Profiling and Reducing Processing Overheads in TCP/IP

Jonathan Kay and Joseph Pasquale

Department of Computer Science and Engineering
University of California, San Diego
San Diego, CA 92093-0114
{jkay, pasquale}@cs.ucsd.edu

Abstract

This paper presents detailed measurements of processing overheads for the Ultrix 4.2a implementation of TCP/IP network software running on a DECstation 5000/200. The performance results were used to uncover throughput and latency bottlenecks. We present a scheme for improving throughput when sending large messages by avoiding most checksum computations in a relatively safe manner. We also show that for the implementation we studied, reducing latency (when sending small messages) is a more difficult problem because processing overheads are spread over many operations; gaining a significant savings would require the optimization of many different mechanisms. This is especially important because, when processing a realistic workload, we have found that non-data-touching operations consume more time in aggregate than data-touching operations.

1.0 Introduction

We analyze TCP/IP [30] and UDP/IP [29] processing overheads given a real workload on a DECstation 5000/200 running Ultrix 4.2a, and we use this information to guide our development of new optimizations. The cost of various processing overheads depend on message size; consequently, our optimizations take into account the message size distributions derived from the network traffic in our environment (which is not atypical of many academic and office environments).

In our analysis of the TCP/IP and UDP/IP LAN and WAN traffic we were able to collect, we find that message sizes are far from uniformly distributed; rather, most messages are either very small or very large. Small messages are usually used to carry control information, whereas large messages typically carry bulk data. Different kinds of optimizations can improve processing speed for each type of traffic; in this paper we discuss both.

Typical processing time breakdowns for short (i.e. 64-128 byte) control messages fundamentally differ from those of long multiple kilobyte data messages. The processing time of large messages is dominated by *data-touching* operations such as copying and computing checksums [4, 8-11, 15, 24, 36] because these operations must be applied to each byte. However, small messages have few bytes of data, and thus their processing time is dominated by *non-data-touching* operations.

To optimize processing of large messages, we describe a checksum redundancy avoidance algorithm that eliminates most checksum processing without sacrificing reliability. Since checksum processing alone consumes nearly half the total processing time (of large messages), this optimization improves throughput considerably.

On both the LAN and WAN we studied, both of which are typical Unix-networking environments, small messages far outnumber large messages. In fact, even though processing a large message requires more time, the large proportion of small messages causes the cumulative non-data-touching processing time to exceed the cumulative data-touching processing time. We show that it would be difficult to significantly reduce the average processing time of non-data-touching overheads, at least for the implementation we studied, because that time is spread over many operations. Achieving a significant savings would require the optimization of many different mechanisms.

The paper is organized as follows. In Section 2 we describe the network traffic traces used to drive our analysis. In Section 3 we present our categorization of major network software processing overheads. The experimental set-up used to obtain our measurements is described in Section 4. Section 5 contains an analysis of network software processing overheads for a full range of message sizes. Section 6 discusses our algorithm for avoiding the computing of checksums when they are deemed redundant. In Section 7 we analyze aggregated processing overheads based on the measured message size distribution. Section 8 contains a finer analysis for each category of overheads, with an emphasis on the most time-consuming. In Section 9 we briefly discuss how our results may apply to other architectures. Finally, we present conclusions in Section 10.

2.0 Network Traffic Traces

To determine TCP and UDP message size distributions that are representative of real traffic, we obtained packet traces from two different FDDI networks, each used for a different purpose. The first trace, which reflects wider usage, is of traffic destined for and generated by a file server on an FDDI LAN of general-purpose Unix workstations in a university computer science department. We refer to this as the "LAN trace". The second trace, which is of wide-area network traffic, is obtained from the FDDI network that feeds the NSFnet backbone node at the San Diego Supercomputer Center (SDSC) (this is the same trace used in [7]). We refer to this second trace as the "WAN trace". The LAN and WAN traces were obtained in September of 1992 and March of

1993, respectively. Message sizes have a bimodal distribution in both traces. This and the other behaviors we observed conform to findings of earlier studies [5, 13, 21].

We define "message size" to mean the amount of user data actually sent, not including protocol headers. The LAN trace contains 139,720 messages. 90% are UDP messages, mostly generated by NFS [33]. The rest are TCP messages. The size distribution of the UDP messages is bimodal: 86% are less than 200 bytes long, while 9% are approximately 8 kilobytes long (the rest are in between). The TCP messages sizes are almost all small; over 99% are less than 200 bytes long.

The observed message size distributions matched our expectations based on previous work on Ethernet-based packet traces [13, 21]. The median message sizes for TCP and UDP messages are 32 and 128 bytes, respectively. The low UDP median reflects the fact that even in the case of UDP, most messages are small (e.g. NFS status messages). The reason for the large number of 8 kilobyte UDP messages is due to NFS. Given the scarcity of large TCP messages, we examined other LANs to check if this was simply an artifact of the network we measured, but we found very similar results. While there are applications that produce large TCP messages, such as X Window System and WWW image transfers and some Network News messages, these messages are infrequent relative to other traffic.

In the WAN trace, we found that most messages are small: 99.7% are no more than approximately 500 bytes long. Of the few large messages (of approximately 1500 bytes) that are present, they are mostly TCP messages probably generated by the few hosts that implement MTU Discovery [23]. Contrary to what we observed on our LAN, most (92%) of the WAN messages are sent using TCP. This is not surprising because TCP contains mechanisms for working smoothly over WANs (and is one reason why some vendors provide a port of NFS to use TCP over WANs).

Few hosts see a WAN-like distribution of traffic. Most messages are generated on a LAN and are destined for another host on the same LAN. However, for completeness, we used the WAN trace to determine the processing profile of hosts communicating over a wide-area network. Also, once again, our size distribution results are not surprising and consistent with [5].

3.0 Overhead Categories

We categorized the major processing overheads in network software as follows (the name abbreviations used in subsequent graphs are in parentheses): checksum computation ("Checksum"), data movement ("DataMove"), data structure manipulations ("Data Struct"), error checking ("ErrorChk"), network buffer management ("Mbuf"), operating system functions ("OpSys"), and protocol-specific processing ("ProtSpec"). Other studies have shown some of these overheads to be expensive [4, 8-11, 15, 36].

Checksum: Computing checksums is accomplished by a single procedure, the Internet checksum routine [2], which is performed on data in TCP and UDP, and on the header in IP.

DataMove: There are three operations contributing to data movement: copying data between user and kernel buffers (Usr-Krnl Cpy), copying data out to the FDDI controller (Device Copy), and cache coherency maintenance (Cache Clear).

DataStruct: These are operations that manipulate network data structures: the socket buffer (Socket Buffer), IP defragmentation queue (Defrag Queue), and interface queue (Device Queue) data structures. Mbuf manipulation is covered by its own category.

ErrorChk: The category of operations checks for user and system errors, such as parameter checking on socket system calls.

Mbuf: Network software subsystems generally require some type of buffer descriptor that allows headers to be prepended and messages to be defragmented efficiently. Berkeley Unix-based network subsystems buffer network data in a data structure called an *mbuf* [19]. All mbuf operations are part of this category. The allocating and freeing of mbufs are the most time-consuming mbuf operations.

OpSys: Operating system overhead includes support for sockets, synchronization overhead (sleep/wakeup), and other general operating system support functions.

ProtSpec: This category is of protocol-specific operations, such as setting header fields and maintaining protocol state, which are not included in any of the other categories. This category is a comparatively narrow definition of protocol-specific processing. For example, although computing checksums is a part of TCP, UDP, and IP, we categorize the checksum computation separately because of its high expense, and because it is not limited specifically to any one of these protocols.

Other: This final category of overhead includes all the operations that are too small to measure. An example of this is the symmetric multiprocessing (SMP) locking mechanism, which is called frequently in the DEC Ultrix 4.2a kernel. SMP locking executes hundreds of times per message, but each execution consumes less time than even the execution time of our probes. Thus we are unable to tell with certainty how much time is consumed by that mechanism. In the processing times we present, the time due to "Other" is the difference between the total processing time and the sum of the times of the categories listed above.

4.0 Experimental Setup

We instrumented the TCP/IP protocol stacks (including UDP/IP) in the DEC Ultrix 4.2a kernel. All measurements were taken on a DECstation 5000/200 workstation, a 19.5

SPECint MIPS RISC machine. We connected a HP 1652B Logic Analyzer to the DECstation I/O bus to obtain software processing time measurements with a resolution of 40 nanoseconds (the DECstation clock cycle time). We instrumented the kernel by placing C preprocessor macros at the beginning and end of the source code for each operation of interest. Each macro execution causes a pattern plus an event identifier to be sent over the DECstation I/O bus. We programmed the logic analyzer to recognize the pattern and store the event identifier, along with a timestamp. The measurement software causes minimal interference, generating overhead of less than 1% of the total network software processing time.

We measured the total message processing time for various components of network software when receiving and then sending the same-sized message. The experimental system, shown in Figure 1, consists of two workstations connected by an FDDI network with no other workstations and no network traffic other than that generated by the experiment. An experiment consists of one workstation sending a message to the *system under test*, which then sends the same message back. All measurements are made on the system under test, which is executing a probed kernel and is hooked up to the logic analyzer. Each experiment is carried out for 40 message sizes evenly spaced from 1 byte to 8192 bytes. Experiments are repeated at least 100 times at the same message size to obtain statistical significance in the results (the average percentage of standard error over all categories is less than 5%).

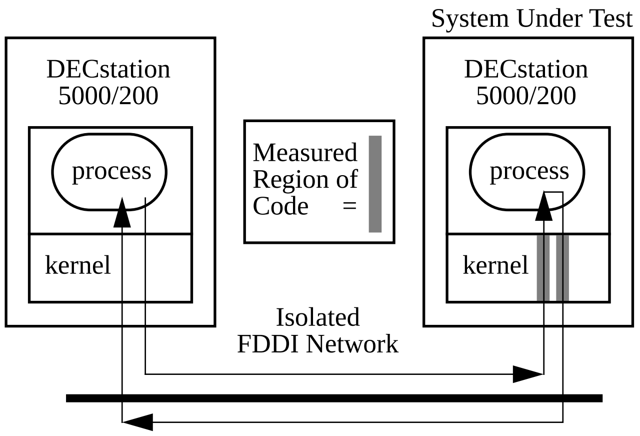
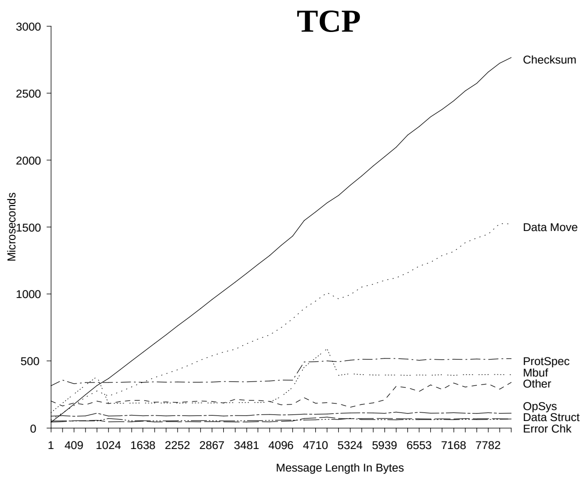


Figure 1: The experimental system consists of two DECstation 5000/200 workstations connected by an FDDI network.

The experiments are designed to capture only the CPU time spent processing network messages, ignoring other sources of delays in network communications. For example, our timings do not include network transmission times. Nor do we count time that a message is held up by flow or congestion control, except for the processing time needed to decide not to hold up the message; our workload does not provoke flow or congestion controls into operation. We note that the workload we generate has the result that TCP acknowledgments are always piggybacked.

5.0 Effects of Message Size on Processing Time Breakdowns

In this section we present results on processing overhead times by message size. These results provide a basis for understanding performance effects given a network workload. Figure 2a-b shows the per-message processing times versus message size for the various overheads for TCP and UDP messages, for a large range of message sizes, 1 to 8192 bytes. The processing time is the accrued time spent in the kernel (which includes the network protocols) when receiving and then sending the same-sized message.



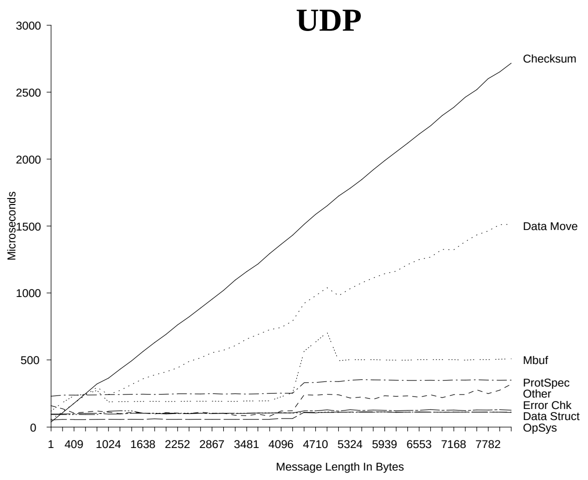


Figure 2a-b: Breakdown of operation processing times to receive and send TCP (2a) and UDP (2b) messages with sizes ranging from very small to very large, 1-8192 bytes.

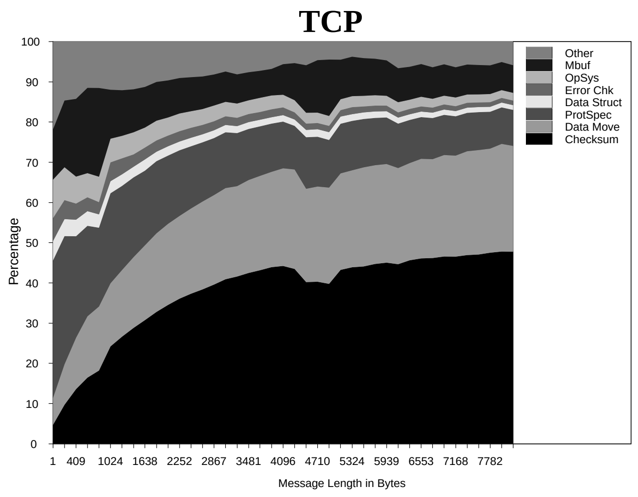
One can distinguish two different types of overheads: those due to *data-touching* operations (Data Move and Checksum) and those due to *non-data-touching* operations (all other categories). Data-touching overheads dominate the processing time for large data transfer messages, whereas non-data-touching operations dominate the processing time for small control transfer messages. This is because, generally, data-touching overhead times scale linearly with message size, whereas non-data-touching overhead times are relatively constant.

Actually, the observation that non-data-touching overheads are constant is simplistic; Figure 2a-b shows variations in non-data-touching processing times by message size. Fragmentation is the reason for the jump at 4312 bytes: FDDI cannot transmit packets larger than 4352 bytes, so TCP and IP fragment messages larger than 4312 bytes (4312 = 4352 - TCP header length - IP header length) into multiple packets. Fragmentation results in more non-data-touching processing time because it is necessary to execute device driver routines for each fragment.

Mbufs are another reason for the humps. Ultrix 4.2a network software (as well as other network software based on Berkeley UNIX) uses a data structure called mbufs for buffering. An mbuf is a 128-byte structure. In Ultrix on a DECstation 5000/200, it can either store up to 108 bytes of data or a pointer to a 4 kilobyte page if more space is needed. Mbufs containing a pointer to a page are called “cluster mbufs” and do not store any data in the original 128-byte data structure. All mbufs contain linked-list pointers so they can be connected into an “mbuf chain,” either to store messages larger than 4 kilobytes, to prepend protocol headers to mbuf chains, or to string fragments into a single chain. Allocation of mbufs to hold messages is done in a complex manner. To reduce internal fragmentation, the socket transmission code allocates a string of up to 10 small mbufs to hold messages no greater than 1 kilobytes long, calling a memory allocation routine for each mbuf. A message larger than 1024 bytes (up to 4 kilobytes) is held in a single cluster mbuf; this causes two calls to the memory allocator, one for the mbuf and another for the page. For a message larger than 4 kilobytes, the algorithm is repeated until enough mbufs have been allocated to hold the entire message. The bimodal operation of the mbuf allocation algorithm causes the hump between 1 byte and 1024 bytes and part of the hump between 4 kilobytes and 5 kilobytes, largely because of the resulting numbers of calls to the mbuf allocation routine. Others have also remarked on this bimodality [4, 13].

The breakdowns of TCP and UDP processing times shown in Figure 2a-b are very similar. Even TCP protocol-specific processing is only slightly more expensive than UDP protocol-specific processing. The differences are small because, even though TCP is the most complicated portion of the TCP/IP implementation, it is only a relatively small part of the executed layers of network software. Sending or receiving either a TCP or UDP message involves executing IP, the socket layer, the FDDI driver, and numerous support routines.

Figure 3a-b presents a different view of the data, showing the breakdown of processing overhead times expressed as cumulative percentages of the total processing overhead time. Notice that the lower two regions are due to the data-touching overheads; for large message sizes these operations consume approximately 70% of the total processing overhead time. However, as messages get smaller, the non-data-touching overhead times become more prominent, as expected. In fact, for single byte messages, data-touching overheads contribute only 11% of the total processing overhead time.



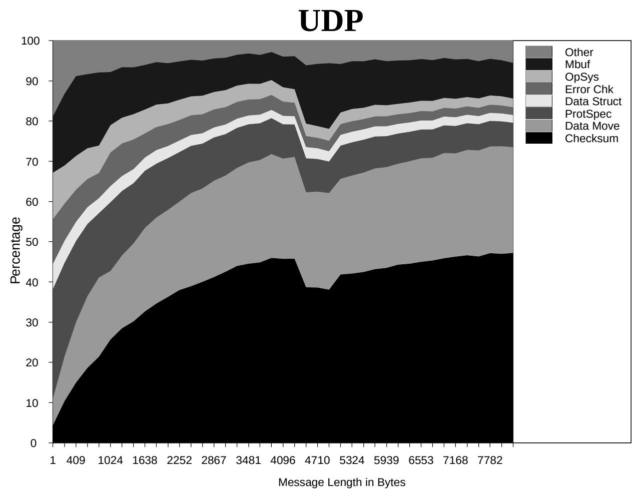
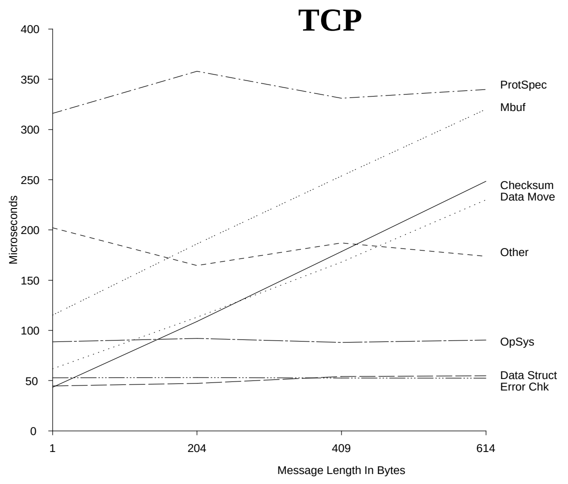


Figure 3a-b: Breakdown of operation processing times expressed as cumulative percentages of the total processing time for TCP (3a) and UDP (3b) messages.

In Figure 4a-b, we magnify the left-most regions of the graphs in Figure 2a-b to focus on small messages. The message sizes range from 1 to 614 bytes, which include the most common sizes of messages [5] sent over the Internet. For small messages, especially those smaller than 128 bytes (typical of remote procedure calls and TCP acknowledgments), the non-data-touching overheads clearly dominate.



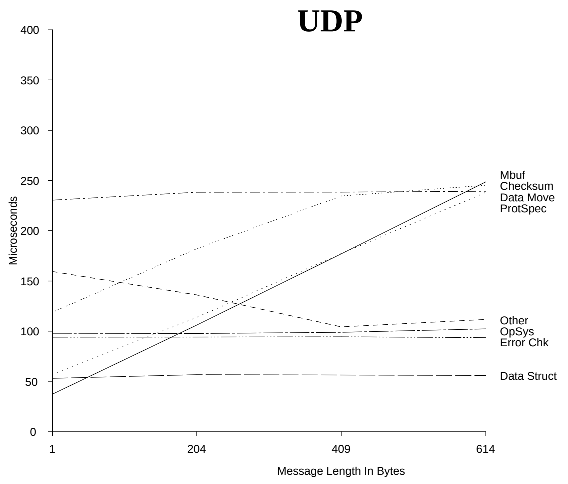


Figure 4a-b: Breakdown of operation processing times for small TCP (4a) and UDP (4b) messages.

6.0 Raising Throughput: Checksum Redundancy Avoidance

It is clear from Figures 2a-b and 3a-b that the largest bottleneck to achieving high throughput in this system is due to computing checksums. The simplest way to eliminate checksum overheads is to eliminate the checksum computation altogether. This is sufficient to produce a large reduction in processing time for large messages. Options already exist in the Ultrix-based networking code to disable the computing of checksums. However, the Internet checksum exists for a good reason: messages are occasionally corrupted during transmission, and the checksum is needed to detect corrupted data. Some vendors have disabled the checksum computation in UDP by default, risking a loss in packet integrity and in contravention of Internet host requirements [3].

6.1 Eliminating Checksum Redundancy

One can argue that computing checksums in software, to a large degree, is redundant. For example, Ethernet and FDDI networks implement their own 32-bit cyclic redundancy check. Thus, messages sent directly over an Ethernet or FDDI network are already protected from data corruption. In fact, there is evidence that the CRC is considerably stronger than the Internet checksum [25]. The high cost of the software checksum computation found in this study illustrates that this redundant, weaker checksum is expensive. *We suggest that in cases of complete redundancy the Internet checksum computation not be performed.* We devised a scheme, called Checksum Redundancy Avoidance (CRA), for avoiding most situations of computing checksums in software.

One must be careful, though, about deciding when the Internet checksum is redundant. We believe that it is reasonable to turn off checksums when crossing a single network which implements its own checksum in its hardware adapters. Since the destinations of most TCP and UDP messages are on the same LAN over which they are sent, this policy would eliminate computing checksums of most TCP and UDP messages.

Such a policy differs somewhat from traditional TCP/IP implementations in one aspect of protection against corruption. Always performing a software checksum in host memory protects against errors in data transfer over the I/O bus in addition to the protection between network interfaces given by the Ethernet and FDDI checksums. However, since data transfers over the I/O bus for such common devices such as disks are routinely assumed to be correct and are not checked in software, we argue that such a reduction in protection against I/O bus transfer errors for network devices is not unacceptable, especially in light of its high cost.

However, turning off checksum protection in any wider area context seems unwise without considerable justification. Not all networks are protected by checksums, and it is difficult to see how one might check that an entire routed path is protected by checksums without undue complications involving IP extensions. A more fundamental problem is that network checksums only protect a packet between network interfaces; errors may arise while a packet is in a router. Although the likelihood of such corruption is small for a single machine, the overall probability that some data corruption occurs somewhere over the end-to-end path increases with the number of routers a packet crosses according to 1 -

p^n (where p is the probability of corruption in a single router, n is the number of routers, and assuming independence).

6.2 UDP Implementation

The implementation of CRA in UDP uses the standard UDP checksum-nonuse protocol [29]: if the checksum field of an incoming message is zero, then it is assumed that there is no checksum on the message. All that remains is to decide on the details of when the sender should send a zero checksum field. Our implementation involves adding a flag to each network interface that supports some form of CRC. For Berkeley Unix, routing information is already available to UDP, and the route structure includes a flag indicating whether the next hop in the route is a router or the destination host. CRA checks whether the next hop is a destination, and whether the network interface that will transmit the packet supports some type of CRC. Computing checksums is skipped if both of conditions hold. We also implement a new socket option that allows applications to disable the checksum computation on a per-socket basis, even for sessions involving a large number of hops between hosts.

6.3 TCP Implementation

Extending TCP to implement checksum elimination is more difficult than for UDP since a checksum is always supposed to be applied to TCP data, and zero is a valid checksum value. An additional mechanism is required so that implementations can agree to disable checksums. An experimental alternate checksum option already exists [37]. The alternate checksum option provides a generic mechanism for TCP implementations to agree to use an alternate checksum. It is straightforward to use this option to define an alternate checksum value specifying no data checksum.

The scheme works as follows. The side that wishes to create a connection starts the negotiation. The same series of checks are used in TCP as in UDP to decide whether to send an option to turn off checksums. When a server receives a TCP connection request, it responds with the checksum-avoidance option if it gets a checksum-avoidance request in the connection-setup packet. Each side of a conversation only disables computing checksums if it receives a checksum elimination option from the other host in the conversation.

As with UDP, we added a socket option that allows the disabling of checksums even when the checksum is not redundant. Since the sender of the option makes the decision as to whether to negotiate the disabling of checksums, and receivers knowledgeable of this option automatically agree to requests for turning off checksums, only one side of a conversation needs to exercise the option.

6.4 Performance Improvement

Table 1 summarizes the throughput resulting from the improved checksum techniques (CRA+: CRA plus improvements to the checksum computation algorithm [18]). The measurements reflect absolute throughput for each configuration, and the percentage values in parenthesis are the percentage improvements relative to unmodified Ultrix 4.2a.

Table 1: CRA+ Throughput Improvements.

System	TCP	UDP
Ultrix 4.2a	17 Mbit/s	19 Mbit/s
With CRA+	25 Mbit/s (47%)	33 Mbit/s (74%)

6.5 Detecting Whether the Destination is on the Same LAN

The CRA scheme as described above is not always certain that a remote host is on the same network as the local host [31]. Bridging [14] and Proxy ARP [6] both make it difficult for a host to be certain that another host is actually on the same network. CRA works fine for most bridges because they forward frames using the CRC field that the frames were received with. However, a small number of bridges forward frames using a recalculated CRC field, which introduces an unknown likelihood of undetectable error arising during forwarding. Proxy ARP is a threat because it makes it possible for packets to cross a router without CRA (as described above) knowing about it.

We are investigating a strategy for deciding whether a remote host is on the same network as a local host. The key to the strategy is the IEEE spanning tree protocol used for routing among bridges [14]. Bridges are not allowed to pass packets using this protocol, and IP routers ignore such packets. Thus, reception of a spanning tree protocol packet from a host is good evidence that that host is on the same physical network as the local host.

Hosts implementing this scheme would broadcast a spanning tree packet right after responding to ARP requests. The contents of the spanning tree packets would be set such that no bridge will expect a host to actually bridge MAC packets.

This revised protocol is reasonably safe. Four conditions must apply to the same packet for a CRA-supporting host to fail to catch an error that would have been caught without CRA. A packet crossing a bridge or proxy router must become corrupted (with an error that the Internet checksum would catch) during forwarding, the bridge must recalculate CRCs, the bridge must indiscriminately broadcast spanning tree packets, and the packet must be one that would otherwise be considered local. This is a possible, but highly unlikely, scenario.

6.6 Related Work

There are a variety of other ways to reduce time spent computing checksums. One is to implement the Internet checksum on the network adapter [1, 17]. This approach requires extra hardware, and, unless the checksum is computed as the data is transferred to the adapter, it may not decrease latency.

Another scheme is to merge the copying and checksum operations to reduce transfers between memory and the CPU [8, 11, 26]. On some machines this strategy is sufficient to result in the effective elimination of checksum overhead [16]. The degree of improvement will depend on a number of factors related to CPU architecture and memory system performance among others.

The advantage of CRA is that it results in the elimination of checksum overheads for all classes of machines; it does not depend on special properties of adapters, CPU architecture, or memory system. Furthermore, it certainly does not prevent the use of the approaches above in addition to it to realize even greater performance improvements.

7.0 Latency Considerations: Aggregate Overhead Times

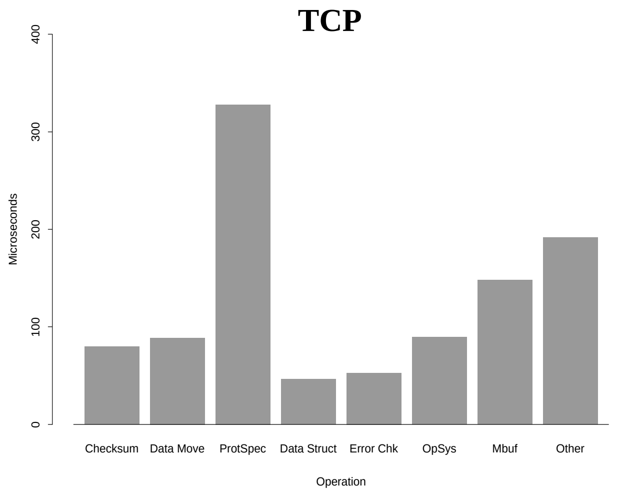
We know that:

- network software processing overhead times for small messages depend mainly on non-data-touching operations
- the distribution of message sizes differs by transport protocol (TCP or UDP) and environment (LAN or WAN)

Thus, the categories of processing overheads we defined in Section 3 have different relative costs depending upon protocol and environment because of the differing distributions of message sizes. We now consider the aggregate processing overhead times based on the message size distributions from our LAN and WAN traces. Figure 5a-b shows the TCP and UDP aggregate processing overhead times for the LAN trace, while Figure 6a-b shows the same for the WAN trace.

7.1 The Importance of Non-Data-Touching Overheads

Figure 5a shows that only a small fraction (16%) of the total time spent processing TCP messages on a LAN is due to the checksum computation and data movement. Representing 40% of the total processing time, computing checksums and moving data are significant factors for UDP on a LAN, as shown in Figure 5b, but they do not overwhelm the other categories of overhead. Consequently, the non-data-touching operations have a significant effect on performance.



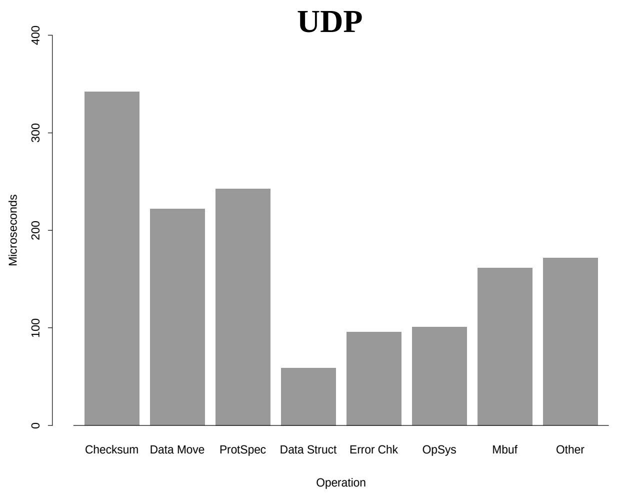
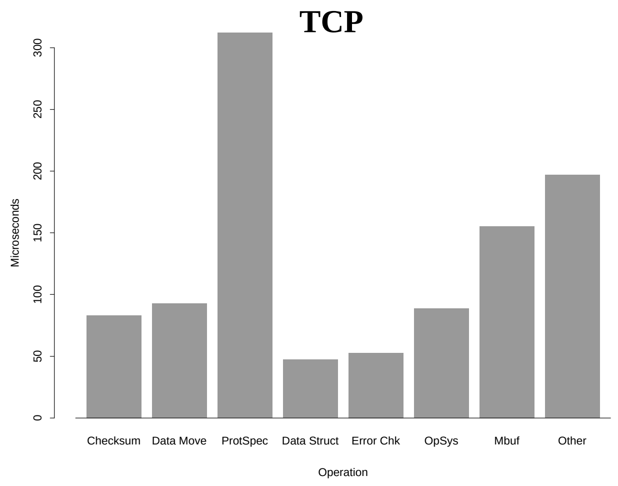


Figure 5a-b: Profiles of aggregate processing times for TCP (5a) and UDP (5b) operations in the LAN trace. For TCP, since most TCP messages are small, more time is spent on protocol overheads than on data movement. For UDP, since there are a number of large UDP messages, computing checksums and copying are large.

Because messages tend to be far smaller on WANs than on LANs, data-touching operations consume even less time: in the WAN trace (Figure 6a-b), 17% of the time required to process TCP messages is spent on data-touching operations, and 13% for UDP messages.



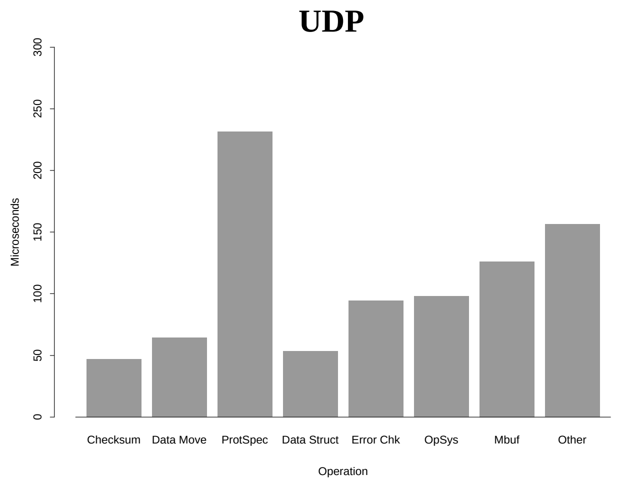


Figure 6a-b: Profiles of aggregate processing times for TCP (6a) and UDP (6b) operations in the WAN trace. Most of the packets in the traces are small, so it is not surprising that computing checksums and data copying consume relatively small times. Unlike the LAN trace where UDP traffic dominates, 92% of the packets in this trace are those of TCP.

In considering reasons for the large amount of overhead for non-data-touching operations, we do not believe that the implementation of the Berkeley Unix network subsystem (at our disposal) is poorly tuned. Observe that Berkeley Unix TCP/IP has rich functionality. Even the UDP/IP protocol stack has significant functionality, despite UDP's reputation as a minimal protocol: the UDP/IP protocol stack (in Berkeley Unix) has its own port space, checksum integrity, scalable internetworking, fragmentation, dynamic LAN address resolution, two-level input buffering and queuing, an error-checked system call interface to user programs, etc. Most of this functionality is due to layers other than UDP: IP, sockets, the link layer, and the device driver. Each item of functionality carries a cost, the sum of which is significant. Consider that the TCP portion of the ProtSpec (i.e. protocol-specific processing) category accounts for only 13% of the total processing time for single-byte messages (whose processing times are least affected by data-touching operations); all the other time is consumed by functionality provided by other layers and by the operating system.

Some researchers [32, 34, 35] have built specialized protocols with impressively low latency. These protocols achieve low latency by supporting some absolute minimal feature set needed by its application, resulting in far simpler code than TCP/IP.

7.2 Difficulty of Optimizing Non-Data-Touching Overheads

Improving overall performance by trying to optimize non-data-touching operations is difficult. For example, because of the relative scarcity of large TCP messages, the most prominent overhead category in the profile in Figure 5a is ProtSpec. ProtSpec consists of the protocol-specific processing overhead from each protocol layer (TCP, IP, Link layer, and FDDI driver). The largest component of ProtSpec is that of TCP, which as mentioned above consumes 13% of the total processing time; however, TCP protocol-specific processing is actually made up of a large number of smaller operations, as are the Mbuf and Other categories. Thus, a wide range of improvements would be needed to produce a significant improvement in performance for the non-data-touching operations.

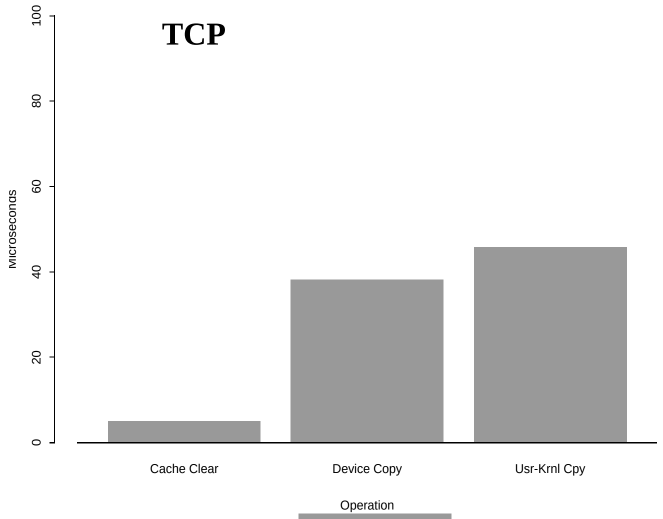
8.0 In-Depth Analysis

In Section 7 we presented an overview of the analysis of aggregate processing overhead times. In this section, we take a closer look at the LAN trace, analyzing the aggregate times for individual categories of processing overhead, in decreasing order of importance. Each category is examined in detail and the most time-consuming factors are explained. Differences between TCP and UDP time breakdowns are also explained. In general, the differences either result from differences between the distributions of message sizes or differences in TCP and UDP protocol complexity.

8.1 Touching Data

Checksum and DataMove are the operations that touch each byte of the message; thus, their aggregated operation times increase with message size. The data-dependent routines are the same for TCP and UDP.

As Figures 2a-b and 3a-b illustrate, Checksum is the dominant overhead for large messages. We see in Figure 5a-b that the aggregate time spent computing checksums is relatively low for TCP and relatively high for UDP. This is because in our LAN trace, almost all TCP messages are small whereas there is a significant fraction of large UDP messages. Since the Checksum category is itself a single operation, we do not analyze it any further.



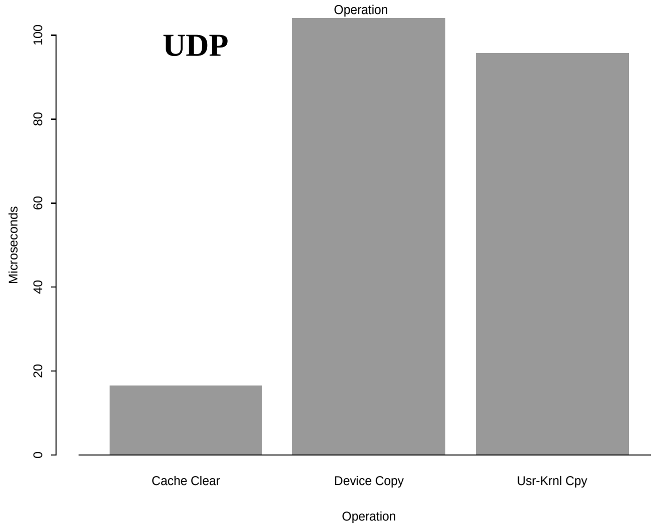


Figure 7a-b: Aggregate data movement times for TCP (7a) and UDP (7b) messages. Copying times are much higher for UDP because of the greater number of large UDP messages.

DataMove is the next-largest time-consuming category of overheads for large messages; Figure 7a-b shows a breakdown of this category into three operations: User-Kernel Copy, Device Copy, and Cache Clear. User-Kernel Copy is the amount of time spent copying incoming messages from kernel to user buffers and outgoing messages from user to kernel buffers. Device Copy and Cache Clear concern movement of data between kernel buffers and the FDDI controller. The controller we used for our measurements does not support send-side DMA, so the CPU must copy data to the controller. Device Copy is the amount of time needed to copy each message from kernel buffers to buffers in the FDDI controller. The controller does support receive-side DMA, so there is no receive-side equivalent to Device Copy. A cache coherency protocol is needed; Cache Clear is the amount of time used to insure cache consistency when the FDDI controller puts incoming packets into memory.

The amount of time spent copying data between the device and memory dominates the other times, with user-kernel copying time coming in a close second. In fact, Device Copy and Kernel Copy do roughly the same amount of work, but caching improves the latter's time.

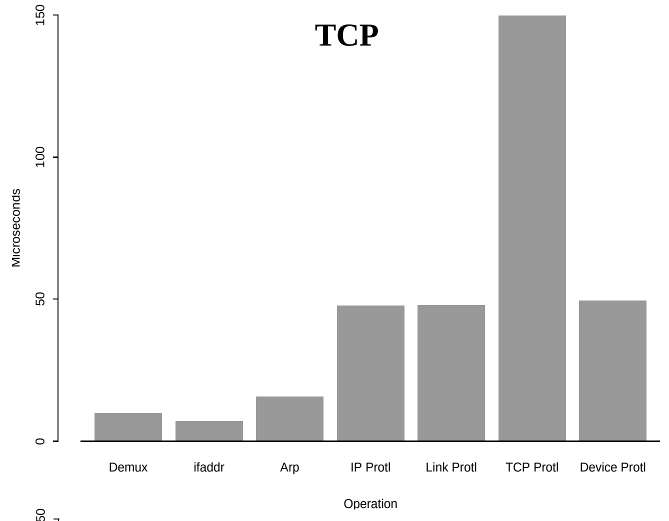
Since the time consumed by DataMove overheads increase with message size, the higher times for UDP reflect the higher average UDP message length. Another smaller source of difference between TCP and UDP is that Device Copy consumes less time than User-Kernel Copy in TCP, but the pattern is reversed for UDP. This is because each of the data-touching overhead times effectively has a constant component and a component rising linearly with message size. User-Kernel Copy has a larger constant component but a smaller linear component than Device Copy, and thus the reversal in relative sizes is due to the different TCP and UDP message size distributions.

8.2 Protocol-Specific Processing

As seen in Figure 5a-b, ProtSpec is the dominant category for TCP and is prominent for UDP. For TCP messages, ProtSpec consumes nearly half the total processing overhead time.

A breakdown of the ProtSpec category is shown in Figure 8a-b. Demux is the operation of finding a protocol control block given a TCP or UDP header (`in_pcblookup`), ifaddr is a set of operations to find and get information from local network interface adapters given specific IP addresses, and Arp is the Address Resolution Protocol [28]. IP Protl is the IP layer, Link Protl is part of the IEEE 802 encapsulation layer, TCP (UDP) Protl is the TCP (UDP) layer, and Device Protl is in the device driver layer. Finally, (Dis)connect (PCB) includes the operations for checking session information for a socket, setting up the protocol control block to reflect current connection state properly, and removing session information (`in_pcbconnect` and `in_pcbdisconnect`).

In Figure 8a, TCP Protl dominates the ProtSpec category. This is in contrast to UDP (Figure 8b), where UDP Protl is a rather small portion. However, despite TCP Protl's size, it only consumes 13% of the *total* network software processing time. Reducing the TCP protocol-specific processing time would be useful, but it is questionable whether the performance improvement would be worth the major effort required. A substantial reduction in time spent in the *entire* protocol-specific processing category would produce a more significant performance improvement, since the category as a whole consumes 32% for TCP and 18% for UDP. However, achieving such a reduction would require significant improvements across the entire stack of protocols.



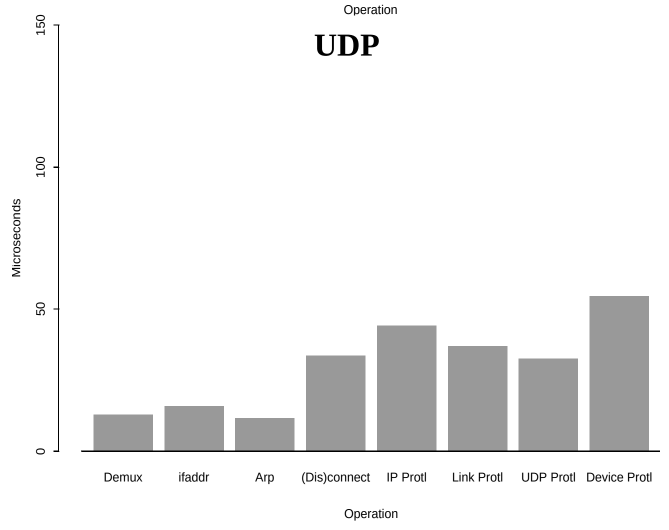
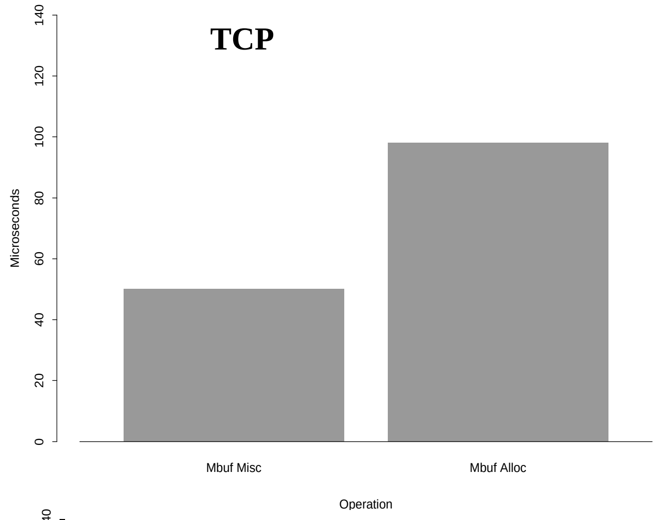


Figure 8a-b: Aggregate protocol processing (Protls) time for TCP (8a) and UDP (8b) messages. TCP protocol processing time is large, but less than half of total protocol processing time. UDP layer overheads are very small.

It is worth noting that Demux is small, under 20 microseconds, despite reports that this operation can be a bottleneck [10, 20]. This is because our single-user testing environment only has a single connection, and hence a single entry in the list of protocol control blocks.

8.3 Mbufs

Mbuf is the second largest of the non-data-touching categories of overhead. The mbuf data structure supports a number of operations, the most costly being Mbuf Alloc. Figure 9a-b shows breakdowns of the Mbuf category into mbuf allocation and deallocation (Mbuf Alloc) and all other mbuf operations (Mbuf Misc).



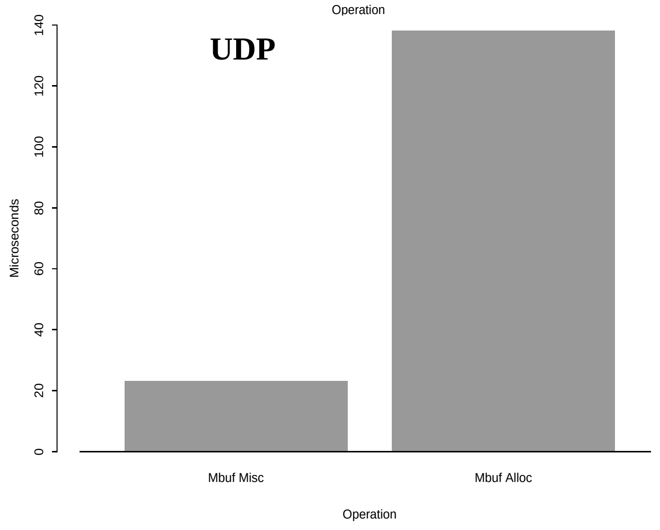


Figure 9a-b: Aggregate network buffer (i.e. "mbuf") management times for TCP (9a) and UDP (9b) messages. The large cost of the several memory allocation operations performed by the mbuf packages is one of the largest single operation costs.

Mbuf Alloc is more expensive for two reasons: memory allocation is an inherently expensive operation and is performed a number of times per message. Mbuf Alloc consumes more time for UDP because of the mbuf allocation strategy. In Ultrix, messages less than 1024 bytes long are stored in as many small buffers as necessary, each individually allocated to store at most 108 bytes. The overwhelming majority of TCP messages fit within a single mbuf, while a significant number of UDP messages are long enough to require at least two mbufs for data storage.

The other mbuf operations constituting Mbuf Misc are simpler operations such as copying mbuf chains, defragmentation (implemented by joining two linked lists of mbufs), and message length checking. TCP spends more time in Mbuf Misc than UDP because TCP must make a copy of each message in case retransmission is necessary.

8.4 Operating System Overheads

Figure 10a-b shows the breakdown of operating system overheads. Proc Restart is the time needed for a sleeping process to start running again. Socket Misc is miscellaneous processing in the socket layer. Sleep is the operating system procedure that a process uses to block itself. Wakeup is the call used to awaken a process. Softint Handler is the software interrupt handler that dequeues incoming packets and calls IP to handle them. Sched Soft Intr is the operation of scheduling a software interrupt to process an incoming packet. Perhaps the most interesting aspect of this category is that the various transfer-of-control operations are so cheap relative to the other overhead categories (see Figs. 5 and 6).

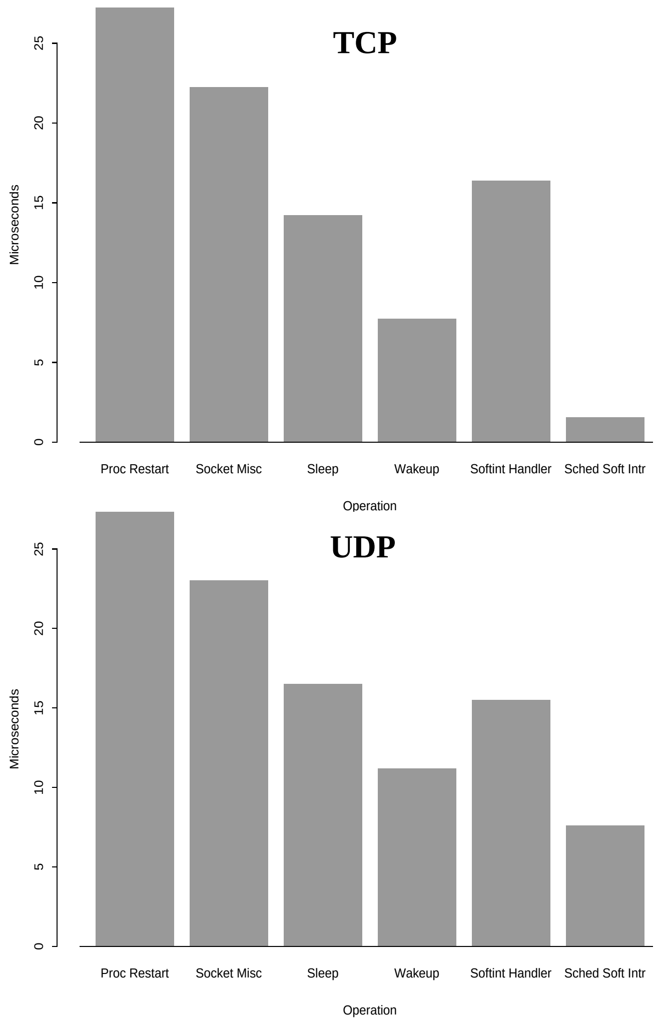


Figure 10a-b: Aggregate operating system overhead times for TCP (10a) and UDP (10b) messages. Transfer of control is surprisingly inexpensive.

8.5 Error Checking

ErrorChk is the category of checks for user and system errors. Figure 11a-b shows the breakdown of overheads for this category. Skt Errchk contains assorted checks for errors within the socket layer. Syscall Argchk is checking specifically for incorrect user arguments to the system calls used to send and receive messages. Interestingly, Syscall Argchk consumes a relatively large amount of time. This is mostly spent verifying that a user buffer is within a span of virtual memory accessible by the user process.

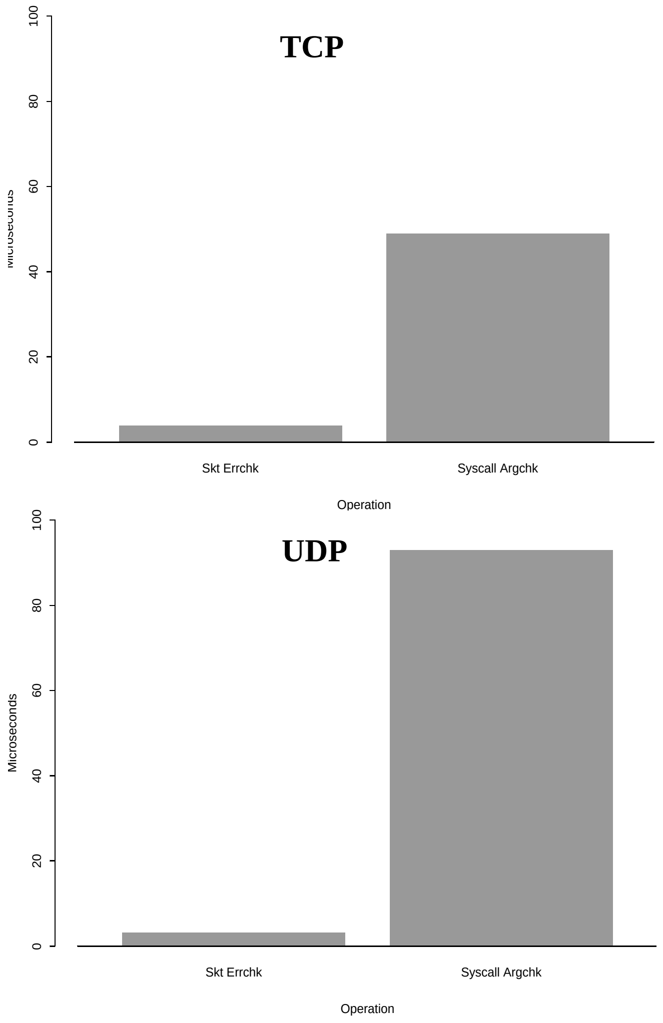


Figure 11a-b: Aggregate error checking times for TCP (11a) and UDP (11b) messages. Checking whether a correct receive buffer is specified is expensive and imposes a large penalty on large receive buffers.

8.6 Data Structure Manipulations

The DataStruct category consists of manipulations of various data structures whose cost do not justify individual scrutiny as did mbufs. Figure 12a-b shows the breakdown of overheads for this category. The socket buffer is the data structure in which a limited amount of data is enqueued either for or by the transport protocol. TCP makes heavier use of this data structure than UDP, which makes no use of the socket buffer structure for outgoing messages, and only uses it as a finite-length queue upon reception. In contrast, TCP uses the socket buffer to implement windowing flow control on both send and receive sides.

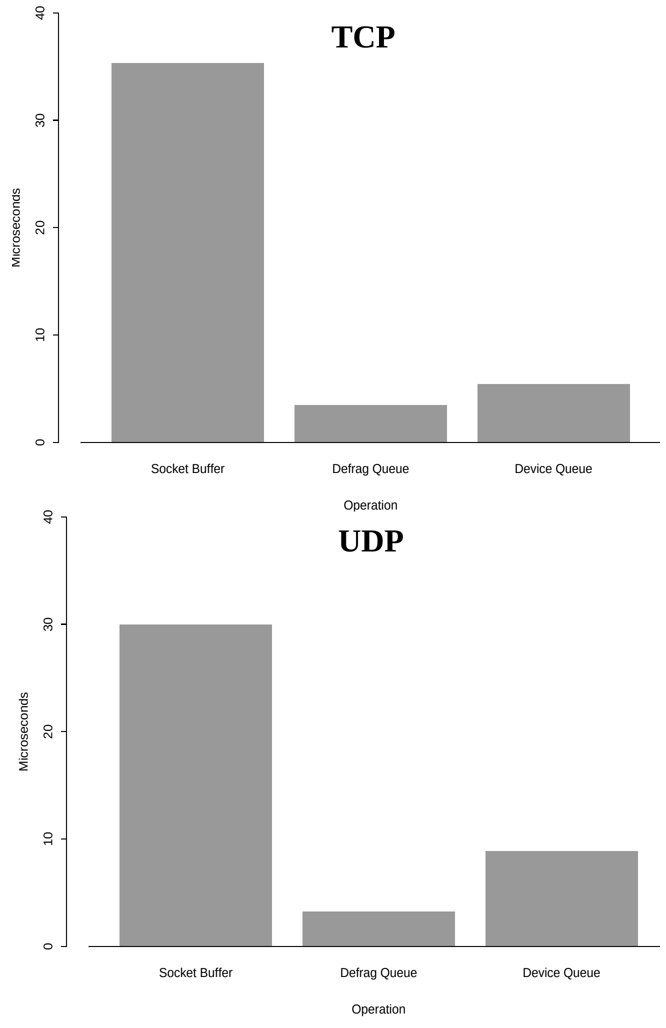


Figure 12a-b: Aggregate data structure manipulation times for TCP (12a) and UDP (12b) messages. TCP makes more extensive use of the socket buffer's properties than does UDP.

The Defrag Queue is the data structure used to defragment IP messages. The times for Defrag Queue processing require explanation. In general, messages larger than the FDDI MTU must be sent in multiple pieces (fragmented and defragmented). Fragmentation is implemented in both IP and TCP. UDP messages are fragmented by IP, but TCP does its own fragmentation specifically to avoid IP fragmentation. Thus, it is surprising that the code checking the defragmentation queue is called at all for TCP messages; this is an example of where a check for matching fragments could have been avoided in the IP code. Even more interesting is the fact that the average amount of time spent in Defrag Queue for UDP is not significantly greater than the amount of time spent in Defrag Queue by each unfragmented message. Less than 10% of UDP messages are large enough to be fragmented; the extra time spent defragmenting large messages is not sufficient to noticeably raise the average cost of defragmentation.

The Device Queue is the data structure in which outgoing data is enqueued by the link layer until the network controller is prepared to process it, and incoming data is enqueued by the device driver until IP is ready to process it in response to a software interrupt. UDP messages spend more time in Device Queue processing because of fragmentation. If a UDP message is fragmented into two FDDI packets, then twice as much work must be performed by the Device Queue to send that message.

9.0 Applicability of these Results to other Architectures

This study considered a single implementation of two related protocol stacks on a single processor architecture and operating system: TCP/IP and UDP/IP on the DECstation 5000/200 running Ultrix 4.2a. Nonetheless, we believe that many of this paper's major observations are applicable to a variety of other architectures and even different protocol suites. For example, it is well known that memory speeds are not improving as rapidly as processor speeds, making the reduction/elimination of data-touching operations important in all systems. At the very least, our observations can serve as a guide for potential bottlenecks to look out for in other systems.

It is important to note that many TCP/IP implementations are ports of the Berkeley Unix implementation. We know this to be true of SunOS until release 5, OSF/1, HPUX, IRIX, AIX, VxWorks, and of course the direct Berkeley Unix ports such as Ultrix itself, NetBSD, and BSDI. Even the SVR4 streams-based TCP/IP implementation is based on Berkeley Unix, although with significant rewriting to change the BSD facilities to streams-style facilities. Of course, these implementations have changed over time and are improved, in some cases significantly, over the original implementation from which they are derived. Yet, especially where software structure has not been radically changed, one can expect to find many of our observations applicable.

As an important example, there is good reason to believe that the significance of the non-data-touching overheads we observed is not simply a unique characteristic of our test environment. It takes considerable effort to minimize the number of control messages present in a distributed environment, whether they be RPC responses, acknowledgments, or information requests. This is reflected by the prevalence of NFS control messages in our LAN trace. There is no evidence that this is getting any better. Although a new version of NFS claims to reduce control messages [27], HTTP exhibits even worse properties in this regard than NFS [22].

10.0 Conclusions

We measured various categories of processing overhead times of the TCP/IP and UDP/IP protocol stacks on a DECstation 5000/200, and used the results to guide our search for bottlenecks in achieving high throughput and low latency. As predicted by others, the data-touching operations, computing checksums and copying, are the bottlenecks when trying to achieve high throughput. To improve throughput, we presented a scheme called Checksum Redundancy Avoidance which avoids the software computation of checksums when it would be redundant with the CRC computed by most LAN adapters.

Non-data-touching overheads are a bottleneck when trying to achieve low latency. Furthermore, because most messages observed in real networks are small, non-data-touching overheads consume a majority of the total software processing time. Unfortunately, unlike the breakdown for data-touching overheads which only consists of computing checksums and a small number of types of copying, time is evenly spread among the various types of non-data-touching overheads. Reducing a single non-data-touching overhead, such as TCP protocol-specific processing, will not have a major effect on overall performance improvement.

11.0 References

[1] D. Banks, M. Prudence, " A High-Performance Network Architecture for a PA-RISC Workstation," IEEE Journal on Selected Areas in Communications, pp. 191-202, February 1993.

[2] R. T. Braden, D. A. Borman, and C. Partridge, "Computing the Internet Checksum," Internet RFC 1071, September 1988.

[3] R. T. Braden, ed., "Requirements for Internet Hosts - Communication Layers," Internet RFC 1122, October 1989.

[4] L.-F. Cabrera, E. Hunter, M. J. Karels, D. A. Mosher, "User-Process Communication Performance in Networks of Computers," *IEEE Transactions on Software Engineering*, 14(1), 38-53, January 1988.

[5] R. Caceres, P. B. Danzig, S. Jamin, D. J. Mitzel. "Characteristics of Wide-Area TCP/IP Conversations," *Proceedings of the SIGCOMM '91 Symposium on Communications Architectures and Protocols*, pp. 101-112, August 1991.

[6] S. Carl-Mitchell, J. Quarterman, "Using ARP to implement transparent subnet gateways," Internet RFC 1027, October 1987.

[7] K. Claffy, "Internet Workload Characterization," Ph.D Thesis, University of California, San Diego, June 1994.

[8] D. D. Clark, "Modularity and Efficiency in Protocol Implementation," Internet RFC 817, 1982.

[9] D. D. Clark, "The Structuring of Systems Using Upcalls," *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pp. 171-180, December 1985.

[10] D. D. Clark, V. Jacobson, J. Romkey, H. Salwen, "An Analysis of TCP Processing Overhead," *IEEE Communications Magazine*, pp. 23-29, June 1989.

[11] D. D. Clark, D. L. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," *Proceedings of the SIGCOMM '90 Symposium on Communications Architectures and Protocols*, pp. 200-208, September 1990.

[12] P. Desnoyers, "Re: bypassing TCP checksum," comp.protocols.tcp-ip, Usenet, June 1993.

[13] R. Gusella, "A Measurement Study of Diskless Workstation Traffic on an Ethernet," *IEEE Transactions on Communications*, 38(9), pp. 1557-1568, September 1990.

[14] ANSI/IEEE, Std 802.1d, "Information Processing Systems - Local and Metropolitan Area Networks - Part 1d," ANSI/IEEE Std 802.1d, 1992.

[15] V. Jacobson, "BSD TCP Ethernet Throughput", comp.protocols.tcp-ip, Usenet, 1988.

[16] V. Jacobson, "Some Design Issues for High-speed Networks," *Networkshop '93*, November 1993.

[17] H. Kanakia, D. R. Cheriton, "The VMP Network Adapter Board (NAB): High-Performance Network Communication for Multiprocessors," *Proceedings of the SIGCOMM '88 Symposium on Communications Architectures and Principles*, pp. 175-187, 1988.

[18] J. Kay, J. Pasquale, "Measurement, Analysis, and Improvement of UDP/IP Throughput for the DECsta-

tion 5000," *Proceedings of the Winter 1993 USENIX Conference*, pp. 249-258, January 1993.

[19] S. J. Leffler, M. K. Mckusick, M. J. Karels, J. S. Quarterman, *The Design and Implementation of the 4.3 BSD UNIX Operating System*, Addison-Wesley, November 1989.

[20] P. E. McKenney, K. F. Dove, "Efficient Demultiplexing of Incoming TCP Packets," *Proceedings of the SIGCOMM '92 Symposium on Communications Architectures and Protocols*, pp. 269-279, August 1992.

[21] J. Mogul., "Network Locality at the Scale of Processes," *Proceedings of the SIGCOMM '91 Symposium on Communications Architectures and Protocols*, pp. 273-284, August 1991.

[22] J. Mogul, "The Case for Persistent-Connection HTTP," *Proceedings of the ACM SIGCOMM '95 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 299-313, September 1995.

[23] J. Mogul, S. Deering, "Path MTU Discovery," Internet RFC 1191, November 1990.

[24] J. Ousterhout, "Why Aren't Operating Systems Getting Faster As Fast As Hardware," *Proceedings of the Summer 1990 USENIX Conference*, pp. 247-256, June 1990.

[25] C. Partridge, J. Hughes, J. Stone, "Performance of Checksums and CRCs over Real Data", *Proceedings of the ACM SIGCOMM '95 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 68-76, August 1995.

[26] C. Partridge, S. Pink, " A Faster UDP," IEEE/ACM Transactions on Networking, pp. 429-440, August 1993.

[27] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, "NFS Version 3 Design and Implementation," *Proceedings of the Summer 1994 USENIX Conference*, June 1994.

[28] D. C. Plummer, "Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware," Internet RFC 826, November 1982.

[29] J. Postel, "User Datagram Protocol," Internet RFC 768, August 1980.

[30] J. Postel, "Transmission Control Protocol," Internet RFC 793, September 1981.

[31] V. J. Schryver, "Re: bypassing TCP checksum," comp.protocols.tcp-ip, Usenet, June 1993.

[32] A. Z. Spector, "Performing Remote Operations Efficiently on a Local Computer Network," *Communications of the ACM*, pp. 246-260, April 1982.

[33] Sun Microsystems, Inc., "NFS: Network File System Protocol Specification," Internet RFC 1094, March 1989.

[34] C. A. Thekkath, H. M. Levy, "Limits to Low-Latency Communication on High-Speed Networks," ACM Transactions on Computer Systems, pp. 179-203, May 1993.

[35] T. von Eicken, D. Culler, S. Goldstein, K. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

[36] R. W. Watson, S. A. Mamrak, "Gaining Efficiency in Transport Services by Appropriate Design and Implementation Choices," *ACM Transactions on Computer Systems*, 5(2), pp. 97-120, May 1987.

[37] J. Zweig, C. Partridge, "TCP Alternate Checksum Options," Internet RFC 1146, March 1991.