5. PERFORMANCE OF IPC

5.1 INTRODUCTION

There are some important reasons for measuring performance of a complex system such as the IPC implementation in Unix: (1) it forces the acquisition of a fairly detailed understanding of the kernel internals and protocols; and (2) 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

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 to 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.

The experiments first investigate the performance of IPC from the application point of view and then move to the IPC internal layers. In Experiment 1, the study of the IPC mechanism is motivated by showing that even though Sparcstation 1 workstations are theoretically capable of exceeding the Ethernet rate, the Ethernet driver performance did not allow this. In Experiment 2, experiments that can be performed without the use of a probing mechanism are presented. These experiments are simple but some understanding of the kernel structure is required to interpret the results. The results from Experiment 2 are useful but insufficient for detailed understanding of IPC. In Experiment 3, the behavior of the IPC queues is examined with the aid of probes. In Experiment 4, the queue with the higher delay is identified and the source of the delay and its effect on TCP is investigated. In Experiment 5, the effect of computation load on communication is investigated. The investigation includes throughput measurements and the use of probes to depict the effect on the internal queues. In Experiment 6, the process scheduling by the operating system is investigated when the OSM application is running.

5.2 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 of two IPC configurations is compared: in the first communication is over two physically separate machines, and in the second over 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.

5.2.1 Throughput and Socket Buffer Size

There are two distinct socket buffers associated with each socket, one for sending and one for receiving. Figure 5.1 depicts the location of these buffers relative to the queues in

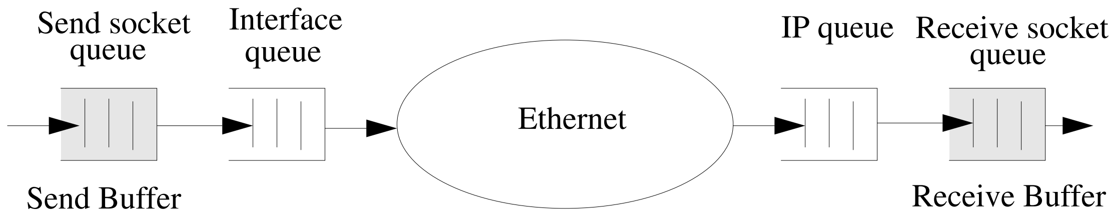

Figure 5.1: Socket buffers and queues in the queueing network

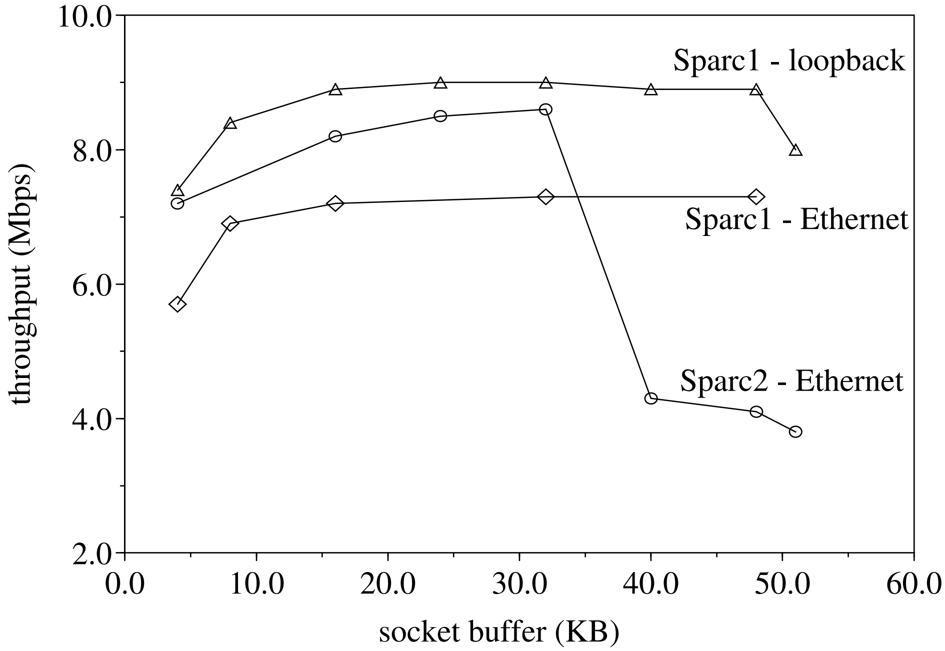

Figure 5.2: Throughput vs. Socket Buffer Size

the queueing model presented in Chapter 4. The sizes of the buffers determine the available slots in the socket queues. 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 buffers reside in kernel space and may be of different size. Data to be transmitted are copied from user space into the send buffer for access by the protocol layers. Data received by the protocols are placed into the receive buffer until copied to the application space. The minimum of the two buffer sizes is the maximum window TCP can use (for a more detailed description please refer to Chapter 4).

Figure 5.2 shows the throughput obtained 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

1. The throughput measurements were obtained using custom software

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 bug in SunOS 4.1. The same behavior was observed by others who also had no explanation of this phenomenon[1]. The unavailability of source code for SunOS 4.1 prevented further investigation of this problem[2].

Observations

The graphs show that throughput of Sparcstation 1 machines over the Ethernet improves significantly 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.

Increasing the socket buffer size is not only beneficial for existing IPC, but it will be required in order to effectively use TCP over high bandwidth networks. The reason is that 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), thus on the receiving end the protocol can receive only four packets with every window, which leads to a kind of stop-and-wait protocol[3] with four segments.

---

1. Private email with Paul Liu, Distributed Computing Evaluation Group, Dept.45262, Bell Labs
2. The problem has been fixed in later versions of SunOS
3. In a stop-and-wait protocol the transmitter sends a packet of data and waits for an acknowledgment before sending the next packet.

Another potential problem with small socket buffers is that they lead to a significant increase in acknowledgment traffic because at least one acknowledgment is required for every window. The number of acknowledgments generated with 4 kilobyte 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 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.

5.2.2 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 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 5.2 shows the throughput over the loopback interface vs. the socket buffer size. The throughput reaches rates close to 9 Mbps. This 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 5.2, 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's running SunOS 4.1 were also using 1 kilobyte packets (as opposed to 1460 which is the default) to make the two cases comparable.

5.2.3 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 during the next experiment.

5.3 EXPERIMENT 2: NON-PROBING MECHANISMS

Inserting probes in the kernel to explore the performance of the IPC mechanism involves overhead. First, there is the complexity of studying the kernel code to determine the best location for the probes. Second, the probes may interfere with the mechanism being monitored. Therefore it may be worthwhile to explore the possibility of using other methods to investigate the performance of the IPC mechanism.

These experiments attempt to gain some insight into the performance of IPC by using non-probing methods. These methods provide some interesting results and are thus worth presenting. However, the information provided is not sufficient for a detailed understanding of the IPC performance.

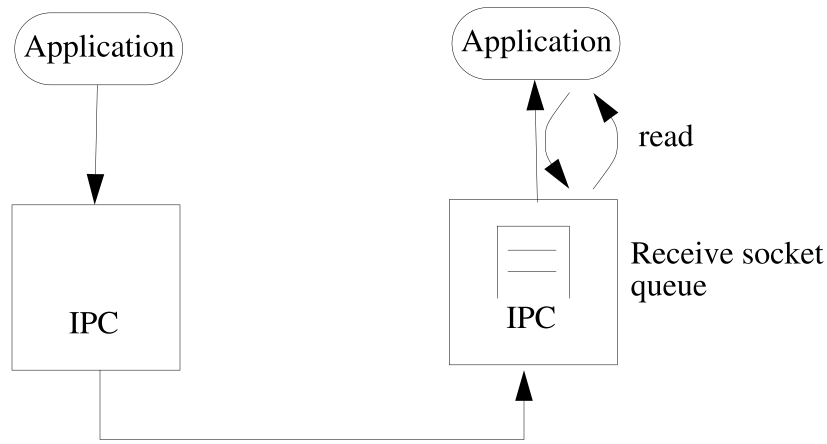

Figure 5.3: Receiver performs multiple calls to read large data

Two methods are presented. The first method attempts to deduce information about the performance of the receive socket interface by observing the amount of data returned to the application with each read call. The second employs a tool which captures packets from the network in real time to study the performance of the network driver.

5.3.1 Investigating the Read Call

In this experiment the information about queueing at the receive socket queue is investigated. As mentioned in Experiment 1, queueing at the receive socket buffer is worth investigating, since it may have a significant effect on performance. The presence of queueing can be detected by studying the amount of data returned by each application read call.

TCP uses stream sockets whose semantics specify that each application read request will return a portion of the data from the stream. No guarantees are given about how much data will actually be read. The only guarantee is that no more than the requested amount will be returned. If the application then wishes to receive more data, it has to perform consecutive reads until satisfied. This process is depicted in Figure 5.3.

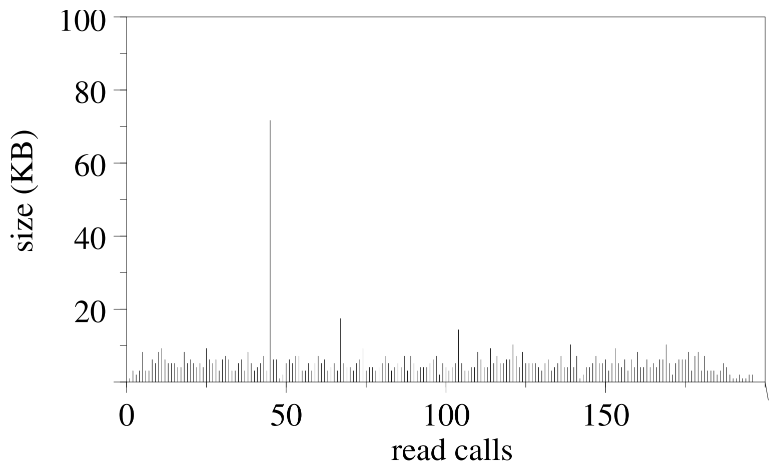

Figure 5.4: Server read calls and their size in a unidirectional connection

Assuming a large data transfer, the amount returned by each call is the amount of data present in the socket buffer at the time the call was made. However, new data may be added to the buffer while old data are being copied to the application space. Thus it is possible for a read call to return more data than can fit in the socket buffer. In addition, the amount returned to the application is also an indication of how fast the data were processed by the IPC layers. If data are processed quickly a large number of read calls will occur, each returning a small amount of data. If the processing is slow the opposite will be observed. Therefore, the frequency and size of read calls will give an indication of the socket queue activity. If the read calls return more than a packet of data, then there was queueing present at the socket buffer. If exactly a packet of data is returned, the IPC mechanism was fast enough to deliver data to the application at the rate it arrives.

To investigate queueing at the receive socket buffer an experiment was performed measuring the frequency and size of read calls during communication on a Sparcstation 1. For the experiment a unidirectional connection was set up, and data was sent from the client side to the server. Figure 5.4 shows the amount of data each read call returned vs. the number of read calls required to receive 1 Mbyte of data. Each vertical bar represents a read call, and the length of the bar represents the data returned by that call. The graph shows that there is queueing at the socket queue. About 200 read calls were executed, most of them returning about 5 - 10 kilobytes of data. The Sparcstation 1 is not fast enough to deliver each packet to the application as it arrives from the Ethernet, thus data are queued at the socket buffer. This may cause some delay. However, the queueing present is relatively low, and the IPC mechanism when the processor is devoted to communication, appears to perform quite well. To verify that queueing was not the result of time slice granularity, the same experiment was performed on a Sparcstation 2. It was observed that each read call returned exactly 1024 bytes of data, indicating that a Sparcstation 2 is able to deliver packets to the application as they arrive from the network.

5.3.2 The Performance of the Network Driver

Experiment 1 has shown that a Sparcstation 1 in loopback mode can achieve throughput rates approaching 9 Mbps, indicating that the Ethernet rate is probably be the limiting factor during communication between two machines. The experiment however, excluded the network interface driver performance. This experiment aims at investigating the performance of the driver.

Theoretically, the network driver may output Ethernet frames at 10 Mbps. If data in the packets are 1024 bytes, the time to transmit a packet is 882 μS. This number is calculated as follows: The Ethernet header is 14 bytes, and the TCP/IP headers are 40 bytes. The preamble, the checksum and the start-of-frame delimiter take 12 more bytes. There is also a 9.6 μS inter-frame gap, which corresponds to the transmission time for 12 bytes. So if the data portion of a frame is equal to 1024 bytes, the total time required to transmit a frame is:

(1024 bytes + 40 bytes + 14 + 12 + 12 bytes) x 8 bits) / 10 Mbps = 882 μS

At 10 Mbps there should be a frame on the Ethernet every 882 µS. However, it has been observed that the Ethernet cannot deliver 100% efficiency[1] when multiple hosts are contending. When the contention is low, the theoretical Ethernet efficiency is about 90% [1]. In Experiment 1 rates up to 8.7 Mbps were achieved with Sparcstation 2 workstations, which translates to an 87% measured efficiency. To achieve this level of efficiency one frame must be transmitted every millisecond approximately.

Tcpdump can be used to investigate the Ethernet interface driver performance by measuring the frequency that frames appear on the Ethernet. Tcpdump is a tool used to capture packets from the Ethernet and is described in Chapter 4. The experiment used tcpdump to capture packets from the network while a Sparcstation 1 was sending 1 MByte of data to another Sparcstation 1. From the collected data, a histogram of the delay between frames was created and is shown in Figure 5.5.

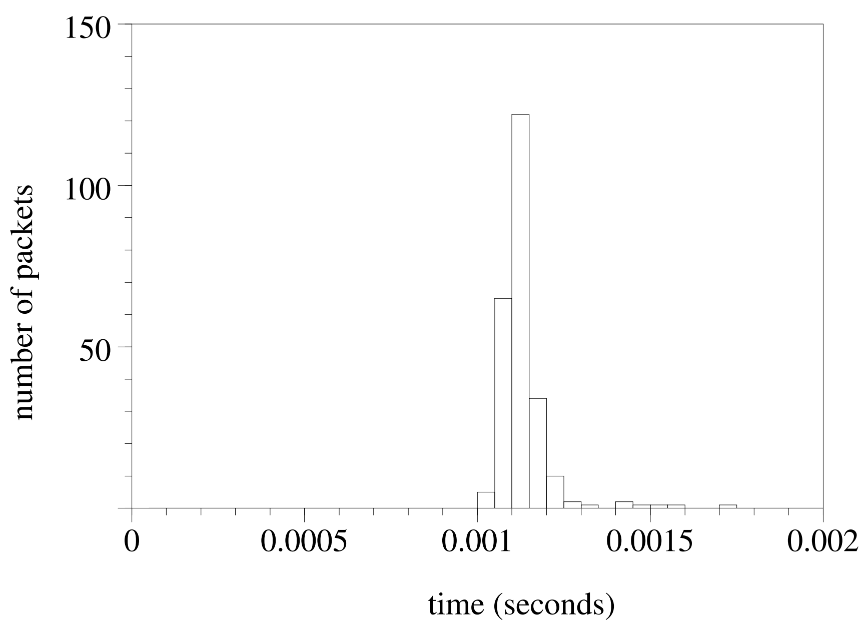

Figure 5.5: Sparcstation 1 inter-packet gaps

1. Some researchers however, have reported on Usenet that they have observed 98 - 100% efficiency on Ethernet, with carefully tuned implementations of the network interface.

The histogram shows the inter-frame delay on the x-axis collected in bins of 50 microseconds. The y-axis shows how many such delays were observed. The graph shows that most packets experienced a delay of about 1150 microseconds. Therefore, the Ethernet driver seems able to achieve about 7.7 Mbps. However, this result is close to the 7.5 Mbps observed in Experiment 1. This means that the network driver may indeed be the reason the throughput is limited to 7.5 Mbps.

As an attempt to confirm the above observation, throughput measurements were taken between Sparcstation 1 and Sparcstation 2 workstations. Experiment 1 has established that Sparcstation 2 machines can achieve about 8.7 Mbps. The measurements show that with this configuration the throughput was still limited to 7.5 Mbps. Admittedly, there may be protocol implementation differences in SunOS 4.0.3 and 4.1 that play an important role, but it is assumed that if anything, the newer version is able to perform at least as well as the old one, especially when running on a faster machine.

5.3.3 Summary

Using tcpdump, an external view of communication in the form of packet trace was produced during this experiment. This view has suggested that the network driver is a bottleneck at the sending side of the communication and limits the throughput to 7.5 Mbps. The network however, may achieve as much as 8.7 Mbps.

5.4 EXPERIMENT 3: INVESTIGATING IPC WITH PROBES

In this experiment the developed probing mechanism is used to get a better understanding of the behavior of IPC. For easy reference the queueing network depicting the probe location which was first shown in Chapter 4 is shown again in Figure 5.6.

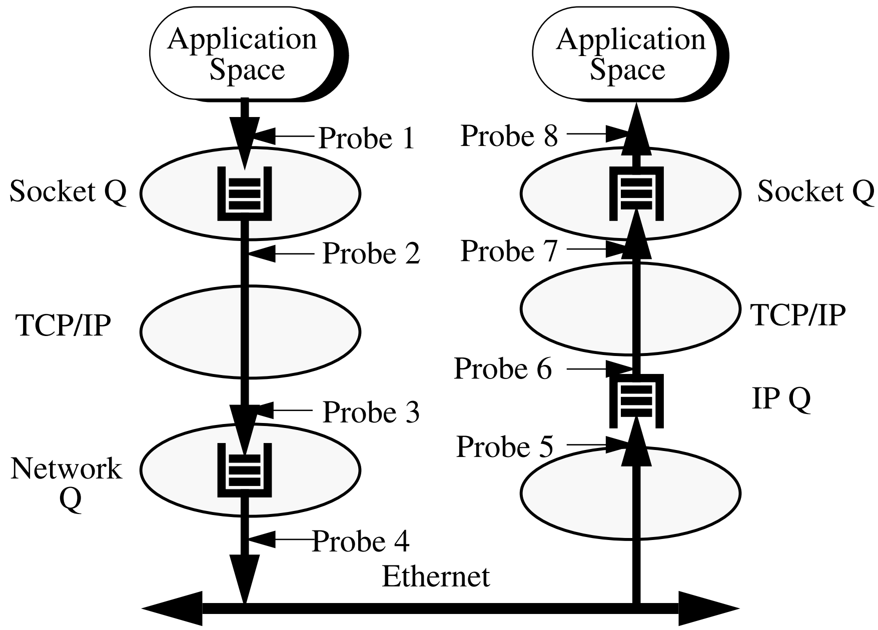

Figure 5.6: Queuing network and probes

The experiment has two parts: the first consists of setting up a connection and sending data as fast as possible in one direction. The data size used for this part was 256 KB, matching the slice size of the OSM application. 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.

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 either Sun's *traffic* or *xenetload*, which are tools capable of displaying the Ethernet utilization. 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.

5.4.1 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 four sets of graphs: (1) a packet trace of the connection; (2) queue length of the four queues; (3) delay in the queues; and (4) protocol processing delay.

Packet Trace

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

The first set of four graphs shows 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 of as a measure of the packet sequence number since for this experiment a packet carries one kilobyte of data. Each packet is represented by a vertical bar, to provide a visual indication of packet activity. For the sender the clock starts ticking when the first SYN packet is passed to the network, and for the receiver when the SYN packet is received. The offset introduced is about 1ms, which does not affect the results significantly.

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

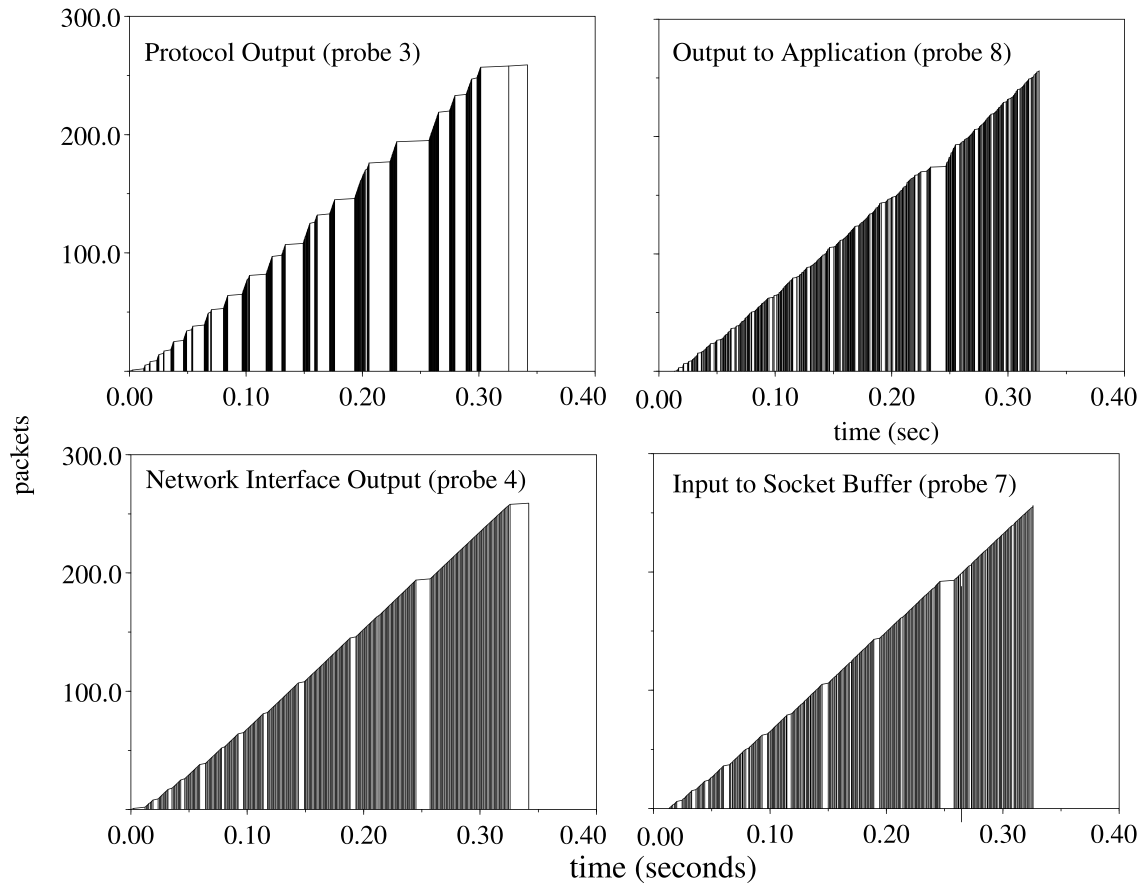

Figure 5.7: Packet Trace with maximum socket buffers

Packet Trace with Maximum Size Buffers

The examination of the protocol output in Figure 5.7 reveals the burstiness 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 periods 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 is not 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 will be presented shortly.

Packet Trace with Default Size Buffers

The second set of packet trace graphs with the default socket buffer size (4096 bytes) is shown in Figure 5.8. The effect of the smaller window size is immediately apparent. The packet flow from the protocol to the network does not exhibit the bursty behavior observed with the bigger size buffers. The packet flow from protocol to network appears smoother. However, the time to transmit the data has increased from 0.3 to approximately 0.4 seconds, showing a degradation in performance.

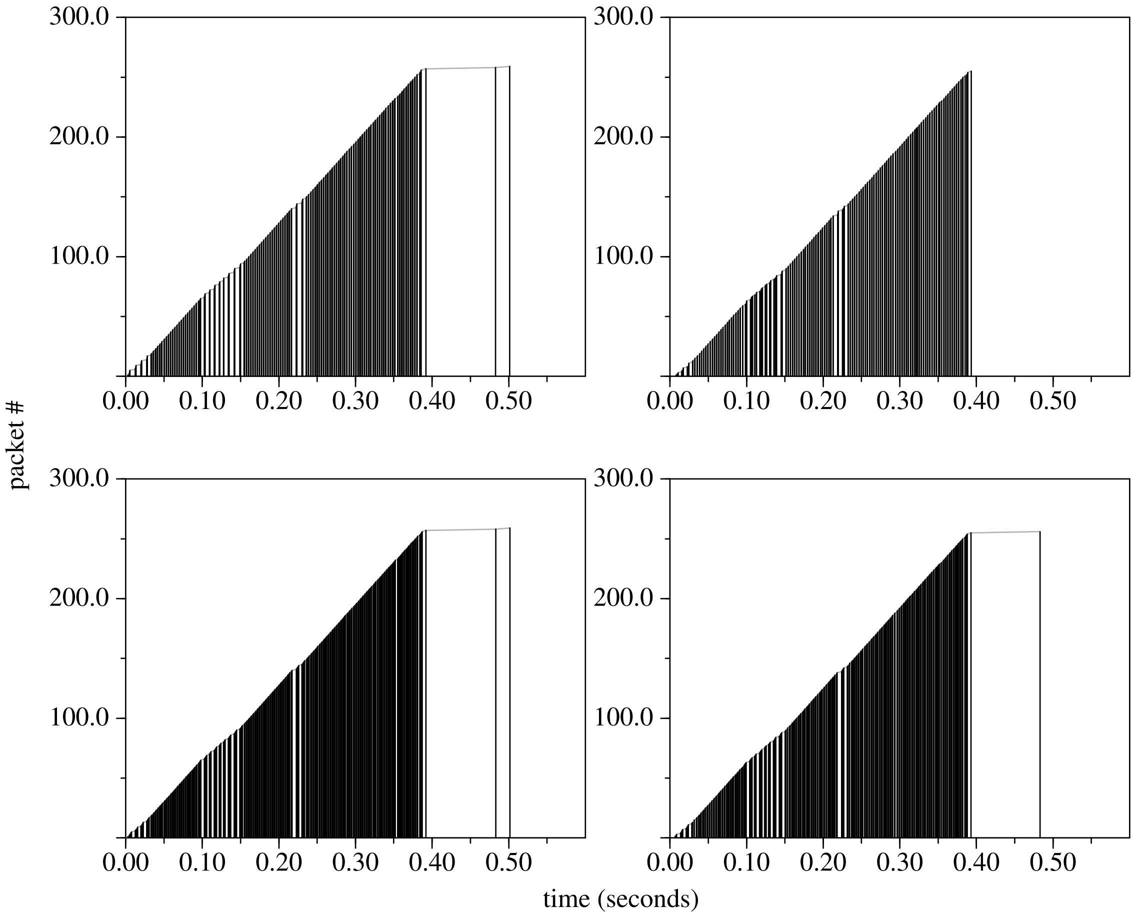

Figure 5.8: Packet trace with default socket buffers

Queue Length

The second set of graphs in Figure 5.9 shows the queue length at the various queues as a function of time (please note the difference in the y-axis 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 in 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 any 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 were plotted instead.

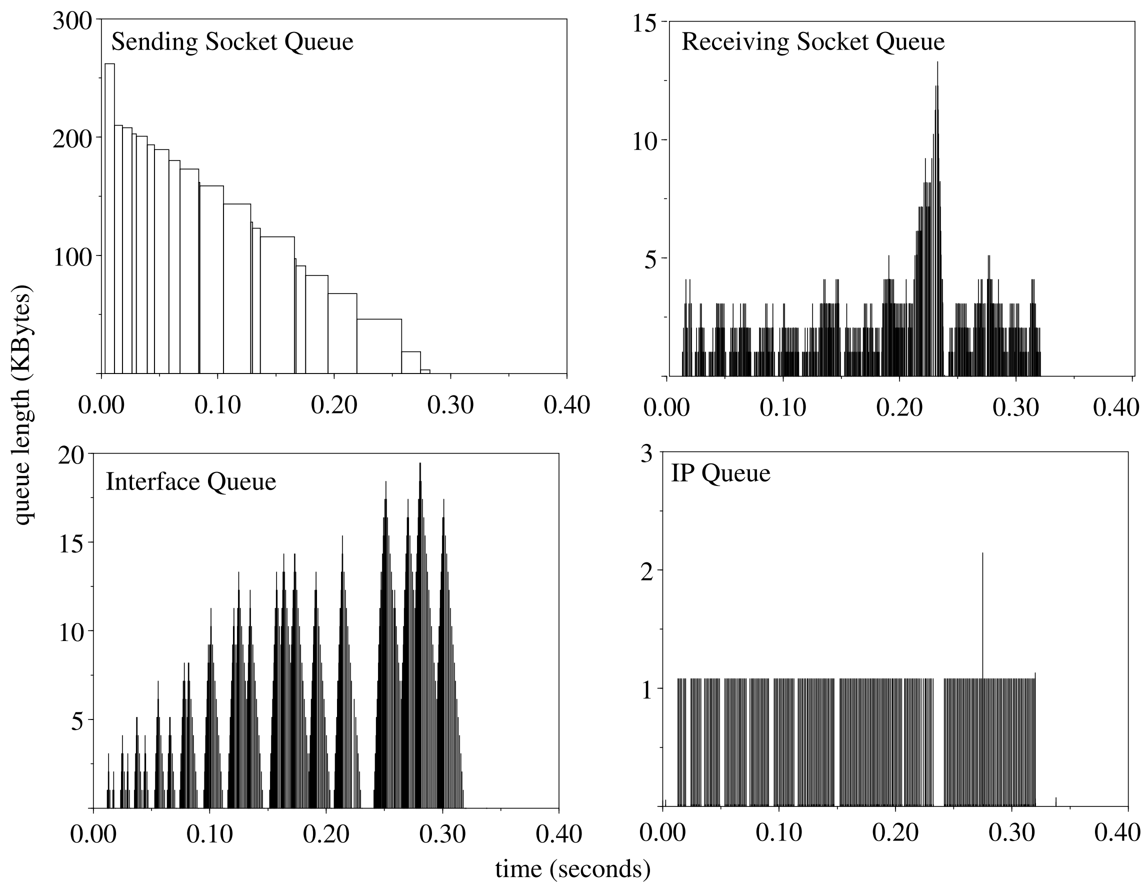

Figure 5.9: Queue Length during congestion window opening

The sending socket queue length is shown to monotonically decrease as expected, but not at a constant rate since draining is controlled by acknowledgment activity. Note that the point the queue appears to be 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 remain in the socket buffer until acknowledged.

The interface queue reflects the packet generation by the protocol. It is immediately apparent that there is burstiness in packet generation, as manifested by the queue length peaks. The queue build-up suggested by the packet trace graphs is clearly visible. Also the queue length increases with every new burst, indicating that each new burst contains more packets than the previous one. This behavior is attributed to the slow start strategy of the congestion control mechanism [10]. 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 next one arrives.

The queue length graph for the receive socket reveals some queueing. 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 build-up 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 are removed from the socket buffer.

Congestion Control Mechanism

The congestion control mechanism has two parts: slow start and congestion avoidance. The sender maintains two windows: a flow control window based on feedback from the receiver, and a congestion window based on congestion in the network. The sender uses MIN (flow control window, congestion window). At the beginning of a new connection the window is set to four segments. The slow start mechanism increments the window by one with each new acknowledgment received, so the increase in window size is exponential. If during data transfer a packet loss is detected, the congestion window is reduced to one and the slow start mechanism is activated again. This time, when the window reaches half the original window the congestion avoidance mechanism is activated. During congestion

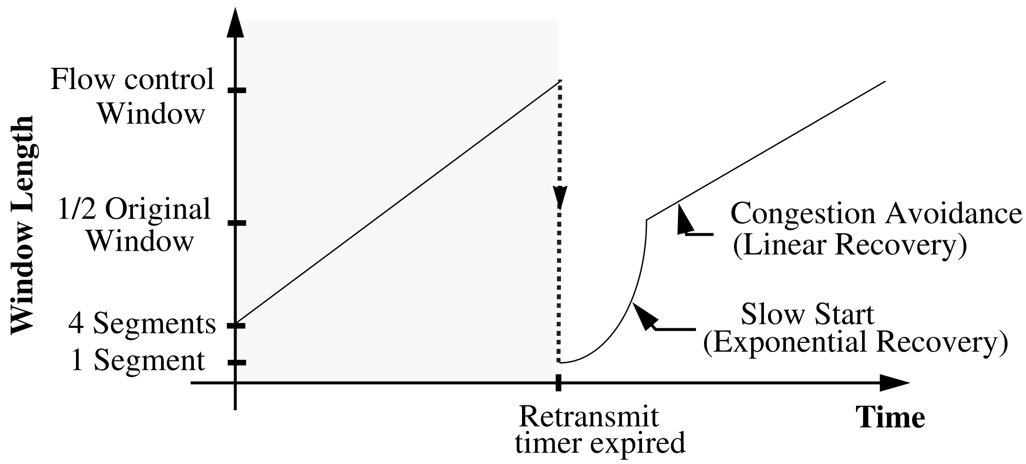

Figure 5.10: Congestion control: Slow-start and congestion avoidance

avoidance the window is incremented by one only when all the segments in the previous window have been acknowledged, leading to a linear increase in window size. The effect of the congestion control mechanism on window size is depicted in Figure 5.10. The x-axis shows the time and the y-axis the window size. The slow start mechanism can also be observed in the interface queue length plot in Figure 5.9.

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 5.11, 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 5.12 shows the packet delay at the four queues. The x-axis shows the elapsed time and the y-axis the delay. Each packet is again represented as a vertical bar. The position of the bar on the x-axis shows the time the packet entered the

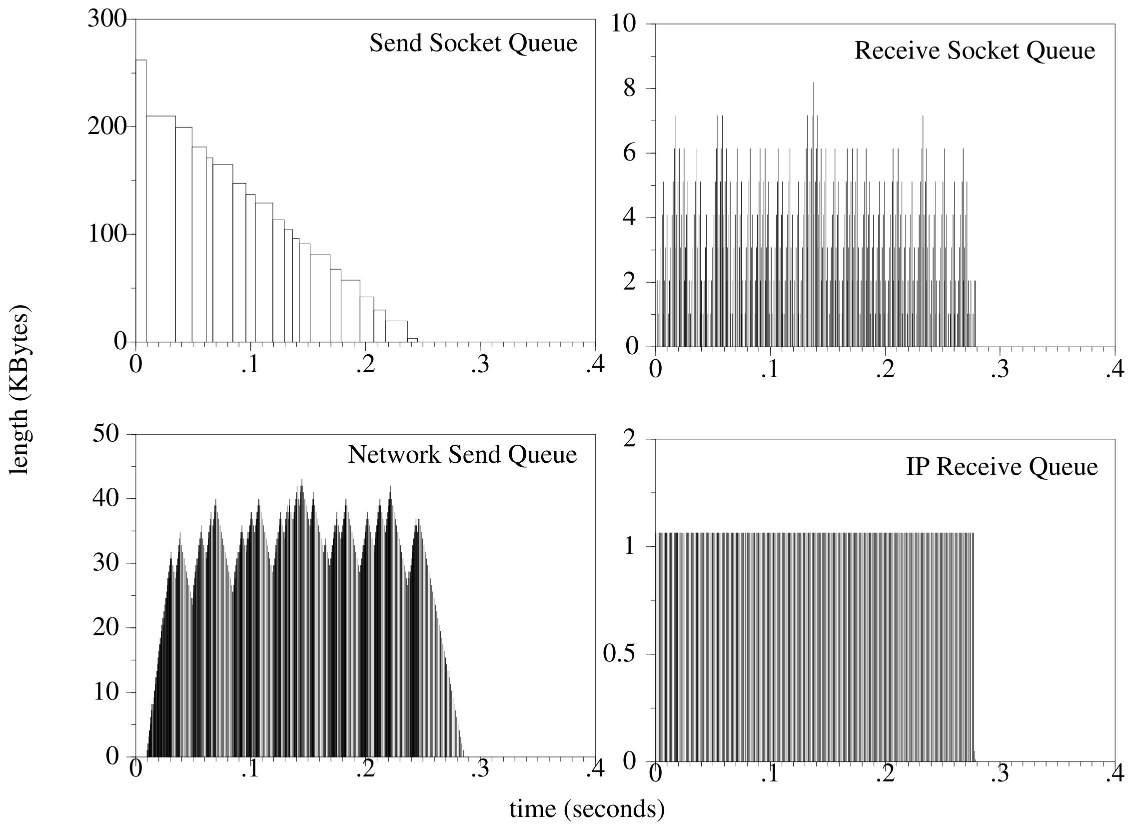

Figure 5.11: Queue Length after congestion window has opened

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 mil-

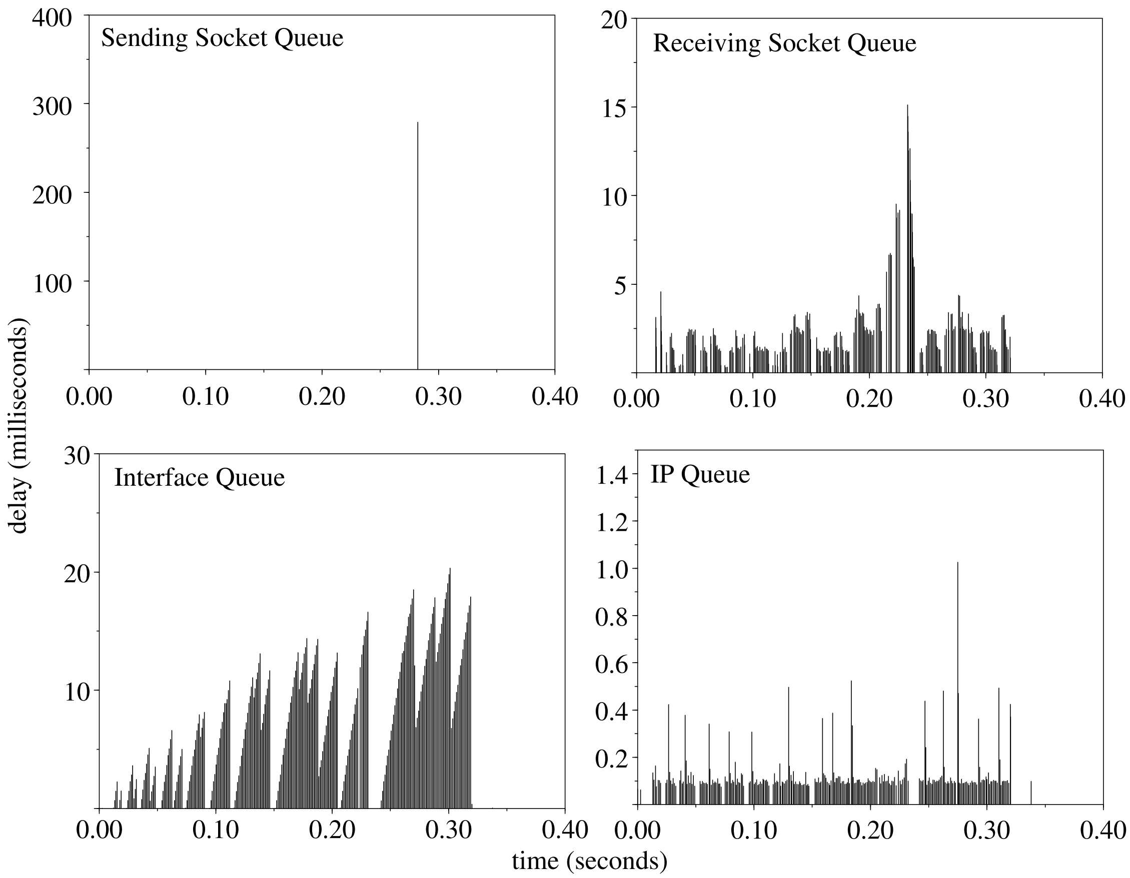

Figure 5.12: Queue Delay during congestion window opening

liseconds.

The third graph shows the IP queue delay which is a measure of how fast the protocol layer can remove and process packets from the input queue. The packet delay appears very small, with the majority of packets remaining in the queue for about 120 microseconds. Most of the packets are removed from the queue well before the 1 ms interval[1] that it takes for the next packet to arrive.

The fourth graph shows the delay at the receive socket queue. Here the delays are much higher. This delay includes the time to copy data from mbufs to user space. Since the receiv-

1. Determined when measuring the driver performance

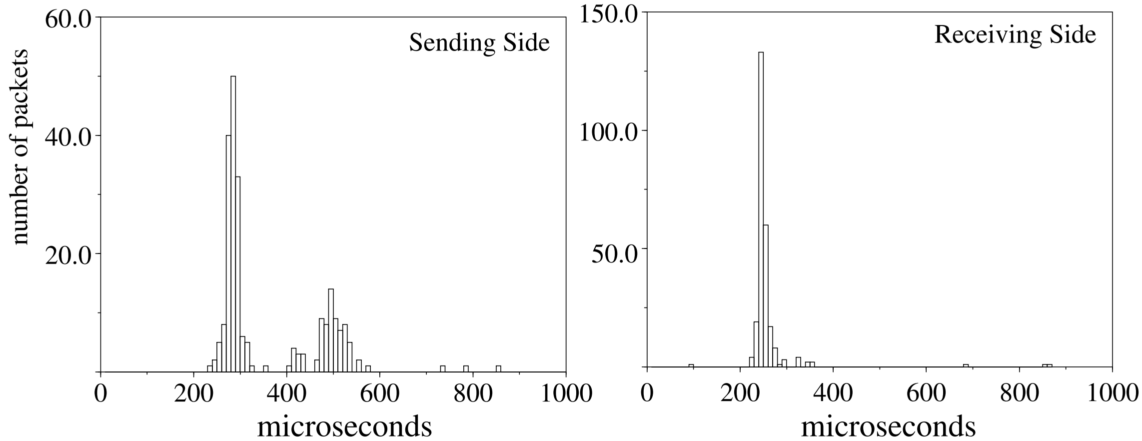

Figure 5.13: Protocol Processing Delay Histograms

ing 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 cannot keep up with the data arrival.

Protocol Processing Delay

The two histograms in Figure 5.13 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 which are 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 microseconds 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) requires about 250 to 400 microseconds of processing, and the second 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 a Sparcstation 1 when the CPU is devoted to communication with no data copying and network driver overhead, is estimated 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.

5.4.2 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 processing. Moreover, asynchronous interrupts due to packet reception or transmission may steal CPU cycles from higher layers. To isolate layer processing this experiment was performed, where consecutive packets are sent after introducing some artificial delay between them. The aim is to allow the IPC mechanism enough time to send a packet and receive an acknowledgment before supplied with the next packet.

The experiment was set up as follows: the same pair of Sparcstation 1's was used again in a 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 kilobyte, 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

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

results obtained in the previous experiment when 256 kilobytes of data were transmitted.

On the sending side, the send socket queue delay shows that there is a delay of about

280 µS 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 is estimated at 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 in section 5.3.2 it was found that it takes an additional 1150 µS for the packet to be transmitted.

On the receive end, the IP queue delay is also quite small. The delay has not changed significantly from the earlier experiment, when 256 kilobytes of data were transferred. The receive socket queue delay is about 412 µS, out of which 130 are 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.

5.4.3 Probe Overhead

The measurements presented above include probe overhead. The overhead is small, dominated by the timestamp call. To estimate on the overhead, throughput measurements were taken with and without the probes. The results showed that the throughput dropped slightly when the probes were present. The drop was about 8 percent. Without probes, throughput values in the range of 7 to 7.5 Mbps were obtained, dropping to about 6.6 Mbps with the probes 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.

5.5 EXPERIMENT 4: EFFECT OF QUEUEING ON THE PROTOCOL

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

5.5.1 Effect of Receive Socket Queueing

A detailed description of the actions taken at the receive socket in order to receive data is presented in Chapter 4. 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 a very important protocol action - the generation of acknowledgments - 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 calls the TCP user-request function, which in turn calls the TCP output function if an acknowledgment must be sent. During normal data transfer (no retransmissions) an acknowledgment is sent after the application (1) removes at least 2 maximum segments from the receive buffer, or (2) leaves the buffer empty, or (3) 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 in all existing connections. For each connection with the TF_DELACK flag set, the timer routine changes the flag 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.

---

1. See description of tcp_output() in Chapter 4 for more details

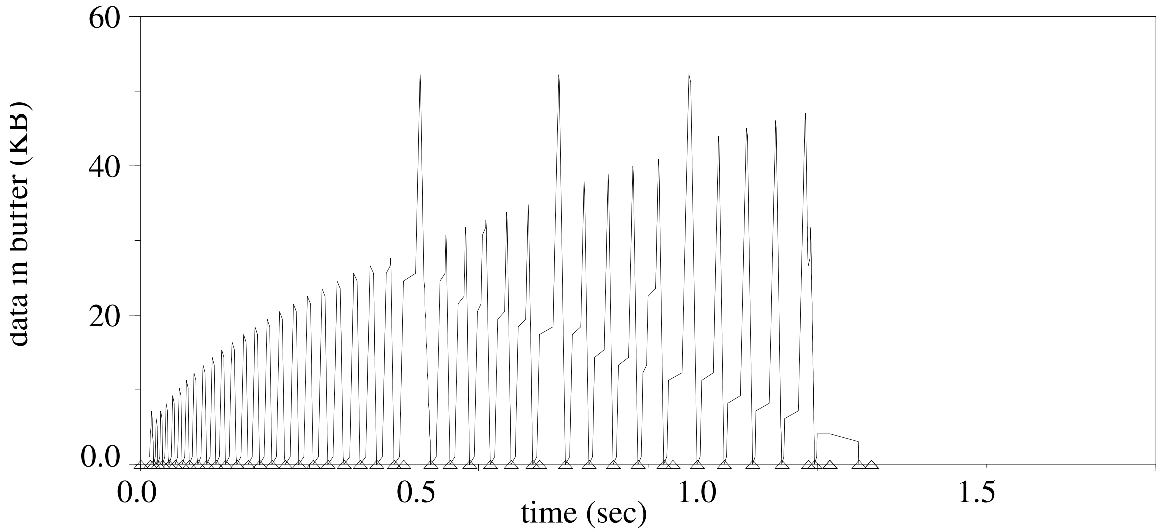

Figure 5.14: Socket queue and acknowledgments on loopback

To verify the above an experiment was performed monitoring the receive socket buffer and the generated acknowledgments. The experiment again consisted of 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 5.14. 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 exactly to the full window size, and then drops down to zero where an acknowledgment is generated causing a new window of data to come 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 an acknowledgment is sent before data in the receive buffer are removed, causing the sender to transmit more data that reach the socket buffer and cause these peaks. There do not seem to be other instances when the protocol sends acknowledgments. 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.

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 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.

5.5.2 Summary

This experiment has shown that queueing at the receive socket layer may affect the protocol acknowledgment generation. It is conceivable that at the extreme, 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.

5.6 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 on single-processor machines) computation and communication will affect each other. For example, as described in Chapter 3, the computation part of the OSM application affects the communication part.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.

5.6.1 Communication vs. Computation

The experiment was performed by setting up a unidirectional connection, running the workload, and blasting data from one Sparcstation to another. There are two parts in the experiment: in the first the workload is placed 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

The main requirement for the artificial workload was that it be CPU intensive. The assumption is that if a task needs to be distributed, it will most likely be CPU intensive. The chosen workload consists of a program that forks a specified number of processes calculating prime numbers. For the purposes of this experiment the artificial workload level is defined to be the number of prime numbers processes running at the same time.

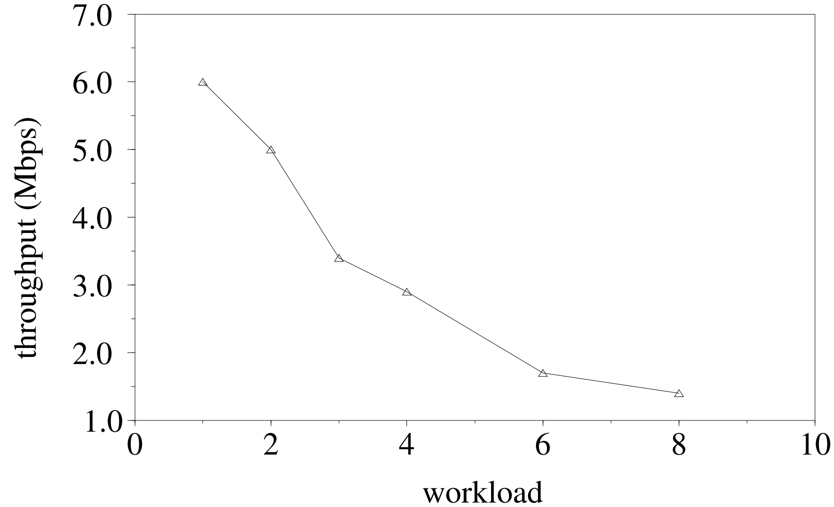

Figure 5.15: Throughput vs. server background load

Loading the Server

Figure 5.15 shows the effect of the artificial workload on throughput when the workload runs on the server. 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.

Figures 5.18 - 5.16 show the internal queues during data transfer with a loaded server. For these graphs the data size was reduced to 1 Mbyte to keep the size of the graphs reasonable. Four different results are shown for every queue, corresponding to equivalent levels of workload. The x-axis shows the time and the y-axis the queue length. Figure 5.18 shows the receive socket queue. It shows that when data arrives at the queue it is delayed for an amount of time proportional to the workload level. 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. Figure 5.17 shows

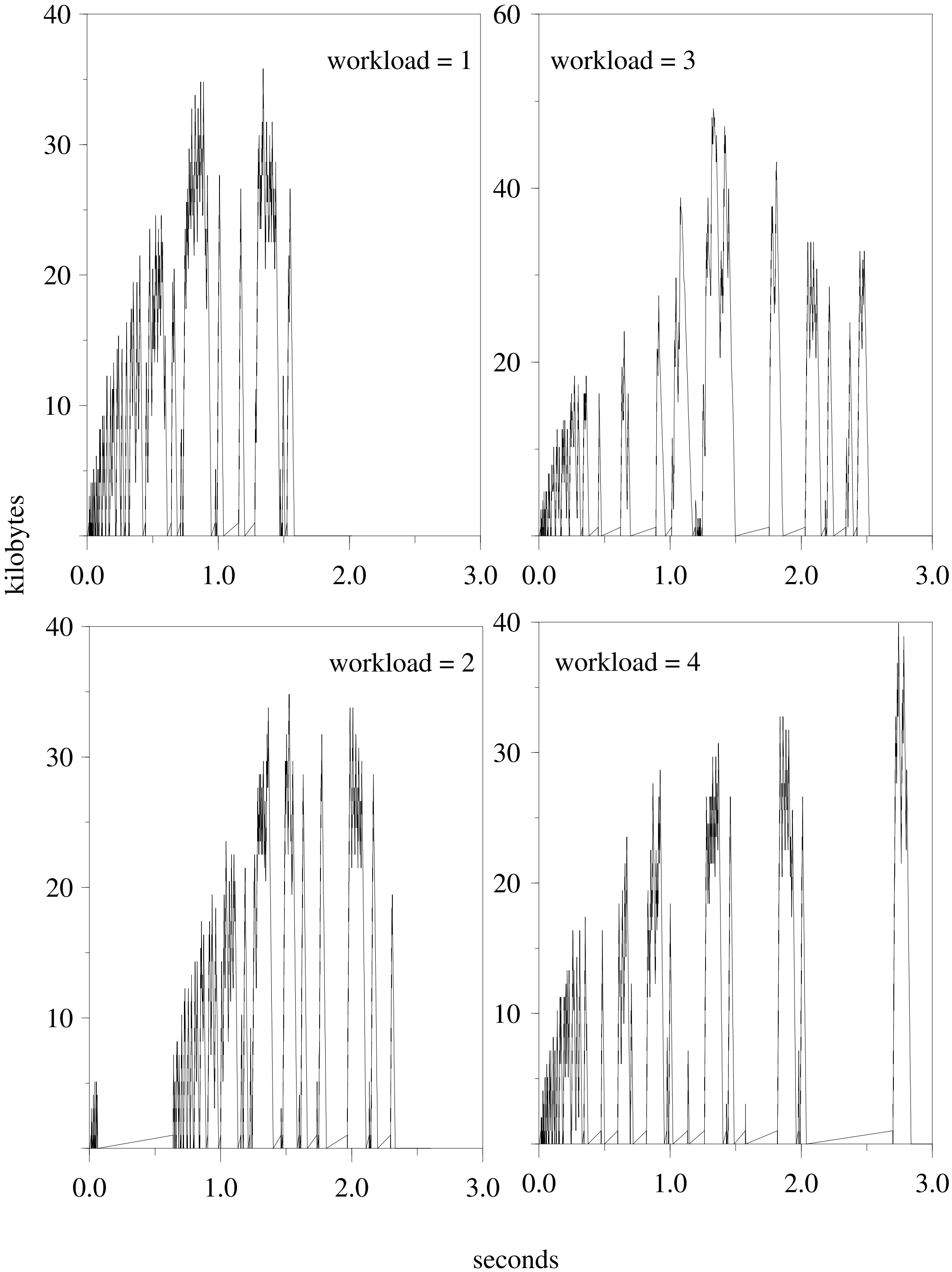

Figure 5.16: Effect of workload on send interface queue

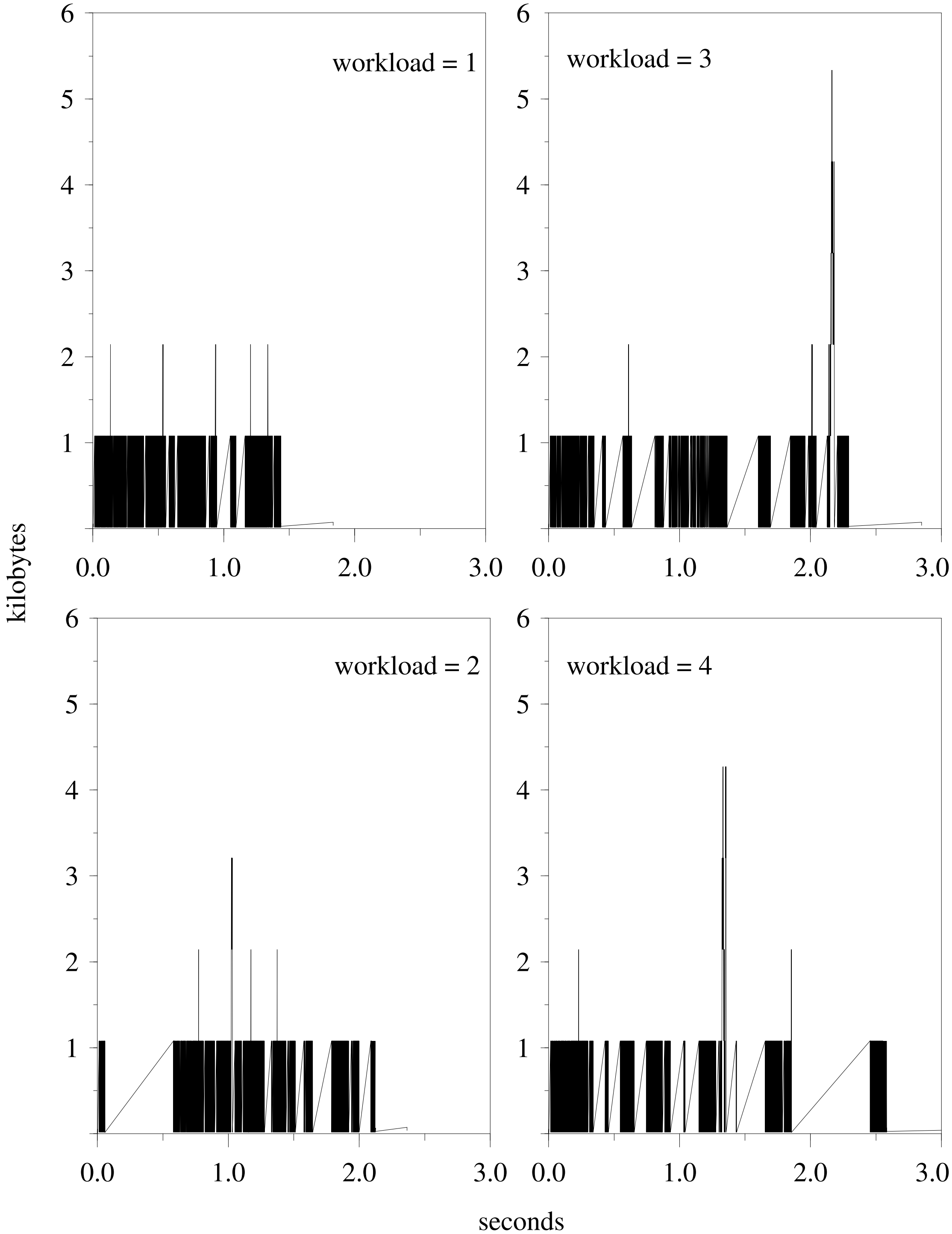

Figure 5.17: Effect of server workload on receive IP queue

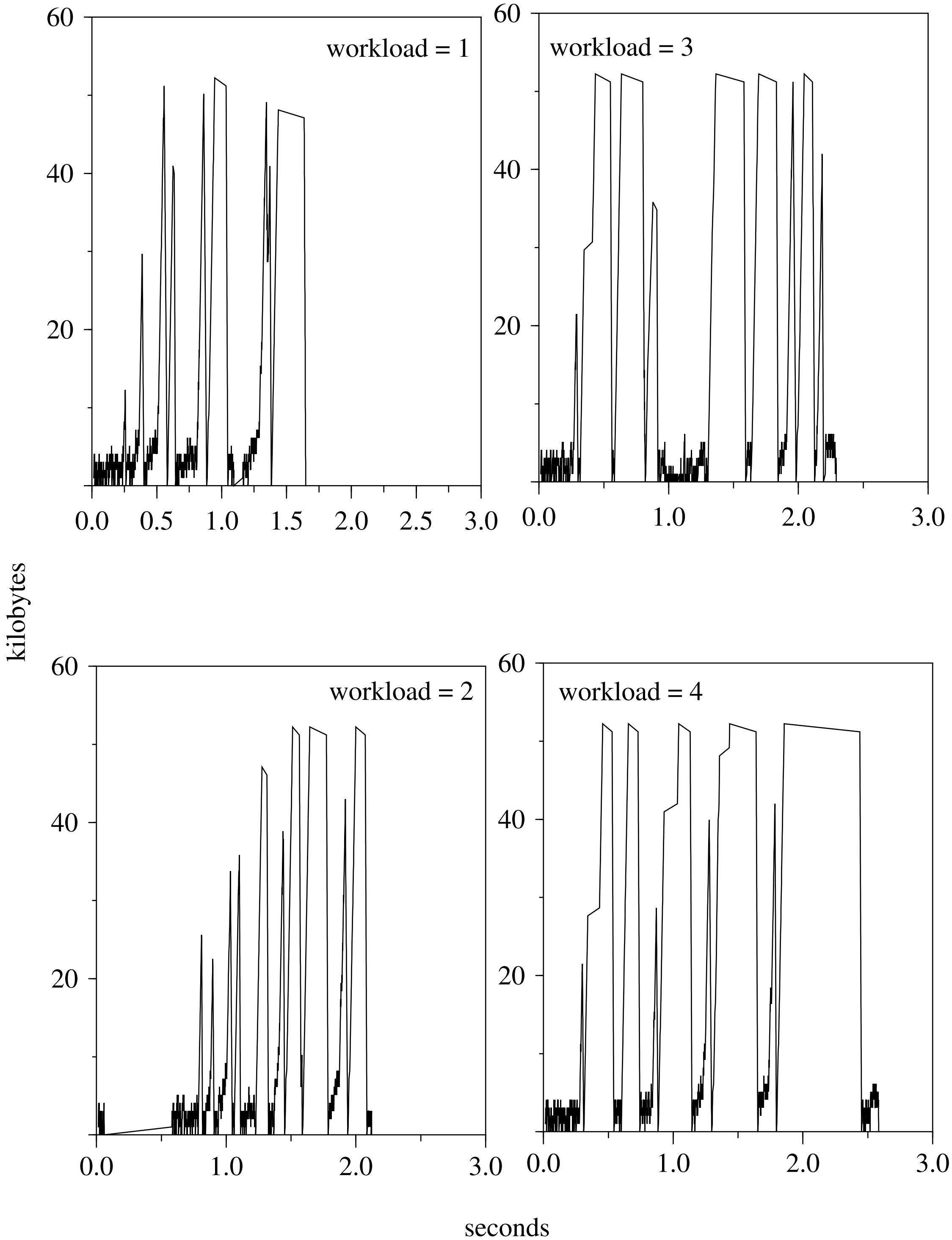

Figure 5.18: Effect of server workload on the receive socket queue

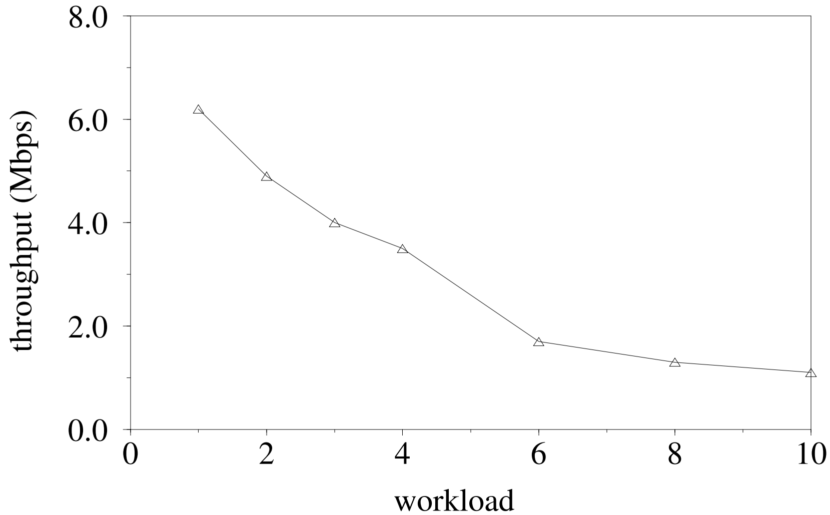

Figure 5.19: Throughput vs. client background load

that 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. As the workload increases data begin to accumulate, although the accumulation is still very small compared to the other queues. Figure 5.16 shows the interface queue on the sending side. The queue reflects the effect introduced by the delay in acknowledgment from the receiver.

Loading the Client

The next graph in Figure 5.19 shows the effect of client artificial workload on the throughput. The graph was obtained by 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. Again, there is a sharp drop in throughput as the workload is increased.

Figures 5.20 and 5.21 show the internal queues when the workload is on the client. Again workload levels of one to four were used and the data transferred was 1 MByte. The graphs show that the sending side is now the bottleneck by introducing delays at the sending end. These delays can be observed in Figure 5.20 which shows the send socket queue,

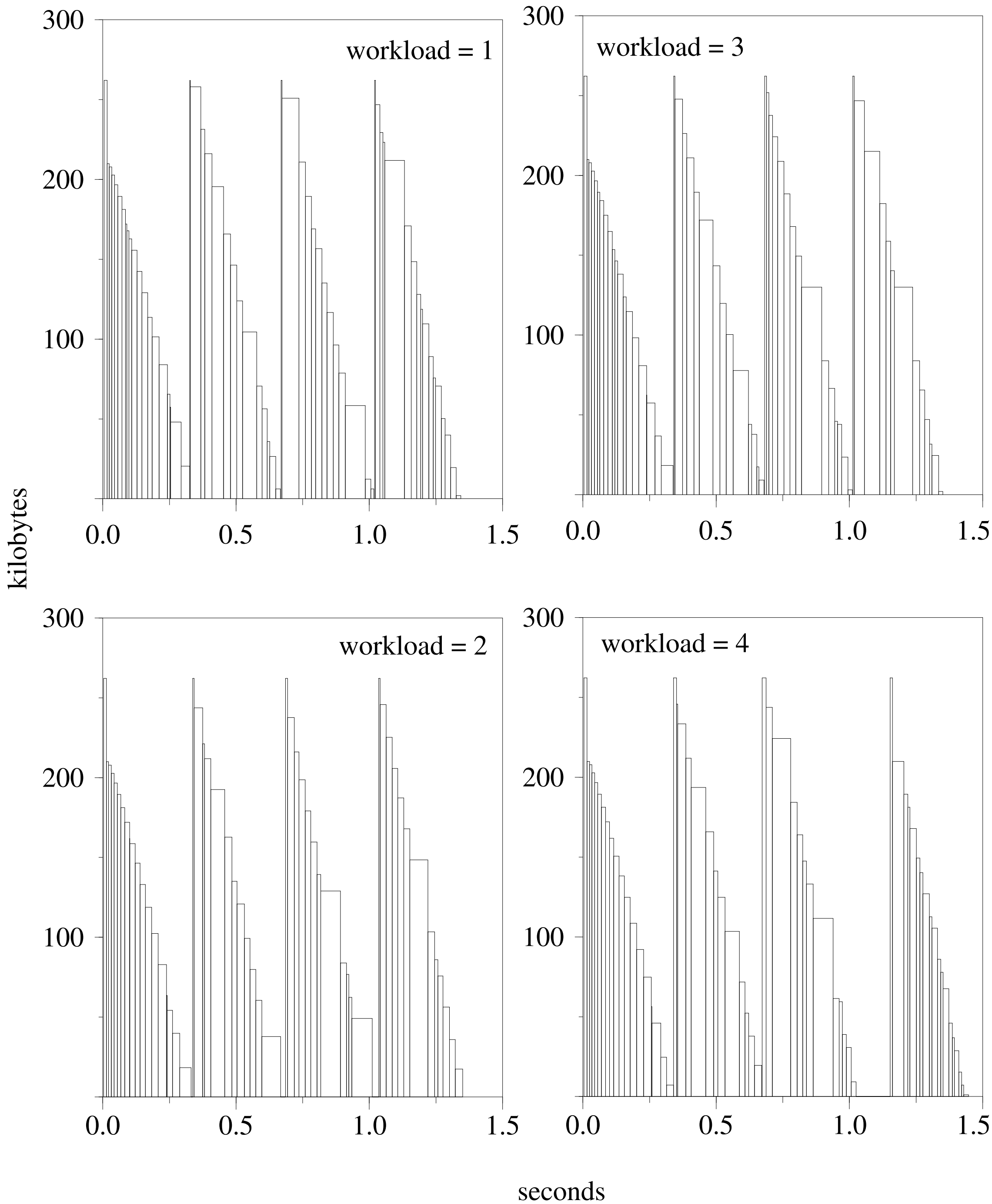

Figure 5.20: Effect of client workload on the send socket queue

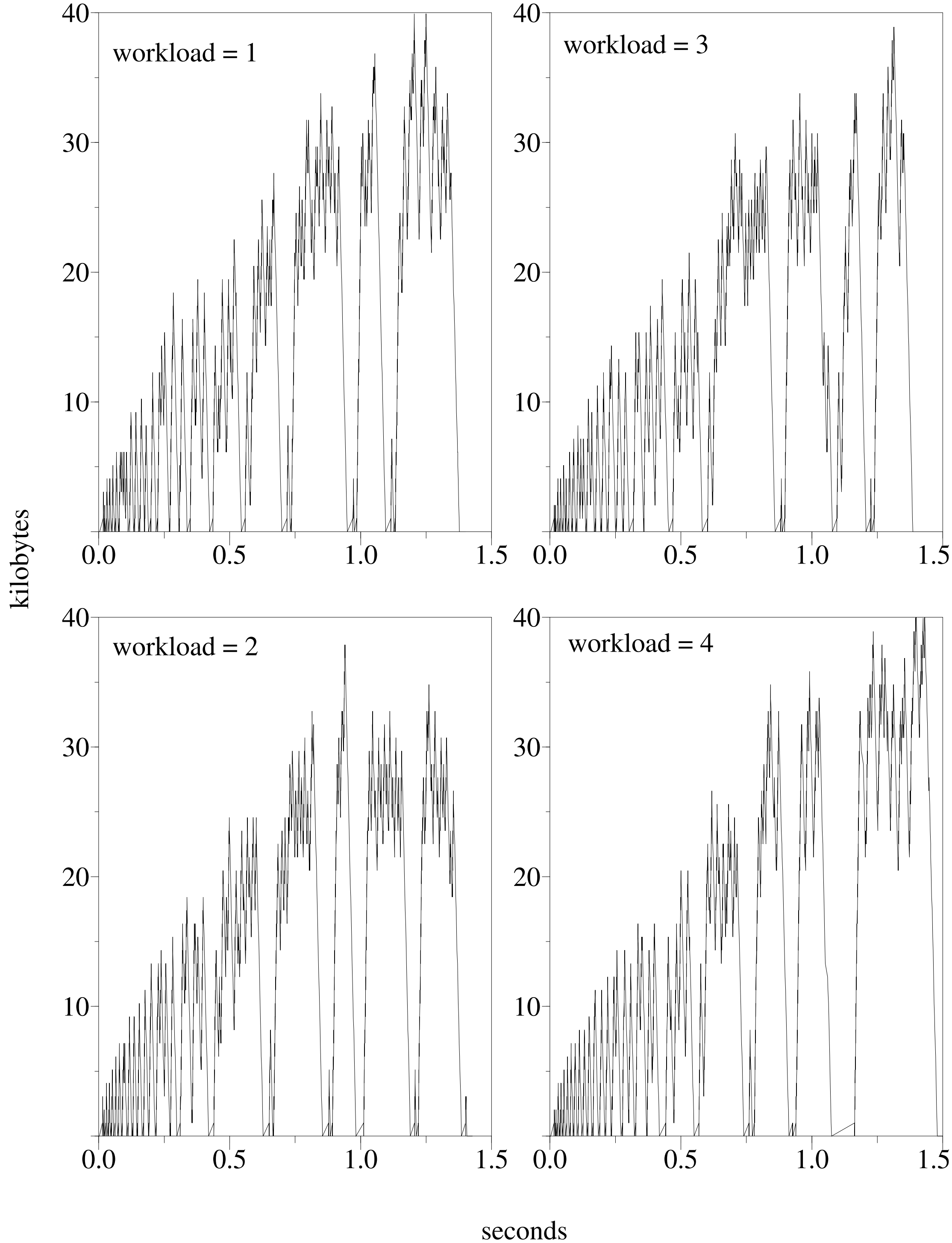

Figure 5.21: Effect of client workload on the send interface queue

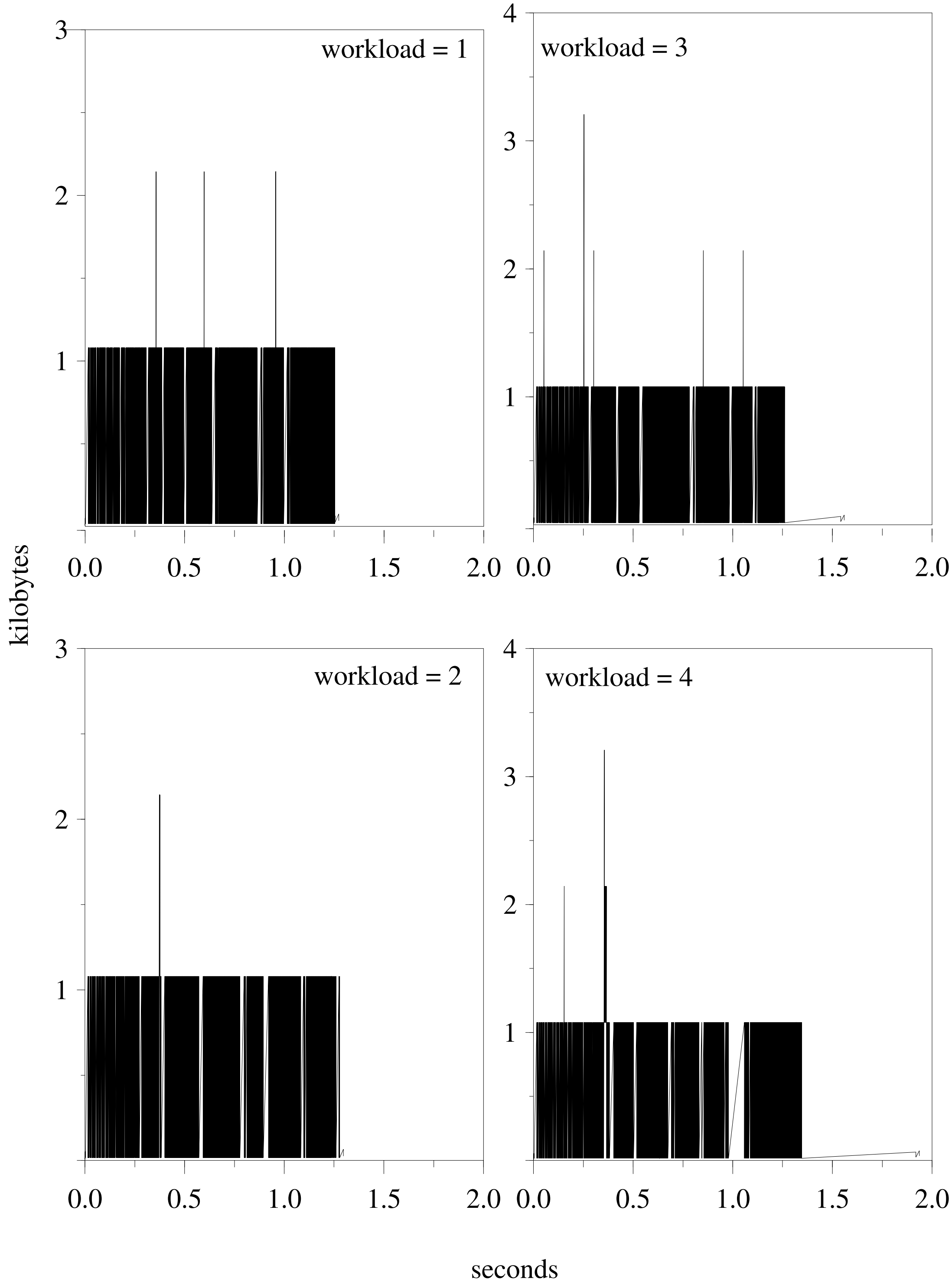

Figure 5.22: Effect of client workload on the receive IP queue

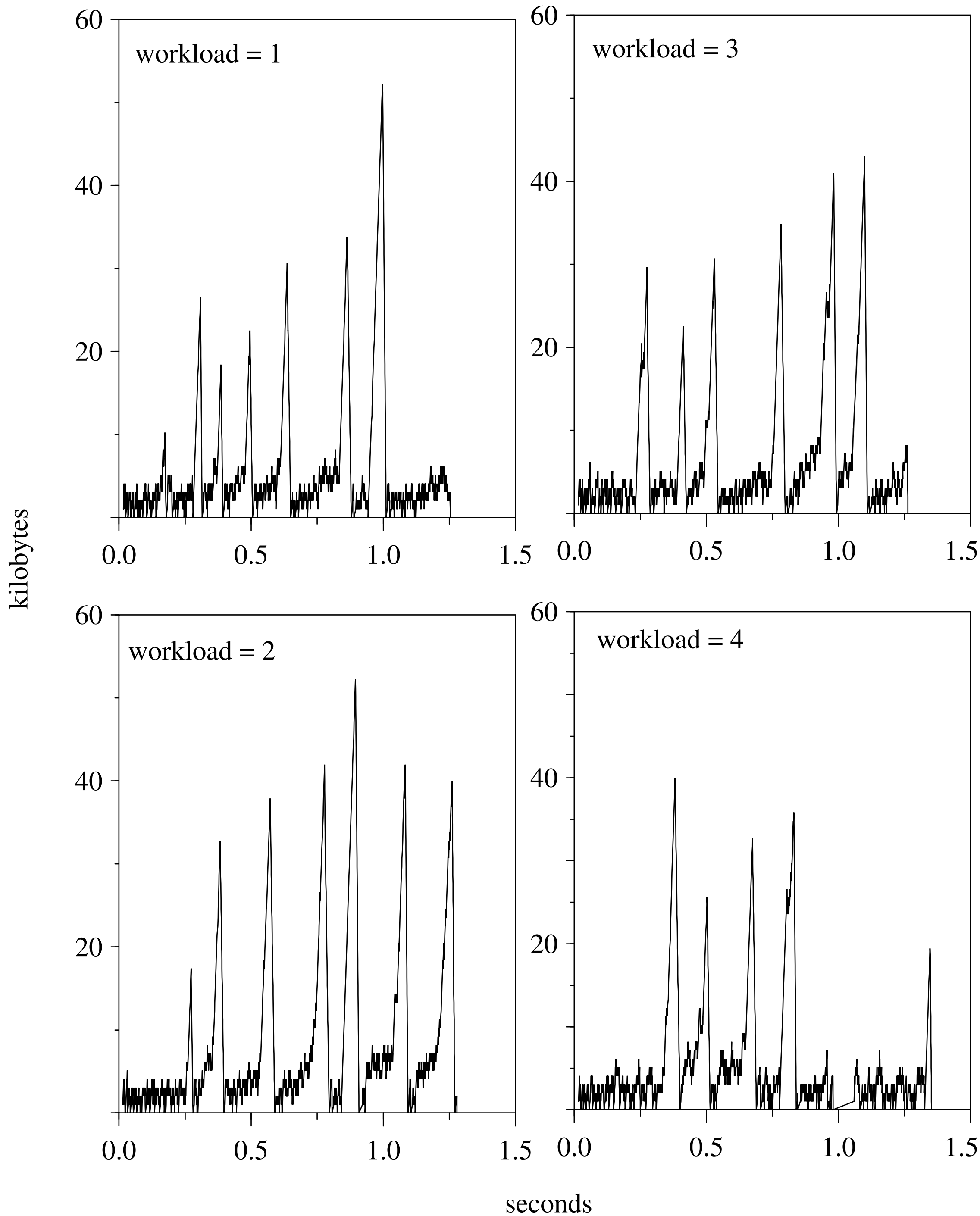

Figure 5.23: Effect of client workload on the receive socket queue

and in Figure 5.21 which shows the interface queue. Both figures show that there are delays introduced to the transfer of data from the socket layer to the protocol. The behavior of both queues is not as smooth as before when there was no load on the machines. Figures 5.22 and 5.23 show the IP queue and the receive socket queue respectively. The graphs show that loading on the client does not change the behavior of these queues compared to the case when there is no extra 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 the increase was small (about 10%) in protocol processing on either side. This was not surprising since the protocol routines run with the network interrupts masked.

5.6.2 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 above hypothesis was confirmed in the next experiment where plots of the various queue lengths are shown while the OSM application was running. It was also confirmed separately by performing the following experiment:

A unidirectional connection was set up to blast data between two Sparcstation 1 machines. The data size was set to 1 MByte. The first part of the experiment was identical to the previous experiments, i.e. the artificial workload process was run and then immediately a connection was established and the data were transferred. In the second part the same procedure was repeated, but this time the workload process was allowed to execute for about 30 seconds before transferring the data. The throughput obtained showed that when all processes start within a small interval (1 second) of each other, the throughput drops as the previous experiments indicate. However, in the second part the throughput was affected very little by the extra load. The throughput has been measured to be very close to the result obtained without any background load and was generally unaffected by the workload magnitude.

The throughput measurements presented in this experiment should be viewed with the above considerations in mind.

5.7 EXPERIMENT 5: RUNNING THE OSM APPLICATION

The last experiment presents snapshots from the activity of the three processes constituting the threshold module of the OSM application described in Chapters 2 and 3. 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 very often preempted by the communication processes.

Figure 5.24 shows two graphs which depict the duty cycle of the three processes in the threshold module of the OSM application. The second graph superimposes the duty cycle graphs along with the interface queue in order to show the queue length during the module operation. The x-axis shows the elapsed time while processing 3-D images consisting of 4 slices each. The y-axis shows the interface network queue length. The activity of the three processes in the module is tracked by monitoring the time they send messages to the message queue (see description of module operation in Chapter 2). A process is assumed to be active at the moment it receives a message from the global message queue, and is assumed to be idle after it sends a message to the queue. The graphs depict these 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 first graph shows the activity of the three processes after steady state is reached. The processing cycle seems to be 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 after receiving a message, while the compute process is loosing on CPU time from the communication processes.

The second graph also shows the cycles of the three processes, but with the interface queue superimposed in the write cycle. The queue appears to build up considerably as data gathers awaiting transmission.

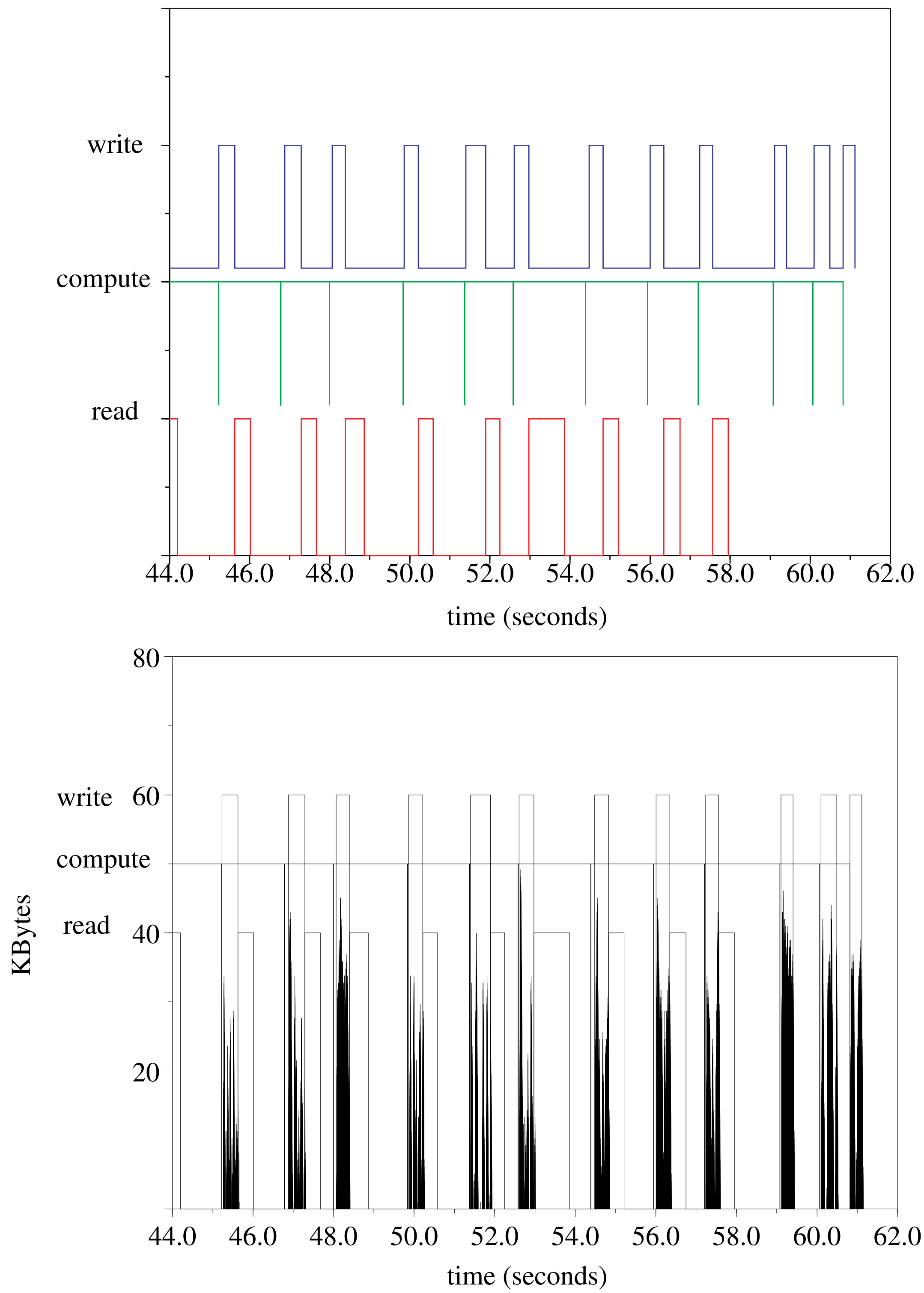

Figure 5.24: Process activity and send interface queue

5.8 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. As shown in Experiment 2, 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 on 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 to the limit of the network interface queue size. The network interface queue shown in Figure 5.6, 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 packet loss due to overflow for 12 simultaneous connections (12 connections with a maximum window of 4). If the default socket buffers are increased to 16 kilobytes 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. This was calculated as follows: 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

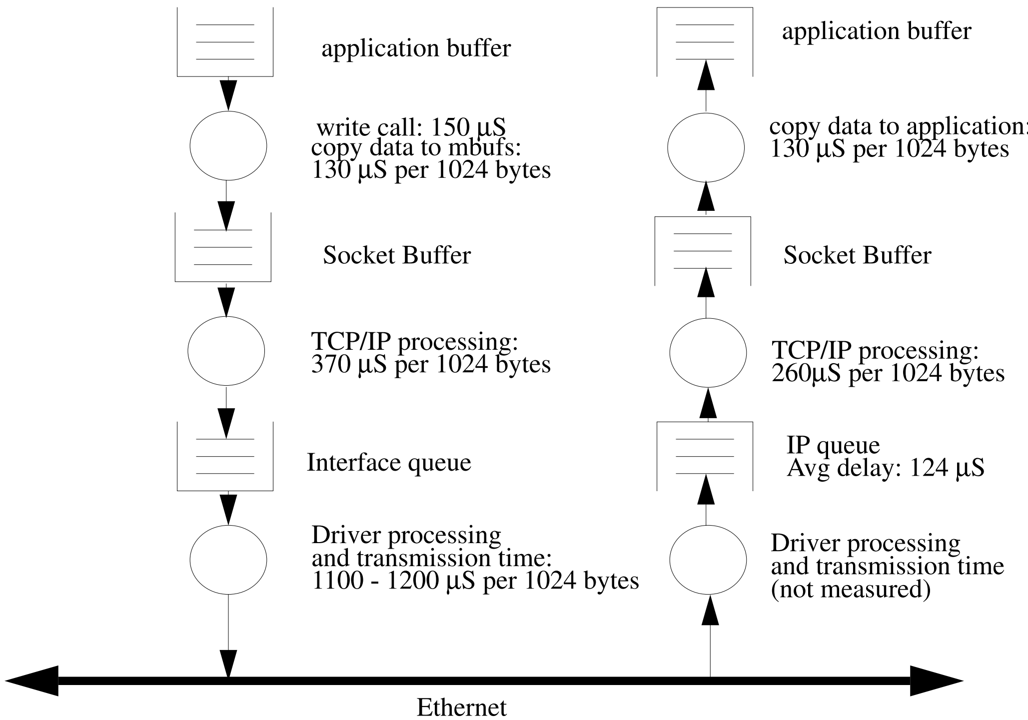

Figure 5.25: Delays in IPC

processing packets at the rate they arrive until they reach the socket queue, where they are queued for delivery to the application. Figure 5.25 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. For the sending side this is about 16 Mbps, and for the receiving side about 21 Mbps.

During large data transfer TCP depends on the socket layer for notification of data reception by the application in order to send acknowledgments. However, the socket layer notifies TCP only when all data have 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-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.