Report Number: WUCS-92-23

1992-01-01

Experimental Evaluation of SUNOS IPC and TCP/IP Protocol Implementations

Christos Papadopoulos and Gurudatta M. Parulkar

Progress in the field of high speed networking and distributed applications has led to the debate in the research community on suitability of existing protocols such as TCP/IP for emerging applications over high speed networks. Protocols have to operate in a complex environment comprising of various operating systems, host architectures, and a rapidly growing and evolving internet of large number of heterogeneous subnetworks. Thus, evaluation of protocols is definitely a challenging task and cannot be achieved by studying protocols in isolation. This paper presents results of a research project that was undertaken with the following objectives: 1) Characterization of... **Read complete abstract on page 2.**

Experimental Evaluation of SUNOS IPC and TCP/IP Protocol Implementations

Christos Papadopoulos and Gurudatta M. Parulkar

Complete Abstract:

Progress in the field of high speed networking and distributed applications has led to the debate in the research community on suitability of existing protocols such as TCP/IP for emerging applications over high speed networks. Protocols have to operate in a complex environment comprising of various operating systems, host architectures, and a rapidly growing and evolving internet of large number of heterogeneous subnetworks. Thus, evaluation of protocols is definitely a challenging task and cannot be achieved by studying protocols in isolation. This paper presents results of a research project that was undertaken with the following objectives: 1) Characterization of the performance of TCP?IP protocols for communication intensive applications. Components to be studied include control mechanisms (such as flow and error control), per-packet processing, buffer requirements, and interaction with the operating system, by systematic measurement and extrapolation. 2) An attempt to investigate the scalability of existing protocols to future networks and applications.

Experimental Evaluation of SUNOS IPC and TCP/IP Protocol Implementations

Christos Papadopoulos and Gurudatta M. Parulkar

WUCS-92-23

July 1992

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

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.

EXPERIMENTAL EVALUATION OF SUNOS IPC AND TCP/IP PROTOCOL IMPLEMENTATIONS[*]

Christos Papadopoulos
christos@dworkin.wustl.edu
+1 314 935 5087

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 led to the debate in the research community on suitability of existing protocols such as TCP/IP for emerging applications over high speed networks. Protocols have to operate in a complex environment comprising of various operating systems, host architectures, and a rapidly growing and evolving internet of large number of heterogeneous subnetworks. Thus, evaluation of protocols is definitely a challenging task and cannot be achieved by studying protocols in isolation. This paper presents results of a research project that was undertaken with the following objectives:

- Characterization of the performance of TCP/IP protocols for communication intensive 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 and extrapolation.
- An attempt to investigate the scalability of existing protocols to future networks and applications.

[*]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 [5]. These efforts suggest that networks that can support data rates of a few 100 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, such applications require not only high speed networks but also carefully engineered end-to-end protocols and their implementations within the constraints of a given operating system and host architecture. There has been considerable debate in the research community regarding suitability of existing protocols such as TCP/IP for emerging applications over high speed networks [3,13,14]. One group of researchers believe that existing protocols such as TCP/IP are suitable and can be adopted for use in high speed environments [4,11]. Another group claims that the TCP/IP protocols are complex and their control mechanisms are not suitable for high speed networks and applications [1,2,7,16,17]. It is important, however, to note that both groups agree that efficient protocol implementation and appropriate operating system support are essential.

Precools operate in a complex environment. For example, the underlying communication substrate is in fact a constantly evolving internet of a large number of heterogeneous networks; operating systems that incorporate protocol implementations are complex systems with a number of interfering components such as memory management, interrupt processing, process scheduling, and others; and performance of protocol implementations is also dictated by the underlying host architecture. Thus, evaluation of protocols such as TCP/IP for newer applications over high speed networks is definitely a challenging task and cannot be achieved by studying protocols in isolation. This paper presents results of a research project that was undertaken with the following objectives:

- Characterization of the performance of TCP/IP protocols for communication intensive 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 and extrapolation.
- An attempt to investigate the scalability of existing protocols to future networks and applications.

There are some important reasons for measuring performance of a complex system such as the interprocess communication (IPC) implementation on TCP/IP protocols in Unix: (1) it forces the acquisition of a fairly detailed understanding of the kernel internals and protocols; (2) and it allows the assessment of the performance of the real system with all the pieces in place and fully functional, possibly revealing unexpected interactions that could not be seen by studying the pieces individually

The test environment consists of SunOS 4.0.3 IPC using stream sockets on top of TCP/IP. The hardware used were two Sparcstation 1 workstations connected on the same Ethernet segment using the AMD Am7990 LANCE Ethernet Controller. Occasionally, two Sparcstation 2 workstations running SunOS 4.1 were used in the experiments. However, only SunOs 4.0.3 IPC could be thoroughly studied due to lack of source code for SunOS 4.1.

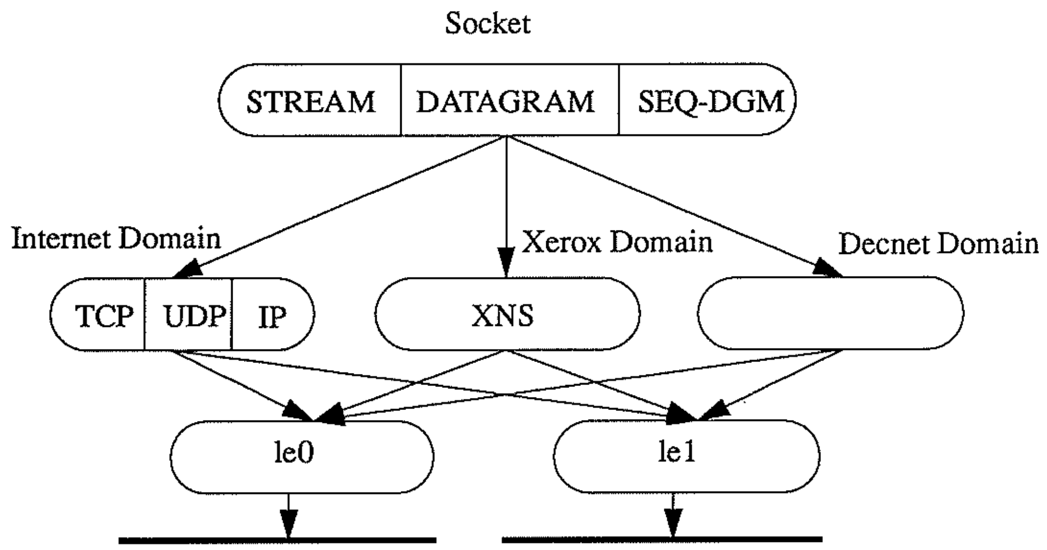

Figure 1: Inter-process Communication (IPC) layering

The paper is organized as follows: Section 2 presents an overview of protocol implementation and IPC model of the Unix kernel. Section 3 presents details of kernel probing which was used for measurements reported in this paper. Section 4 summarizes other tools aimed at protocol evaluation using measurements. Section 5 presents a number of measurement experiments. Each experiment is aimed at characterizing performance of IPC, OS, and TCP/IP under certain pre-specified conditions. Finally Section 6 presents important conclusions of this study including some scalability arguments for TCP/IP and their existing implementation model.

2. UNIX KERNEL BACKGROUND

This section presents an overview of the implementation of IPC using TCP/IP stream sockets in SunOS 4.0.3[†]. The focus of the description is on the internal structure and organization of the IPC mechanism and the underlying protocols. This overview is necessary in order to:

- Help better understand the operation of the IPC mechanism, the protocols and the operating system internals
- Identify the various queues encountered along the data path and use these to locate probes
- Study the performance of IPC and identify any bottlenecks

The study of the performance is the focus the rest of this paper. This section deals with the first two issues.

2.1 INTER-PROCESS COMMUNICATION LAYERS

IPC is organized into 3 layers as shown in Figure 1. The first layer, the socket layer, is the IPC interface

[†]SunOS 4.0.3 is based on 4.3 BSD Unix, and for details on BSD Unix see [15].

to applications. The socket layer has two main purposes: (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 application can create and use to send and receive data. Sockets appear as file descriptors to the application, and the usual functions that operate on file descriptors were extended to operate on sockets. The socket layer supports a number of different types of sockets each of which provides 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.

Depending on application requirements, the socket layer can choose a protocol from the family of protocols the application has selected, which has the required properties. This collection of protocols form the protocol layer. Moreover, protocols are grouped in domains. For example all Internet protocols are grouped together in the Internet domain. The application can specify the protocol it wishes to use, or leave that selection to the socket layer. Currently, the main protocols of the Internet protocol domain are TCP, UDP and IP. Other domains, like Xerox NS and Decnet, provide other protocols.

The third layer of IPC is the network interface layer. Hardware device drivers are part of this layer. This is where interaction with the network hardware occurs. A host may have more than one network interface, particularly if it is acting as a gateway. As part of routing, the protocols decide on which network interface to output their packets on.

To summarize, the Unix IPC facilities are implemented in three layers. The first layer is the socket layer, which provides endpoints that provide certain types of service. The second layer consists of the protocols that implement those services. Finally, the third layer comprises of a collection of interfaces to the networks to which the host is attached. Sockets, when created are bound to a protocol, which in turn will choose an interface to send data. An example configuration would be the application requesting a stream socket in the Internet domain, the socket layer choosing TCP, and communication taking place over the local Ethernet interface.

2.2 MEMORY MANAGEMENT: MBUFS

Interprocess communication and network protocols impose special requirements on memory allocation because of special needs. Both fixed and variable size blocks of memory are required. Fixed size blocks are needed for data structures like control blocks, and variable size blocks for storing packets. Moreover, efficient allocation and deallocation of such blocks is essential since such operations occur very frequently (at least once on every packet reception). These reasons motivated the IPC developers in 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 and the routines used to manipulate the mbuf memory. The mbuf memory is created during system initialization, where a part of the kernel memory is permanently allocated to the mbuf memory management. Mbuf memory is always in main memory and is never paged out.

Mbufs are fixed size memory blocks 128 bytes long. Up to 112 out the 128 bytes can be used for storing data. The remaining 16 bytes contain control fields which are used to manipulate both the data and the mbufs. The structure of an mbuf is shown in Figure 2.

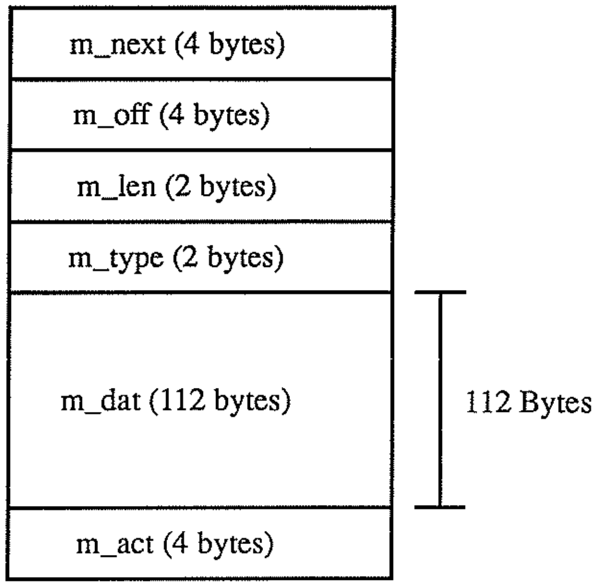

Figure 2: Mbuf Data Structure

These mbufs are called small mbufs because the amount of data that can be stored internally is limited to 112 bytes. Another type of mbuf, called a cluster mbuf is also available, where data is stored externally in a page associated with each particular cluster mbuf. Pages in SunOS 4.0.3 are 1024 bytes long. Data in cluster mbufs are stored exclusively in the associated page. The internal space of a cluster mbuf is never used for data.

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

2.3 IPC FUNCTION CALLS FOR DATA EXCHANGE

There are a fair number of functions involved in exchanging data. Figure 3: shows the functions involved in the order which they get invoked. Also shown are their locations with regard to layering. The arrows in the figure imply the sequence in which functions get called. The data flow is implied by the system call used on each side: write() sends the data, and read() receives it.

2.4 QUEUEING MODEL

Figure 4 shows the data distribution into mbufs at the sending side. In the beginning, data reside in application space in contiguous buffer space. Then data are copied into mbufs in response to a user send request, and are 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 are queued in this buffer until acknowledged. The pro-

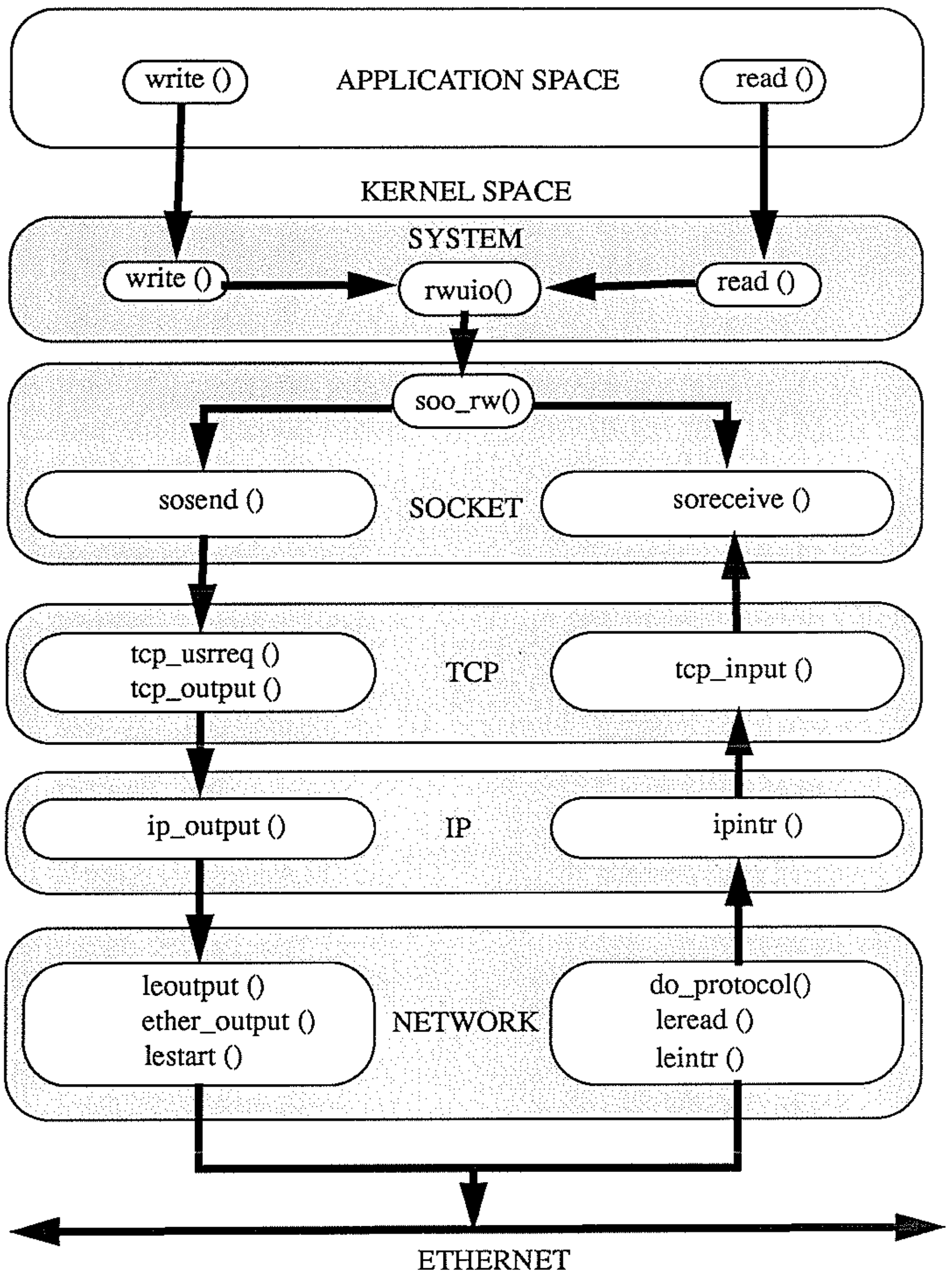

Figure 3: Function call sequence

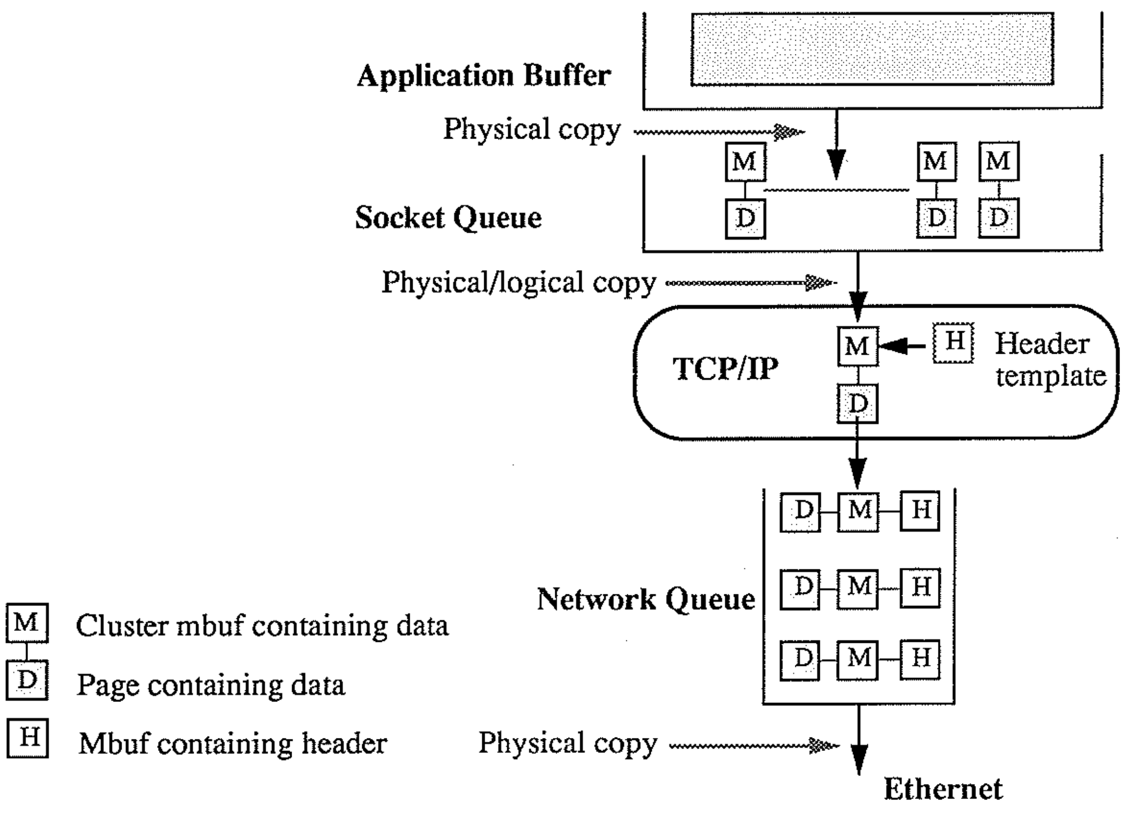

Figure 4: Data distribution into Mbufs

tocol retrieves data equal to the size of available buffering minus the unacknowledged data, breaks it in packets, adds the appropriate headers and passes them 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, when packets arrive at the network interface the ethernet header is stripped, and the remaining packet is appended to the protocol receive queue. The protocol is notified via a software interrupt, which 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 socket receive queue and wakes up the application. The four queueing points identified above are depicted in Figure 5.

3. PROBE IMPLEMENTATION

To monitor the queue activity, probes were inserted in the SunOS Unix network code. This section describes issues relevant to probe design and implementation. First, each probe's location is presented, with a brief description of what part of IPC the probe is monitoring. Then the parameters recorded by the probes are listed. The issue of minimizing probe overhead is discussed next. Then probe activation is discussed, and finally, the internal structure of the probes is presented.

3.1 Probed Parameters

Each probe when activated records a timestamp, and the amount of data passing through its checkpoint.

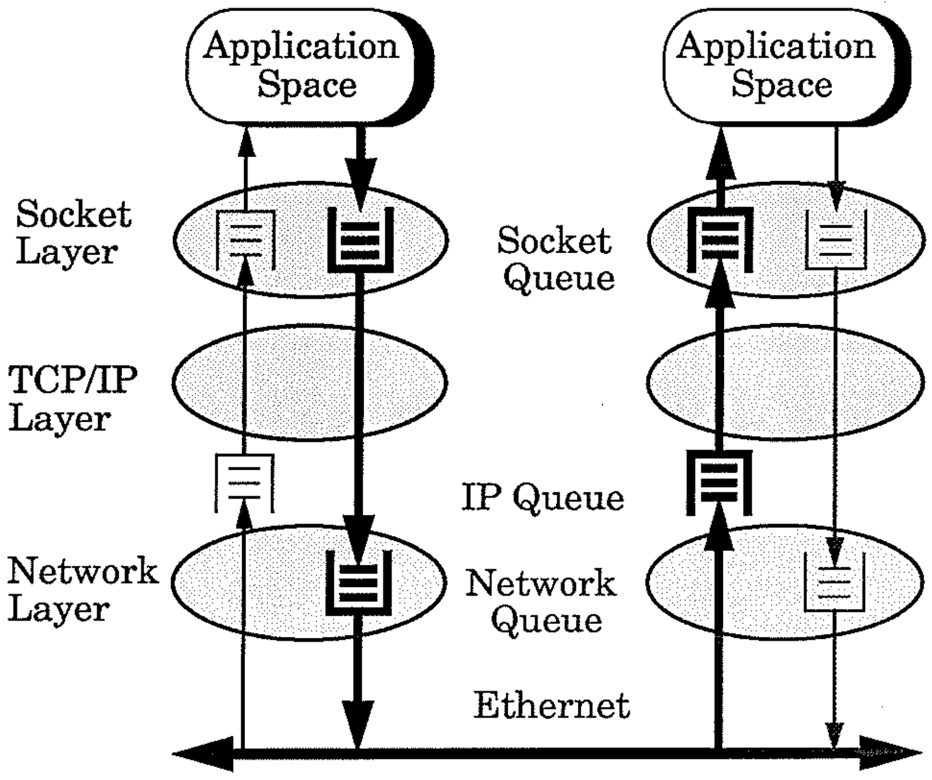

Figure 5: Queueing Model

It also fills a type field to identify which probe produced the record. The structure of a record is shown in Figure 6.

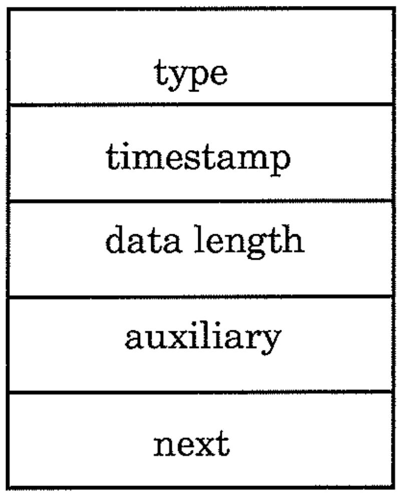

Figure 6: Structure of a record produced by a probe

The auxiliary field is used to record other information that might be required. 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 fragmentation occurs.

3.2 Probe Location

After identifying the location of the queues in the network code, the position of the probes in the code was determined. The selected probe locations are depicted in Figure7.:

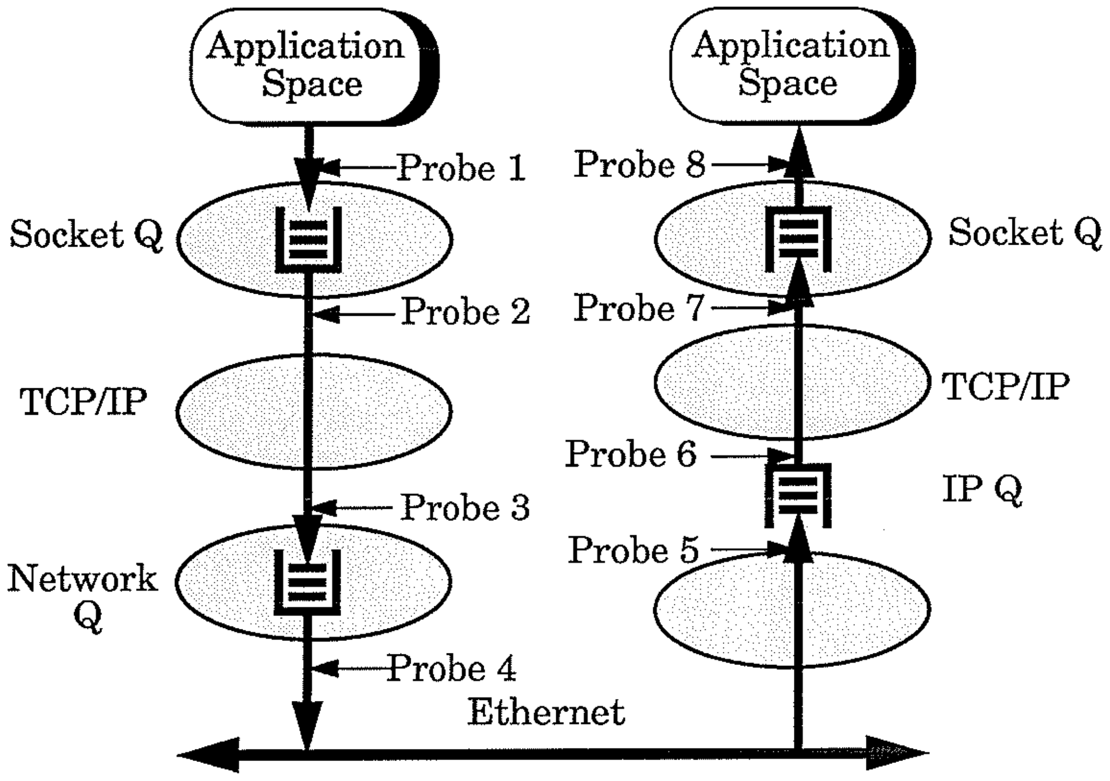

Figure 7: Probe Location

The following paragraphs explain how the probes are used to monitor the queues and compute various important statistics.

3.2.1 Send Socket Queue

Probes 1 and 2 monitor the sending activity at the socket layer. The records produced by probe 2 show how data is broken up into transmission requests. The records produced by probes 1 and 2 can be used to plot queue delay and queue length graphs for the socket queue.

3.2.2 Send Protocol Processing Delay

Probe 3 monitors the rate packets reach the interface queue. This rate is essentially the rate packets are processed by the protocol. The probe 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 inter-packet gap during a burst.

3.2.3 Send Interface Queue

The interface queue delay is given as the difference in timestamps between probes 3 and 4. The queue length at various instances can also be obtained from probes 3 and 4, as the difference of packets logged by the two probes.

3.2.4 Receive IP Queue

On the receiving side, probes 5 and 6 monitor the activity of the IP queue, providing a measure of 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 can be determined using probe 5. This rate is strongly dependent on Ethernet load.

3.2.5 Receive Protocol Processing Delay

The difference in timestamps between probes 6 and 7 will give the protocol processing delay.

3.2.6 Receive Socket Queue

Probes 7 and 8 monitor the socket receive buffer activity. Packets arrive in this queue after the protocol has process them, and depart from the queue as the application reads them. Probes 7 and 8 can give the socket receive queue delay and length.

3.2.7 Monitoring Acknowledgments

The probes can also monitor acknowledgment activity. 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 half of the probes will be recording acknowledgments.

3.3 Probe Operation and Overhead

An issue of vital importance is that probes incur as little overhead as possible to the normal operation of the network code. There are three kinds of overhead associated with probe operation:

1. overhead due to probe execution time
2. overhead due to the recording of measurements
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 source, it was decided that probes should not write records to a file, but rather should store records in the kernel virtual space. To accomplish this, a circular list of empty records residing in kernel space is used as depicted in Figure 8. The list is initialized at system

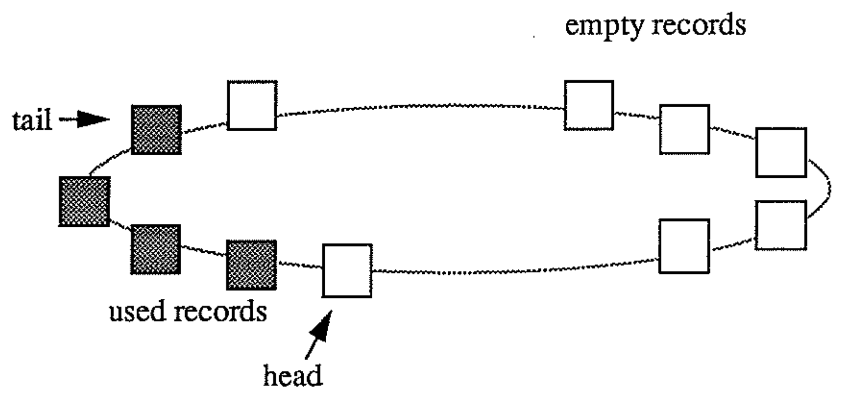

Figure 8: Kernel Circular List holding records

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 the probes. The head pointer points to the first empty record on the list and the tail pointer to the last record in use.

The third source of overhead, probe activation, refers to determining 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 mechanism is described next.

3.3.1 Probe Activation

The probes must be activated by the application with a call to 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, broadcasting, etc. The setsockopt() call was extended by adding another socket option named SO_LOG. An application wishing to activate logging on a connection calls setsockopt() with this option. In the current implementation, logging initiated at the sending end will automatically initiate logging at the receiving end. The receiver can also initiate logging on its own side if it wishes so, but in this case only the socket layer and network interface probes will be activated. The IP probes rely on information sent by the sending end to be activated.

Setting the logging option on a socket has two effects: first, a flag is set in the socket structure marking the socket as logging. Then, in the protocol layer a flag is set in the IP packet header so that probes can identify packets at the lower layer. The reasons for this double marking are 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. However, this still leaves the problem of notifying the network layer that a packet should be logged. 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 through that locate the socket structure. However, this requires performing a search for each outgoing packet, which would be too expensive to be practical. Instead, it was decided that the packet itself should carry the information to activate the probes. This technique has the significant benefit of solving the problem of probe activation at the network layer at the receiving side.

The logging information is carried in the IP header of an outgoing packet. The IP header contains a field called "tos" (type of service) which has 2 bits currently not used. IP implementations simply ignore them. One of those bits is used to carry the logging information. The network layer is able to 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. 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.

As already mentioned, initiating logging from the sending side will activate all the probes on both the sending and receiving end. The reason for this is that sometimes the receiver was not able to activate its probes before the arrival of the first 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 packet sent for connection establishment has the logging flag set.

3.3.2 Probe Internal Structure

The code for a typical probe is shown in Figure 9, and 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" statement. Probes having no access to the socket structure would test the IP header "tos" field. Then the probe masks processor 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 processor interrupts are unmasked after obtaining the timestamp since the remaining operations are not critical.

4. OTHER MONITORING TOOLS

There are two other public domain tools that can be used to monitor network conversations and extract information. These are tcpdump and NETMON. Tcpdump is quite different from our probing mechanism, and does not provide the same level of information. NETMON is quite similar to our probes, but does not provide all the information we require. Moreover, it was not written with the purpose of monitoring the internal activity of all the communication layers and their interaction with the kernel. The tools and their differences with our probing mechanism are discussed next.

4.1 TCPDUMP

Tcpdump[‡] is a network monitoring tool that captures packet headers from the Ethernet. The captured headers are filtered to display only those of interest to the user, or if desired, the packet

[‡]Tcpdump is a tool developed at Lawrence Berkeley Laboratory by Van Jacobson, Steven McCanne and Graig Leres. It is based on Sun's Etherfind

```
#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 9: Code for a typical probe

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 for each header. A typical setup using tcpdump is shown in Figure 10.

Two important advantages of tcpdump are that it does not interfere with the communication since it runs on a separate machine, and it does not require any changes to the kernels of the machines involved (some minor changes might be required to the kernel of the machine on which tcpdump runs on). The headers can be dumped either in a file or on the screen. A sample trace generated by tcpdump is shown below:

```
root_flora[29]# tcpdump ip host zen and patti
1524 packets received by filter
138 packets dropped by kernel
14:26:34.215034zen.1409 > patti.2000: S 1827328000:1827328000(0) win 4096 <mss
1460>
14:26:34.215869   patti.2000   >   zen.1409:   S   1419328000:1419328000(0)   ack
1827328001 win 4096 <mss 1460>
14:26:34.216717 zen.1409 > patti.2000: . ack 1 win 4096
14:26:34.218431 zen.1409 > patti.2000: . 1:1461(1460) ack 1 win 4096
```

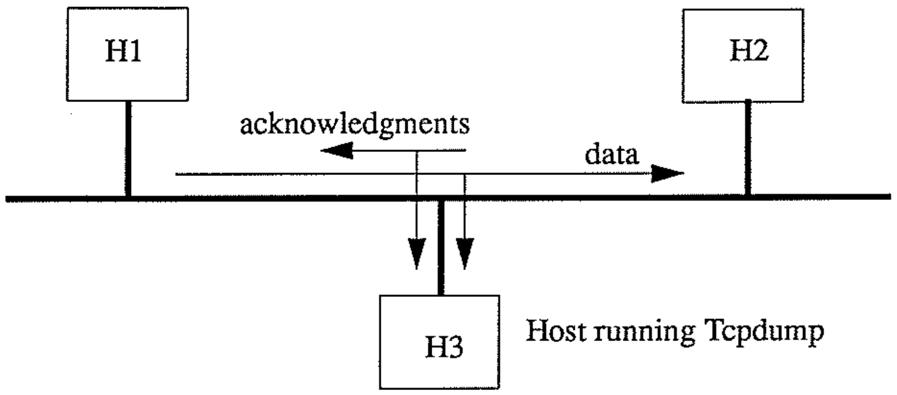

Figure 10: Capturing packets with Tcpdump

```
14:26:34.219856 zen.1409 > patti.2000: . 1461:2921(1460) ack 1 win 4096
...
...
14:26:34.455187 zen.1409 > patti.2000: . 247917:249377(1460) ack 1 win 4096
14:26:34.456349 zen.1409 > patti.2000: . 249377:250837(1460) ack 1 win 4096
14:26:34.457762 zen.1409 > patti.2000: . 250837:252297(1460) ack 1 win 4096
14:26:34.458201 patti.2000 > zen.1409: . ack 249377 win 16384
14:26:34.481345 patti.2000 > zen.1409: F 1:1(0) ack 262146 win 16384
14:26:34.482184 zen.1409 > patti.2000: . ack 2 win 4096
```

Our experience has shown that to obtain accurate timestamps, the third machine must be at least as fast or faster than the machines being monitored. Packet traces created by tcpdump are very useful in examining protocol-to-protocol communication in order to detect possible aberrations. The trace can be used to generate a graph to visualize the packet activity. However, no information about internals of the IPC mechanism can be extracted from such packet traces. Tcpdump presents only an external view of the communication, which is filtered by the network, and thus cannot provide information about the internal queues. For example, the burstiness of the windowing mechanism is almost completely invisible. In addition, although the time the acknowledgments reached the network can be observed, no information exists about when they were actually generated by the protocol[**]. Information such as window burstiness and acknowledgment generation is necessary when trying to evaluate the performance of the IPC implementation.

4.2 NETMON

NETMON[††] is a system for recording network protocol information for monitoring and/or

[**]It is possible that there may be a delay 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.

[††]Developed by Allison Mankin of MITRE Corp.

analysis, that runs under Unix BSD 4.3. NETMON is similar to our probing mechanism. It employs probes in the kernel code to record information about network protocol activity, and a user process to recover the information into a file. However, NETMON monitors only IP level and interface queue information, whereas our probing mechanism adds probes to higher layers. 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 at providing information on network protocol activity. It can also be used to monitor gateway activity. Since it concentrates on the network layer, it does not provide any information on the activity of higher layers, like the socket layer.

4.3 REMARKS

The above network monitoring tools concentrate on monitoring some specific layers of communication and therefore do not emphasize monitoring the mechanism as a whole. A probing tool that monitors all the communication layers as a unit during operation will provide more insight into the internals of IPC. Such a tool can be used to investigate the interaction among the different components to bring out aspects that may not become visible by examining the components individually.

5. PERFORMANCE OF IPC

This section presents a number of experiments aimed at characterizing performance of various components of IPC, including the underlying TCP/IP protocols. Note that these experiments evaluate the performance of SunOS 4.0.3 IPC using stream sockets on top of TCP/IP. The hardware used were two Sparcstation 1 workstations connected on the same Ethernet segment using the AMD Am7990 LANCE Ethernet Controller. Occasionally, two Sparcstation 2 workstations running SunOS 4.1 were used in the experiments. However, only SunOs 4.0.3 IPC could be thoroughly studied due to lack of source code for SunOS 4.1.

To avoid interference from users or the network, the experiments were performed with no other users on the machines (but still in multi-user mode). During the experiments, the Ethernet was monitored using a tool capable of displaying the Ethernet utilization, either Sun's *traffic* or *xenetload.* With these tools it was ensured 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.

5.1 EXPERIMENT 1: THROUGHPUT

The 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 by extrapolation, for applications that will be running on faster networks in the future. In the second part, the performance for two IPC configurations is compared: in the first, processes are running on two physically separate machines, and in the second, 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 less than the maximum Ethernet rate.

5.1.1 Throughput and Socket Buffer Size

There are two distinct socket buffers associated with each socket: one for sending and one for receiving as shown 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. The minimum of the buffer sizes is the maximum window TCP can use.

Figure 11 shows various plots of 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[*]. Plots are shown for three configurations: (1) processes on two Sparcstation 1 workstations communicating over the Ethernet; (2) processes on a single Sparcstation 1 communicating through the loop-back interface; (3) and processes on two Sparcstation 2 workstations communicating over the Ethernet. The unexpected dive in throughput for the third plot around 32 kilobytes 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 an explanation [REF] of this phenomenon. The unavailability of source code for SunOS 4.1 prevented further investigation of this problem[†].

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

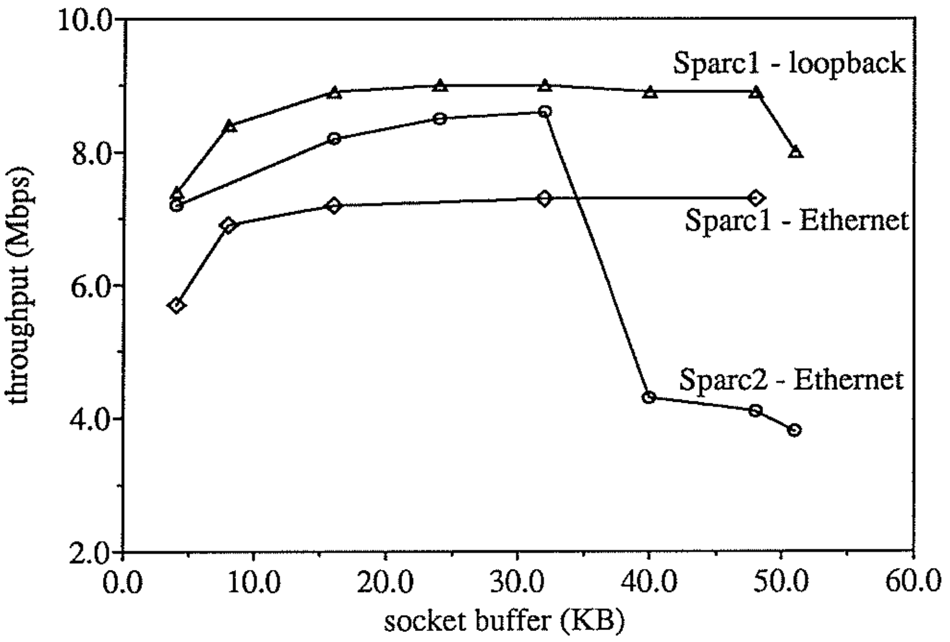

Figure 11: Throughput vs. Socket Buffer Size

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 socket buffers (to 16 kilobytes), followed by some smaller subsequent improvement.

Observations

Increasing the socket buffer size is beneficial for existing IPC. Moreover, it will be required in order to effectively use TCP over high bandwidth networks because the window used by TCP is directly affected by the size of these buffers. Networks with high bandwidth-delay product store large 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 the window cannot exceed 4096 bytes (which corresponds to 4 packets). On the receiving end the protocol receives four packets with every window, which leads to a kind of stop-and-wait protocol‡ with four segments.

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. 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 the maximum allowable socket buffer size, which is approximately 51 kilobytes. Any deviations will be clearly stated in the text.

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

5.1.2 Performance of Local vs. Remote IPC

Local communication happens when the communicating processes reside on the same machine The IPC interface is designed so that the location of the peer is transparent to processes. For this reason the socket and protocol layers of the IPC mechanism for both local and remote communication are the same. Only the network layer is different. The protocol layer is not aware of this fact because in the implementation the local communication is disguised as a network interface. This 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 as it would to other network interfaces.

Figure 11 shows the throughput over the loopback interface vs. the socket buffer size. The throughput reaches rates close to 9 Mbps. The result implies that a machine running only one side of the communication can comfortably reach rates which are 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 11, shows that with these machines rates up to 8.7 Mbps are indeed possible over the Ethernet. It is important to note that in this experiment the Sparcstation 2 machines running SunOS 4.1 were also using 1 kilobyte packets (as opposed to 1460 which is the default) to make the two cases comparable.

5.1.3 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 Sparcstation 1 workstation 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 reasons are investigated in subsequent experiments.

5.2 EXPERIMENT 2: INVESTIGATING IPC WITH PROBES

In 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 part consists of setting up a connection and sending data as fast as possible in one direction. The data size used for this part was 256 KB. In the second part of the experiment, a unidirectional connection was set up again, but packets are sent one at a time in order to isolate and measure the packet delay at each layer.

5.2.1 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**. The results are presented in four sets of graphs: the first set is a packet trace of the connection; the second is the queue length of the four queues; the third is the delay in the queues; and the last is the protocol processing delay.

**Data collected with the probes allows the detection of retransmissions.

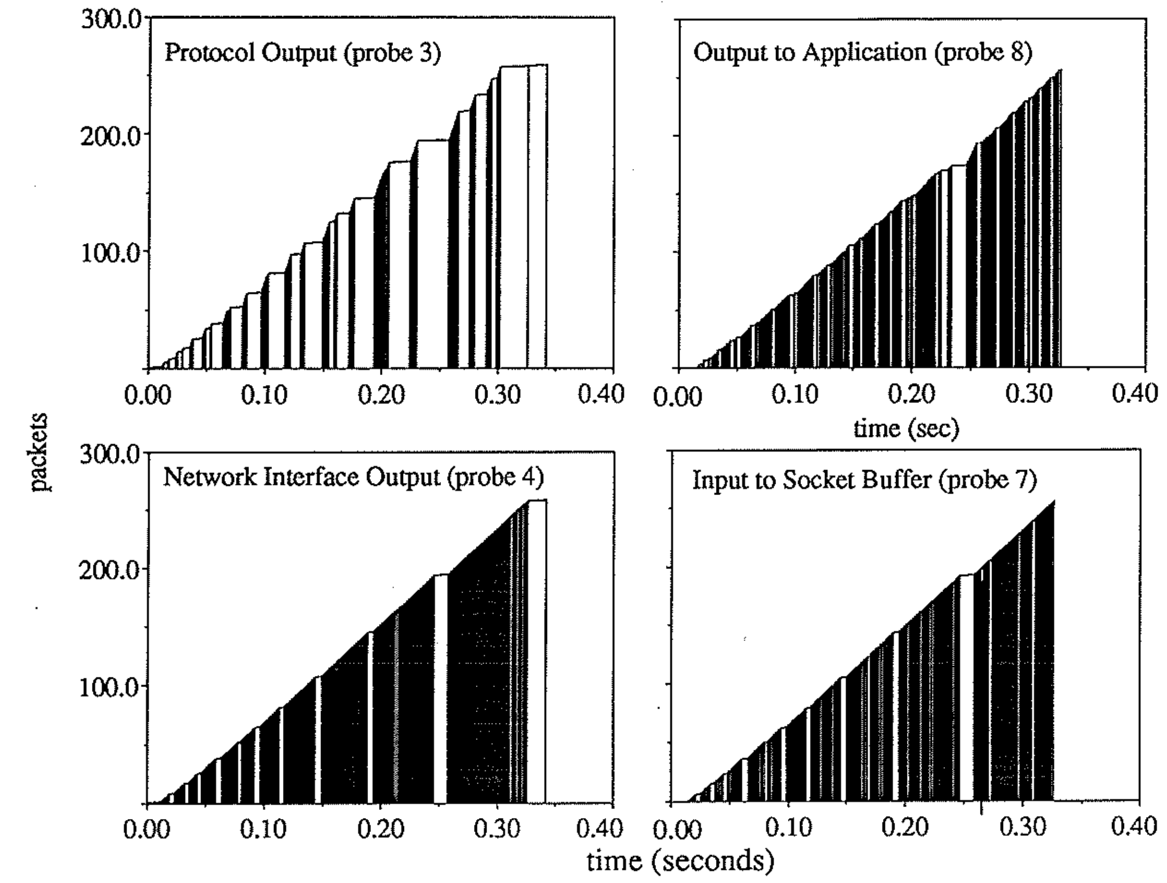

Figure 12: Packet Trace with maximum socket buffers

Packet Trace

Two sets of graphs are shown in this part of the experiment. One set was created with maximum size socket buffers, and another with the default size.

The first set of four graphs shows the packet traces as given by probes 3, 4, 7 and 8, which are the most interesting. The first two probes are on the sending side. The first monitors the protocol output to the network interface, and the second monitors the network driver output to the Ethernet. The other probes are on the receiving side, and they monitor the receiving socket input and output activity 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, a packet carries 1 KByte 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 passed to the network, and for the receiver when this SYN packet is received. The offset introduced is about 1ms, which does not affect the results in a significant manner.

Packet Trace with Maximum Size Buffers

The examination of the protocol output in Figure 12 reveals the burstiness of transmission introduced by the windowing mechanism. The blank areas in the graph represent periods the protocol is idle awaiting acknowledgment of its outstanding window. 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 (which is essentially the same graph tcpdump would have produced) 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 reaching the Ethernet. The reason for this is that the protocol is able to produce packets faster than the network can absorb, which leads to queuing at the interface queue. The network interface driver is kept busy outputting packets as fast as it can.

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 does not seem to be much queueing delay introduced by the IP queue. This also indicates that the protocol at the receive side is able to process packets at the rate they arrive. This is not surprising since the rate was reduced by the network, resulting in a packet arriving approximately 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 traced after examining the queue length graphs which is presented next.

Queue Length

The second set of graphs in Figure 13 shows the queue length at the various queues as a function of time (Please note the difference in scale in these graphs). The queue length is shown in number of bytes in the queue instead of packets, for the sake of consistency, since the socket layer there is no real division of data into packets at the sending side. Packets only come to existence after the protocol layer. This does not lead to loss of information, since for this experiment the packets were all of the same size and the graphs would still look the same if the number of packets was plotted instead.

The sending socket queue length is shown to monotonically decrease as expected, but not at a constant rate since its draining is controlled by acknowledgment activity. Note that the point the queue appears to become empty is the point where the protocol has taken possession of all the data (i.e. all data were copied to the socket buffer). The data remains in the socket buffer until acknowledged.

The interface queue shows the packet generation by the protocol. It is immediately apparent that there is burstiness in packet generation, as manifested by the peaks at the queue length. The queue build up suggested by the packet trace graphs is clearly visible. Also, the queue length increases with every new burst, indicating that a new burst contains more packets than the previous one. This behavior is attributed to the slow start strategy of the congestion control mechanism. The operation of the mechanism is explained at the end of this section.

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 network. As the graph shows, the queue practically never builds up, each packet being removed and processed before the

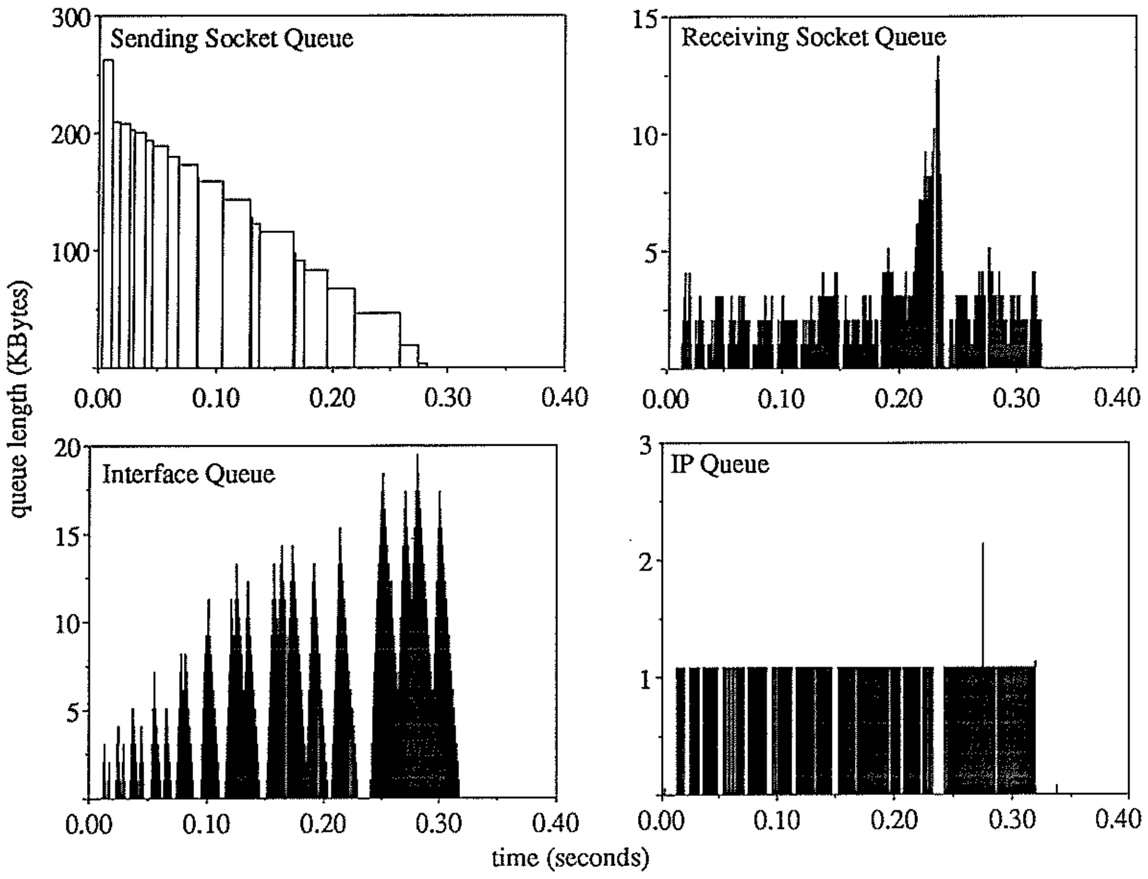

Figure 13: Queue Length during congestion window opening

next one comes in.

The queue length graph for the receive socket shows some queueing activity. It appears that there are durations over which data are accumulated in the socket buffer, followed by a drain of the buffer. During that time the transmitter does not send any data, since no acknowledgment is sent by the receiver. This queue buildup leads to "empty spots" in packet transmission, as witnessed earlier by the packet trace plots. These spots are a result of the receiver delaying the acknowledgment until data is removed from the socket buffer.

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 14, which shows the four queues after the congestion window has opened up. The graph shows that the IPC mechanism is performing very well.

Queue Delay

The next set of graphs in Figure 15 shows the packet delay at the four queues. The x-axis shows the

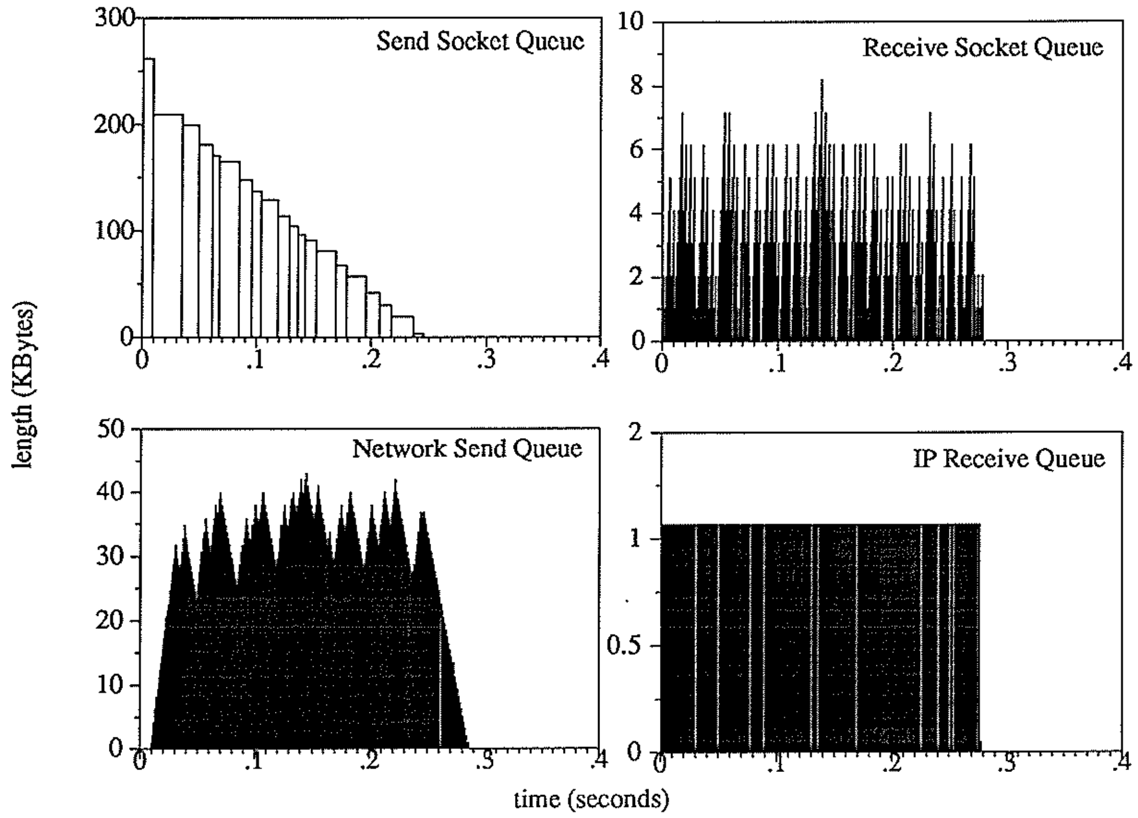

Figure 14: Queue Length after congestion window has opened

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 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 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 a very small delay, while subsequent packets are queued and therefore delayed significantly. There is a wide range of delays experienced by different packets, ranging 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 to 200 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.

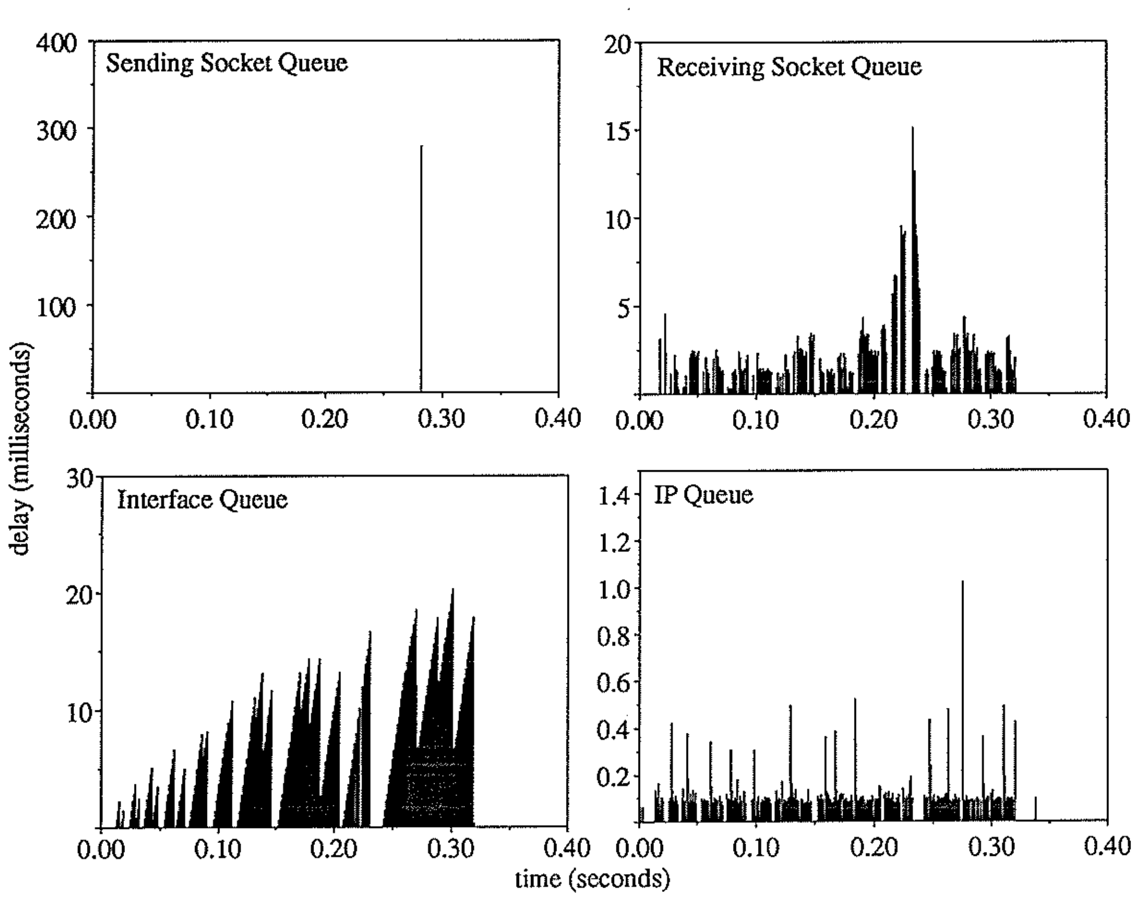

Figure 15: Queue Delay during congestion window opening

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 data accumulates in the buffer, shows that the process runs at a frequency less than the data arrival.

Protocol Processing Delay

The two histograms in Figure 16 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 are simply ignored. This problem does not appear at the receiving end, where the protocol receives and forwards distinct packets to the socket layer.

The graphs show histograms of the protocol delays. The x-axis is divided into bins which are 50 micro-

††Determined when measuring the driver performance

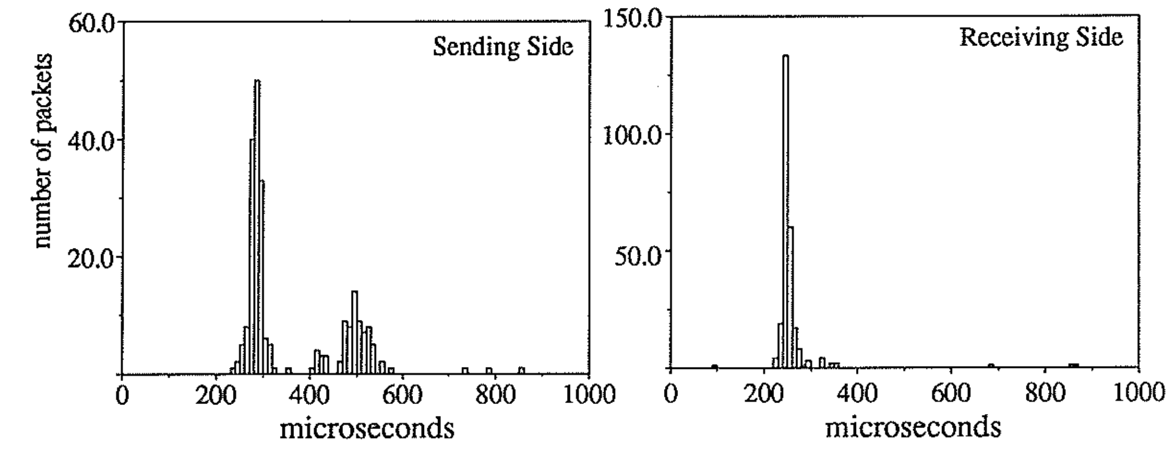

Figure 16: Protocol Processing Delay Histograms

seconds wide, and the y-axis shows the number of packets into each bin. The protocol delays appear to be different at the two ends: on the sending end processing delay divides the packets into two distinct groups, the first, the larger one, taking about 250 to 400 microseconds to process, and the second taking about 500 to 650 microseconds. At the receiving end the packet delay is more evenly distributed, and the protocol takes about 250 to 300 microseconds for most packets, with some taking between 300 and 400. 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. Thus the theoretical maximum TCP/IP can attain on Sparcstation 1's when the CPU is devoted to communication with no data copying and network driver overhead, appears to be about 22 Mbps.

Part 1 of the experiment has shown that the IPC mechanism can achieve about 75% utilization of the Ethernet. Once the congestion window has opened up, the windowing mechanism can make good use of the Ethernet. It was however, confirmed that there is queueing present at the receive socket queue. The effect of queueing at the socket queue on throughput is the focus of Experiment 4.

5.2.2 Part 2: Incremental Delay

In this experiment the delay experienced by a single packet moving through the IPC layers is isolated and measured.

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 an experiment was performed, where consecutive packets are sent after introducing some delay between them. The aim is to give the IPC mechanism enough time to send the previous packet before supplied with the next.

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

The results of this experiment are summarized in Table 5.1:, and are compared to the results obtained

Table 5.1: 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

in the previous experiment, when 25 KBytes of data were transmitted.

The send socket queue delay shows that there is about 280 μS delay 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.

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 actually not reported in this paper, it was found that it takes about 1150 μS more to be transmitted [12].

The IP queue delay is also quite small. The delay has not changed significantly from the earlier experiment, when 256 KBytes of data were transferred.

The receive socket queue delay is about 412 mS, 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. This result could not be obtained from the earlier experiment.

The protocol layer processing delay 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 delay for the sending side has increased by about 73 μS (443 vs. 370 μS). This is due to the fact that the TCP output function for this experiment must be called every time a packet is transmitted. In the previous setting the TCP output function was called with a large transmission request, and entered a loop forming packets and transmitting them in window bursts.

5.2.3 Probe Overhead

The measurements presented above include extra overhead from the probes. The delay introduced by the probes is small, and is dominated by the timestamp call.

In order to get an estimate on the overhead the probes introduce to the IPC mechanism, throughput measurements were performed, first without the probes and then with the probes active. The results showed that the throughput was not significantly affected by the presence of probes. Using custom software the throughput was measured in both cases, and showed that it dropped about 8 percent. Without probes, throughput values in the range of 7 to 7.5 Mbps were obtained, dropping to about 6.6 when the probes were active.

The presence of probes also affected the acknowledgment generation. The slower processing at the receive end resulted in about one third reduction to the number of generated acknowledgments.

5.3 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 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 interface, 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 is investigated.

5.3.1 Effect of Receive Socket Queueing

The actions taken at the receive socket in order to receive data are summarized briefly. The receive function enters a loop traversing the mbuf chain copying data to the application space. The loop is exit 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 with the PRU_RCVD request so that protocol specific actions, like sending an acknowledgment can 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 user removes data from the socket buffer. As mentioned earlier, when this happens the socket layer will call the TCP user-request function, which in turn will call the TCP output function if an acknowledgment must be sent. During normal data transfer (no retransmissions) an acknowledgment is sent after the application removing at least 2 maximum segments from the receive buffer leaves the buffer empty, or whenever a window update would advance the sender's window by at least 35 percent‡‡.

‡‡As per the tcp_output () of SunOS. See [12] for details.

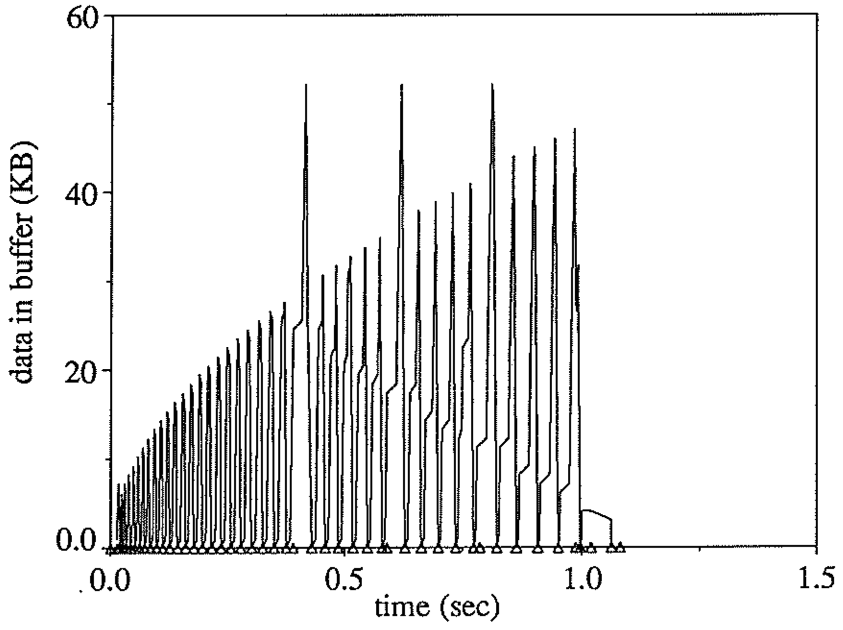

Figure 17: Socket queue and acknowledgments on loopback

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 in 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.

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 was again a unidirectional connection sending and receiving data as fast as possible. To isolate the protocol from the Ethernet, the experiment was conducted using the loopback interface. The data size sent was set to 1 MByte, and the result is shown in Figure 17. This graph shows the length of socket receive buffer and the generated acknowledgments throughout the connection 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 full 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 due to the delayed acknowledgment timer process that runs every 200 mS. The effect of this timer is that when it runs it sends an ack before the data in the receive buffer is removed, causing the sender to transmit more data, which reaches the socket buffer and causes these peaks. There do not seem to be other times when the protocol sends acks. It should be noted, however, that during remote data transfer, more acknowledgments would be generated, since the receive buffer would become empty much more often than during loop-back transfer.

5.3.2 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 only when the receive buffer becomes empty. If data arrive 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 are 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 process may cause more data to be received before the old window of data was passed to the application, causing peaks at the socket buffer during the slow start period. This however, does not seem to have any detrimental effects.

5.3.3 Summary

This experiment has shown that queueing at the receive socket layer may affect the protocol acknowledgment generation. It is conceivable that at its worst, this effect may interfere with the constant packet flow the window mechanism is designed to produce, forcing the protocol to a stop-and-wait operation.

5.4 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. It is aimed at determining which parts of the IPC mechanism are affected when additional load is present on the machine.

5.4.1 Communication vs. Computation

The experiment was performed by setting up a unidirectional connection, running the workload, and blasting data from one Sparcstation1 to another. There are two parts to the experiment: the first, places the workload on the receiver machine, and the second, on the sender machine. The results reported include graphs depicting how the various queues are affected by the extra workload during data transfer. The effect on protocol processing and throughput measurements are also reported

The Artificial Workload

There 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 numbers processes running at

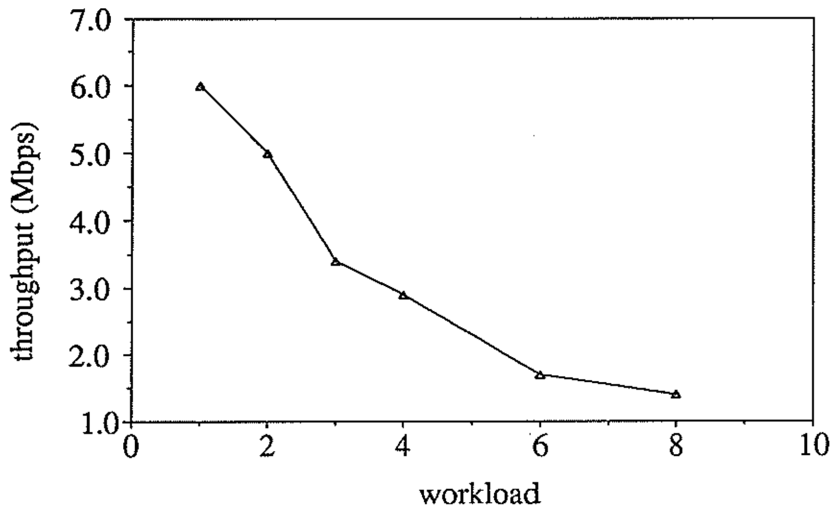

Figure 18: Throughput v.s. server background load

the same time.

Loading the Server

Figure 19 shows the effect of the artificial workload on communication throughput when the server machine runs the workload. The experiment was performed by first starting the artificial workload on the server machine, and then immediately performing a data transfer of 5 MBytes. The graph shows that the throughput drops sharply as the number of processes increases. This suggests that the current protocols and their implementations require significant amount of CPU cycles, and if an application's computation and communication have to compete for the same CPU cycles, both suffer leading to poor overall performance.

Figure19 shows the internal queues during data transfer with a loaded server. For plotting these graphs the data size was reduced to 1 Mbyte to keep the size of the graphs reasonable. The workload level for this graph is 4, meaning that there were four prime number processes running. The x-axis shows the time and the y-axis the queue length. 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 presence of the workload causes data to accumulate in the IP queue. The accumulation however, is still very small compared to the other queues. The interface queue on the sending side reflects the effect introduced by the delay in acknowledging data by the receiver.

Loading the Client

The graph in Figure 20 shows the effect of client artificial workload on throughput. The graph was

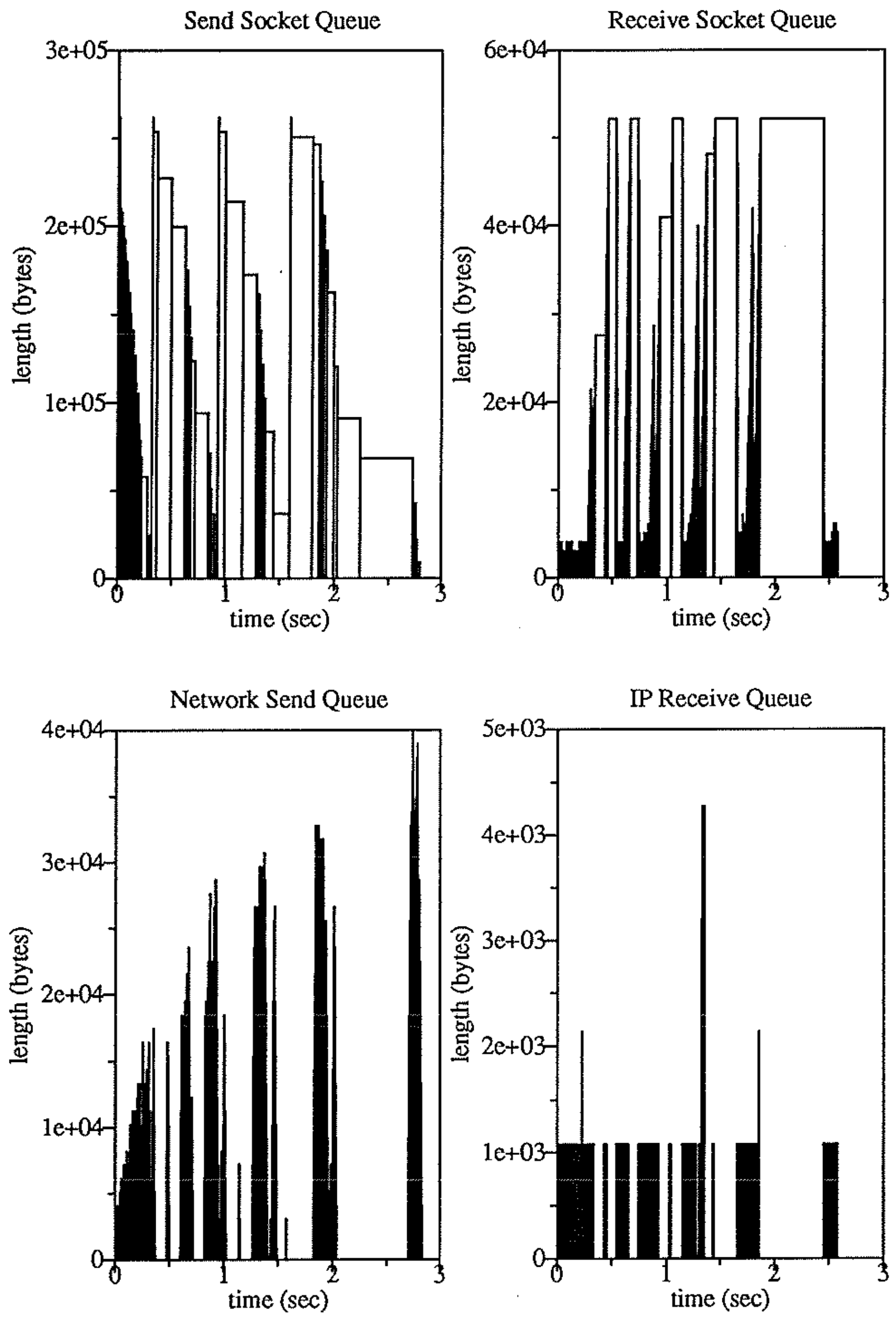

Figure 19: Queue behavior with server background load

obtained using the same setup as for loaded server. It shows that the effect of loading the client is similar to the effect of loading the server.

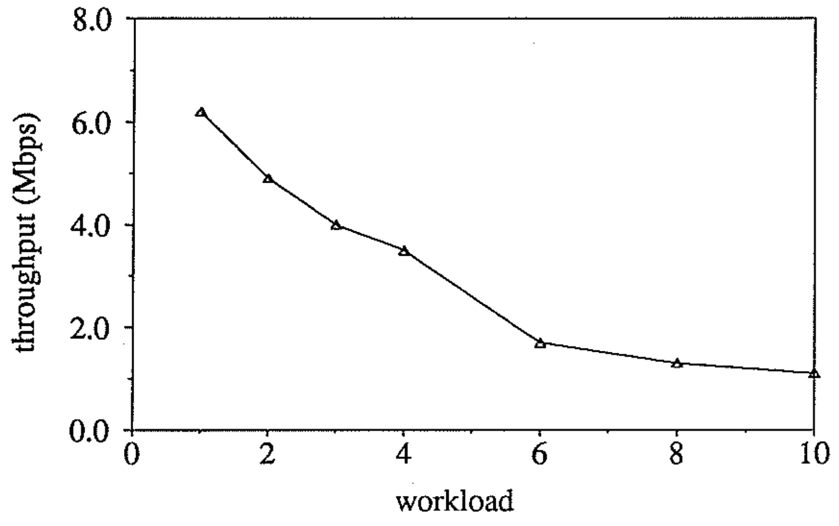

Figure 20: Throughput v.s. client background load

Protocol Processing Delay

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 run with the network interrupts masked.

5.4.2 The Effect of Process Scheduling on Throughput

In the presence of background load, the Unix scheduling mechanism affects communication throughput considerably. The mechanism is designed to favor short and light 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 priority is lowered, in order to prevent it from monopolizing CPU time.

As an example, assume that a distributed pipeline is implemented as three concurrent processes (one reading new segments, the next performing the desired operation, and the third writing the segments downstream). The scheduling policy will have significant impact on the performance of the pipeline. If the computation requires more CPU time than the communication (which is a very reasonable assumption), whenever a segment becomes available for transmission or reception, the communication processes will preempt the computing process. Thus during the relatively short amount of time required for communication, the CPU allocation will favor the communication processes, which will run at near maximum speed.- The throughput measurements presented in this experiment, should be viewed with the above considerations in mind.

5.5 EXPERIMENT 5: RUNNING A DISTRIBUTED APPLICATION

The last experiment presents snapshots from the activity of the three processes constituting a stage in a pipelined distributed application. In fact, this experiment was performed with a real pipelined remote visualization application developed as a part of this research. Details of this application can be found in [12]. Each application stage consists of a process receiving data from the previous stage, another process performing some computation on the data, and a third process transmitting the data to the next stage. The network interface queue activity is superimposed with the activity of the other processes in the module, in order to observe the behavior of the queue while the module is running. The graphs show that the computation process is usually preempted by the communication processes. This again suggests that the existing protocols and their implementations, which of course include memory management, data copying and other OS overhead, are CPU intensive and can adversely affect overall performance of a distributed application.

Figure 21 depicts the duty cycle of the three processes in the stage. The x-axis shows the elapsed time,

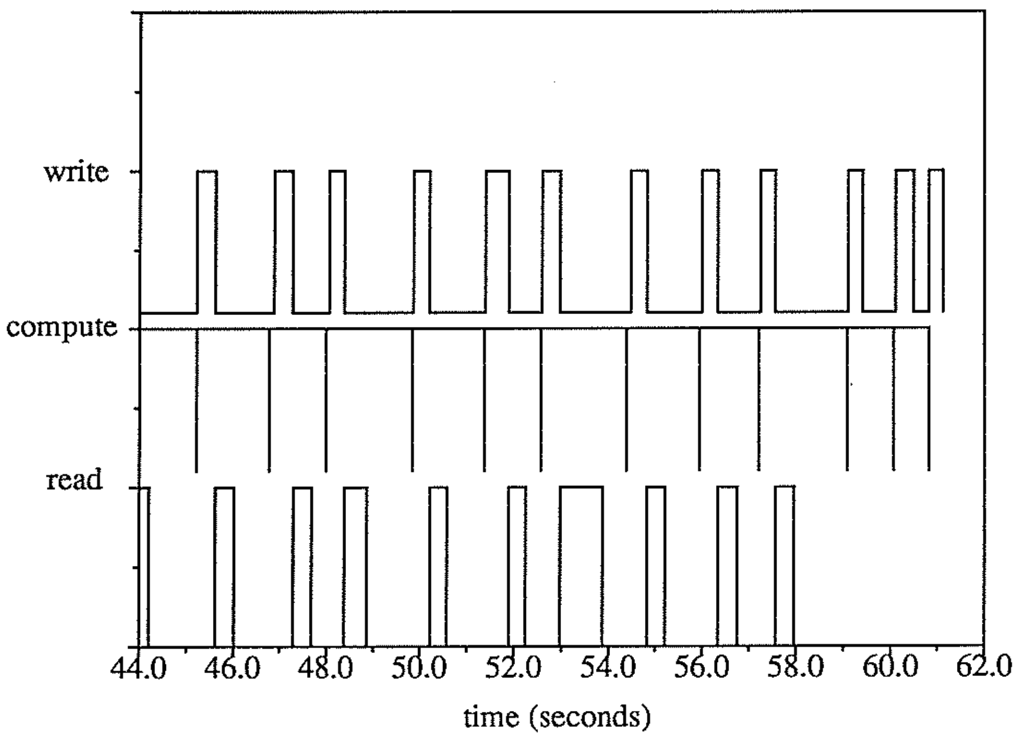

Figure 21: Process activity and send interface queue

and the y-axis shows the interface network queue length. The graphs depict duty cycles as on/off states. Each state has a different "on" level to make the graph more readable. The lower level state represents the read process, the next level up the compute process, and the last the write process.

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 a longer 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, filling up the slot freed by the write. The two communication processes appear to be scheduled immediately receiving a message, while the compute process looses on CPU time.

6. 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 [12]. Therefore, a better driver implementation will increase throughput further.

Some improvements of the default IPC are still possible. Experiment 1 has shown that increasing the default socket buffer size from 4 kilobytes 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 network interface queue, shown in Figure 5, is the point where outgoing packets are queued for transmission. Currently its size is limited to 50 packets which with the default buffers guarantees no loss of packets for 12 simultaneous connections (12 connections with a maximum window of 4). If the default socket buffers are increased to 16 kilobytes, then this number drops to 3.

The performance of IPC for local processes on a Sparcstation 1 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 was measured 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 receiving side has the potential of processing packets at the rate they arrive up to the socket queue, where it is queued for delivery to the application. Figure 22 shows the distribution of the various delays. From these numbers the potential performance of the protocol can be estimated. The performance is determined by the protocol processing and the data copy. The performance for the sending side is about 16 Mbps, and for the receiving side about 21 Mbps.

During large data transfer, TCP depends on the socket layer for a notification of data reception by the application in order to send acknowledgments. However, the socket layer notifies TCP only when all data has been removed from the socket buffer. This introduces delay in sending data which is equal to one round trip time. This is a small problem on the Ethernet, but it will be unacceptable on future high bandwidth x delay networks.

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 competition increases. Conversely, the presence of communication affects computation as shown in Experiment 5, where communication is shown to steal 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.

6.1 SCALABILITY

Though it is very tempting to make a definite statement about the scalability of TCP/IP protocols for the

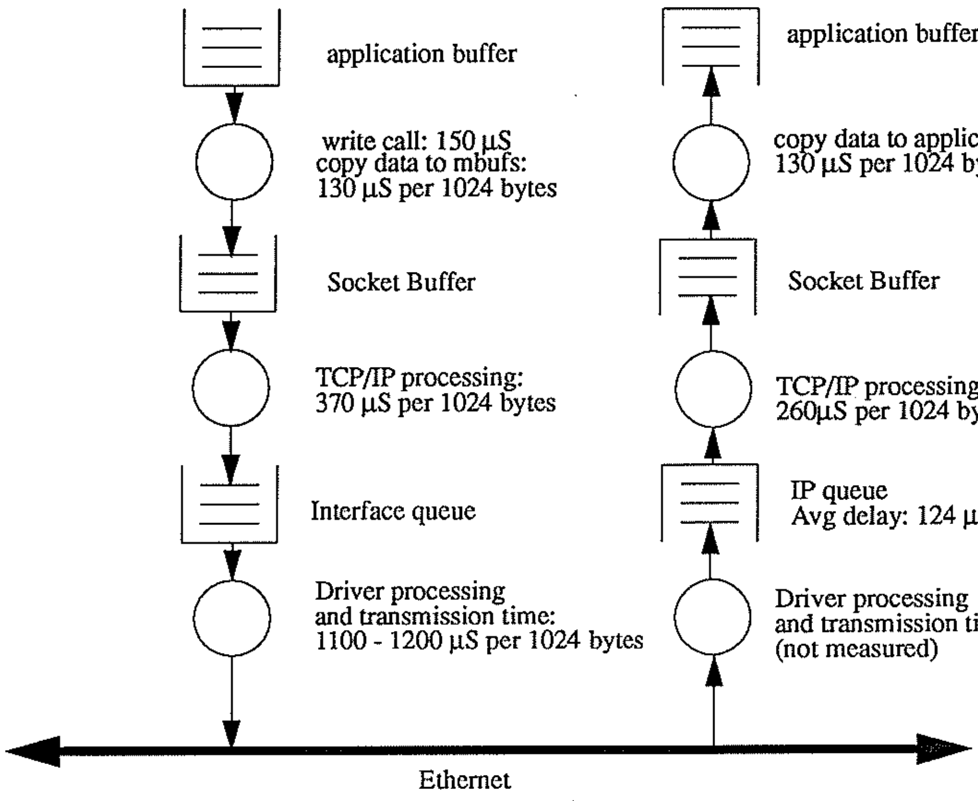

Figure 22: Delays in IPC

emerging high bandwidth applications over high speed networks, it is a very difficult task because of rapid changing technological trade-offs. For example, a typical CPU MIPS rating has been improving and multiprocessor workstations are becoming available; memory and buffer management schemes of operating systems are undergoing revisions, and building ASIC (Application Specific Integrated Circuit) hardware is becoming affordable and easy to do. In the following paragraphs we attempt to state claims about the existing protocols and their existing implementation model as presented in this paper.

Some areas of weakness have already been identified with existing TCP for high speed networks, and extensions to TCP were proposed to address them [10]. 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 [8]. We and other groups are currently evaluating these options [6].

For the test environment reported in this paper, TCP 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, the network speeds will also increase. For example, network speeds are expected to increase to 1 Gbps or more, which is an increase of a factor of 100. Therefore, even with processors 100 times faster than processors available today, TCP will continue to require a large share of the CPU for communication, unless some drastic modifications are made to the implementation. Also, it is important to note that the CPU MIPS ratings are based on high cache hit ratios, however, the protocol processing may not be able to sustain such high cache hit ratios, and thus, all of CPU MIPS cannot be put to use for high speed communication.

For every packet TCP performs checksum operations twice, one at the sending and another at the receiving end of the protocol. In order to perform checksumming, data needs to be accessed in main memory. The fact that the TCP checksum resides in the header and not at the end of the packet prevents the use of hardware to perform checksumming as data are sent to, or arrive from the network.

With the current implementation, IPC may encounter some performance problems if used without further modifications in high-speed networks. As shown by the results in Experiment 4, a Sparcstation 1 during a unidirectional data transfer receives packets at a rate higher than the rate packets can be delivered to the application. This leads to queueing at the receive socket queue, which can become significant, especially if the machine is running other jobs at the same time. The resultant reduction in acknowledgments could degrade the performance of TCP to a form of stop-and-wait protocol. This may result in packets being transmitted in bursts which are interspersed by gaps equal to one round-trip delay.

This reduction in acknowledgments could be especially harmful during congestion window opening on high bandwidth high-delay networks. The situation will be exacerbated especially if the TCP large windows extension is implemented. Since the congestion window opens only after receiving acknowledgments, there may not be enough acknowledgments generated to open the window quickly, resulting in the slow-start mechanism becoming too slow. In addition to causing long delay during the beginning of data transfer, the same behavior will be replicated after each packet loss.

Finally, one may question suitability of TCP's point-to-point 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 [9].

7. 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] Dittia, Zubin, "Implementation and Evaluation of TCP Extensions," MS Thesis, Department of Electrical Engineering, Washington University in St. Louis. In progress.

[7] Feldmeier, D.C., "Multiplexing Issues in Communication System Design," Proceedings of SIGCOMM '90, pages 209-19 Philadelphia, PA, September, 1990.

[8] 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.

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

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

[11] 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.

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

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

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

[15] 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.

[16] 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.

[17] 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.