Experimental Evaluation of SunOS IPC and TCP/IP Protocol Implementation[1]

Christos Papadopoulos
christos@dworkin.wustl.edu
(314) 935-4163

Gurudatta M. Parulkar
guru@flora.wustl.edu
(314) 935-4621

Computer and Communications Research Center
Department of Computer Science
Washington University
St. Louis MO 63130-4899

Abstract

Progress in the field of high speed networking and distributed applications has led to a debate in the research community on the suitability of existing protocols such as TCP/IP for emerging applications over high-speed networks. Protocols have to operate in a complex environment comprised of various operating systems, host architectures, and a rapidly growing and evolving internet of several heterogeneous subnetworks. Thus, evaluation of protocols is definitely a challenging task that cannot be achieved by studying protocols in isolation. This paper presents results of a study which attempts to characterize the performance of the SunOS Inter-Process Communication (IPC) and TCP/IP protocols for distributed, high-bandwidth applications.

1. Introduction

Considerable research and development efforts are being spent in the design and deployment of high speed networks. These efforts suggest that networks supporting data rates of a few hundreds of Mbps will become available soon. Target applications for these networks include distributed computing involving remote visualization and collaborative multimedia, medical imaging, teleconferencing, video distribution, and other demanding applications. Progress in high speed networking suggests that the raw data rates will be available to support such applications. However, these applications require not only high speed networks but also carefully engineered end-to-end protocols implemented efficiently within the constraints of various operating systems and host architectures. There has been considerable debate in the research community regarding suitability of existing protocols such as TCP/IP [3,9,10] for emerging applications over high speed networks. One group of researchers believe that existing protocols such as TCP/IP are suitable and can be adopted for use in high speed environments [4,7]. Another group claims that the TCP/IP protocols are complex and their control mechanisms are not suitable for high speed networks and applications [1,2,12,13]. It is important, however, to note that both groups agree that efficient protocol implementation and appropriate operating system support are essential.

Inter-Process Communication (IPC) which includes protocols, is quite complex. The underlying communication substrate, for example, is a constantly evolving internet of many heterogeneous networks with varying capabilities and performance. Moreover, protocols often interact with operating systems which are also complex and have additional interacting components such as memory management, interrupt processing, process scheduling and others. Furthermore, the performance of protocol implementations may be affected by the underlying host architecture. Thus, evaluation of IPC models and protocols such as TCP/IP is definitely a challenging task and cannot be achieved by studying protocols in isolation. This paper presents the results of a study aimed at characterizing the performance of TCP/IP protocols in the existing IPC implementation in SunOS for high bandwidth applications. Components to be studied include the control mechanisms (such as flow and error control), per-packet processing, buffer management and interaction with the operating system by systematic measurement.

The software examined is the SunOS 4.0.3 IPC using BSD stream sockets on top of TCP/IP. SunOS 4.0.3 is based on 4.3 BSD Unix (for further details on BSD Unix IPC implementation, see [11]). The hardware were two Sun

[1]This work was supported in part by the National Science Foundation and an industrial consortium of Bellcore, BNR, DEC, Italtel SIT, NEC, NTT, and SynOptics.

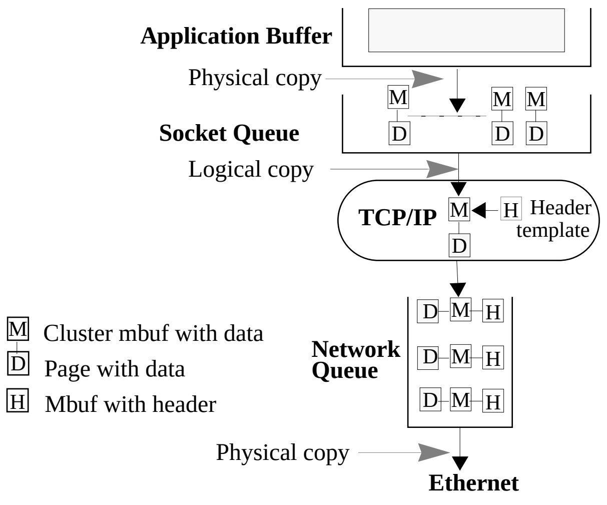

Figure 1: Data distribution into Mbufs

Sparcstation 1 workstations connected on the same Ethernet segment via the AMD Am7990 LANCE Ethernet Controller. Occasionally two Sparcstation 2 workstations running SunOS 4.1 were also used in the experiments. However, only SunOS 4.0.3 IPC could be studied in depth due to the lack of source code for SunOS 4.1.

2. Unix Inter-Process Communication (IPC)

In 4.3 BSD Unix, IPC is organized into 3 layers. The first layer, the socket layer, is the IPC interface to applications and supports different types of sockets each type providing different communication semantics (for example, *STREAM or DATAGRAM* sockets). The second layer is the protocol layer, which contains protocols supporting the different types of sockets. These protocols are grouped in domains, for example the TCP and IP protocols are part of the Internet domain. The third layer is the network interface layer, which contains the hardware device drivers (for example the Ethernet device driver). Each socket has bounded send and receive buffers associated with it, which are used to hold data for transmission to, or data received from another process. These buffers reside in kernel space, and for flexibility and performance reasons they are not contiguous. The special requirements of interprocess communication and network protocols require fast allocation and deallocation of both fixed and variable size memory blocks. Therefore, IPC in BSD 4.3 uses a memory management scheme based on data structures called MBUFs (Memory BUFfers). Mbufs are fixed size memory blocks 128 bytes long that can store up to 112 bytes of data. A variant mbuf, called a cluster mbuf is also available, in which data is stored externally in a page 1024 bytes long, associated with each cluster mbuf. All data to be transmitted is copied from user space into mbufs, which are linked together if necessary to form a chain. All further protocol processing is performed on mbufs.

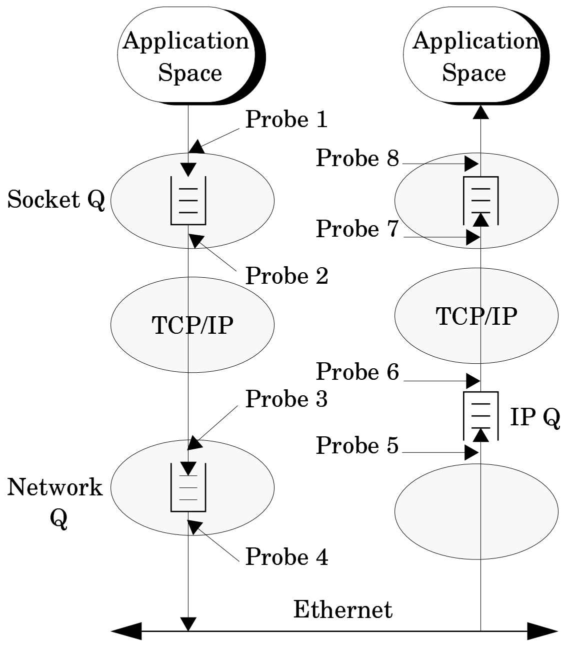

Figure 2: Queue and Probe Locations

2.1 Queueing Model

Figure 1 shows the data distribution into mbufs at the sending side. In the beginning, data resides in application space in contiguous buffer space. Then data is copied into mbufs in response to a user send request, and queued in the socket buffer until the protocol is ready to transmit it. In the case of TCP or any other protocol providing reliable delivery, data is queued in this buffer until acknowledged. The protocol retrieves data equal to the size of available buffering minus the unacknowledged data, breaks the data in packets, and after adding the appropriate headers passes packets to the network interface for transmission. Packets in the network interface are queued until the driver is able to transmit them. On the receiving side, after packets arrive at the network interface, the Ethernet header is stripped and the remaining packet is appended to the protocol receive queue. The protocol, notified via a software interrupt, wakes up and processes packets, passing them to higher layer protocols if necessary. The top layer protocol after processing the packets, appends them to the receive socket queue and wakes up the application. The four queueing points identified above are depicted in Figure 2. A more comprehensive discussion of IPC and the queueing model can be found in [8].

3. Probe Design

To monitor the activity of each queue, probes were inserted in the SunOS Unix network code at the locations shown in Figure 2. A probe is a small code segment placed at strategic locations in the kernel. Each probe when activated records a timestamp and the amount of data passing through its checkpoint. It also fills a field to identify itself. The information recorded is minimal, but can nevertheless provide valuable insight into queue behavior. For example, the timestamp differentials can provide queue delay, queue length, arrival rate and departure rate. The data length can provide throughput measurements, and help identify where and how data is fragmented into packets.

At the sending side, probes 1 and 2 monitor the sending activity at the socket layer. The records produced by probe 2 alone, show how data is broken up into transmission requests. The records produced by probes 1 and 2 together, can be used to plot queue delay and queue length graphs for the socket queue. Probe 3 monitors the rate packets reach the interface queue. This rate is essentially the rate packets are processed by the protocol. Probe 3 monitors the windowing and congestion avoidance mechanisms by recording the bursts of packets produced by the protocol. The protocol processing delay can be obtained by the interpacket gap during a burst. The interface queue delay is given as the difference in timestamps between probes 3 and 4. The queue length can also be obtained from probes 3 and 4, as the difference in packets logged by each probe.

At the receiving side, probes 5 and 6 monitor the IP queue, measuring queue delay and length. The rate at which packets are removed from this queue is a measure of how fast the protocol layer can process packets. The rate packets arrive from the Ethernet, is strongly dependent on Ethernet load, can be determined using probe 5. The difference in timestamps between probes 6 and 7 give the protocol processing delay. Probes 7 and 8 monitor the socket queue delay and length. Packets arrive in this queue after the protocol has processed them and depart from the queue as they are copied to the application space.

The probes can also monitor acknowledgments. If the data flow is in one direction, the packets received by the sender of data will be acknowledgments. Each application has all 8 probes running, monitoring both incoming and outgoing packets. Therefore, in one-way communication probes at the lower layers record acknowledgments.

3.1 Probe Overhead

An issue of vital importance is that probes incur as little overhead as possible. There are three sources of potential overhead: (1) due to probe execution time; (2) due to the recording of measurements; and (3) due to probe activation (i.e. deciding when to log).

To address the first source of overhead, any kind of processing in the probes besides logging a few important parameters was precluded. To address the second source, it was decided that probes should store records in the kernel virtual space in a static circular list of linked records, initialized during system start-up. The records are accessed via global head and tail pointers. A user program extracts the data and resets the list pointers at the end of each experiment. The third source of overhead, probe activation, was addressed by introducing a new socket option that the socket and protocol layer probes can readily examine. For the network layer probes, a bit in the “tos” (type of service) field of the IP header in the outgoing packets was set. The network layer is able to access this field easily because the IP header is always contained in the first mbuf of the chain either handed down by the protocol, or up by the driver. This is a non-standard approach, and it does violate layering, but it has the advantage of incurring minimum overhead and is straightforward to incorporate.

4. Performance of IPC

This section presents some experiments aimed at characterizing performance of various components of IPC, including the underlying TCP/IP protocols. To minimize interference from users and the network, the experiments were performed with one user on the machines (but still in multi-user mode), and the Ethernet was monitored using either Sun’s *traffic* or *xenetload*, which are tools capable of displaying the Ethernet utilization. With the aid of these tools it was ensured that background Ethernet traffic was low (less than 5%) during the experiments. Moreover, the experiments were repeated several times in order to reduce the degree of randomness inherent to experiments of this nature.

4.1 Experiment 1: Throughput

This experiment has two parts. In the first, the effect of the socket buffer size on throughput is measured when setting up a unidirectional connection and sending a large amount of data (about 10 Mbytes or more) over the Ethernet. The results show that throughput increases as socket buffers are increased, with rates up to 7.5 Mbps achievable when the buffers are set to their maximum size (approximately 51 kilobytes). This suggests that it is beneficial to increase the default socket buffer size (which is 4 kilobytes) for applications running on the Ethernet, and by extrapolation, for applications that will be running on faster networks in the future. The largest increase in throughput was achieved by quadrupling the socket buffers (to 16 kilobytes), followed by some smaller subsequent improvements as the buffer got larger.

In the second part of the experiment, the results of the first part are compared to the results obtained when the two processes exchanging data reside on the same machine. This setup bypasses completely the network layer, with packets from the protocol output looped back to the protocol input queue. Despite the fact that a single machine handles both transmission and reception, the best observed local IPC throughput was close to 9 Mbps, suggesting that the mechanism is capable of exceeding the Ethernet rate. However, as the previous part showed, the best observed throughput across the Ethernet was only 7.5 Mbps, which is only 75% of the theoretical Ethernet throughput. The bottleneck was traced to the network driver which could not sustain rates higher than 7.5 Mbps. The actual results of this experiment can be found in [8].

Increasing the socket buffer size can be beneficial in other ways too: the minimum of the socket buffer size is the maximum window TCP can use. Therefore, with the default size the window cannot exceed 4096 bytes (which corresponds to 4 packets). On the receiving end the protocol can receive only four packets with every window which leads to a kind of stop-and-wait protocol[2] with four segments. Another potential problem is that the use of small buffers leads to a significant increase in acknowledgment traffic, since at least one acknowledgment for every window is required. The number of acknowledgments generated with 4K buffers was measured using the probes, and was found to be roughly double the number with maximum size buffers. Thus, more protocol processing is required to process the extra acknowledgments, possibly contributing to the lower performance obtained with the default socket buffer size. In light of the above observations, all subsequent experiments were performed with the maximum allowable socket buffer size.

4.2 Experiment 2: Investigating IPC with Probes

For this experiment the developed probing mechanism is used to get a better understanding of the behavior of IPC. The experiment has two parts: the first consists of setting up a connection and sending data as fast as possible in one direction. The data size used was 256 KB. In the second part, a unidirectional connection is set up again, but now packets are sent one at a time in order to isolate and measure the packet delay at each layer.

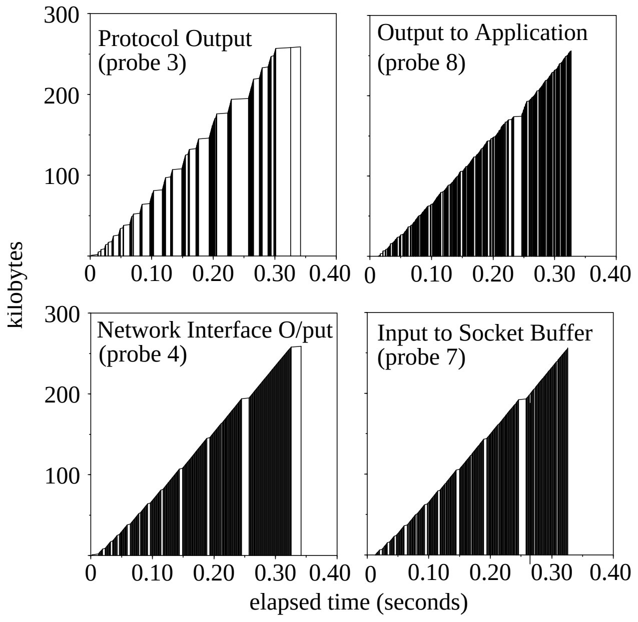

Figure 3: Packet Trace

For the first part, the graphs presented were selected to show patterns that were consistent during the measurements. Only experiments during which there were no retransmissions were considered, in order to study the protocols at their best. The results are presented in four sets of graphs: the first is a packet trace of the connection; the second is the length of the four queues vs. elapsed time; the third is the delay in the queues vs. elapsed time; and the last is the protocol processing delay.

The graphs in Figure 3 show the packet traces as given by probes 3, 4, 7 and 8. Probes 3 and 4 are on the sending side. The first probe monitors the protocol output to the network interface and the second monitors the network driver output to the Ethernet. Probes 7 and 8 are on the receiving side and monitor the receiving socket input and output respectively. The graphs show elapsed time on the x-axis and the packet number on the y-axis. The packet number can also be thought as a measure of the packet sequence number since for this experiment all packets carried 1 kilobyte of data. Each packet is represented by a vertical bar to provide a visual indication of packet activity. For the sender the clock starts ticking when the first SYN packet is transmitted, and for the receiver when this SYN packet is received. The offset introduced is less than 1 ms and does not affect the results in a significant manner.

[2]In a stop-and-wait protocol the transmitter sends a packet of data and waits for an acknowledgment before sending the next packet.

Examining the first graph (protocol output) in the set reveals the transmission burstiness introduced by the windowing mechanism. The blank areas in the graph represent periods the protocol is idle awaiting acknowledgment of outstanding data. The dark areas represent the period the protocol is sending out a new window of data (the dark areas contain a much higher concentration of packets). The last two isolated packets are the FIN and FIN_ACK packets that close the connection. The second graph (interface output) is a trace of packets flowing to the Ethernet. It is immediately apparent that there is a degree of "smoothing" to the flow as packets move from the protocol out to the network. Even though the protocol produces bursts, when packets reach the Ethernet these bursts are dampened considerably, resulting in a much lower rate of packets on the Ethernet. The reason is that the protocol is able to produce packets faster than the network can transmit, which leads to queuing at the interface queue.

On the receiving side, the trace of packets into the receive socket buffer appears to be similar to the trace produced by the sender, meaning that there is no significant queueing delay introduced by the network or the IP queue. This also indicates that the receive side of the protocol is able to process packets at the rate they arrive, which is not surprising since this rate was reduced by the network (to approximately a packet every millisecond). This lower rate is possibly a key reason why the IP queue remains small. The output of the receive socket appears similar to the input to the socket buffer. Note however, that there is a noticeable idle period, which seems to propagate back to the other traces. The source for this is explained after examining the queue length graphs, next.

The set of graphs in Figure 4 shows the queue length at the various queues as a function of time (note the difference in scale of the y-axis). The queue length is shown as the number of bytes in the queue instead of packets for the sake of consistency, since in the socket layer at the sending side there is no real division of data into packets. Data is fragmented into packets at the protocol layer. The sending socket queue length is shown to monotonically decrease as expected, but not at a constant rate since the draining of the queue is controlled by the reception of acknowledgments. Note that the point the queue appears to become empty is the point where the last chunk of data is copied into the socket buffer and not when all the data was actually transmitted. The data remain in the socket buffer until acknowledged. The interface queue shows the packet generation by the protocol. The burstiness in packet generation is immediately apparent and manifested as peaks in the queue length. The queue build-up suggested by the packet trace

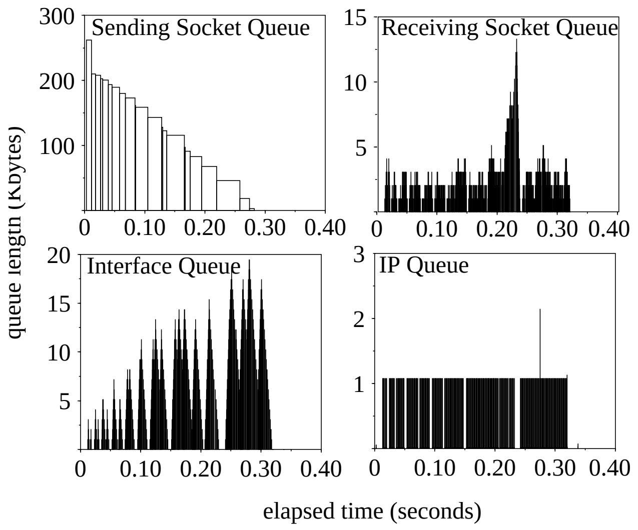

Figure 4: **Queue Length during congestion window opening**

graphs is clearly visible. Also, the queue length increases with every new burst, indicating that each new burst contains more packets than the previous one. This is attributed to the slow start strategy of the congestion control mechanism [5]. The third graph shows the IP receive queue, and confirms the earlier hypothesis that the receive side of the protocol can comfortably process packets at the rate delivered by the network. As the graph shows the queue practically never builds up, each packet being removed and processed before the next one arrives. The graph for the receive socket queue shows some queueing activity. It appears that there are durations over which data accumulates in the socket buffer, followed by a complete drain of the buffer. During that time the transmitter does not send any data, since no acknowledgment is sent by the receiver. This queue build-up leads to gaps in packet transmission, as witnessed earlier by the packet trace plots. These gaps are a result of the receiver delaying the acknowledgment until data is removed from the socket buffer.

The next set of graphs in Figure 5 shows the packet delay at the four queues. The x-axis shows the elapsed time and the y-axis the delay. Each packet is again represented as a vertical bar. The position of the bar on the x-axis shows the time the packet entered the queue. The height of the bar shows the delay the packet experienced in the queue. As mentioned earlier, there are no packets in the first queue, therefore the first (socket queue) graph simply shows the time it takes for data in each write call to pass from the application layer down to the protocol layer. For this experiment, there is a single write call, so there is only one entry in the graph. The second graph shows the network interface queue delay. Packets entering an empty queue at the begin-

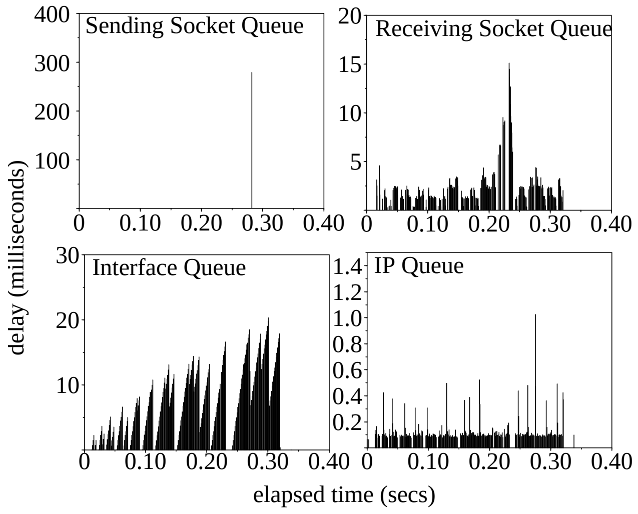

Figure 5: **Queue Delay during congestion window opening**

ning of a window burst experience a very small delay, while subsequent packets are queued and thus delayed. The delays range from a few hundred microseconds to several milliseconds. The third graph shows the IP queue delay, which is a measure of how fast the protocol layer can remove and process packets from its input queue. The delay of packets appears very small, with the majority of packets remaining in the queue for about 100 microseconds. Most of the packets are removed from the queue well before the 1 ms interval that it takes for the next packet to arrive. The fourth graph shows the delay at the receive socket queue. Here the delays are much higher. This delay includes the time to copy data from mbufs to user space. Since the receiving process is constantly trying to read new data, this graph is an indication of how often the operating system allows the socket receive process which copies data to user space to run. The fact that there are occasions where data accumulates in the buffer, shows that the process runs at a frequency less than the data arrival.

The two histograms in Figure 6 show the protocol processing delay at the send and receive side respectively. For the sending side the delay is assumed to be the packet interspersing during a window burst. The reason is that since the protocol fragments transmission requests larger than the maximum protocol segment, there are no well-defined boundaries as to where the protocol processing begins for a packet and where it completes. Therefore, packet interspersing was deemed more fair to the protocol (the gaps introduced by the windowing mechanism were removed). This problem does not appear at the receiving end where the protocol receives and forwards distinct packets to the socket layer. The x-axis in the graphs is divided into bins

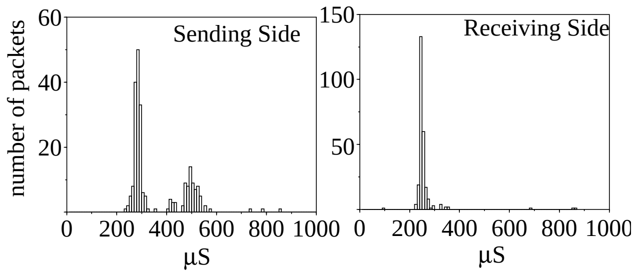

Figure 6: **Protocol Processing Delay Histograms**

which are 50 microseconds wide, and the y-axis shows the number of packets in each bin. Examining the graphs shows that the protocol delays are different at the two ends: at the sending end processing delay divides the packets into two distinct groups with the first, the larger one, requiring about 250 to 400 microseconds to process and the second about 500 - 650 microseconds. At the receiving end the packet delay is more evenly distributed, and processing takes about 250 - 300 microseconds for most packets, with some taking between 300 and 400. Thus, the receiving side is able to process packets faster than the sending side. The average delays for protocol processing are 370 microseconds for the sending side and 260 microseconds for the receiving side. Thus the theoretical maximum TCP/IP can attain on Sparcstation 1's when the CPU is devoted to communication with no data copying (but including checksum) and without the network driver overhead is estimated to be about 22 Mbps.

In the second part of the experiment the delay experienced by a single packet moving through the IPC layers is isolated and measured. A problem with the previous measurements is that the individual contribution of each layer to the overall delay is hard to assess because layers have to share the CPU for their processing. Moreover, asynchronous interrupts due to packet reception or transmission may steal CPU cycles from higher layers. To isolate layer processing consecutive packets are sent after introducing some artificial delay between them. The idea is to give the IPC mechanism enough time to send the current packet and receive an acknowledgment before supplied with the next.

The experiment was set up as follows: the pair of Sparcstation 1's was used again for unidirectional communication. The sending application was modified to send one 1024 byte packet and then sleep for one second to allow the packet to reach the destination and the acknowledgment to come back. To ensure that the packet was sent immediately, the TCP_NODELAY option was set. Moreover, both sending and receiving socket buffers were set to 1 kilobyte, thus effectively reducing TCP to a stop-and-wait protocol.

The results of this experiment are summarized in Table 1, and are compared to the results obtained in the previous experiment, when 250 kilobytes of data were transmitted. The send socket queue delay shows that a delay of about 280 μS is experienced by each packet before reaching the protocol. This includes data copy, which was measured to be about 130 μS. Therefore, the socket layer processing takes about 150 μS. This includes locking the buffer, masking network device interrupts, performing various checks, allocating plain and cluster mbufs and calling the protocol. The delay for the sending side has increased by about 73 μS (443 vs. 370 μS). This is due to the fact that the work required before calling the TCP output function with transmission request is replicated with each new packet. Earlier, the TCP output function was called with a large transmission request, and entered a loop forming packets and transmitting them until the data was exhausted. The interface queue delay is very small: it takes about 40 μS for a packet to be removed from the interface queue by the Ethernet driver when the latter is idle. The IP queue delay is also quite small. The delay has not changed significantly from the earlier experiment, when 256 kilobytes of data were transferred. The protocol layer processing at the receiving end has not changed much from the result obtained in the first part. In this part the delay is 253 μS, while in part 1 it was 260 μS. The receive socket queue delay is about 412 μS, out of which 130 will be due to data copy. This means that the per packet overhead is about 292 μS which is about double the overhead at the sending side. The main source of this appears to be the application wake-up delay.

Table 1: Delay in IPC Components

Location	Average delay stop-and-wait transfer (μS)	Average delay with continuous transfer (μS)
Send socket (with copy)	280	-[a]
Protocol - sending side	443	370
Interface queue	40	-[a]
IP queue	113	124
Protocol - receiving side	253	260
Receive socket queue (with copy)	412	-[a]
Byte-copy delay[b] (for 1024 bytes)	130	130

a. varies with data size

b. measured using custom software

4.3 Experiment 3: Effect of Queueing on the Protocol

The previous experiments have provided insight into the queueing behavior of the IPC mechanism. The experiments established that the TCP/IP layer in SunOS 4.0.3 running on a Sparcstation 1 has the potential of exceeding the Ethernet rate. As a result the queue length graphs show that noticeable queueing exists at the sending side (at the network interface queue) which limits the rate achieved by the protocol. At the receiving end, queueing at the IP queue is low compared to the other queues because the packet arrival rate is reduced by the Ethernet. However, queueing is observed at the socket layer, showing that packet arrival rate is higher than the rate packets are processed by IPC. In this experiment the effect of the receive socket queueing on the protocol was investigated.

Briefly, the actions taken at the socket layer for receiving data are as follows: the receive function enters a loop traversing the mbuf chain and copying data to the application space. The loop exits when either there is no more data left or the application request is satisfied. At the end of the copy-loop the protocol user request function is called so that protocol specific actions like sending an acknowledgment can take place. Thus any delay introduced at the socket queue will delay protocol actions. TCP sends acknowledgments during normal operation when the user removes data from the socket buffer. During normal data transfer (no retransmissions) an acknowledgment is sent if the socket buffer is emptied after removing at least 2 maximum segments, or whenever a window update would advance the sender's window by at least 35 percent[3].

The TCP delayed acknowledgment mechanism may also generate acknowledgments. This mechanism works as follows: upon reception of a segment, TCP sets the flag TF_DELACK in the transmission control block for that connection. Every 200 mS a timer process runs checking these flags for all connections. For each connection that has the TF_DELACK flag set, the timer routine changes it to TF_ACKNOW and the TCP output function is called to send an acknowledgment. TCP uses this mechanism in an effort to minimize both the network traffic and the sender processing of acknowledgments. The mechanism achieves this by delaying the acknowledgment of received data in hope that the receiver will reply to the sender soon and the acknowledgment will be piggybacked onto the reply segment back to the sender.

[3]As per the tcp_output () of SunOS. See [8] for details.

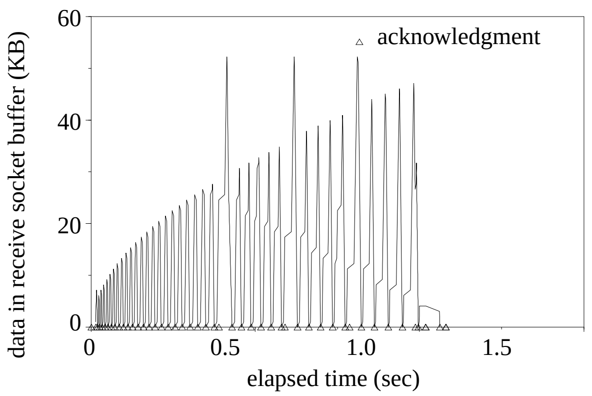

Figure 7: Receive socket queue length and acknowledgment generation

To summarize, if there are no retransmissions, TCP may acknowledge data in two ways: (1) whenever (enough) data are removed from the receive socket buffer and the TCP output function is called, and (2) when the delayed ack timer process runs.

To verify the above, the following experiment was performed: a unidirectional connection was set up again, but in order to isolate the protocol from the Ethernet the connection was made between processes on a single machine (i.e. using local communication). The data size sent was set to 1 Mbyte, and the probes were used to monitor the receive socket queue and the generated acknowledgments. The result is shown in Figure 7, which shows the queue length v.s. elapsed time. The expansion of the congestion window can be seen very clearly with each new burst. Each window burst is exactly one segment larger than the previous one. Moreover, the socket queue grows by exactly the window size, and then it drops down to zero, where an acknowledgment is generated and a new window of data comes in shortly after. The peaks that appear in the graph are caused by the delayed acknowledgment timer process that runs every 200 mS. The effect of this timer is to send an acknowledgment before the data in the receive buffer is completely removed, causing more data to arrive before the buffer is emptied. However, during data transfer between different machines more acknowledgments would be generated because the receive buffer becomes empty more often since the receive process is scheduled more often and has more CPU cycles to copy the data.

The experiment shows how processing at the socket layer may slow down the TCP by affecting the acknowledgment generation. For a bulk data transfer most acknowledgments are generated after the receive buffer becomes empty. If data arrives in fast bursts, or if the receiving machine is slow or loaded, data may accumulate in the socket buffer. If a window mechanism is employed for flow control as in the case of TCP, the whole window burst may accumulate in the buffer. The window will not be acknowledged until the data is passed to the application. Until this happens, TCP will be idle awaiting the reception of new data, which will come only after the receive buffer is empty and the acknowledgment has gone back to the sender. So there may be gaps during data transfer which will be equal to the time to copy out the buffer plus one round trip delay for the acknowledgment and the new data to arrive.

4.4 Experiment 4: Effect of Background Load

The previous experiments have studied the IPC mechanism when the machines were solely devoted to communication. However, it would be more useful to know how extra load present on the machine would affect the IPC mechanism. In a distributed environment (especially with single-processor machines), computation and communication will affect each other. This experiment investigates the behavior of the various queues when the host machines are loaded with artificial load. The experiment is aimed at determining which parts of the IPC mechanism are affected and how when additional load is present on the machine. The results reported include the effect on throughput and graphs depicting how the various queues are affected by the extra workload during data transfer.

The experiment was performed by setting up a unidirectional connection, running the workload, and sending data from one Sparcstation to another. The main requirement for the artificial workload was that it be CPU intensive. The assumption is that if a task needs to be distributed, it will most likely be CPU intensive. The chosen workload consists of a program that forks a specified number of processes that calculate prime numbers. For the purposes of this experiment the artificial workload level is defined to be the number of prime-number processes running at the same time.

Figure 8 shows the effect of the artificial workload on communication throughput when the workload is run on the server machine. The experiment was performed by first starting the artificial workload and then immediately performing a data transfer of 5 Mbytes. The graph shows that the throughput drops sharply as the number of background processes increases, which suggests that IPC requires a significant amount of CPU cycles. Although not actually measured, the effect on computation also appears significant. Thus when computation and communication compete for CPU cycles they both suffer, leading to poor overall performance.

Figure 8 shows the internal queues during data transfer

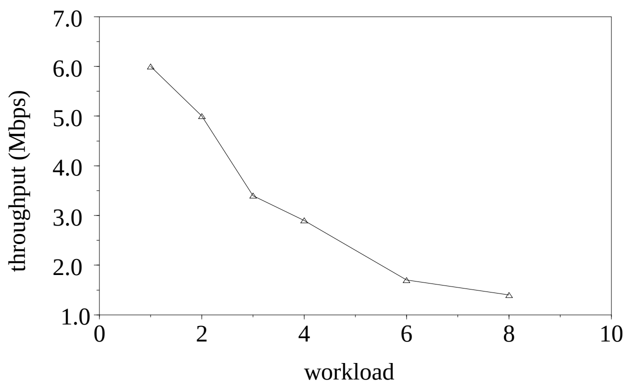

Figure 8: **Throughput v.s. server background load**

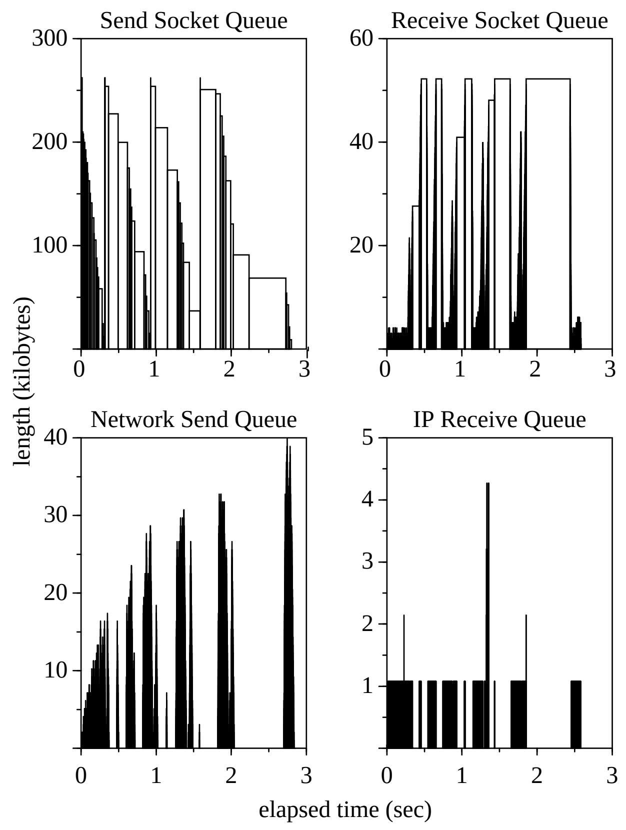

Figure 9: **Queue length v.s server background load**

with a loaded server. For these graphs the data size was reduced to 1 Mbyte to keep the size of the graphs reasonable. The workload level is 4, meaning that there were four prime-number processes running. The x-axis shows the elapsed time and the y-axis the queue length. Examining the receive socket queue plot shows that when data arrives at the queue is delayed significantly. The bottleneck is clearly shown to be the application scheduling. The receiving process has to compete for the CPU with the workload processes, so it is forced to read the data at a lower rate. The extra load seems to affect the IP queue also. In previous experiments with no extra computation load, data would not accumulate in the IP queue. The accumulation however, is still very small compared to the other queues. The interface and socket queues on the sending side reflect the effect introduced by the delay in acknowledging data by the receiver. Although not reported here, note that the effect of loading the client was similar to the effect of loading the server [8].

5. Conclusions

In general, the performance of SunOS 4.0.3 IPC on Sparcstation 1 over the Ethernet is very good. Experiment 1 has shown that average throughput rates of up to 7.5 Mbps are achievable. Further investigation has shown that this rate is not limited by the TCP/IP performance, but by the driver performance. Therefore, a better driver implementation will increase throughput further.

Some improvements to the default IPC are still possible. Experiment 1 has shown that increasing the default socket buffer size from 4 to 16 kilobytes or more, leads to a significant improvement in throughput for large data transfers. Note however, that increasing the default socket buffer size necessitates an increase in the limit of the network interface queue size. The queue, shown in Figure 5, holds outgoing packets awaiting transmission. Currently its size is limited to 50 packets and with the default buffers guarantees no overflow for 12 simultaneous connections (12 connections with a maximum window of 4). If the default socket buffers are increased to 16 kilobytes, then the number of connections drops to 3.

The performance of IPC for local processes is estimated to be about 43% of the memory bandwidth. Copying 1 kilobyte takes about 130 microseconds, meaning that memory-to-memory copy is about 63 Mbps. Local IPC requires two copies and two checksums. Assuming that the checksum takes about half the cycles as memory copy (only reading the data is required), the maximum theoretical limit of local IPC is 63/3 = 21 Mbps. The IPC performance measured was close to 9 Mbps which is about 43% of the memory bandwidth. When processes running on different machines, the sending side of IPC is able to process packets faster than the Ethernet rate, which leads to queueing at the network interface queue. At the receiving side, the network and protocol layers are able to process packets as they arrive. At the socket layer however, packets are queued before delivered to the application. In both local and remote communication, the performance of TCP/IP is determined by protocol processing and checksum. Ignoring the data copy, this was measured to be about 22 Mbps and is limited by the sending side.

While transferring large amounts of data, TCP relies on the socket layer for a notification that the application received the data before sending acknowledgments. The socket layer notifies TCP after all data has been removed from the socket buffer. This introduces a delay in receiving new data which is equal to the time to copy the data to application space plus one round trip time to send the acknowledgment and the new data to arrive. This is a small problem on the current implementation where the socket queues are small, but may be significant in future high bandwidth-delay networks where queues may be very large. Experiments 3 and 4 show that queueing at the receive socket queue can grow to fill the socket buffer, especially if the machine is loaded. This reduction in acknowledgments could degrade the performance of TCP especially during congestion window opening or after a packet loss. The situation will be exacerbated if the TCP large windows extension is implemented. The congestion window opens only after receiving acknowledgments, which may not be generated fast enough to open the window quickly, resulting in the slow-start mechanism becoming too slow.

The interaction of computation and communication is significant. Experiment 4 has shown that the additional load on the machine dramatically affects communication. The throughput graphs show a sharp decrease in throughput as CPU contention increases which means that IPC requires a major portion of the CPU cycles to sustain high Ethernet utilization. Even though machines with higher computational power are expected to become available in the future, network speeds are expected to scale even faster. Moreover, some of these cycles will not be easy to eliminate. For example TCP calculates the checksum twice for every packet, once at the sending and once at the receiving end of the protocol. The fact that in TCP the checksum resides in the header and not at the end of the packer means that this time consuming operation cannot be performed using hardware that calculates the checksum and appends it to the packet as data is sent to, or arrives from the network. Conversely, the presence of communication affects computation by stealing CPU cycles from computation. The policy of favoring short processes adopted by the Unix scheduling mechanism allows the communication processes to run at the expense of computation. This scheduling policy makes communication and computation performance unpredictable in the Unix environment.

Some areas of weakness have already been identified with existing TCP for high speed networks, and extensions to TCP were proposed to address them [6]. The extensions are: (1) use of larger windows (greater that TCP's current maximum of 65536 bytes) to fill a high speed pipe end-to-end; (2) ability to send selective acknowledgments to avoid retransmission of the entire window; (3) and inclusion of timestamps for better round trip time estimation [7]. We and other groups are currently evaluating these options. However, one may question the suitability of TCP's point-to-point 100% reliable byte stream interface for multi participant collaborative applications with multimedia streams. Additionally, if more and more underlying networks support statistical reservations for high bandwidth real time applications, TCP's pessimistic congestion control will have to be reconsidered.

References

[1] Beirsack, E.W. and Feldmeier, D. C., ''A Timer-Based Connection Management Protocol with Synchronized Clocks and its Verification,'' Computer Networks and ISDN Systems, to appear.

[2] Chesson, Greg, ''XTP/PE Design Considerations'', IFIP WG6.1/6.4 Workshop on Protocols for High Speed Networks, May 1989, reprinted as: Protocol Engines, Inc., PEI~90-4, Santa Barbara, Calif., 1990.

[3] Comer, Douglas, *Internetworking with TCP/IP*, Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1991.

[4] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An analysis of TCP processing overhead", *IEEE Communications Magazine*, Vol. 27, No. 6, Jume, 1989, pp. 23-29.

[5] Jackobson, Van, "Congestion Avoidance and Control", *SigComm '88, Symp., ACM*, Aug. 1988, pp. 314-329.

[6] Jacobson, V., Braden, R.T. and Zhang, L., "TCP extensions for high-speed paths", RFC 1185, 1990.

[7] Nicholson, Andy, Golio, Joe, Borman, David, Young, Jeff, Roiger, Wayne, "High Speed Networking at Cray Research," ACM SIGCOMM Computer Communication Review, Volume 2, Number 1, pages: 99-110.

[8] Papadopoulos, Christos, "Remote Visualization on Campus Network," MS Thesis, Department of Computer Science, Washington University in St. Louis, 1992.

[9] J. Postel, "Internet Protocol-DARPA Internet program protocol specification", Inform. Sci. Inst., Rep. RFC 791, Sept. 1981.

[10] J. Postel, "Transmission Control Protocol", USC Inform. Sci. Inst., Rep. RFC 793, Sept. 1981.

[11] Leffler, Samuel J., McKusick, Marshall K., Karels, Michael J., and Quarterman, John S., *The Design and Implementation of the 4.3 BSD Unix Operating System*, Addison-Wesley Publishing Company, Inc., Redding, Massachusetts, 1989.

[12] Sterbenz, J.P.G., Parulkar, G.M., "Axon: A High Speed Communication Architecture for Distributed Applications," Proceedings of the Ninth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'90), June 1990, pages 415-425.

[13] Sterbenz, J.P.G., Parulkar, G.M., "Axon: Application-Oriented Lightweight Transport Protocol Design," Tenth International Conference on Computer Communication (ICCC'90), Narosa Publishing House, India, Nov. 1990, pp 379-387.