multiple TCP segments. The use of multiple segments increases the protocol processing overhead and significantly contributes to poor performance of the ULTRIX implementation compared to the BSD implementation.

2.2.2 Mbuf Manipulation

For transfers of less than 1KB, one to eight mbufs are used in the BSD implementation. Beyond this size, cluster mbufs are used. The measured time to allocate and free an mbuf (independent of type) is just over 7 μs, making the mbuf manipulation a small cost relative to the overall cost of sending or receiving data.

Compared to the round trip time, this is not a significant portion of the round trip time, which is in disagreement with [2].

Latency characteristics change in a number of ways as the data transfer size grows above 1K. For example, on transmit the time spent copying data from the socket mbufs into mbufs destined for the driver in the 500 byte transfer is greater than in the 1400, 4000 (ULTRIX and BSD), and 8000 byte (BSD) transfers. We attribute this to mbuf manipulation overheads. However, these effects are artifacts of a particular buffer management implementation choice rather than inherent protocol behavior.

2.2.3 Checksum

The checksum does not scale linearly with the small transfer sizes because the checksum is done over the data and the TCP/IP header (20 bytes for TCP header + 20 bytes for IP overlay + length of TCP options). Also, as transfer sizes grow, the checksum calculation begins to dominate most other costs in sending the data, indicating that the checksum is an attractive place for optimization.

2.2.4 Data Copies

The times in three rows of the tables (**User**, **mcopy**, and **ATM**) include the cost of a data copy: the **User** time includes copying data between kernel space and user space; the **mcopy** rows in Tables 3a and Tables 3b contain the time to copy the data for supporting retransmissions; and the **ATM** row includes the time spent copy data between the host and the device.

Therefore, the data is copied at least twice on both sends and receives. The copy in **mcopy** only occurs on sends, is made from the mbuf chain for retransmissions. When packets sizes are large and cluster mbufs are used to hold the data, the **mcopy** simply increments a reference count. In these cases, it does not need to touch all the data and is less expensive than the other copies. Eliminating the checksum (discussed below) opens the possibility of alleviating these data copying costs, given a network adapter that supports DMA. With a combined copy and checksum, Clark and Jacobson [2] discuss a network adapter design that eliminates the need for a second copy.

2.2.5 Scheduling

The scheduling times for switching contexts are independent of data transfer size, both in scheduling the software interrupt for IP queue processing and in scheduling the user process to return the received data from the socket buffers. Nevertheless, these times are costly for small data transfers. Scheduling costs are 14% (140 μs out of 1040 μs) of the 4 byte and 20 byte round-trip times using the BSD implementation.

2.3 Measurement Summary

The detailed measurements have shown the contributions to latency of the various layers used in TCP communicaton. For large packet sizes, the TCP segment size, data copies, and checksum calculation significantly affect the overall processing time. For small packet sizes, the scheduling time and the time to do the TCP processing (other than the checksum and data copy on transmit) become significant. Overall, the mbuf allocation and deallocation time is not significant.

For the TCP layer in particular, the protocol processing time can be split into the time to perform the checksum, the time to do the copy during transmit, and the remainder. Although we do not further address the issue of the data copy, we address the problem of reducing the remaining protocol processing time using header prediction in the next section and the problem of optimizing the checksum in a subsequent section.

3 Header Prediction

Header prediction has often been suggested as a performance benefit for TCP[2]. There are two distinct kinds of optimizations that are often called header prediction. The first, involving prefilling parts of the transport header, is a known optimization for lowering latency [11, 8], and is not discussed further here. The second technique involves exploiting traffic locality to predict the next incoming packet and to avoid the protocol control block (PCB) lookup cost. Others have studied using traffic locality to improve throughput for bulk data transfer protocols [1, 13].

In the BSD implementation, the TCP input processing keeps a single entry cache of the most recently used PCB. If the incoming packet is from the same connection as the previous packet, the call to the PCB lookup routine is avoided. The BSD 4.4 TCP also precomputes what values it expects to find in the next incoming packet header, and can then execute a faster processing path if the prediction is correct.

A related issue is the organization of PCBs, so that lookup is efficient in the case where there is a miss in the PCB cache. The insertion algorithm for the linked list of PCBs places the most recent creation at the head of the list. The lookup algorithm for the PCBs is just a linear search through the linked list of PCBs. McKenney and Dove study alternative data structures for PCB lookup, and analyze these data structures by the expected average search length[9]. However, they do not discuss how long a search of any given length will take. While this facilitates comparisons, it is difficult to study the absolute effect of header prediction. We measured the cost of a search for a variety of lengths and show the results in Table 5. The data in the table suggest that the cost per element on a DECstation 5000/200 is just less than $1.3\mu s$. In addition, the typical number of active PCBs appears to be quite modest. For example, our departmental mail server had less than 250 active PCBs, and all of the thirty workstations we sampled had less that 50. Given the relatively small memory requirements (even for 1000 PCBs), it seems that a simple hash table implementation could eliminate the lookup problem entirely.

In light of the above discussion, we decided to neglect the cost of the lookup and analyze the overall benefit of header prediction given that lookups are free. We built a kernel where both the PCB cache and the precomputation of the next incoming packet header were disabled. By default, in our test environment, there will only be a very small number of TCP connections, because our machines are only running the standard ULTRIX daemons and our test program.

Table 6 shows the results of this experiment, comparing a kernel with header prediction disabled to a kernel with it enabled. For all the cases less than 8000 bytes, we notice only a very small improvement

Search Length (N)	**PCB Lookup Times (μs)**
20	26
50	64
100	127
200	254
500	633
1000	1280

Table 5: Time to search through N entries in the PCB list.

	Round Trip Times (μs)	
Size (bytes)	**Prediction**	**No Prediction**
4	1021	1110
20	1039	1127
80	1289	1324
200	1520	1560
500	2140	2186
1400	2976	2962
4000	5891	5950
8000	10636	11477

Table 6: Effects of Header Prediction.

with header prediction, which is basically independent of data size. This small improvement is caused by a hit in the PCB cache, since the header precomputation and check fails in these cases (explained below). In the 8000 byte case, the larger difference comes from the hit in the PCB cache and from half of the header precomputation and checks succeeding. The savings from the PCB cache hit are not large because the number of PCBs is small and the TCP connection for our test program is likely to be near the head of the PCB list, since recently created connections go at the head of the list. Even if there were many connections, a hash table implementation of PCBs would yield similar results.

The precomputation and check of the next header fails in all cases except the 8000 byte tests, where it succeeds half the time. In the 8000 byte case, this accounts for a small but noticeable difference. This is because two packets are being sent in the 8000 byte case, so the precomputation and check succeeds for the second packet. Upon closer inspection of the header prediction code, we discovered that the BSD 4.4 TCP header prediction only works in the two common cases of unidirectional data transfer. As the sender in a unidirectional transfer, header prediction succeeds when receiving an in-sequence acknowledgment with no data. As the receiver in a unidirectional transfer, header prediction succeeds when receiving an in-sequence data segment with no acknowledgment. Our test code creates the common case for a round-trip RPC style of communication where one receives data with a piggybacked acknowledgment, and this does not arise in a single sender, high throughput style of communication, which is what this code has clearly been optimized for.

To summarize our results concerning header prediction, we found that the PCB cache accounted for

Size (bytes)	Checksum and Copy Measurements (μs)			
	ULTRIX Checksum	ULTRIX Kernel bcopy	ULTRIX Total	Integrated Checksum and Copy
4	5	4	9	3
20	7	5	12	5
80	20	11	31	10
200	43	20	63	24
500	104	47	151	56
1400	283	124	407	153
4000	807	350	1157	430
8000	1605	698	2303	864

Table 7: Checksum and Copy Measurements.

a only a small improvement in latency, and that the current implementation of header precomputation does not improve latency in a bidirectional RPC style of communication.

4 TCP Checksums

4.1 Optimizing the Checksum

An optimization suggested in [2] is to combine the checksum calculation with one of the data copies. In ULTRIX, data is copied at least twice on both send and receive. One copy moves the data between user and kernel space. The other copy moves the data between kernel and device memory.

The measurements in Table 7 were calculated by a user-level test program, not in the kernel. However, the performance is indicative of the real, in-kernel implementation. The first thing to note is that our combined algorithm is faster than the ULTRIX checksum routine alone, primarily due to our use of loop unrolling, and word rather than halfword memory accesses. In the 8000 byte case, the effective bandwidth limitation imposed by the combined copy/checksum loop is just above 9MB/s on the DECstation 5000/200.

For comparison, on a Sun-3 (20MHz 68020) for 1KB of data, Van Jacobson reported 130 μs for the checksum, and 140 μs for the memory to memory copy[2]. The combined cost was 200 μs. On the DECstation 5000/200, using the standard ULTRIX kernel routines to do the checksum takes 207 μs, and the copy takes 91 μs. The combined checksum and copy takes 111 μs. This relative performance is not very surprising and is consistent with the observations by Ousterhout[10].

4.1.1 Kernel Implementation Issues

On the transmit side, we first investigated deferring the checksum calculation until the copy from kernel to device memory. However, the design our ATM interface makes this impossible. Recall that it uses a simple memory mapped transmit FIFO. As soon as a single cell has been copied into the FIFO memory, the device begins to send it as later cells are still being copied to the device. Therefore, there is no explicit action by the device driver to trigger the send. To compute the checksum, one must copy all of the data, and then write the checksum into the header of the packet. Therefore, it is impossible to combine the checksum and copy loops at the driver level given the FORE interface design.

Next, we investigated calculating the checksum during the copy from user to kernel space. The only tricky part is that the socket layer of the kernel needs to know the underlying TCP segment size for the particular connection in order to calculate the checksum for the correct amount of data.

On the receive side, it will be difficult to postpone the checksum calculation until the kernel to user space copy, because the protocol processing needs to know whether or not the incoming data is corrupt. Therefore, we think that the device memory to kernel memory copy is the right place to calculate the checksum in the receive path. The disadvantage of this is that the device driver for each network interface needs to be modified to support this.

We have a kernel implementation of the combined checksum and copy on the transmit side, and the receive side implementation is in progress. It is clear from our analysis that as the size of the data transfers increases, the checksum calculation becomes a large component of TCP processing overhead. In the 8000 byte case, if the improvement is close to what we expect from our user-level measurements, then replacing the checksum with a combined copy and checksum on both the send and receive cases could improve the overall latency from greater than 10 ms to less than 5 ms.

4.2 Eliminating the Checksum

The previous section has demonstrated the reduction in latency arising from combining the checksum calculation with a data copy. However, it is clear that latency can be further reduced by eliminating the checksum calculation altogether for local area traffic. It is already common practice to eliminate the UDP checksum for NFS traffic, although the mechanism does not distinguish between local traffic and traffic through routers. Kay and Pasquale[5] describe a mechanism to implement this change in the protocol in general. We therefore restrict ourselves to an analysis of the error characteristics of eliminating the checksum, the remaining issue left unaddressed.

The original environment in which TCP was developed provided very little support for detecting link-level errors in hardware, necessitating the use of the TCP checksum. However, current local area networks such as Ethernet and ATM calculate a link-level CRC in hardware.

To examine the effectiveness of the CRC compared to the TCP checksum, we observed the error characteristics of a typical Ethernet local area network. For 42 DECstations running Ultrix 4.2A on the same subnet in our department, we counted the errors detected by the link layer and the TCP layer on those machines. Both local area as well as wide area traffic was measured. Table 8 lists the average of these errors, where **Packets** is the number of packets received; **Errors** is the number of error packets detected; and **Error Rate** is the rate of errors defined as bad packets/total packets.

	Detected Errors		
Layer	**Packets**	**Errors**	**Error Rate**
TCP	483,147,997	162	3.4×10^{-7}
Link	628,068,556	19986	320×10^{-7}

Table 8: Errors on Ethernet from 42 machines on the same subnet in our department.

Although we do not know the number of TCP packets that both the CRC and checksum calculations missed, the last column of Table 8 shows that on a typical Ethernet the TCP checksum only contributes 1% to the detected error. Eliminating the checksum would therefore decrease the current rate of error detection by an negligible amount.

Since ATM networks are not yet in widespread use, we cannot perform the same experiment to measure the effects of removing the checksum on ATM. However, the link error rate of fiber networks is on the order of 10^{-14} bit-errors/s (i.e., one bit error in 11 days if the network is run at a bandwidth of 100 Mbits/s). Since the checksum only detects errors that the link-level CRC misses, the absolute number of errors that will be missed if the checksum calculation is eliminated is extremely small and, therefore, tolerable for many applications.

The error rates given in Table 8 are conservative in that they include errors generated from wide area network traffic and we argue for eliminating the TCP checksum calculation for local area network traffic only. Note also that the link-level errors are inherently local since a CRC is calculated on each hop, and the errors we observe are those errors generated on the LAN during the last hop. To get a feeling for what the checksum error detection rate is for LANs, we performed the following experiment. For each of the 42 machines used above, we artificially generated LAN traffic from those machines to one machine singled out to only handle LAN traffic. In a round-robin fashion we repeatedly sent 200 512-byte packets every 6 seconds from one of the 42 machines to the "local" machine during the busiest part of two days (from 8 a.m. to 8 p.m.). After receiving 2.9 million local packets (1.5 Gbytes of data), no errors were detected by the TCP checksum.

Table 9 shows the results of eliminating the checksum on round trip measurements. The packet sizes are in bytes, and all times are in microseconds. The **Checksum** column shows the average round-trip latency when the checksum is calculated; **No Checksum** shows the average round-trip latency when the checksum is not calculated; and **Ratio** is the result of dividing **No Checksum** by **Checksum**. On the 4 byte case where the checksum overhead is minimal, nothing is gained. But, the latency of the 8000 byte case is reduced by 40%.

5 Conclusions

We characterized the latency costs of TCP communication on the FORE ATM network, and investigated various methods for reducing those costs. A recent study [5] concludes that the costs are well

	Average ATM Round Trip Time Without Checksum		
Size (bytes)	Checksum	No Checksum	Ratio
4	1021	1020	1.0
20	1039	1020	0.98
80	1289	1233	0.96
200	1520	1392	0.92
500	2140	1808	0.84
1400	2976	2083	0.70
4000	5891	3633	0.62
8000	10636	6233	0.59

Table 9: Comparison of round trip latencies over ATM with and without the TCP checksum calculation.

balanced among the different layers, and that improving the overall latency will be difficult since the costs are evenly distributed. However, we observed that careful protocol implementation does make a difference. Simply replacing the ULTRIX TCP implementation with the latest BSD version improved the latency by as much as 20%. Others with experience designing high performance "lightweight" RPC systems have noticed that controller design has a significant impact on performance[14]. We discovered that controller design has a large impact on latency even using a relatively "heavyweight" protocol such as TCP. Operating system services such as memory allocation had less impact than we expected, yet context switching had more of an impact at small packet sizes than expected. Header prediction did not have a significant impact on latency, and in the future we will investigate modifying the BSD implementation to improve latency. We have found that computing the TCP checksum is a major cost of the overall TCP processing. We observed that, in local area networks, the TCP checksum does not contribute significantly to the detection of errors. We have quantified the potential benefits of (1) eliminating the checksum, (2) combining the checksum and copy, and (3) eliminating one of the copies. We have discussed the implementation issues and difficulties of these optimizations.

6 Acknowledgements

We would like to gratefully acknowledge Ed Lazowska for his encouragement and helpful comments on this project and report.

References

[1] John B. Carter and Willy Zwaenopoel. "Optimistic Implementation of Bulk Data Transfer." In *Proceedings of the 1989 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1989.

[2] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. "An Analysis of TCP Processing Overhead." *IEEE Communications Magazine* (June 1989), 23-39.

[3] Dan Dobberpuhl, R. Witek, et al. "A 200 MHz 64 bit Dual Issue CMOS Microprocessor." *International Solid-State Circuits Conference 1992*, February 1992.

[4] FORE Systems. *TCA-100 TURBOchannel ATM Computer Interface, User's Manual*, 1992.

[5] Jonathan Kay and Joseph Pasquale. "A Performance Analysis of TCP/IP and UDP/IP Networking Software for the DECstation 5000" *Tech Report, CSL U.C. San Diego/Sequoia*, December 1992.

[6] Jonathan Kay and Joseph Pasquale. "Measurement, Analysis, and Improvement of UDP/IP Throughput for the DECstation 5000" *Tech Report, CSL U.C. San Diego/Sequoia*, January 1993.

[7] V. Jacobson, R. Braden, and D. Borman. "TCP Extensions for High Performance." RFC 1323, LBL, USC/ISI, and Cray Research, May 1992.

[8] David B. Johnson and Willy Zwaenopoel. "The Peregrine High-Performance RPC System." To appear in *Software Practice and Experience.*

[9] Paul E. McKenney and Ken F. Dove. "Efficient Demultiplexing of Incoming TCP Packets." In *Proceedings of SIGCOMM '92*, Maryland, USA.

[10] John K. Ousterhout. "Why Aren't Operating Systems Getting Faster As Fast as Hardware?" In *Proceedings of the USENIX 1990 Summer Conference*, June 1990, pp. 247-256.

[11] M.D. Schroeder and M. Burrows. "Performance of Firefly RPC." *ACM Transactions on Computer Systems*, 8(1):1-17, February 1990.

[12] M. Schroeder, A. Birrell, M. Burrows, H. Murray, R. Needham, T. Rodeheffer, E. Satterthwaite, and C. Thacker. "Autonet: A High-Speed Self-Configuring Local Area Network Using Point-to-Point Links. *IEEE Journal on Selected Areas in Communications*, 9(8):1318-1335, October 1991.

[13] Cheng Song and Lawrence Landweber. "Optimizing Bulk Data Transfer Performance: A Packet Train Approach." In *Proceedings of SIGCOMM '88*, September 1988.

[14] Chandramohan Thekkath and Henry Levy. "Limits to Low-Latency Communication on High-Speed Networks." Technical Report 91-06-01, Department of Computer Science, University of Washington, June 1991.