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

Abstract

In this paper we characterize the latency of the BSD 4.4 alpha implementation 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 of TCP. We also characterize the impact on latency of some widely discussed improvements to TCP, such as header prediction and the combination of the checksum calculation with data copying.

1 Introduction

In this paper we investigate the latency characteristics of the TCP transport protocol on an Asynchronous Transfer Mode (ATM) network[6]. 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[6].
- The wide use of applications and subsystems (like RPC) that can benefit from reduced latency.

Prior studies have concentrated on characterizing and optimizing the throughput of TCP on substantially different hardware or networks than the ones we describe here (e.g., [3, 8]). In addition to focusing on an ATM network, we investigate how optimizations previously suggested for improving throughput affect latency. 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., [4])? 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

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

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.

transmit FIFO as soon as there is one complete cell in the FIFO. The ATM driver and adapter implement the Class 3/4 ATM Adaptation Layer (AAL), which is responsible for all segmentation and reassembly of datagrams and the detection of transmission errors and dropped cells.

We used the ULTRIX 4.2A kernel as a foundation and replaced its TCP implementation with the BSD 4.4 alpha TCP implementation. The BSD TCP implementation has lower latency than the ULTRIX implementation, in part because of its use of one fewer mbuf for small packets and its use of a protocol control block cache[1]. Since ULTRIX 4.2A is a BSD derivative, substituting the new BSD TCP implementation was relatively straightforward because the two implementations have nearly identical interfaces to the rest of the protocol stack.

1.2 Measurement Techniques

Many of our experiments measured the round-trip latency of two user-level processes roughly simulating client-server communication. Unless stated otherwise, the processes ran on otherwise idle machines and communicated over a switchless private ATM network. The client connected to the server using TCP, started a timer, and 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 its value. For all the round-trip measurements in this paper, we ran 40000 iterations for at least 3 repetitions and took the average to get the final result.

Our measurements used a range of packet sizes. Based upon previous studies of RPC and TCP traffic behavior, we chose a variety of packet lengths sized 500 bytes and smaller [1, 8]. We also measured packets of 1400 bytes (the Ethernet MTU minus protocol headers), 4000 bytes (fits on a single memory page including protocol headers), and 8000 bytes (fits on two memory pages including protocol headers, also close to our ATM MTU of 9K).

Latency measurements typically involve estimates of small code paths that take on the order of microseconds. To measure at this level of granularity, we used a real time clock on a TurboChannel card with a 40ns period[2]. One advantage of using this clock is that we avoid instruction counting as a technique for estimating the execution time of small code sections. One disadvantage of this approach, however, is that our measurements include cache effects.

The clock is initialized at boot time, and user-level processes gain access to 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 to measure events that started in user space and ended in the kernel, or vice-versa.

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 motivated by the results in Section 2. These modifications are not new and have been suggested by others to improve throughput[4, 9]. However, our focus here is on the effect of these modifications on latency.

2 Measurement of the Baseline System

The baseline system that we measured is the BSD 4.4 alpha TCP release operating on an ATM network. From our measurements, we investigate: (1) the contribution to latency of the network driver and adapter; (2) the cost of TCP protocol processing; and (3) the overhead of protocol-independent operating system mechanisms.

2.1 Effect of the Network on Latency

To demonstrate the effects of the network driver, adapter, and physical link 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 1. For the small transfer sizes, the network has a large effect on overall latency (e.g., a 919 μs difference in the 4 byte case). For large transfer sizes, much of the effect can be attributed to the lower bandwidth of the Ethernet driver, adapter, and physical link.

[1] A comparison of the two implementations can be found in University of Wash. CSE Dept. Tech. Report #93-03-02.

[2] The TurboChannel card is the AN-1 controller from DEC SRC[16]. Note that we did not employ the AN-1 network in this study, only the clock on its controller.

Size (bytes)	Round Trip Times (μs) Ethernet	ATM	Percentage Decrease (%)
4	1940	1021	47
20	2337	1039	55
80	2590	1289	50
200	2804	1520	45
500	4101	2140	47
1400	6554	2976	54
4000	13168	5891	55
8000	22141	10636	52

Table 1: Comparison of ATM versus Ethernet latencies.

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 measure both sides. The results for the transmit side are shown in Table 2, and the results for the receive side are shown in Table 3.

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	5.1	5.7	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 2: Breakdown of BSD 4.4 alpha Transmit Side Latency

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 copying data from user space into kernel mbufs at the socket layer.

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 Table 3 have similar meanings. The **User** time span refers 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 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 receiving and reassembling incoming ATM cells.

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 3: Breakdown of BSD 4.4 alpha Receive Side Latency.

We also introduced two more time spans on the input side. 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 it runs.

The nonlinear response of the ATM adapter, as observed in the **ATM** rows of the receive data, is due to overlap between sending the data and receive processing. When the sending ATM adapter is sending a large number of cells, the receiving ATM adapter can process the first cells while the sending adapter is still sending the later cells. We only measure the portion of the receive processing that actually contributes to the overall latency. This is the time from 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. 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.

The following subsections present an analysis of the data in these tables.

2.2.1 Mbuf Manipulation

One to eight mbufs are used for transfers of less than 1 KB. Beyond this size, cluster mbufs are used. Cluster mbufs are used for large transfers because they hold 4 KB of data, the size of a memory page, whereas normal mbufs hold only 108 bytes of data. 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.

The nonlinear response between the 500 and 1400 byte transfer sizes of the **User** and **mcopy** rows of Table 2 is due to a switch in the use of mbuf types in the ULTRIX 4.2A socket layer. Once the data transfer size grows above 1 KB, ULTRIX uses cluster mbufs to store user data.

In the **User** row, the copy from the user buffer to the mbuf takes less time because user data does not have to be fragmented into multiple mbufs.

In the **mcopy** row, using cluster mbufs reduces latency because the mbuf-to-mbuf copy semantics of cluster mbufs differs from normal mbufs. When normal mbufs are copied, the data is actually copied into separately allocated mbufs. However, cluster mbufs use reference counts for copying; no storage is allocated or data copied. Since TCP makes a copy of the mbufs passed from the socket layer on the transmit path, the copy for transfers larger than 1 KB takes less time than for smaller transfers.

We note, however, that these effects are artifacts of a particular buffer management implementation choice rather than inherent protocol behavior.

2.2.2 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 the cost of protocol processing. In a later section, we discuss optimizing the checksum for better latency.

2.2.3 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** row in Table 2 contains the time make copies of the data for retransmissions; and the **ATM** row includes the time spent copy data between the host and the device.

From this breakdown we see that data is copied at least twice on both sends and receives. The copy in **mcopy** only occurs on sends, and is made from the mbuf chain for retransmissions. Eliminating the checksum (discussed in Section 4.2) opens the possibility of eliminating these data copying costs given a network adapter that supports DMA. With a combined copy and TCP checksum, Clark et al. discuss a network adapter design that eliminates the need for a second copy[4]. In a later section, we investigate how combining a copy and checksum affects latency using the ATM adapter.

2.2.4 Scheduling

The scheduling times (the sum of the **IPQ** and **Wakeup** rows) for switching contexts on the receive side are noticeable for small data transfers (68 μs out of 1021 μs, or 6.7% of the round trip time for the 4 byte case), particularly when compared to the costs of mbuf allocation and deallocation. However, scheduling costs do not contribute greatly to the overall latency of transferring large messages.

2.3 Measurement Summary

The detailed measurements have shown the contributions to latency of the various layers used in TCP communication. For large packet sizes, most of the overall processing time is spent in data copies and the checksum calculation significantly. For small packet sizes, the scheduling time and the time to do the TCP processing become noticeable when compared with the overhead of mbuf allocation and deallocation. However, for large transfers, the checksumming and copying data operations dominate the round trip times.

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[4]. 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, 15], and is not discussed further here. The second technique involves exploiting traffic locality to predict the next incoming packet to avoid the protocol control block (PCB) lookup cost. Others have studied using traffic locality to improve throughput for bulk data transfer protocols [2, 17]; we study its impact on latency.

In the BSD implementation, the TCP input processing engine 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 alpha TCP also precomputes the values it expects to find in the next incoming packet header, and can then execute a faster processing path if the prediction is correct. This notion is similar to the RPC "fast path" found in high performance RPC systems such as SRC RPC[15].

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

Size (bytes)	Round Trip Times (μs) No Prediction	Round Trip Times (μs) Prediction	Percentage Decrease (%)
4	1110	1021	8
20	1127	1039	8
80	1324	1289	3
200	1560	1520	3
500	2186	2140	2
1400	2962	2976	0
4000	5950	5891	1
8000	11477	10636	7

Table 4

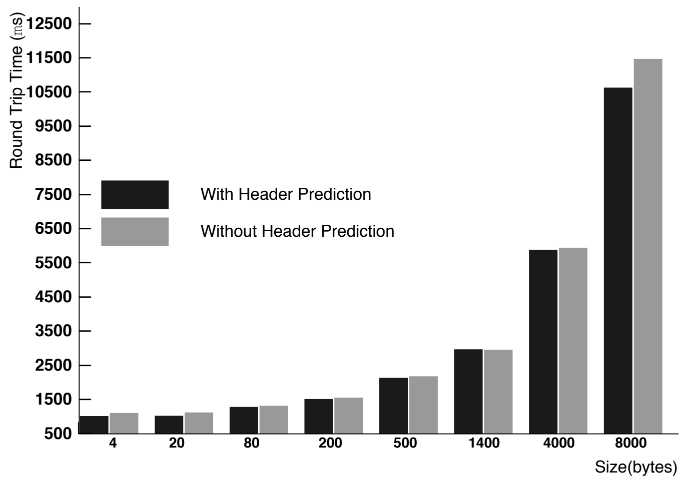

Figure 1

Effects of Header Prediction.

study alternative data structures for PCB lookup, and analyze these data structures by the expected average search length[12]. 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, ranging from 20 entries (26 μs) to 1000 entries (1280 μs), and found that the results scaled linearly. The cost per element on a DECstation 5000/200 is just less than 1.3μs. In addition, the typical number of active PCBs appears to be quite modest. For example, our departmental mail server has less than 250 active PCBs, and sampling thirty of our department workstations we found that all had less than 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 (i.e. the TCP fast path) 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 4 shows the results of this experiment, comparing a kernel with header prediction disabled to a kernel with it enabled. Figure 1 plots the same data graphically. For all the cases less than 8000 bytes, we notice only a very small improvement 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 (TCP fast path) fails in these cases (as explained below). In the 8000 byte case, the larger difference comes from the header precomputation and check succeeding for half the received packets, as well as the hit in the PCB cache. The savings from the PCB cache hit are not large because the number of PCBs is small (1.3 μs per PCB), 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 been optimized for.

To summarize our results concerning header prediction, we found that the PCB cache accounted for a only a small improvement in latency (about 4% on average), and that the current implementation of header precomputation does not improve latency in a bidirectional RPC style of communication.

4 TCP Checksums

From the breakdown of the latency costs above, we see that, for large transfers, the cost of calculating the TCP checksum is a significant portion of round trip latency. In this section we introduce an optimized checksum algorithm and then discuss the kernel implementation issues of combining the checksum with a data copy. We then address the issue of eliminating the checksum for particular combinations of link types and applications.

4.1 Optimizing the Checksum

Others have noted that the ULTRIX 4.2A checksum algorithm could be improved by eliminating halfword accesses and using loop unrolling[9]. We implemented a similar optimized checksum algorithm; the performance of this algorithm and the ULTRIX algorithm at user level are shown in Table 5.

An optimization suggested in [3, 4] combines the checksum calculation with one of the data copies to eliminate redundant movement of data over the memory bus. In ULTRIX 4.2A, data is copied at least twice on both send and receive in addition to calculating the TCP checksum. One copy moves the data between user and kernel space, and the other copy moves the data between kernel and device memory.

We combined our optimized checksum algorithm with a data copy at user level to investigate its potential performance benefits. The results are show in Table 5. The benefits are large: in the 8000 byte case, integrating the checksum and copy is 40% faster than performing the operations separately, and the effective bandwidth limitation imposed by the combined copy and checksum loop is just above 9 MB/s on the DECstation 5000/200.

The graph in Figure 2 shows the relative performance of the three methods for calculating the TCP checksum and copying the data.

We compare the performance of our implementation of the integrated checksum and copy with a user-level implementation on a Sun-3 described in [4]. Their measurements provide an interesting comparison of the scale in performance of a combined checksum and copy algorithm when changing hardware platforms. For example, with 1 KB of data they reported 130 μs to perform the checksum, and 140 μs to perform the memory to memory copy. The cost of their combined algorithm was 200 μs. On the DECstation 5000/200, our optimized checksum takes 96 μs to checksum 1 KB of data, and the copy takes 91 μs. The combined checksum and copy algorithm takes 111 μs. The savings from the combined algorithm on the Sun-3 is 35%, and on the DECstation 5000/200 is 68%. The overall improvement when switching from the Sun to the DECstation is 80%.

Size (bytes)	Copy and Checksum Measurements (μs) ULTRIX Checksum	ULTRIX bcopy	ULTRIX Total	Optimized Checksum	Integrated Copy and Checksum	Savings When Integrated (%)
4	5	4	9	3	3	57
20	7	5	12	4	5	44
80	20	11	31	9	10	50
200	43	20	63	21	24	41
500	104	47	151	49	56	42
1400	283	124	407	134	153	41
4000	807	350	1157	378	430	41
8000	1605	698	2303	754	864	40

Table 5

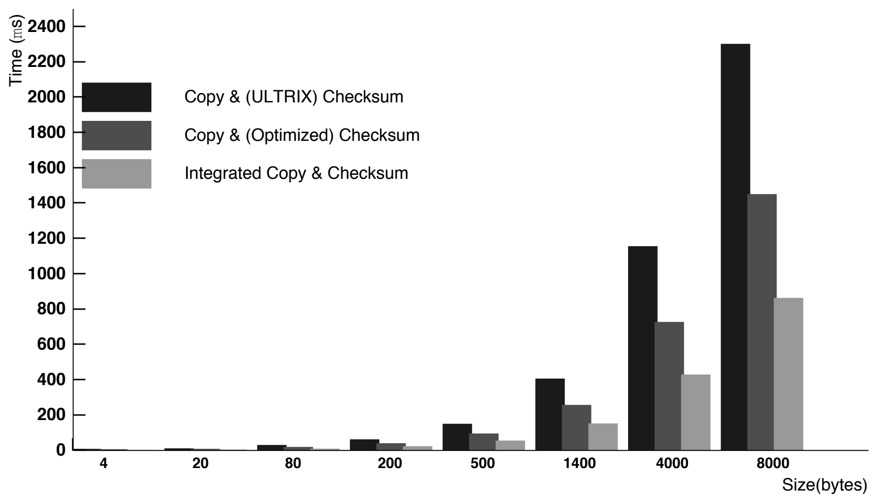

Figure 2

Copy and Checksum Measurements.

4.1.1 Kernel Implementation Issues

On the transmit side, the design of our ATM interface makes it impossible to defer the checksum calculation until the copy from kernel to device memory. 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; 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.

Instead, we chose to integrate calculating the checksum during the copy from user to kernel space. The TCP checksum has the convenient property that one can calculate the checksums for pieces of a packet and then combine those partial checksums later. We calculate the checksum for each chunk of data copied into an mbuf at the socket layer, and store the partial checksum in the mbuf header. As long as all of the data in the mbuf are transmitted in the same TCP segment, then the TCP layer will not have to recalculate the checksum for that data. In our implementation, the socket layer chooses the amount of data to place in each mbuf independent of the current TCP segment size. One possible improvement to this scheme would be for the socket layer to predict future TCP segment sizes based on recent behavior. Another alternative would be to split the data in an mbuf into smaller chunks and calculate more than one checksum per mbuf, thus increasing the chance that a chunk will be transmitted in a single segment.

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 have implemented the combined copy and checksum from the device memory to kernel memory. One disadvantage of this approach is that the device driver for each network interface needs to be modified to support this. Implementing the combined copy and checksum on the receive side is conceptually much simpler than on the send side because all the issues of mbufs and partial checksums disappear. However, the details of the implementation can be quite difficult and heavily dependent on the details of the device driver.

For comparison, the Digital OSF study also dicusses implementing a combined copy and checksum in the kernel[3]. It appears that their combined copy and checksum is only used on the receive side for incoming UDP packets. Also, their implementation combines the checksum with the copy from kernel to user space, rather than from device to kernel memory. This requires that the user enable this code with a socket option because, in the case where the checksum fails, they must overwrite the user buffer with zeros to clear out the data that was just copied.

Size (bytes)	**Round Trip Times (μs)**		
	Standard Checksum (μs)	**Combined Copy and Checksum (μs)**	**Percentage Saving (%)**
4	1021	1249	-22
20	1039	1256	-21
80	1289	1477	-15
200	1520	1707	-12
500	2140	2222	-3.8
1400	2976	2691	10
4000	5891	4644	21
8000	10636	8062	24

Table 6: Comparison of round trip latencies over ATM for the standard checksum and the combined copy and checksum calculation.

As an approximate measure of complexity, we added about 800 lines of code to implement the combined copy and checksum on both send and receive. The assembly language routines that implement the combined copy and checksum algorithm were less than half of the total number of lines of code, the rest was integration with the socket layer, the TCP layer, and the ATM driver.

The performance of our initial kernel implementation of the combined copy and checksum is shown in Figure 6. As expected, when the size of the data transfers increases, the combined checksum calculation provides significant savings in overall latency. In the 8000 byte case, the overall improvement has reached 24%. However, our initial implementation incurs significant costs in the smaller length cases, and the break-even point occurs somewhere between 500 and 1400 bytes.

4.2 Eliminating the TCP Checksum

The previous section has demonstrated that combining the checksum calculation with a data copy reduces latency. However, it is clear that latency can be further reduced by eliminating the checksum calculation altogether. It is already common practice to eliminate the UDP checksum for local area NFS traffic. Kay and Pasquale describe a mechanism using the Alternate Checksum Option to negotiate connections that do not use the checksum[8]. We therefore restrict ourselves to an analysis of the error characteristics and the implications of eliminating the checksum for local-area ATM traffic. We define local-area traffic as packets that go from source host to destination host without passing through any IP routers.

To measure the latency effect of eliminating the TCP checksum, we compare the round-trip times of the various packet sizes with and without the checksum calculation. Table 7 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 **Percentage Saving** is the relative saving when the checksum is eliminated. On the 4 byte case where the checksum overhead is minimal, nothing is gained. But, as the packet size increases, eliminating the TCP checksum significantly improves communication latency, e.g., the latency of the 8000 byte case is reduced by about 40%.

	Average ATM Round Trip Time Without Checksum		
Size (bytes)	Checksum (μs)	No Checksum (μs)	Percentage Saving (%)
4	1021	1020	0.1
20	1039	1020	1.8
80	1289	1233	4.3
200	1520	1392	8.4
500	2140	1808	16
1400	2976	2083	30
4000	5891	3633	38
8000	10636	6233	41

Table 7: Comparison of round trip latencies over ATM with and without the TCP checksum calculation.

With proper support from the host-network interface and the processor-memory subsystem, eliminating the TCP checksum can also benefit throughput oriented applications. For example, having DMA capability in the host-network interface and a snoopy cache as found in [5], allows data to be moved at near bus bandwidth speeds to the application layer. In contrast, as Section 4.1 indicates, even an integrated copy and checksum routine limits bandwidth to about 9% of the bus bandwidth on the DECstation 5000/200.

4.2.1 System Issues in Checksum Elimination

Eliminating the TCP checksum on local-area ATM networks is a delicate system design decision and we discuss some of the relevant system-level issues first before discussing its performance implications.

The "end-to-end argument", a classic principle in system design, says that the two ends of a reliable communication path should not depend on any of the intervening system components for correctness [14]. In other words, to assure the integrity of the communicated data, the communication end points must do a check independent of any checks done by intermediary components. If checks are done by intermediate or internal layers, they serve only as potential performance optimizations and do not subsume the end-to-end correctness check.

Intermediate checks can serve as performance optimizations, for example, by providing an inexpensive check that detects frequently occurring errors that would otherwise invoke a more expensive end-to-end recovery mechanism. On the other hand, an intermediate check can result in overall performance loss if, for example, it is expensive to perform and detects only infrequent errors; in such a case, even a very expensive end-to-end recovery could be preferable since the error seldom happens and the recovery cost can therefore be amortized.

In cases where TCP is used by a higher level service that performs its own checks, such as RPC systems that check their arguments, there is some debate on whether it is prudent to eliminate TCP checksums. The original environment that TCP was developed in used low-bandwidth links with little support for link-level error detection in hardware. Thus, the cost of detecting an error at the application layer impacted performance significantly. The use of the TCP checksum was therefore a performance optimization, consistent with the spirit of the end-to-end argument.

However, ATM networks with fiber optic links have very low error rates. For example, the bit error rate is on the order of 10^{-12} errors/s (i.e., one bit error in 3 hours if the network is used continuously at a bandwidth of 100 Mbits/s)[7]. Further, standard ATM adaptation layers (e.g., AAL3/4 and AAL5) specify *end-to-end* CRC checksums on the data, and host-network interfaces implement these in hardware. Thus, in cases where TCP is used as an intermediate layer to deliver data from an ATM network to a higher layer that will perform its own data integrity checks, eliminating the TCP checksum seems acceptable both on the grounds of performance and the end-to-end principle.

The main purpose of layering a TCP checksum over a link-level CRC is to potentially detect errors that the CRC does not catch. These errors can arise from four sources: (1) errors introduced by switches in transferring data between their input and output ports, (2) errors introduced by the network controllers in moving data between host and controller memories, (3) erroneous data injected into the network through external gateways or bridges, and (4) errors introduced by the link that are theoretically not detectable by the CRC because of the properties of the erroneous bit pattern and the CRC.

The first source of errors is not a problem since AAL payload checksums are end-to-end, i.e., intermediate switches do not recompute the checksum. The second type of errors is a potential problem, if for example, a buggy network controller introduces errors in transferring data between host and controller memory. We view this as a hardware problem in the controller that can be fixed using a hardware solution, e.g., incorporating parity or ECC into the controller memory.

The impact of the third type of errors can be eliminated by the application selectively using the TCP checksum elimination option only for local-area traffic. In fact, experiments conducted with and without wide-area traffic on our departmental Ethernet indicate that TCP detects two orders of magnitude fewer errors than the Ethernet CRC when wide-area traffic is included. Without wide-area traffic, TCP detected no checksum errors. We expect similar behavior on a local ATM network with quieter fibers. The quieter fiber also reduces the likelihood of errors from the fourth source.

We do not believe that eliminating the TCP transport entirely will be an option because the ATM network does not guarantee freedom from cell loss and some form of reliable transport mechanism will be required. The cost savings of optionally eliminating the TCP checksum, though, may be attractive to some applications. Other applications that are unable or unwilling to perform the necessary application-specific checks to guard against data corruption can continue to use the checksum at a cost.

4.2.2 Summary

To summarize, our measurements involving the TCP checksum suggest that there are definite performance advantages in making it optional. Further, for certain combinations of link types and applications, we believe it is possible to eliminate the TCP checksum without compromising error detection efficiency or violating established system design principles.

5 Conclusions

In this paper we have investigated the latency characteristics of TCP, and how optimizations originally proposed to improve throughput affect latency.

In previous experience with designing high performance “lightweight” RPC systems, we found that network driver and adapter design have a significant impact on performance[18]. For TCP, we also found that the network driver, adapter, and physical link have a significant impact on the latency.

In this paper, we first characterized the latency costs of TCP by breaking down round trip times for data transfers ranging in size from 4 bytes to 8000 bytes. We found that operating system services such as memory allocation contributed little to protocol processing time, particularly compared to the cost of scheduling. We also found that data-touching operations, such as copying and checksumming, dominate latency for transfers larger then 200 bytes.

We also found that header prediction had only a small impact on latency, primarily because the TCP fast path for input processing is not used for the round trip style of communication we measured.

We have found that computing the TCP checksum is a major cost of the overall TCP processing, and have characterized the effect of two optimizations to the checksum process. The first is a optimized implementation of the checksum algorithm that uses loop unrolling. The second optimization combines the optimized checksum calculation with copying the data. For large transfer sizes, the combined checksum and copy mechanism decreases round trip latency by as much as 24%.

In addition to optimizing the TCP checksum process, we have argued that, for particular combinations of link types and applications, it is possible to eliminate the TCP checksum calculation. Once the checksum is eliminated, round trip latency improves by as much as 41%.

6 Acknowledgements

We would like to gratefully acknowledge Ed Lazowska for his encouragement and comments on this project and report, as well as the helpful suggestions of the program committee. We also wish to thank Brian Bershad for his diligent shepherding that added greatly to the clarity of the paper.

References

[1] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. "Lightweight Remote Procedure Call." In *ACM Transactions on Computer Systems*,8(1):37–55, February, 1990.

[2] John B. Carter and Willy Zwaenepoel. "Optimistic Implementation of Bulk Data Transfer." In *Proceedings of the 1989 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1989, pp. 61–69.

[3] Chran-Ham Chang, Dick Flower, John Forecast, Heather Gray, Bill Hawe, Ashok Nadkarni, K. K. Ramakrishna, Uttam Shikarpur and Kathy Wilde. "High-Performance TCP/IP and UDP/IP Networking in DEC OSF/1 for Alpha AXP." *Digital Technical Journal*, Vol. 5 No. 1, Winter 1993, pp. 44–61.

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

[5] Digital Equipment Corporation, Maynard, MA. *Alpha Architecture Reference Manual*, 1992.

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

[7] Daniel H. Greene and J. Bryan Lyles. "Reliability of Adaptation Layers" *Protocols for High-Speed Networks, III*, Elsevier Science Publishers B.V., 1993, pp. 185–200.

[8] 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.

[9] 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.

[10] Van Jacobson, Robert Braden, and David Borman. "TCP Extensions for High Performance." RFC 1323, LBL, USC/ISI, and Cray Research, May 1992.

[11] David B. Johnson and Willy Zwaenepoel. The Peregrine high-performance RPC system. *Software – Practice and Experience*, 23(2):201–221, February 1993.

[12] Paul E. McKenney and Ken F. Dove. "Efficient Demultiplexing of Incoming TCP Packets." In *Proceedings of SIGCOMM '92*, August 1992, pp. 269–79.

[13] 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.

[14] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

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

[16] Michael D. Schroeder, Andrew D. Birrell, Michael Burrows, Hal Murray, Roger M. Needham, Thomas L. Rodeheffer, Edwin H. Satterthwaite, and Charles P. 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.

[17] Cheng Song and Lawrence Landweber. "Optimizing Bulk Data Transfer Performance: A Packet Train Approach." In *Proceedings of SIGCOMM '88*, September 1988, pp. 134–144.

[18] Chandramohan A. Thekkath and Henry M. Levy. "Limits to Low-Latency Communication on High-Speed Networks." *ACM Transactions on Computer Systems*,11(2):179–203, May 1993.

Alec Wolman is a graduate student at the University of Washington, currently on leave from Digital Equipment Corporation's Cambridge Research Lab. He holds an A.B. in Computer Science from Harvard University. His electronic mail address is wolman@cs.washington.edu.

Geoff Voelker is a graduate student at the University of Washington. He received the B.S. in Electrical Engineering and Computer Science from the University of California at Berkeley in 1992. His electronic mail address is voelker@cs.washington.edu.

Chandramohan A. Thekkath is a candidate for the Ph.D. degree in the Department of Computer Science & Engineering at the University of Washington. His electronic mail address is thekkath@cs.washington.edu.