EXPERIMENTAL EVALUATION OF SUNOS IPC AND TCP/IP PROTOCOL IMPLEMENTATION[1]

Christos Papadopoulos
christos@dworkin.wustl.edu
+1 314 935 4163

Gurudatta M. Parulkar
guru@flora.wustl.edu
+1 314 935 4621

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

ABSTRACT

Progress in the field of high-speed networking and distributed applications has lead 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 SunOS Inter-Process Communication (IPC) and TCP/IP protocol implementation for distributed, high bandwidth applications. Components studied include queueing in different layers, protocol control mechanisms (such as flow and error control), per-packet processing, buffer requirements, and interaction with the operating system.

---

[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.

1. INTRODUCTION

Considerable research and development efforts are being spent in the design and deployment of high speed networks. Many of these efforts suggest that networks supporting data rates of hundreds of Mbps will become available soon [5]. Target applications for these networks include distributed computing involving remote visualization, collaborative multimedia, medical imaging, teleconferencing and video distribution. Progress in high-speed networking suggests that 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 [3,11,12] for emerging applications over high speed networks. Some researchers believe that existing protocols such as TCP/IP are suitable and can be adopted for use in high speed environments [4,9]. Others claim that these protocols are complex and their control mechanisms are not suitable for high speed networks and applications [1,2,15,16]. It is important, however, to note that both groups agree that appropriate operating system support and efficient protocol implementation are essential to support high bandwidth applications on high speed networks.

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 requirements and interaction with the operating system by systematic measurement. Two important reasons for using measurement to evaluate the performance of a complex system such as IPC in Unix are: (1) it allows the direct examination of the real system instead of a model, and (2) it can assess the performance of the system with all the pieces in place and fully functional, thus revealing unexpected interactions (if any) that may not be observed by studying the components (or component models) individually.

The software examined is the SunOS 4.0.3[1] IPC using BSD stream sockets on top of TCP/IP. The hardware consisted of two Sun 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

[1]SunOS 4.0.3 is based on 4.3 BSD Unix. See [13] for further details.

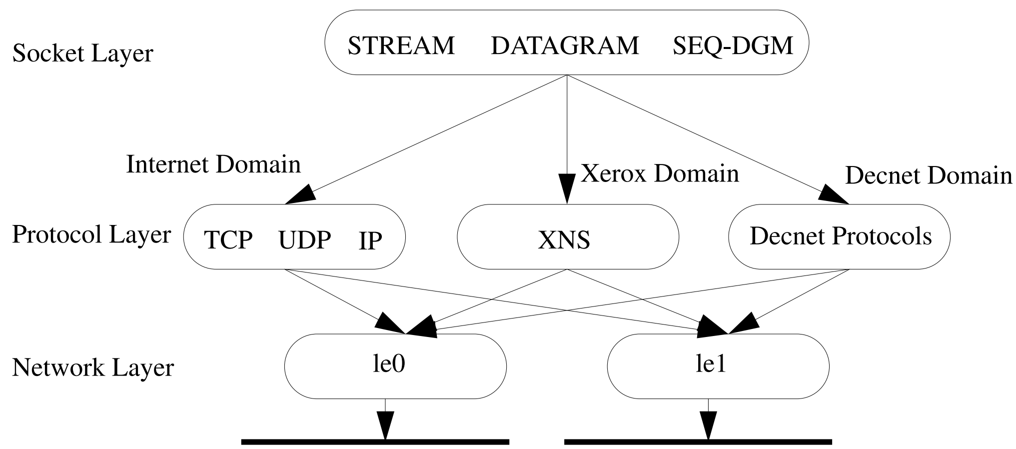

Figure 1: Inter-process Communication (IPC) layering

IPC could be studied in depth due to the lack of source code for SunOS 4.1.

The paper is organized as follows: Section 2 presents an overview of the IPC implementation in the Unix kernel. Section 3 describes two tools traditionally used for protocol evaluation and then presents a new tool used for many of the measurements reported in this paper. Section 4 presents a series of experiments and measurements. Finally, Section 5 summarizes the conclusions of this paper.

2. UNIX KERNEL BACKGROUND

This section presents a brief overview of the IPC implementation in SunOS 4.0.3. The description focuses on the internal structure and organization of IPC and the underlying protocols. The overview is necessary in order to understand the operation of protocols and their interaction with the operating system and identify the various queueing points during data transfer.

2.1 THE INTER-PROCESS COMMUNICATION (IPC) LAYERS

IPC is organized in three layers as shown in Figure 1. The first layer is the *socket* layer which is the IPC interface to applications. The main functions of the socket layer are: (1) to identify different applications on the same host, and (2) to provide send and receive buffers for data transfer. The socket interface provides *sockets*, which are abstract objects that the applications can create, connect and use for sending or receiving data. Sockets appear as file descriptors to the application, thus the usual system calls manipulating file descriptors were extended to operate on sockets. The sockets are typed, with each type providing different communication semantics. For example, a *STREAM* socket provides a reliable connection-based byte stream which may support out-of-band data; a *DATAGRAM* socket provides connectionless, unreliable, packet-oriented communication.

The protocol layer is composed of protocol families. A protocol family is a group of protocols belonging to the same domain. Each domain has its own addressing scheme. The protocols in each family implement the various socket semantics. For example, the Internet family of protocols forms the Internet domain, whose main protocols are TCP, UDP and IP[1]. Other domains, like Xerox NS and DECNET domains, provide their own protocols which may be functionally equivalent but not compatible with the Internet domain protocols. Depending on requirements, the application can select the protocol it wishes to use from a family (domain) of protocols when creating a socket, or leave the choice to the socket layer which will choose an appropriate protocol that implements the requested socket semantics.

The third IPC layer is the network interface layer which sits directly on top of the hardware. It is composed of hardware device drivers and their interfaces (for example the Ethernet driver and card). A host may be connected to more than one network and thus have more than one interface, as in the case of a gateway. The protocols choose a network interface to output packets as part of their routing decision.

To summarize, the Unix IPC facilities are implemented in three layers: (1) the socket layer, which manipulates endpoints that provide certain types of service; (2) the protocol layer, which consists of the protocols that implement those services; and (3) the network layer, which is a collection of interfaces to the networks which the host is attached to. When a socket is created it is bound to a protocol appropriate to the socket semantics, and the protocol chooses an interface to send data. An example configuration is the application creating a reliable (stream) socket in the Internet domain, the socket layer selecting the TCP/IP protocol stack and packets routed over the local Ethernet interface.

2.2 IPC MEMORY MANAGEMENT: MBUFS

Interprocess communication and network protocols impose special requirements on the memory management system. The system must be capable of handling efficiently both fixed and variable size memory blocks. Fixed size memory is needed for data structures like control blocks, and variable size memory for storing packets. Efficient allocation and deallocation of such memory blocks is essential since these operations occur very frequently (at least once for every packet). These requirements motivated the developers of BSD 4.3 to create a new memory management scheme based on data structures called MBUFs (Memory BUFfers).

The main components of mbuf memory management are the mbuf data structure, the mbuf memory and a collection of supporting routines that manipulate the mbuf memory. The memory is allocated during system initialization and is part of the kernel memory permanently reserved for mbufs. Mbufs always reside in physical memory (i.e. they are never paged out). The structure of an mbuf is shown in Figure 2. Mbufs are 128 bytes long, out of which up to 112 bytes can be used for storing data. The remaining 16 bytes contain control fields which are used to manipulate both the mbufs and the data. These are called *small (or plain) mbufs* because the amount of data stored internally is limited to 112 bytes. When more space is required, an external page from a page pool

---

[1] IP however, is not accessed directly by the socket layer but through TCP and UDP, which use IP for sending packets.

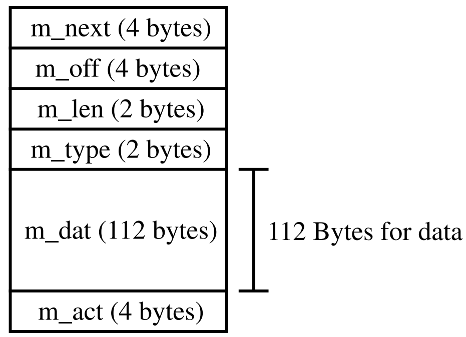

Figure 2: The Mbuf Data Structure

(which is part of the mbuf memory) can be attached to a plain mbuf to form a *cluster mbuf*. These pages can store up to 1024 bytes. Data of a cluster mbuf is stored exclusively in the external page, never in the internal space (112 bytes) of the mbuf. This facilitates an important optimization for copying: the data part of a cluster mbuf is never physically copied; rather each page is associated with a reference counter which keeps track of page usage. The page is free when the counter reaches zero. This optimization is not available for plain mbufs.

SunOS provides yet another type of mbuf which is not part of BSD Unix. These are similar to cluster mbufs, except that data is stored in an external buffer which is not a part of the mbuf memory. A program wishing to allocate such an mbuf must provide the external buffer and a pointer to a function to be used when deallocating this mbuf. This type of mbuf is used by the Ethernet interface driver to facilitate mbuf "loaning", where an mbuf owned by the driver is loaned temporarily to higher level protocol functions. The advantage of loaning is that it avoids data copy from the memory of the interface to mbuf space.

2.3 IPC FUNCTION CALLS FOR DATA EXCHANGE

Several kernel functions are invoked for data exchange. Figure 3 shows these functions in the order in which they get invoked. Also shown is each function's location with regard to layering. Note that the arrows imply the sequence of the function invocation, not the data flow. The data flow is implied by the system call called at the application level on each side: write () sends the data, and read () receives it.

2.4 QUEUEING MODEL

Figure 4 shows the data movement at the sending side. Initially data resides in the application space in a contiguous buffer. After a send request, data is copied (and therefore fragmented) into mbufs in kernel space which are queued at the socket buffer. Protocols remove or copy[1] mbufs from the socket queue and add headers to form packets, which are placed in the network queue.

---

[1]Mbufs are copied when data must be kept until acknowledged, as in the case of reliable transmission. However, for cluster mbufs data copy is a logical copy.

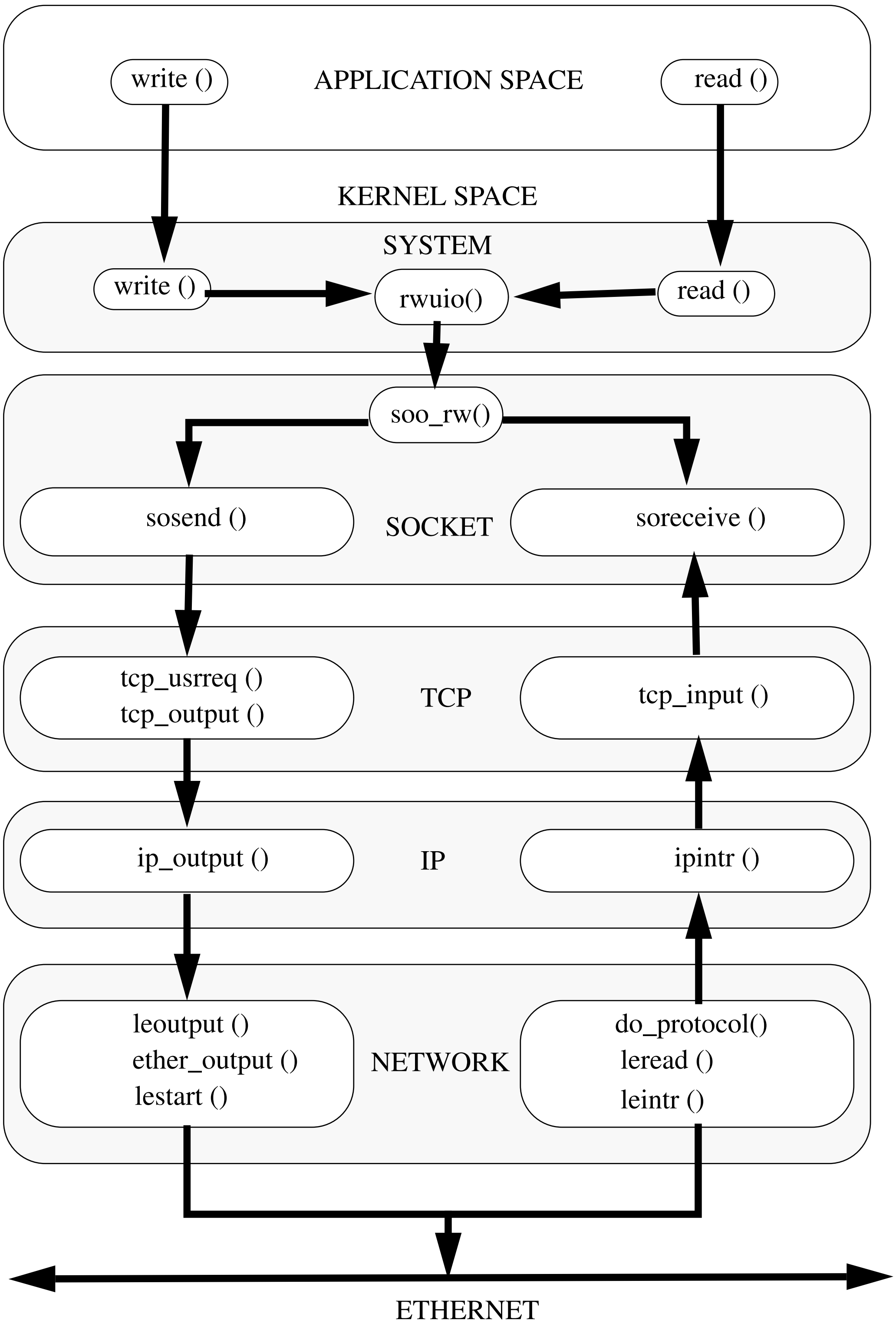

Figure 3: Function call sequence

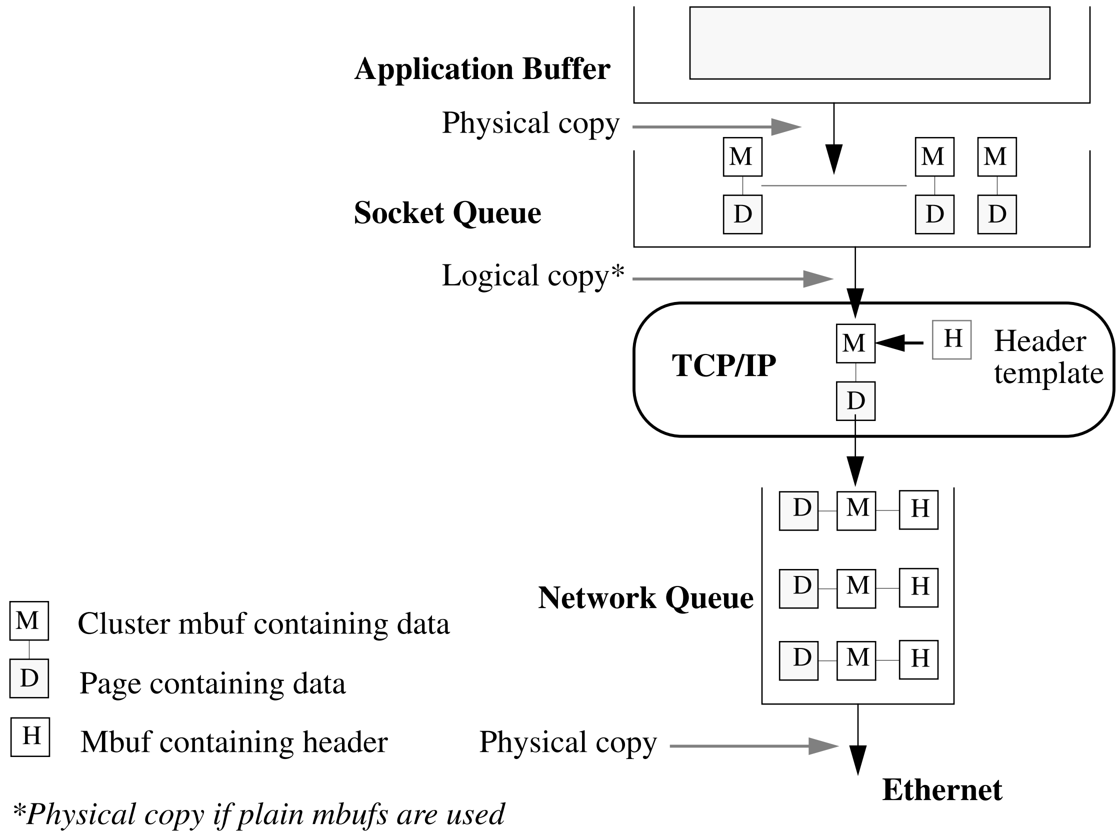

Figure 4: Data distribution into (cluster) mbufs

Packets are queued at the network interface until the driver is able to transmit them. On the receiving side, packets are received by the network interface and placed in one or more mbufs. After the Ethernet header is stripped, the remaining packet is appended to the protocol receive queue (the IP queue for example), and the protocol is notified via a software interrupt. After processing the packets the protocol passes them to higher layer protocols (TCP for example). The top layer protocol after processing appends the packets to the receive socket queue and wakes up the application's read call if it is sleeping in the kernel. The call completes by copying data from mbufs to the application's address space and returns. Thus, there are four queueing points in IPC and are depicted in Figure 5.

3. TOOLS FOR MEASURING IPC PERFORMANCE

This section first examines two well known public domain tools used for IPC measurements, namely TCPDUMP and NETMON. It is argued that although these tools are very useful for their intended applications, they are not suitable for investigating the internals of IPC. The discussion is followed by the presentation of a new tool specifically developed for evaluating IPC.

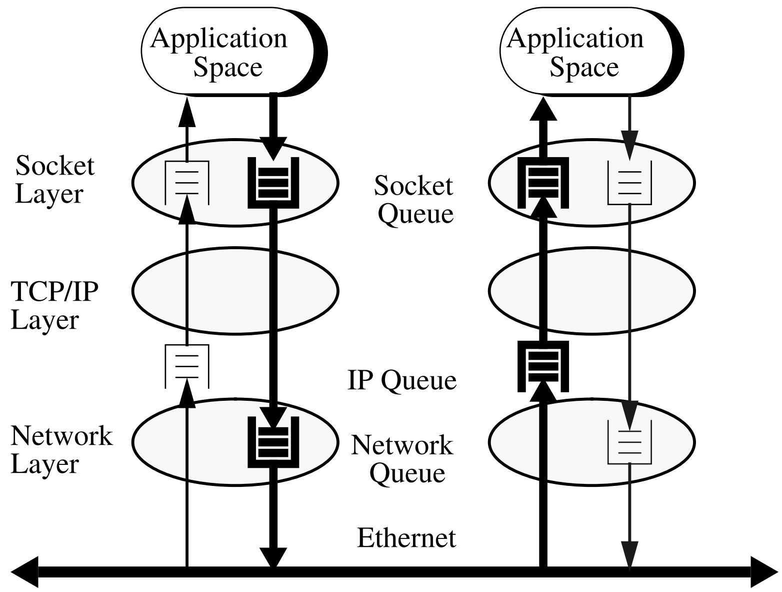

Figure 5: IPC Queueing Model

3.1 TCPDUMP

Tcpdump[1] is a network monitoring tool that captures packet headers from the Ethernet. To avoid interference, the tool usually runs on a machine not involved in the monitored communication. The captured headers are filtered to display those of interest to the user; if desired, the packet traffic can be displayed unfiltered to monitor all activity on the network. Tcpdump prints information extracted from the header of each packet and a timestamp. A typical Tcpdump configuration is shown in Figure 6.

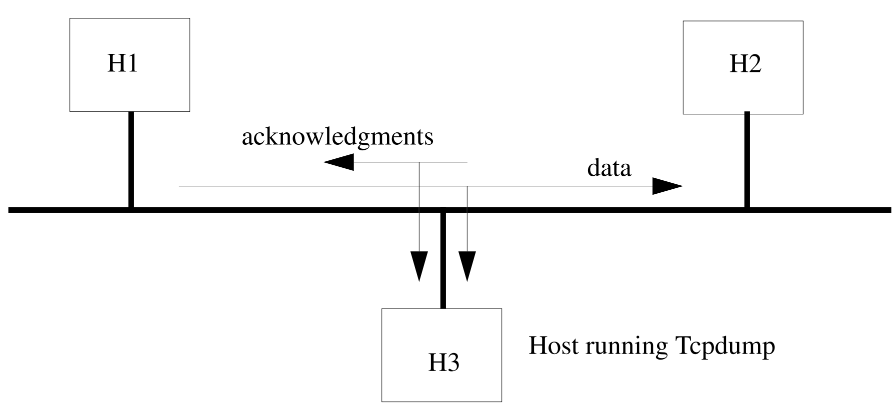

Figure 6: Capturing packets with Tcpdump

[1]Tcpdump was developed at Lawrence Berkeley Laboratory by Van Jacobson, Steven McCanne and Graig Leres. It was inspired by Sun's Etherfind.

Tcpdump has two important advantages: (1) it does not interfere with communication since it runs on a separate machine, and (2) it does not require any changes to the kernels of the machines involved (some minor changes may be required to the kernel of the machine running Tcpdump). The headers are dumped either in a file or on the screen. A sample trace generated by Tcpdump is shown below:

```
root_flora[4]# tcpdump ip host zen and patti
23:12:40.911096 zen.1164 > patti.2000: S 589312000:589312000(0) win 4096 <mss 1460>
23:12:40.911780 patti.2000 > zen.1164: S 1924864000:1924864000(0) ack 589312001 win 4096 <mss 1460>
23:12:40.912218 zen.1164 > patti.2000: . ack 1 win 4096
23:12:40.914468 zen.1164 > patti.2000: . 1:1461(1460) ack 1 win 4096
23:12:40.915609 zen.1164 > patti.2000: . 1461:2921(1460) ack 1 win 4096
23:12:40.916638 zen.1164 > patti.2000: P 2921:4097(1176) ack 1 win 4096
23:12:40.916936 patti.2000 > zen.1164: . ack 1461 win 4096
23:12:40.917213 patti.2000 > zen.1164: . ack 2921 win 4096
23:12:40.918780 zen.1164 > patti.2000: . 4097:5557(1460) ack 1 win 4096
...
...
23:12:40.949691 zen.1164 > patti.2000: . 14317:15777(1460) ack 1 win 4096
23:12:40.950123 patti.2000 > zen.1164: . ack 14317 win 4096
23:12:40.951360 patti.2000 > zen.1164: . ack 15777 win 4096
23:12:40.952889 zen.1164 > patti.2000: P 15777:16385(608) ack 1 win 4096
23:12:40.956877 zen.1164 > patti.2000: F 16385:16385(0) ack 1 win 4096
23:12:40.957190 patti.2000 > zen.1164: . ack 16386 win 4096
23:12:40.957579 patti.2000 > zen.1164: F 1:1(0) ack 16386 win 4096
23:12:40.958517 zen.1164 > patti.2000: . ack 2 win 4096
1637 packets received by filter
7 packets dropped by kernel
```

Our experience has shown that to obtain accurate timestamps the third machine must be at least as fast as the machines being monitored. Packet traces created by tcpdump are very useful in examining protocol-to-protocol communication in order to detect any aberrations. The trace can be used to generate a graph to visualize packet activity on the network. However, no information about the internals of the IPC mechanism can be extracted from such traces. Tcpdump presents only an external view of the communication which may be skewed by the underlying network, and thus cannot provide information about the internal queues. For example, it is not clear whether the rate packets appear on the network is determined by the speed of the hardware interface or that of the protocol. In addition, although the time the acknowledgments reached the network can be observed, no information is available about when they were actually generated by the protocol[1]. Such information is necessary when evaluating the performance of the IPC implementation.

3.2 NETMON

Netmon[2] is a tool running under Unix BSD 4.3 that records network protocol information for monitoring and/or analysis. Netmon employs probes in the kernel code to record information in

---

[1]It is possible that there may be a difference between the time an acknowledgment was generated and the time it appeared on the wire. The LANCE chip for example, will delay transmission of packets if it is receiving back-to-back packets from the network.
[2]Developed by Allison Mankin of MITRE Corp.

memory about network protocol activity that a user process subsequently retrieves into a file. However, Netmon monitors only IP level and network interface queue information. No probes monitoring higher layers are available. A Netmon example output is shown below:

```
"Example of NETMON/iptrace Output"
80202B00 06:57:35.57 ethIn>0 0
80202B00 06:57:35.57 ipInForw>0 qe0 iplen 41 TCP(6) 128.29.1.12.1136 > 10.0.0.78.9
80202B00 06:57:35.58 ipOutForw>-1 dda0 iplen 41 TCP(6) 128.29.1.12.1136 > 10.0.0.78.9
80202B00 2 06:57:35.58 x25Out>1 1
+ + +
80202B00 2 06:57:36.76 x25OutDeq>2 1
```

Netmon is very useful for providing information on network protocol activity. It can also be used to monitor gateway activity. Since its emphasis is on the network layer, it does not provide any information on the activity of higher layers, like the socket layer. Moreover, it was designed to retrieve protocol information which is not relevant to IPC evaluation, thus incurring unnecessary overhead.

3.3 A NEW PROBING TOOL

Tools like Tcpdump and Netmon focus on monitoring some specific layers of communication and therefore do not emphasize monitoring the mechanism as a whole. A tool that monitors all the communication layers at the same time will provide more insight into the internals of IPC. Moreover, such a tool can be used to investigate the interaction among the different components and highlight aspects that are invisible when examining the components individually.

This section presents a tool that was developed specifically to monitor the internals of IPC which is based on probes. Others have used similar mechanisms to monitor different aspects of TCP [6]. A probe is a small segment of code placed strategically in the kernel code to record information during the execution of IPC. The probes are supplemented by data structures and routines that manipulate the recorded information. In the following description each probe's location is identified and the information retrieved is described; then the issue of minimizing probe overhead is discussed, followed by a description of how probes are activated. Finally, the internal structure of the probes is presented.

3.3.1 PROBE LOCATION

Two probes are associated with each IPC queue (see Section 2), one monitoring data appended to the queue and another monitoring data removed from the queue. The selected probe locations are depicted in Figure 7. Each probe when activated records (1) a timestamp, (2) the amount of data passing through its checkpoint, and (3) a type field to identify the location of the probe. The information recorded is minimal, but can nevertheless provide enough information about the behavior of each queue. For example, the timestamp difference provides queue delay, queue length, arrival rate and departure rate. The data length provides throughput measurements, and helps identify where fragmentation occurs. The information collected by the probes is used to compute various important statistics.

On the sending side, probes 1 and 2 monitor the queue delay and length at the socket layer. The

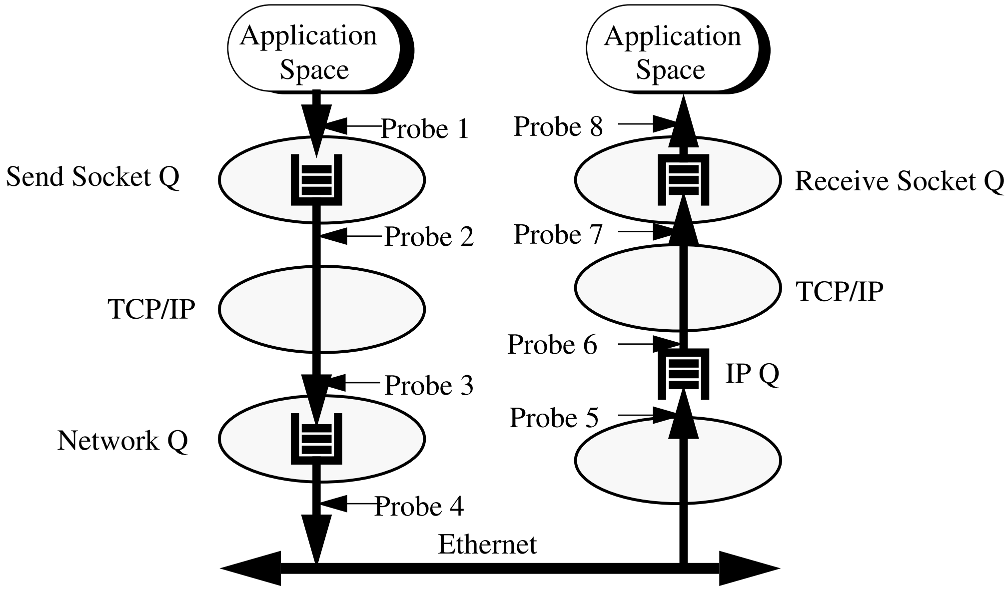

Figure 7: IPC Queues and Probe Location

records produced by probe 2 show how data is broken into protocol transmission requests. Probe 3 monitors packets reaching the interface queue. The rate packets enter the queue is essentially the protocol processing rate. This probe also monitors the windowing and congestion avoidance mechanisms by recording bursts of packets produced by the protocol. The protocol processing delay can be obtained by the inter-packet gap during a burst. Finally, probes 3 and 4 monitor the interface queue delay and length.

On the receiving side, the rate of packets arriving from the Ethernet is monitored by probe 5. Probes 5 and 6 monitor the delay and length of the IP queue. The difference in timestamps between probes 6 and 7 gives the protocol processing delay. Probes 7 and 8 monitor the delay and length of the receive socket queue. Packets arrive in the queue after the protocol has processed them, and depart as they get copied to application space.

The probes can also monitor acknowledgment activity. If the data flow is in one direction, the packets received by the data sender will be acknowledgments. Each application has all eight probes running, monitoring both incoming and outgoing packets. Therefore, in one-way communication some of the probes will be recording acknowledgments.

3.3.2 PROBE OPERATION AND OVERHEAD

An issue of vital importance is that probes incur as little overhead as possible. There are three kinds of overhead associated with the probes: (1) overhead due to probe execution time; (2) overhead due to the storing of measurements; and (3) overhead due to probe activation. To address the first source of overhead, any kind of processing besides logging a few important parameters in the probes was precluded. To address the second overhead, it was decided that probes should store records in the kernel. To accomplish this, a circular list of records residing in kernel space is used. The list is initialized at system start-up by the TCP initialization routine, and two pointers, one to the head and one to the tail of the list are made globally available to all probes. The head pointer points to the first empty record on the list and the tail pointer to the last record in use. Since the records are linked during initialization, only the head pointer needs to be updated after using a record. The third source of overhead is incurred by the work required to determine if a packet should be logged. It is critical that probe activation should interfere as little as possible with normal operation since every packet needs to be examined to determine whether it should be logged or not. The developed activation mechanism is described next.

3.3.3 PROBE ACTIVATION

To activate the probes the application calls setsockopt (). This system call is part of the socket interface and is used by applications to set various properties on sockets, such as the size of the send and receive buffers, out-of-band message capability, etc. The setsockopt () call was extended by adding a socket option for logging, named SO_LOG. Setting the logging option on a socket has two effects: (1) a flag is set in the socket structure marking the socket as logging; and (2) a flag is set in each packet's IP header marking the packets that need to be identified at the lower layers. The reason for this double marking is as follows: The flag set in the socket structure enables the socket and protocol layer routines to quickly determine if a socket is marked for logging, because both layers have direct access to the socket structure. The network layer, however, does not have direct access to this structure. One way of solving this problem is to use the same method the protocol uses to identify incoming packets, i.e. to first locate each packet's corresponding protocol control block and then using it locate the socket structure. However, this requires performing a search for each outgoing packet, which is too expensive to be practical. Instead, it was decided that the packet itself should carry the information to activate the probes. This allows easy identification of marked packets at the network layer of both the sending and receiving end. The logging information is carried in the IP header in the "tos" (type of service) field, which has 2 bits currently unused. Existing IP implementations simply ignore them. One of those bits was chosen to carry the logging information. The network layer is able to easily access this field since the IP header is always contained in the first mbuf of the chain either handed down by the protocol, or up by the driver. Admittedly, this is a non-standard approach and it does violate layering, but it has the advantage of incurring minimum overhead and being straightforward to incorporate.

In the current implementation, logging initiated at the sending end will automatically initiate logging at the receiving end. The reason for this is that sometimes the receiver was not able to activate its probes before the arrival of the initial packets, leading to the logging of incomplete data. To overcome this problem, the kernel was modified to set the socket logging option automatically if the connection request packet has the logging flag set.

3.3.4 PROBE INTERNAL STRUCTURE

The code for a typical probe is shown in Figure 8. It is very simple and efficient. The probe shown is activated if the socket option SO_LOG has been set, which is determined at the first "if"

```
#ifdef LOGGING
if (so->so_options & SO_LOG)
        {
        int     s;
        s = splimp();
        if (log_head->next != log_tail)
                {
                register struct logrec *lr;
                lr = log_head;
                log_head = log_head->next;
                uniqtime(&(lr->tv));
                splx(s);
                lr->type = LOG_LEVEL;
                lr->len = /* data length */
                }
        else    splx(s);
        }
#endif LOGGING
```

Figure 8: Code for a typical probe

statement. Probes having no access to the socket structure would test the IP header "tos" field. Then the probe masks network device interrupts and makes sure that the next record on the list is indeed free. If it is free, its address is assigned to a local pointer, and the list head pointer is advanced. Next the probe records the current time, the length of the data, and an identifier for the location of the probe. Determining the data length may involve traversal of an mbuf chain and adding up the length fields. The interrupts are unmasked after obtaining the timestamp since the remaining operations are not critical.

4. PERFORMANCE OF IPC

This section presents a number of experiments aimed at characterizing the performance of various IPC components including the underlying TCP/IP protocols. The experiments were performed with a single user on the machines (but still in multi-user mode), to avoid interference from other users. During the experiments, the Ethernet was monitored using a tool capable of displaying the Ethernet utilization (Sun's *traffic* or *xenetload*) to ensure that background Ethernet traffic was low (less than 5%) when the experiments were performed. The experiments were also repeated a number of 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 part the effect of the socket buffer size on throughput is measured. The results suggest that it would be beneficial to increase the socket buffer size over the default for applications running on the Ethernet, and for applications that will be running on faster networks in the future. In the second part, the performance of two IPC configurations is compared: in the first the communicating processes are running on two physically separate machines, and in the second the processes are running on the same machine. The observed local IPC performance suggests that the mechanism is capable of exceeding the Ethernet rate. However, the observed throughput across the Ethernet is about 75% of the maximum Ethernet rate.

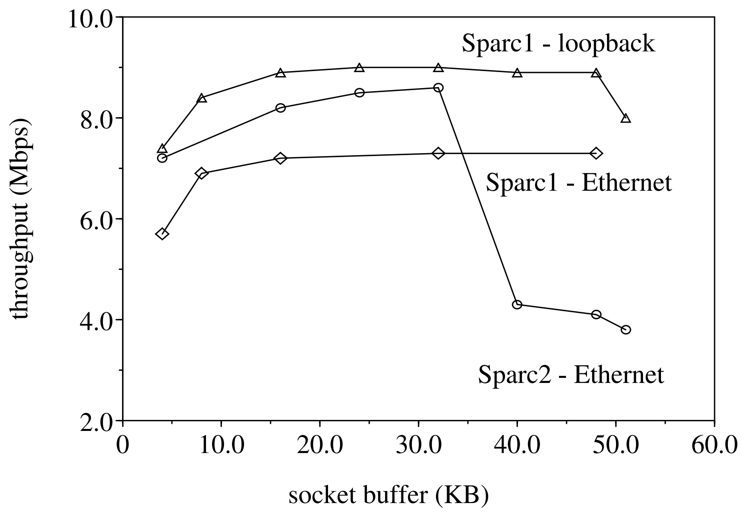

Figure 9: Throughput vs. Socket Buffer Size

4.1.1 THROUGHPUT vs. SOCKET BUFFER SIZE

The socket buffer size is a per-connection user-specified parameter that controls the amount of memory the kernel reserves for buffering. There are two distinct buffers associated with each socket: one for sending and one for receiving (represented by the socket layer queues in Figure 5). The receive socket buffer is the last queue in the queueing network, and therefore may have a decisive effect on the performance of IPC. In addition, the buffer sizes affect TCP directly in that the minimum of the buffers is the maximum window TCP can use.

Figure 9 shows throughput as a function of socket buffer size when both send and receive buffers are of equal size. The x-axis shows the buffer size, and the y-axis the observed throughput[1]. Plots are shown for three configurations: (1) processes on two Sparcstation 1's communicating over the Ethernet; (2) processes on a single Sparcstation 1 communicating through the loop-back interface; and (3) processes on two Sparcstation 2's communicating over the Ethernet. The unexpected dive around 32 kilobytes in throughput for the third plot could not be explained. It is believed to be a manifestation of a problem with SunOS 4.1. The same behavior was observed by other people, who also had no explanation of this phenomenon[14]. The unavailability of source code for SunOS 4.1 prevented further investigation of this problem[2].

The throughput of Sparcstation 1 machines over the Ethernet shows a significant improvement as the socket buffer size increases. The biggest increase in throughput is achieved by quadrupling the buffers (from 4 to 16 kilobytes), followed by some smaller subsequent improvement as the buffers are increased further. These results suggest that increasing the socket buffer size is beneficial even for existing IPC. The increase is necessary in order to effectively use TCP over high bandwidth networks since the TCP window is directly affected by the size of these buffers. Networks

[1]The throughput measurements were obtained using custom software.
[2]The problem has been fixed in later versions of SunOS.

with high bandwidth-delay product store large amounts of data in transit and hence require a large window size to allow the transmitter to fill the pipe. Larger TCP windows are possible only if socket buffers are increased. With the default size however, the window cannot exceed 4096 bytes allowing at most four Ethernet packets to be transmitted with every window, forcing TCP to a kind of four-packet stop-and-wait protocol[1].

Another potential problem is that the use of small socket buffers leads to a significant increase in acknowledgment traffic since at least one acknowledgment for every window is required. Although this may be desirable during congestion window opening in TCP [7], generating acknowledgments too fast may lead to unnecessary overhead. 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, which may contribute to the lower performance obtained with the default socket buffer size.

In light of the above observations, all experiments were performed with this implementation's maximum allowable socket buffer size, which is approximately 51 kilobytes. Any deviations will be clearly stated in the text. Note that current TCP can support a maximum window size of 64 kilobytes.

4.1.2 PERFORMANCE OF LOCAL vs. REMOTE IPC

In most implementations the IPC interface to the network is designed so that protocols are not aware whether their peer is a local process or a process on a different machine. A protocol's routing decision simply involves choosing a network interface to output packets, meaning that the protocol is unaware whether the interface is connected to a real network. The local interface is not connected to any physical network, but instead it loops back the packets to the receiving end of the protocol. The protocol routes packets to the loop-back interface the same way it would route to other network interfaces.

Figure 9 shows the throughput over the loopback interface vs. the socket buffer size. Bigger socket buffers are shown to help even without a real underlying network. The throughput reaches maximum rates close to 9 Mbps. The result implies that a machine running only one side of the communication should comfortably reach rates higher than 9 Mbps. However, the best observed throughput over the Ethernet was around 7.5 Mbps.

To verify that rates higher than 7.5 Mbps are possible on the Ethernet, throughput measurements were taken between two Sparcstation 2 workstations connected on the same Ethernet. The result, also shown in Figure 9, shows that rates up to 8.7 Mbps are indeed possible (it is important to note that in this experiment the Sparcstation 2 machines were also using 1 kilobyte packets as opposed to 1460 bytes which is the default, to ensure that the increase in throughput was not caused by the larger packet size). These results suggest that there is a bottleneck somewhere in the network

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

layer in SunOS 4.0.3. The main bottleneck was traced to the interface driver which was unable to transmit back-to-back packets, and thus had to interrupt the operating system for every new packet. For more details see [10].

4.1.3 SUMMARY

In this experiment it was shown that increasing the socket buffer size leads to improved IPC performance on the Ethernet. It was also shown that the socket and protocol layer on a single Sparcstation 1 are able to exceed the Ethernet rate, and therefore two workstations should be able to achieve high Ethernet utilization. This however, does not happen. The bottleneck was traced to the network interface driver, which was unable to sustain higher rates.

4.2 EXPERIMENT 2: INVESTIGATING IPC WITH PROBES

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

4.2.1 PART 1: CONTINUOUS UNIDIRECTIONAL DATA TRANSFER

The graphs presented below are selected to present patterns that were consistent during the measurements. For this set of graphs only measurements during which there were no retransmissions were considered, in order to study the protocol at its best[1]. The results are presented in sets of four graphs: (1) a packet trace; (2) queue length vs. elapsed time; (3) queue delay vs. elapsed time; and (4) per-packet protocol processing delay.

Packet Traces

The first set of graphs shows packet traces as given by probes 3, 4, 7 and 8. Probes 3 and 4 are on the sending side and monitor the protocol output to the network interface and the network driver output to the Ethernet, respectively. Probes 7 and 8 are on the receiving side and monitor the receive socket queue. The input to this queue is from TCP and the output is to the application. The graphs show elapsed time on the x-axis and the packet number on the y-axis. The packet number can also be thought of as a measure of the packet sequence number. 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[2] packet is passed to the network, and for the receiver when this SYN packet is received. The offset introduced is less than 1ms and does not affect the results in any significant manner.

The first packet trace (protocol output) in Figure 10 shows the transmission bursts introduced

---

[1] Data collected with the probes allows the detection of retransmissions. These however, were infrequent.
[2] A SYN packet carries a connection request.

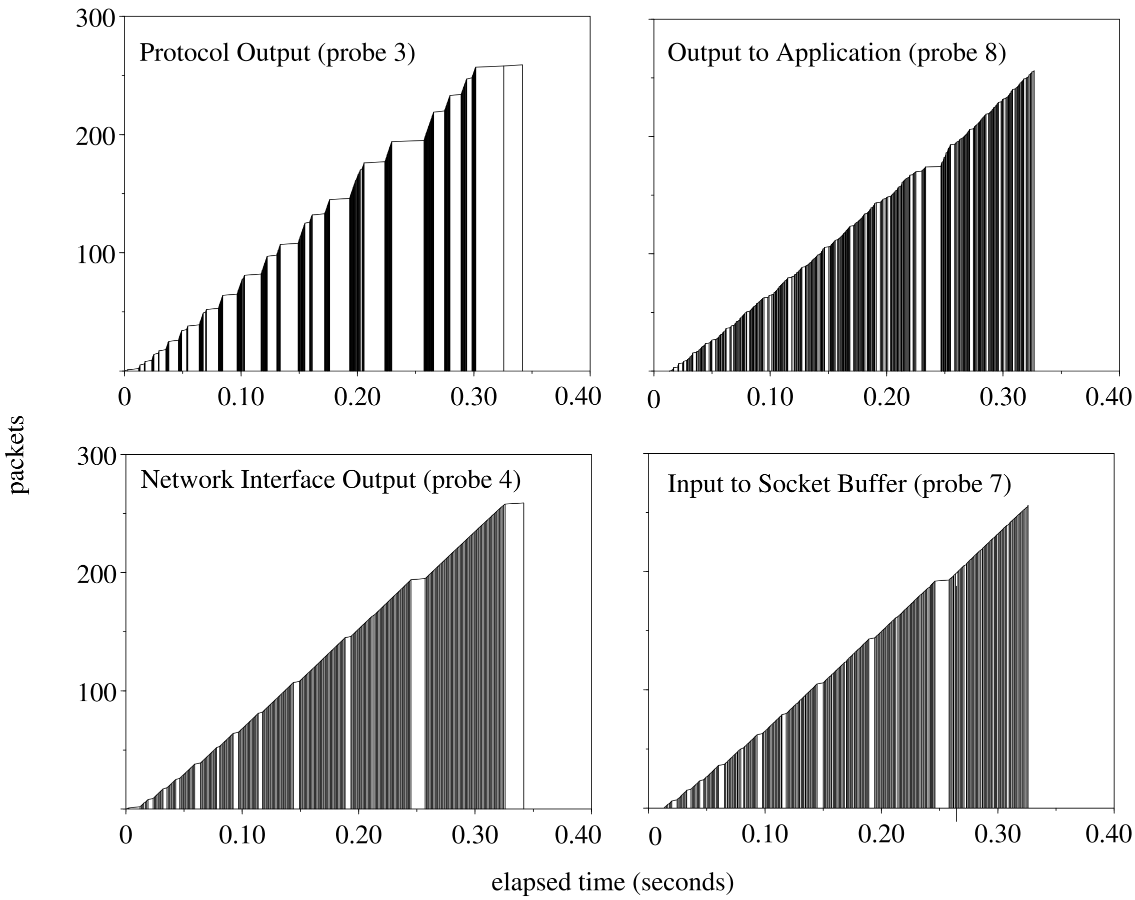

Figure 10: Packet Trace with maximum socket buffers

by the windowing mechanism. The blank areas in the trace represent periods during which the protocol is idle awaiting acknowledgment of the pending window. The dark areas contain a high concentration of packets and represent the periods the protocol is sending out a new window of data. The last two isolated packets are the FIN and FIN ACK packets that close the TCP connection.

The second trace (which is essentially the same as the one Tcpdump would produce) is a trace of packets flowing to the Ethernet. It is immediately apparent that there is a degree of "smoothing" to the flow. The bursts produced by the protocol 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 absorb, which leads to queuing at the interface queue. As a result the interface driver is kept busy most of the time.

On the receiving side, the trace of packets arriving at the receive socket buffer appears to be similar to the trace produced by the Ethernet driver at the sending side. This indicates that there is not much variance in the delay introduced by the IP queue or the Ethernet. It also indicates that the protocol at the receive side is able to process packets at the rate they arrive, which is not surprising since the rate was reduced by the Ethernet. The output rate of the receive socket buffer appears similar to the input to the buffer. Note however, that there is a noticeable idle period, which seems to

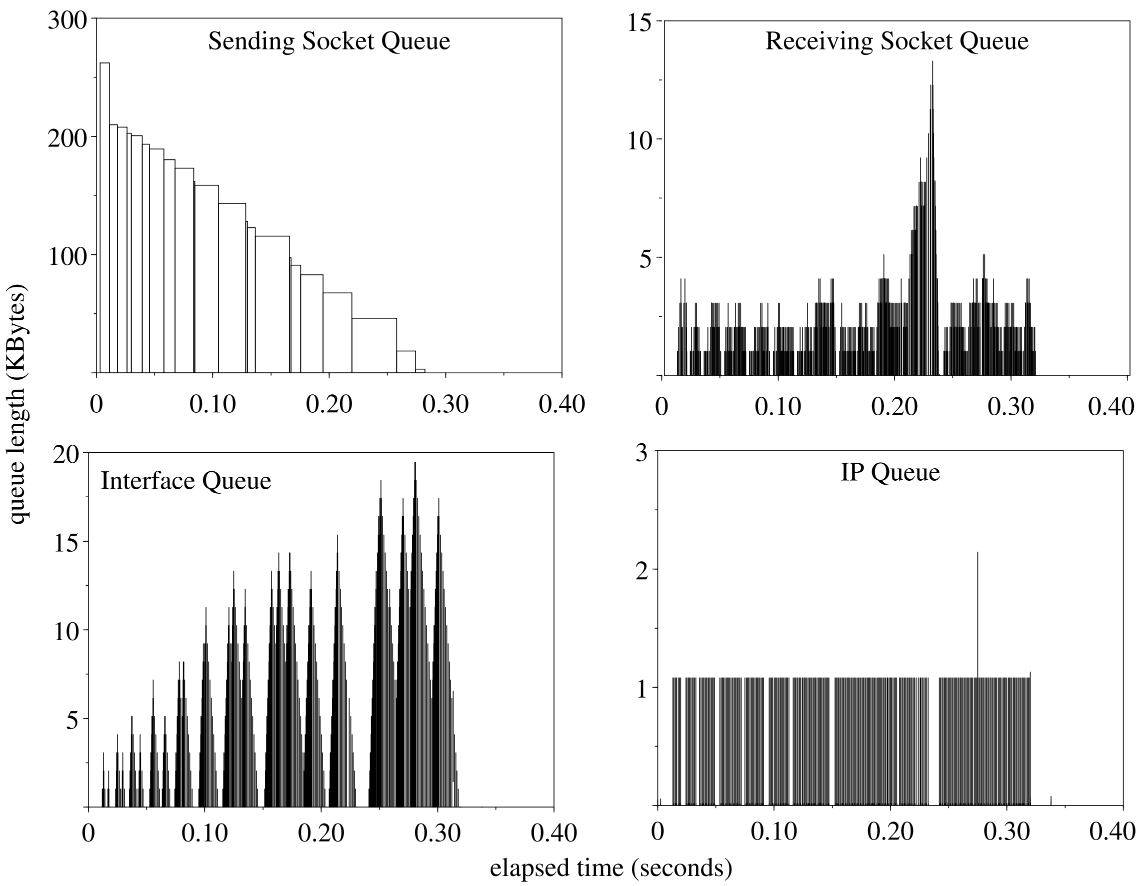

Figure 11: Queue Length during congestion window opening

propagate back to the other traces. The source for this is presented after examining the queue length graphs which are presented next.

Queue Length

The second set of graphs in Figure 11 shows the length of the four queues as a function of elapsed time (note the difference in scale in these graphs). The queue length is shown in number of bytes instead of packets for the sake of consistency, since the socket layer keeps the data as a stream of bytes and not as a queue of packets. Packetization of data is done at the protocol layer. There is no loss of information by displaying the data as bytes instead of packets, since for this experiment the packets were all the same size. It should also be made clear that the data shown in the graph do not actually reside in the socket queue since the queue size is only 51 kilobytes. It is assumed that after the application has called write () the data belongs to the socket layer even if it was not actually copied in the buffer, since the application is put to sleep and cannot access it. Data moves into the socket buffer as space becomes available, i.e. after data is acknowledged.

The length of the sending socket queue is shown to monotonically decrease as expected, but not at a constant rate since its draining is controlled by the window flow control (i.e. by the arrival

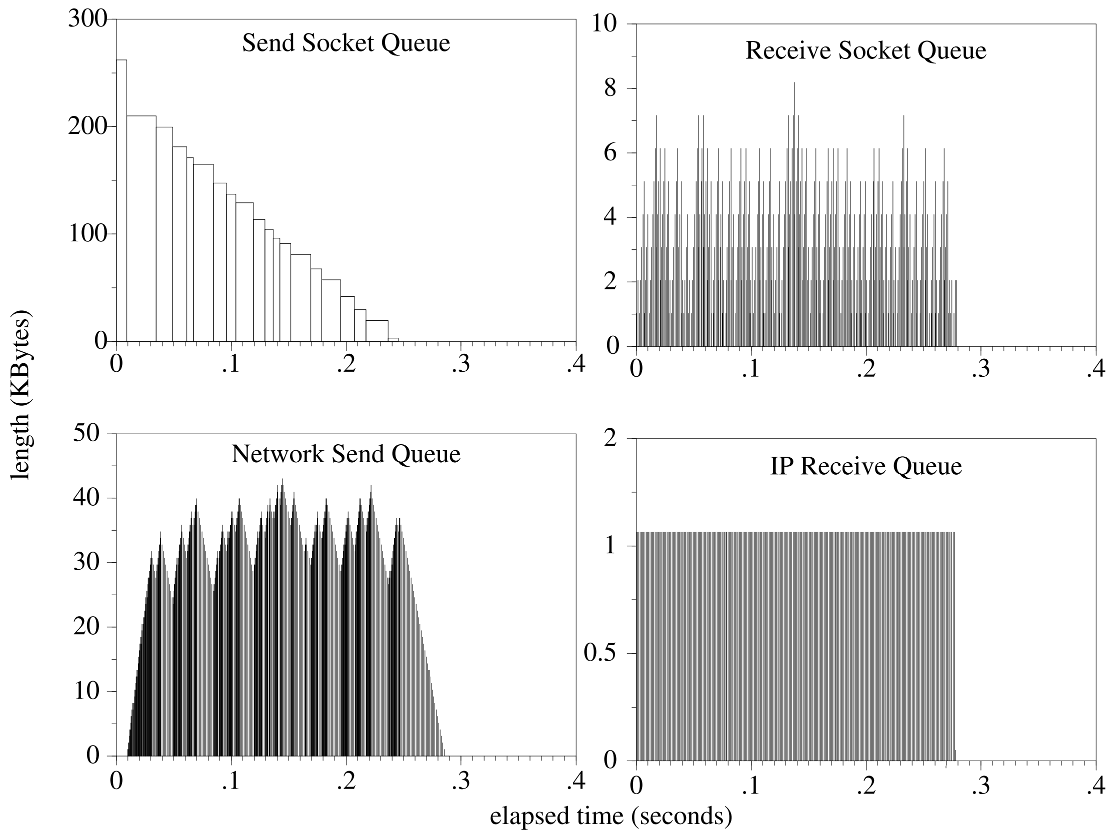

Figure 12: Queue Length after congestion window has opened

of acknowledgments). Note that the point the queue becomes empty is the point where the last chunk of application data is moved into the socket buffer and not when all data was actually transmitted. The data remains in the socket buffer until transmitted and acknowledged by the protocol.

The interface queue holds the packets generated by the protocol. It is immediately apparent that the window bursts generated by the protocol result in peaks in the interface queue length. This was suggested by the packet trace graphs and is now clearly visible. Also, the queue length increases with every new burst, indicating that a new burst contains more packets than the previous one. This is an effect of the slow start strategy of the congestion control mechanism [7].

The third queue length graph, which shows the IP receive queue, confirms the earlier hypothesis that the receive side of the protocol can comfortably process packets at the rate delivered by the Ethernet. The low Ethernet rate is a key reason why packets are not queued in the IP queue. As the graph shows, the queue practically never builds up, each packet being removed and processed before the next one comes in.

The graph for the receive socket queue shows some queueing. It appears that there are instances during data transfer where data accumulates in the socket buffer, followed by a drain of the buffer. During these instances the transmitter does not send more data since no acknowledgment was sent by the receiver. This leads to gaps in packet transmission, as witnessed earlier by the packet trace

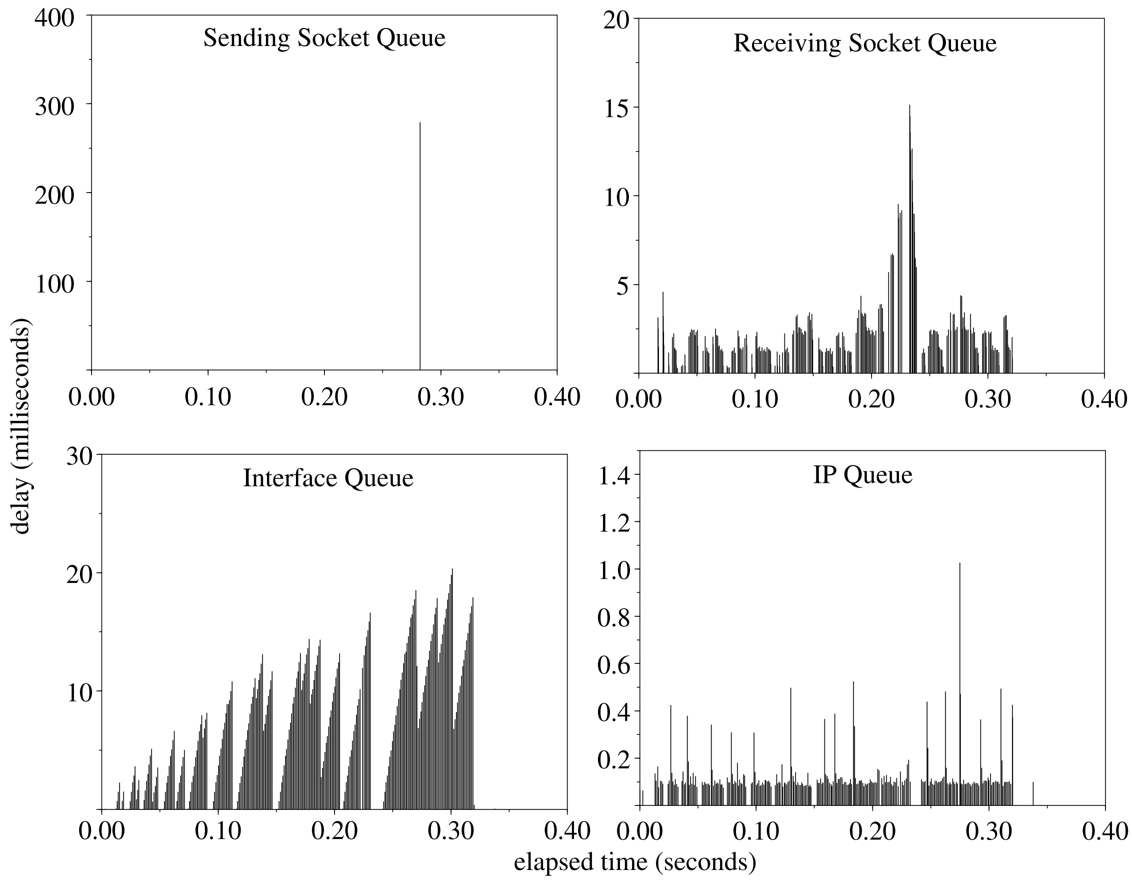

Figure 13: Queue Delay during congestion window opening

plots. These gaps are a result of the receiver delaying the acknowledgment until data is removed from the socket buffer. This is investigated further in experiment 3.

Queue Length after Congestion Window Opening

The previous queue length graphs show the four queues at the beginning of data transfer in a new connection. During this period, the congestion window mechanism is in effect, and the queues show the window is still expanding. For this reason, a second set of queue length graphs is given in Figure 12, which shows the four queues after the congestion window has opened up. The graph shows that the IPC mechanism is performing quite well. The network queue is kept almost full because of the slow Ethernet, IP queueing is still non-existent and the receive socket queue shows moderate queueing.

Queue Delay

The next set of graphs in Figure 13 shows the packet delay. The x-axis again 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.

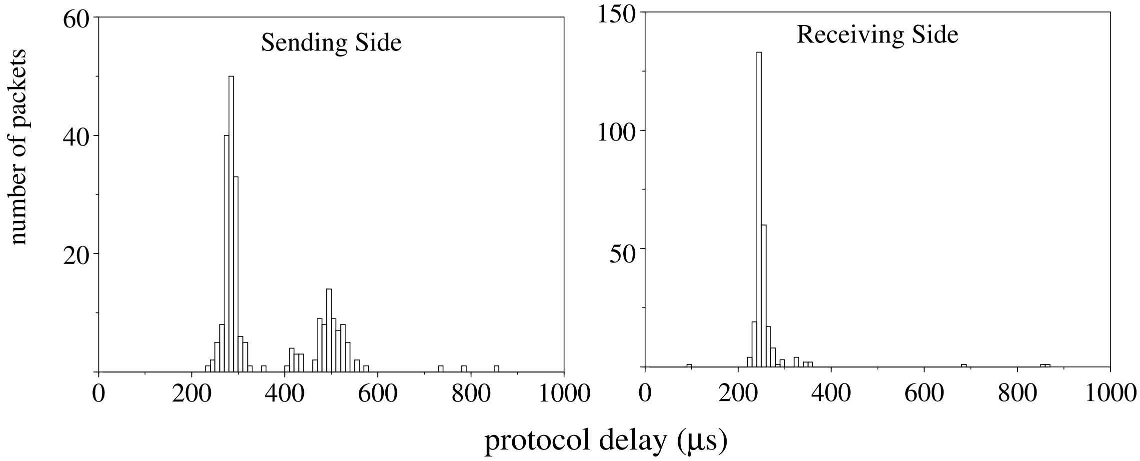

Figure 14: Protocol Processing Delay Histograms

Since the sending socket queue contains a stream of bytes (as mentioned earlier), there are no packets in this queue and thus the delay graph simply shows the time it takes for data in each application write call to pass from the application layer down to the protocol layer. For this experiment, there is a single transmit request (i.e. 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 beginning of a window burst experience very small delay, while subsequent packets are queued and therefore delayed significantly. This leads to a wide range of delays, ranging from a few hundred microseconds to several milliseconds. The third graph shows the IP queue delay. This is essentially a measure of how fast the protocol layer can remove and process packets from the IP queue. The delay of packets appears very small, with the majority remaining in the queue for about 100 to 200 microseconds. Thus most of the packets are removed from the queue well before the arrival of the next packet, which will take approximately 1 ms. The fourth graph shows the delay each packet experiences in the receive socket queue. Here the delays are much higher because they include the time to copy data from mbufs to user space. Since the receiving process is constantly trying to read new data, this graph is also an indication of how often the operating system allows the socket receive process to run. The fact that there are occasional data accumulations in the buffer, shows that the process does not run often enough to keep up with the data arrival.

Protocol Processing Delay

The two histograms in Figure 14 show the protocol processing delay at the send and receive side respectively. The x-axis is divided into bins which are 50 microseconds wide, and the y-axis shows the number of packets into each bin. For the sending side, the delay is measured from the gaps between packets 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. This problem does not exist at the receiving end where the protocol receives and forwards distinct packets to the socket layer.

The protocol delays appear to be different at the two ends: on the sending end processing delay divides the packets into two distinct groups with the larger one taking about 250 to 400 microseconds to process, and the other about 500 to 650 microseconds. At the receiving end the protocol processing is more evenly distributed taking about 250 to 300 microseconds for most packets. These graphs show that 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. From these numbers the theoretical maximum TCP/IP can attain between Sparcstation 1's with the CPU devoted to communication, excluding data copying and network driver overhead, is estimated at about 22 Mbps. It should be noted that the above numbers are approximate, but could be reproduced in several runs of the experiments.

Summary

Part 1 of experiment 1 has shown that the IPC mechanism can achieve about 75% utilization of the Ethernet. Once the TCP congestion window has opened, the IPC mechanism works quite well. However, queueing is present at the interface queue and at the receive socket queue. The former is attributed to protocol processing being faster than the Ethernet, and the latter to the inability of IPC to transfer each packet to application space as it arrives. The effects of queueing at the socket layer are examined later in experiment 3.

4.2.2 PART 2: INCREMENTAL DELAY

The previous measurements have shown the behavior of the IPC queues during large data transfer. One problem with such 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 a new experiment has been performed, where packets are sent one at a time after introducing some delay between them. The idea is to give the IPC mechanism enough time to send the previous packet and receive an acknowledgment before being supplied with the next.

The experiment is set up as follows: again a pair of Sparcstation 1's is used for unidirectional communication. The sending application is modified to send one 1024 byte packet and then sleep for 1 second to allow the packet to reach its destination and for the acknowledgment to return. To ensure that the packet is sent immediately, the TCP_NODELAY option was set. The socket buffers for both sending and receiving are set to 1 kilobyte, 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, where 256 kilobytes of data were transmitted.

The send socket queue delay shows the delay experienced by each packet before reaching the protocol is approximately 280 µS. This includes data copy from application to kernel space which was measured to be about 130 µS as shown in the last row of Table 1. Therefore, the socket layer processing takes about 150 µS. This delay includes locking the buffer, masking network device interrupts, doing various checks, allocating a plain and cluster mbuf and calling the protocol. The

Table 1: Delay in IPC components

Location	Average delay with stop-and-wait transmission(µS)	Average delay with continuous data transfer (µS)
Send socket queue (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

protocol processing delay for the sending side has increased by about 73 µS (443 vs. 370 µS). The extra overhead is due to some additional work performed. For example, the TCP output function must be called every time a packet is transmitted, whereas earlier it was called only a few times with a large transmission request; appending the packet to the socket queue involves two additional function calls, out of which one attempts to do compaction; and the probe at the output of the socket queue now records every packet adding a bit more overhead. 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. From experiments not reported in this paper, it was found that a packet takes about 1150 µS more to be transmitted (including the Ethernet transmission time) [10]. The IP queue delay is also quite small. The delay has not changed significantly from the first part of the experiment. The protocol layer processing delay at the receiving end has not changed much from the earlier result. 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 µS is 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 overhead appears to be the application wake-up delay.

Summary

Part 2 of the experiment measured the overhead of various components of IPC. Perhaps surprisingly, the socket layer overhead appears to be significant, especially on the receive side. The socket overhead is comparable to the protocol overhead.

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 Sparcstation1 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 IPC performance. At the receiving end, though the packet arrival rate is reduced by the Ethernet, there is queueing at the socket layer. This means that the socket layer on the receiving side is not able to keep up with the Ethernet rate in a burst mode. In this experiment the effect of the receive socket queueing on the protocol is further investigated.

4.3.1 EFFECT OF RECEIVE SOCKET QUEUEING

First, the actions taken at the socket layer in response to data arrival are summarized briefly. The receive function enters a loop traversing the mbuf chain in the socket queue and copying data to the application space. The loop exits when either there is no more data left to copy, or the application request is satisfied. At the end of the copy loop, the protocol is notified via the user request function that data was received allowing protocol specific actions like sending an acknowledgment to take place. Thus any delay introduced at the socket queue will delay protocol actions. Next, the effect on the generation of acknowledgments by TCP is investigated.

The TCP Acknowledgment Mechanism

TCP sends acknowledgments during normal operation when the application removes data from the socket buffer. As mentioned earlier, this may cause an acknowledgment to be sent. More specifically, 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[1].

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 is run which checks 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 immediately. 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 order to accumulate more of the forthcoming data and send a cumulative acknowledgment. Additionally, the receiver may reply to the sender soon, in which case the acknowledgment may be piggybacked onto the reply back to the sender.

To summarize, if there are no retransmissions the protocol 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, an experiment was performed monitoring the receive socket buffer and the generated acknowledgments. The experiment again had a unidirectional connection sending and receiving data as fast as possible. To isolate the protocol from the Ethernet, the experiment was

[1] As per the tcp_output () of SunOS. See [10] for details.

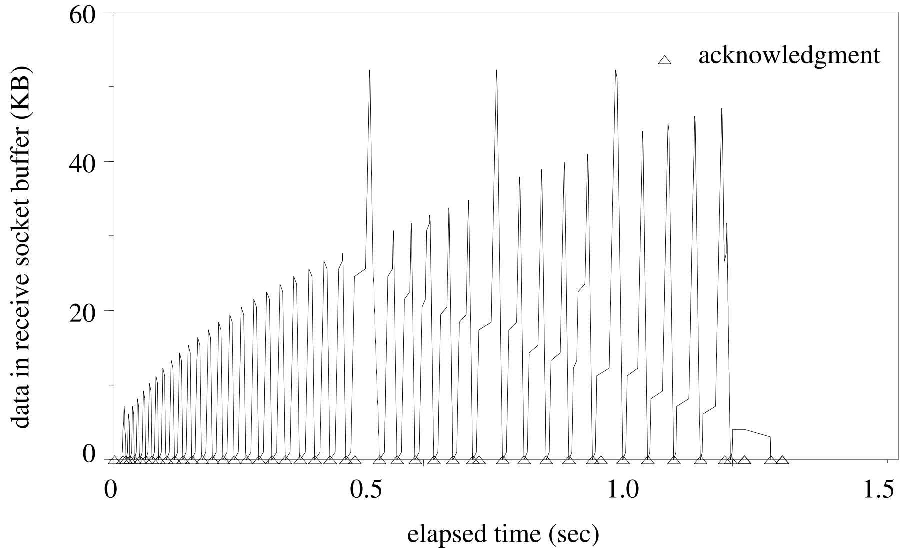

Figure 15: Receive socket queue length and acknowledgment generation with local communication

conducted using the loopback interface. The data size sent was set to 1 MByte, and the result is shown in Figure 15. This graph shows the length of socket receive buffer and the generated acknowledgments throughout the connection's lifetime. 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 ACK before the data in the receive buffer is completely removed, causing more data to arrive before the previous data was removed. However, during data transfer between different machines more acknowledgments would be generated. The receive buffer becomes empty more often during remote transfer because the receive process is scheduled more often and has more CPU cycles to copy the data.

4.3.2 THE EFFECT OF SOCKET QUEUE DELAY ON THE PROTOCOL

The above investigation shows how processing at the socket layer may slow down the protocol by affecting the acknowledgment generation. For a bulk data transfer most acknowledgments are generated when the receive buffer becomes empty. If data arrives in fast bursts, or if the receiving machine is slow, 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 be acknowledged after the data is passed to the application. Until this happens, the protocol 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 idle periods during data transfer which will be equal to the time it takes to copy out the buffer plus one round trip delay for the acknowledgment.

The delayed acknowledgment timer may cause extra acknowledgments to be generated in addition to the acknowledgments generated when the socket buffer becomes empty. Such acknowledgments cause peaks at the socket buffer. These peaks occur only during the slow start period and could conceivably interfere with the slow start behavior. If many connections enter the slow start phase together, as in the case of multiple packet loss due to a buffer overflow at a gateway, these peaks may cause some instability. This possibility however, was not investigated in these experiments.

4.3.3 SUMMARY

This experiment has shown that queueing at the receive socket layer affects the protocol acknowledgment generation. At its worst, this queueing interferes with the continuous packet flow the window mechanism attempts to produce, forcing the protocol to a stop-and-wait operation.

4.4 EXPERIMENT 4: COMMUNICATION VS. COMPUTATION

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 performance. In a distributed environment (especially with single-processor machines), computation and communication do affect each other. This experiment investigates this interaction by observing the behavior of the various queues when the host machines are (1) loaded with some artificial load, and (2) running a real application. It is aimed at determining which parts of the IPC mechanism are affected when additional load is present on the machine.

4.4.1 EFFECT OF BACKGROUND LOAD

In this part of the experiment the machines are loaded with artificial background load. The main selection requirement for the workload is 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 process that forks a specified number of child 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 a particular instance on a single machine.

The experiment is performed by first starting the artificial workload on the server machine, and then immediately performing a data transfer of 5 Mbytes. Figure 16 shows the effect of the artificial workload on communication throughput when the machines run the workload. The graph shows that the throughput drops sharply as the workload level increases which suggests that IPC requires a significant amount of CPU cycles. Thus when computation and communication compete for CPU cycles they both suffer, leading to poor overall performance.

Figure16 shows the internal queues during data transfer with a loaded server. For these graphs the data transferred is 1 Mbyte to keep the size of the graphs reasonable, and the workload level is set at 4. The x-axis shows elapsed time and the y-axis the queue length. The receive socket queue plot shows that data arriving at the queue is delayed significantly. The bottleneck is clearly shown

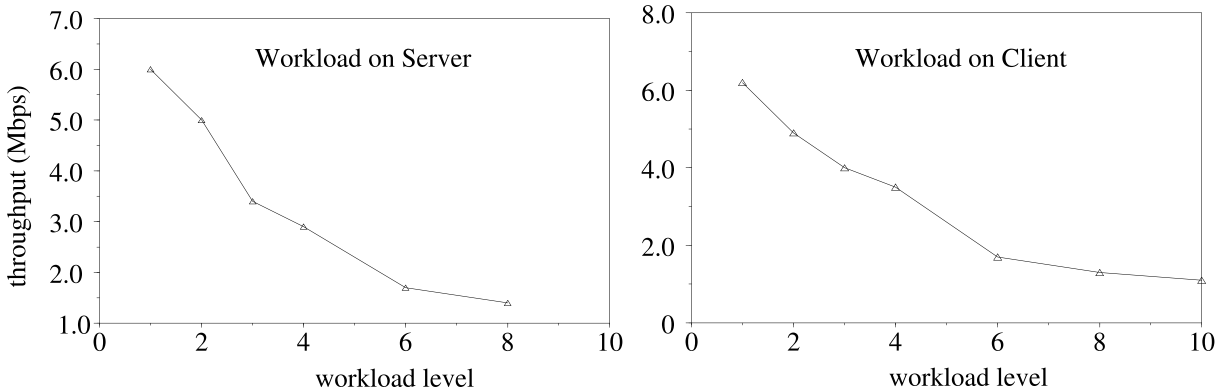

Figure 16: Throughput v.s. server and client background load

to be in the application scheduling. The receiving process has to compete for the CPU with the workload processes thus forced to read the data slower. In addition, the extra load causes data to accumulate in the IP queue. In earlier experiments with no background computation there would be no accumulation in the IP queue. The accumulation however, is still very small compared to the other queues. On the sending side the interface queue reflects the effect introduced by the delay of acknowledgments by the receiver. The queue builds up only after an acknowledgment reception triggers the transmission of a new window. When the workload is on the client, the bottleneck is again at the socket layer on the sending side. The sending process is competing with the workload and its transmission requests are delayed, forcing the data to sit in the application buffer for a longer time.

The protocol processing delay was investigated in this experiment to determine if there was any increase due to the extra load. It was found that there was about 10% increase in the delay in protocol processing on either side. This was not surprising since the protocol routines are short and run in the kernel with the network device interrupts masked.

4.4.2 RUNNING A DISTRIBUTED APPLICATION

The second part of the experiment examines the effect of the Unix scheduling mechanism on a stage in a pipelined distributed application developed in a separate part of this study (details can be found in [10]). Each application stage runs on a single host and consists of three processes: (1) receive data from the previous stage, (2) compute, and (3) transmit to the next stage. Figure 18 depicts the duty cycles of the three processes. The x-axis shows the elapsed time, and the y-axis the process duty cycles as on/off states. A process is in the “on” state when it has work to do, and in the “off” state otherwise. The graph shows the activity of the three processes after steady state is reached. The processing cycle appears regular: first there is a short read followed by computation and finally a short write. The write process appears to start immediately after the compute process and the read process immediately after the write process. Thus the short communication processes are scheduled immediately when they have work to do, while the long compute process gets less CPU cycles.

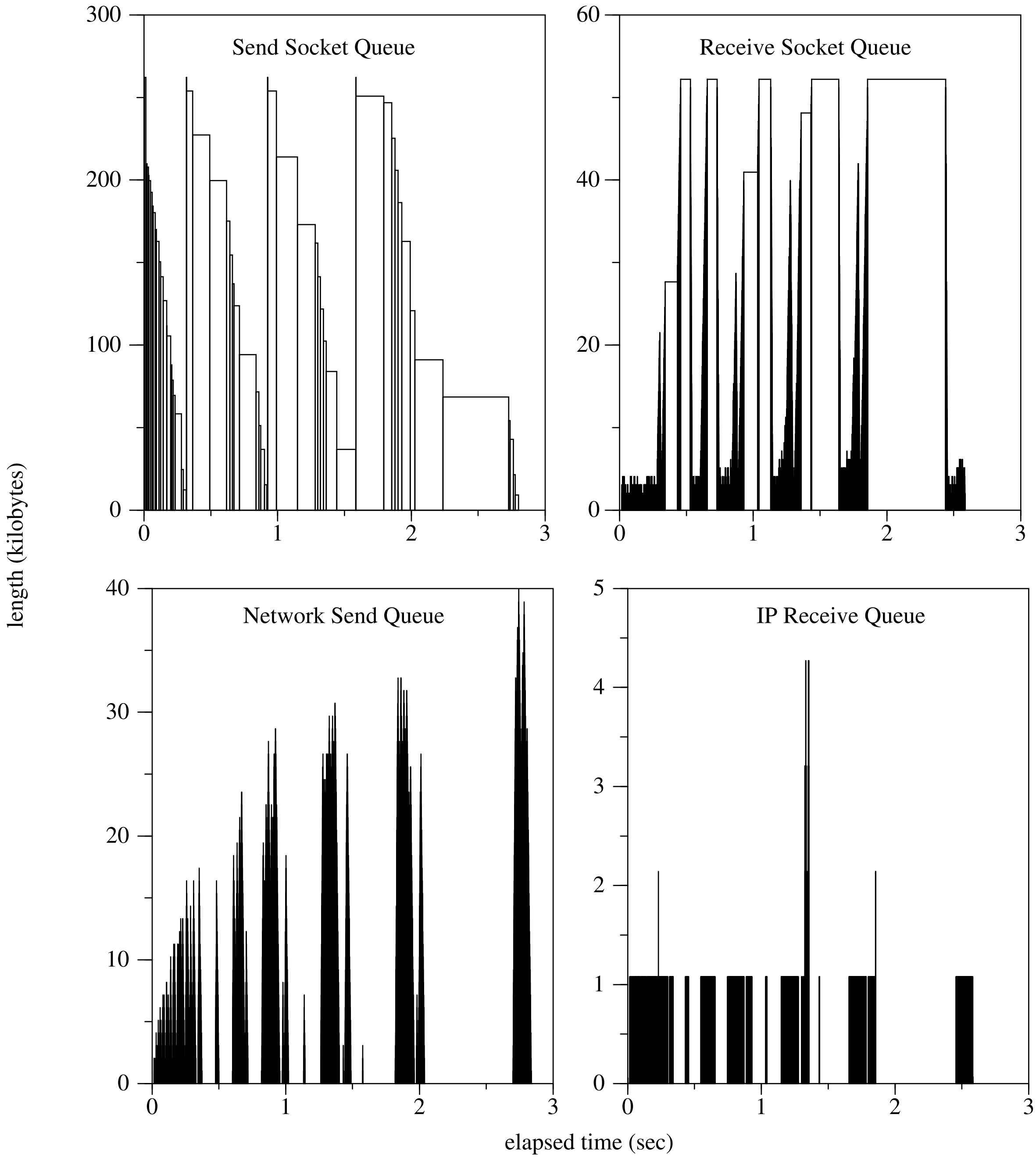

Figure 17: Queue behavior with server background load

4.4.3 SUMMARY

In the presence of background load, the Unix scheduling mechanism affects communication throughput considerably. The mechanism is designed to favor short processes at the expense of long intensive ones. The reason is that it is desirable to provide interactive processes with a short response time to avoid user frustration. This policy has considerable effect on response time as perceived by different user processes. The reason is that as a process accumulates CPU time, its pri-

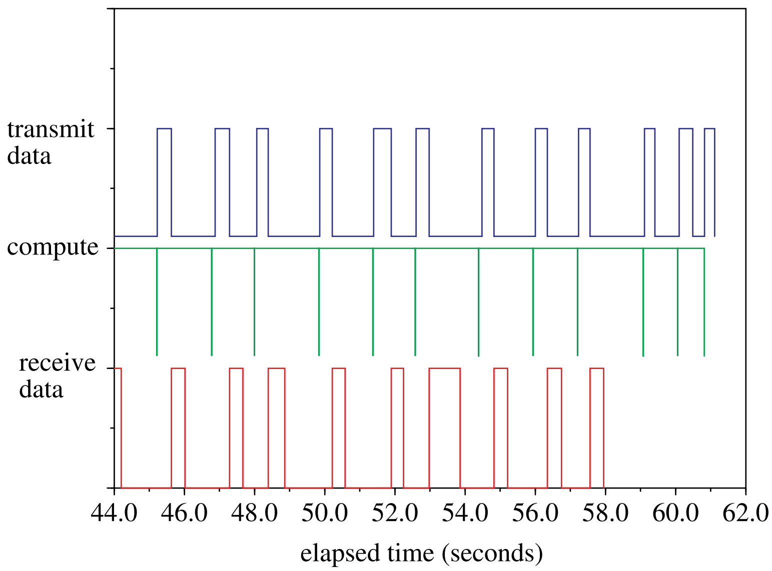

Figure 18: Process activity and send interface queue

ority is lowered in order to prevent it from monopolizing the CPU. If computation requires more CPU time than communication (which is a likely case), whenever data becomes available for transmission or reception, the communication process may run at a higher priority than the computing process since it was idle. Thus in the presence of extra computation, the CPU allocation policy favors the communication processes which run at near maximum speed while the computation process suffers.

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 protocol 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

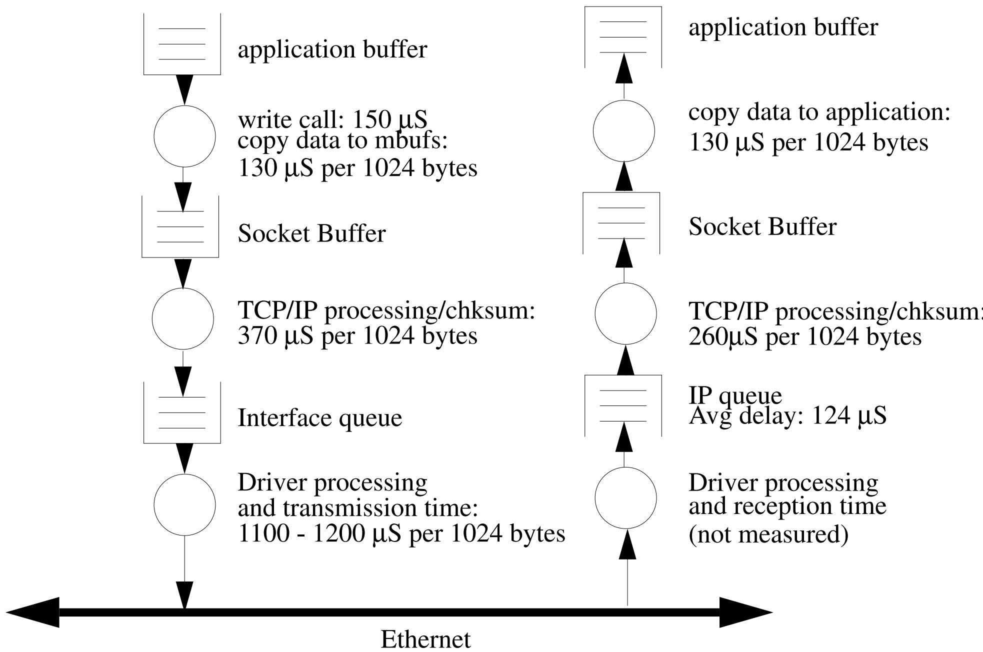

Figure 19: Delays in IPC

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.

The sending side of IPC is able to process packets faster than the Ethernet rate, which leads to queueing at the network interface queue. The lower layers of the receiving side are able to process packets at the rate they arrive. At the socket layer however, packets are queued before being delivered to the application. Figure 19 shows the distribution of delays in the various IPC components. The estimated performance of TCP/IP determined by the protocol processing and checksum and ignoring data copy was measured to be about 22 Mbps.

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 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 for the new data to arrive. This is a small problem on the current implementation where the socket queues are small, but may be significant on future high bandwidth-delay networks where socket queues may be very large. As shown in experiment 4, 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 [8]. 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 [6]. We and other groups are currently evaluating these options. Moreover, 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] Dhas, Chris, Konangi, Vijay, and Sreetharan, M.,‘‘Broadband Switching Architectures, Protocols, Design, and Analysis,’’ IEEE Computer Society Press, Los Alamitos, California, 1991.

[6] D. Sanghi, et al, “A TCP Instrumentation and its use in Evaluating Roundtrip-Time Estimators”, *Internetworking: Research and Experience*, Vol 1, pp. 79-99, 1990.

[7] Jackobson, Van, “Congestion Avoidance and Control”, *SigComm ‘88, Symp., ACM*, Aug. 1988, pp. 314-329.

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

[9] 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.

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

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

[12] J. Postel, “Transmission Control Protocol”, USC Inform. Sci. Inst., Rep. RFC 793, Sept. 1981.

[13] 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.

[14] Liu, Paul (pcl@ihlpz.att.com) private email, Feb 20, 1992.

[15] 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.

[16] 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.