4. UNIX BACKGROUND AND PROBE DESIGN

4.1 INTRODUCTION

This chapter presents an overview of the implementation of Inter-Process Communication (IPC) using TCP/IP stream sockets in SunOS 4.0.3[1]. 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 of Chapter 5. This chapter deals with the first two issues.

The chapter is organized as follows: Section 4.2 presents an overview of the Unix communication layers and their functionality. Section 4.3 presents the memory management facilities provided by the kernel for use in the implementation of these layers. Section 4.4 presents a detailed breakdown of the different kernel routines involved in data exchange. In Section 4.5 a queueing abstraction is extrapolated and used later in analyzing the performance of IPC. Section 4.6 presents a mechanism that uses probes to gather data about IPC performance. Finally in Section 4.7, two related network monitoring tools are presented.

1. SunOS 4.0.3 is based on 4.3 BSD Unix

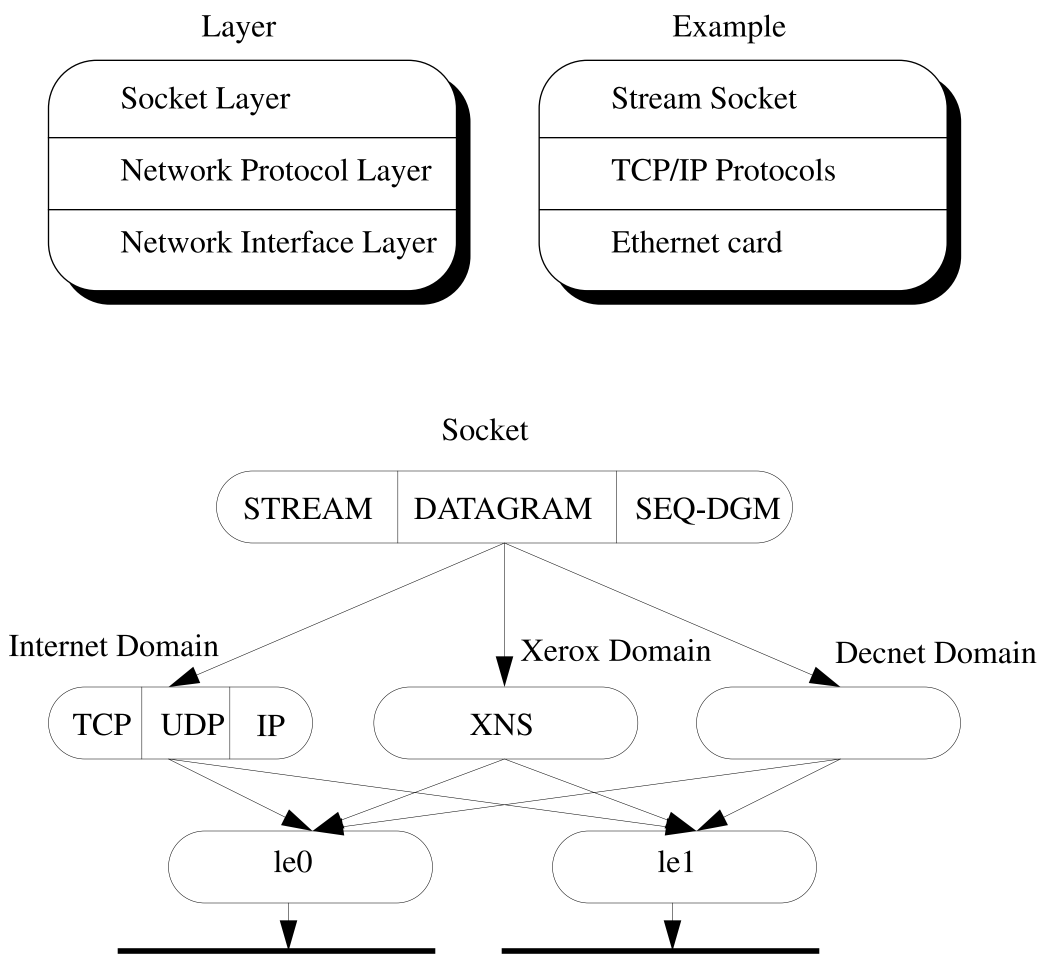

Figure 4.1: Inter-process Communication (IPC) layering

4.2 INTER-PROCESS COMMUNICATION LAYERS

IPC is organized into 3 layers as shown in Figure 4.1. The first layer, the socket layer, is the IPC interface 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 specified by the application, which has the required properties. This collection of protocols forms the protocol layer. In addition, protocols are grouped in domains. For example all Internet protocols are grouped together in the Internet domain. The application can explicitly 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 of communication that supply 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. The protocol will then choose an interface to send data as part of routing. 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.

4.3 MEMORY MANAGEMENT: MBUFS

The memory management subsystem used with IPC can have a tremendous impact on performance This impact is so important that protocol implementers avoid using the general memory management provided by the operating system because its performance incurs too much overhead for IPC. Instead these implementers resort to memory management facilities which bypass those of the underlying operating system. The IPC implementation in Unix 4.3 BSD and SunOS 4.0.3 uses such a memory management facility.

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: (1) the mbuf data structure; and (2) 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 never paged out of main memory.

Mbufs are fixed size memory blocks, 128 bytes long. Up to 112 out of 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 4.2.

The function of the various fields is as follows:

- **m_next**: this is the pointer used to link mbufs together to form a chain. Mbufs

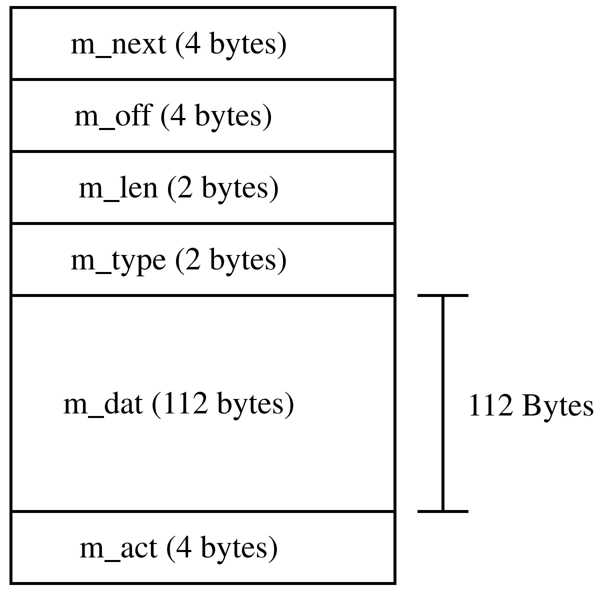

Figure 4.2: Mbuf Data Structure

linked in this manner are assumed by convention to form a single object (for example a packet).

- **m_off**: this is the offset from the beginning of the mbuf where data lies. This field allows efficient deletion of data at the front by simply incrementing the offset value. Protocols routinely need to strip packet headers from incoming packets, and this scheme allows them to do it efficiently.
- **m_len**: this is the length of the data currently residing in the mbuf. In addition it allows efficient deletion of data at the back to remove trailers.
- **m_type**: this is the type of an mbuf. Free mbufs are marked MT_FREE. Other mbufs are marked depending upon their use (for example MT_HEADER).
- **m_act**: this field is used when it is desirable to link together objects (mbuf chains) to form a list. For example, this would be used to hold together individual packets queued in the socket buffers received by a protocol which preserves message boundaries.

The mbufs presented above 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. In cluster mbufs the m_off field is again used to point to the data, but this time it points to a memory location outside the mbuf structure. Data in cluster mbufs are stored exclusively in the associated page. The internal space of a cluster mbuf is never used for data.

Storing data in external pages allows for efficient copy operations by using the following trick: each page is associated with a global reference counter. When a copy of a cluster mbuf is desired the data in the page is not actually copied, but instead the page's global counter is incremented. Freeing the page simply involves decrementing the page reference counter. A page becomes free when its reference counter reaches zero.

SunOS provides yet another type of mbuf. In this type 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 and a pointer to a function to be used when deallocating this mbuf. These mbufs are used in the Ethernet interface driver and facilitate mbuf "loaning". Loaning occurs when an mbuf owned by the driver is loaned temporarily to higher level routines. The advantage of this scheme is that a copy from the memory mapped to the network interface to mbuf space is avoided.

4.3.1 MBUF Operations

Mbufs are allocated from a pool of free mbufs through calls to m_get() or m_getclr(). They are freed using m_free() which frees a single mbuf, or m_freem() which frees a chain of mbufs. In addition the following operations are also available:

- **m_copy()**: make a copy of some or all of an mbuf chain
- **m_adj()**: adjust the data either at the front or the back of an mbuf by manipulating the m_off and m_len fields in the mbuf
- **mtod()**: take a pointer to the mbuf header and return a pointer to the data cast to the desirable type
- **dtom()**: do the reverse operation from `mtod()`
- **m_pullup()**: rearrange an mbuf so that a specified size of data stored internally in the mbuf is contiguous, and can be treated as a normal data structure (This is useful when accessing the different fields in protocol headers)

4.4 IPC FUNCTION CALLS FOR DATA EXCHANGE

To better understand how data flow from the sending process' address space to the receiving process' space, a trace of the journey of data on their way from the sending process to the receiving process is presented in this section. The focus of the presentation is on clarifying the operations in each layer. As an aside this description will also help anyone studying the internals of the IPC mechanism and ease the laborious task of reading kernel code. The discussion is limited to the data exchange part of IPC. It does not consider connection setup, tear-down, routing, or any other related issues. It assumes that a connection has already been established, and each process has an open socket through which it can communicate. During the discussion a number of auxiliary functions are mentioned. For brevity the operation of these functions is not discussed.

There is a fair number of functions involved in exchanging data. Figure 4.3 shows the main functions 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() implies sending data, and read() implies receiving.

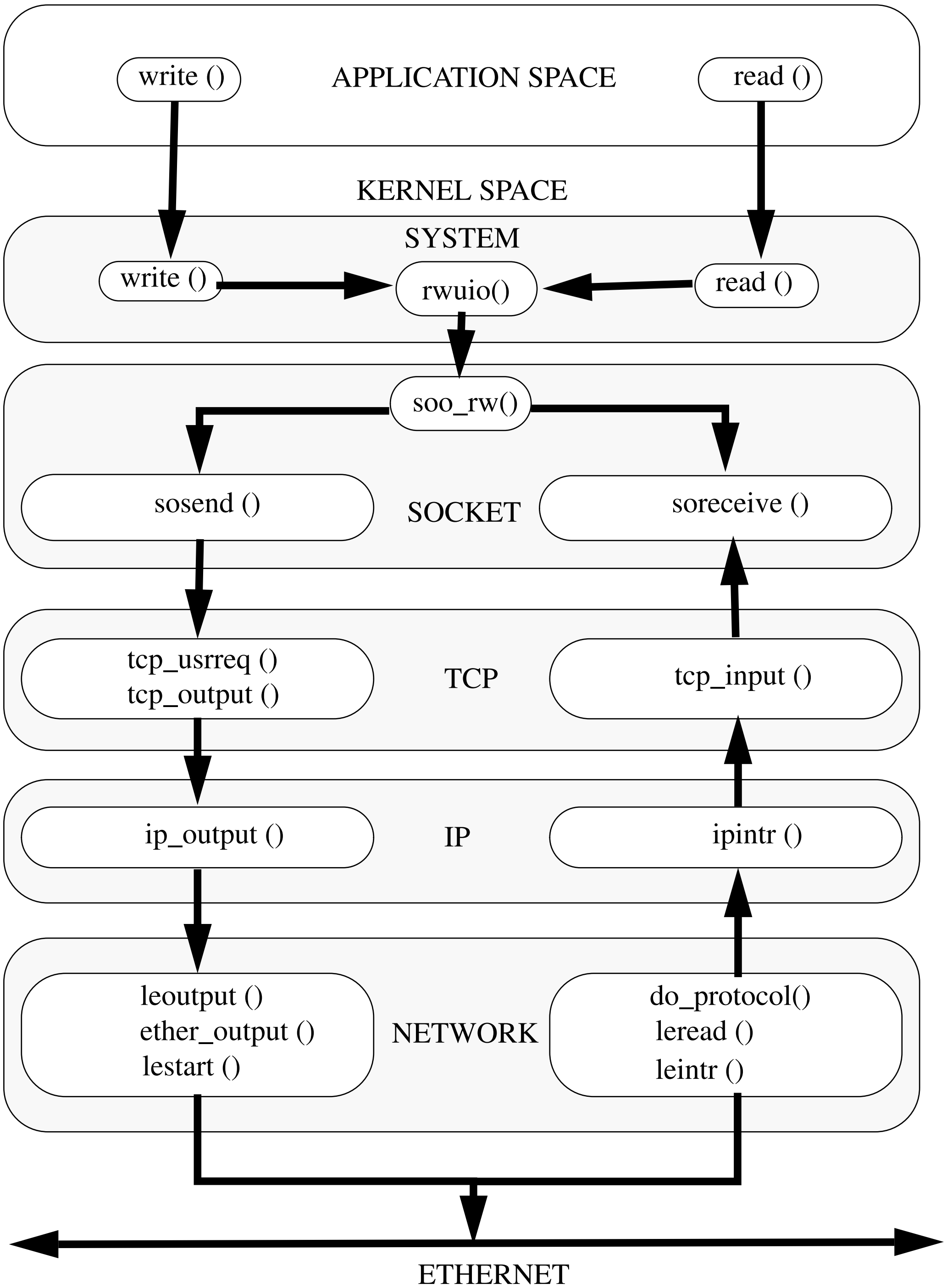

Figure 4.3: Function call sequence

4.4.1 Data Transmission

User level function write (): Data to be transmitted by an application reside in a buffer in user memory. Since all networking services reside in the kernel address space, the process notifies the kernel that it wishes to send data by means of the system call write(). Write() takes as parameters the socket identifier the data should go out on, a pointer to the buffer containing the data, and the size of data in bytes. At this point the operating system pushes the parameters on the stack and switches to the kernel address space.

System level function write (): In kernel space a kernel function also called write() is invoked. This function takes no parameters as it expects to find them on the stack. Its job is to encapsulate the data in a uio (universal input/output) structure and then call the function rwuio() (read/write universal I/O). A parameter in the call specifies the action (writing in this case).

System level function rwuio (): In rwuio() after some checking of the data, the file descriptor is examined to find out which I/O function should be called next. In the case of a socket the appropriate function is soo_rw(). Note that neither write() nor rwuio() are functions specific to IPC, but are generic I/O functions used by the operating system.

Socket layer function soo_rw (): The function soo_rw() is the first function in the socket layer. It is simply a switch statement that switches between reads and writes, calling the function appropriate for each case. For a write that function is sosend().

Socket layer function sosend (): Sosend() is responsible for converting a user transmission request (which can be arbitrarily large) into one or more protocol transmission requests (which must be equal or less than the remaining space in the socket buffer). Therefore, if data in the user request plus data queued for transmission exceed the socket's buffer size, sosend() will break it up into smaller requests. For some socket types it is unacceptable to break up the request. If for example, message boundaries must be preserved (IP datagrams) and there is not enough space in the send queue, sosend returns with an error. Stream sockets are usually allowed to block the execution of the process if there is not enough buffer space available. In this case the process is put to sleep until more space becomes available. Finally, the data copy from user to kernel space takes place in this function.

The steps in sosend can be summarized as follows: First sosend locks the socket's send buffer and enters the sending loop. While in the loop it masks network interrupts to block network - protocol modules from changing the state of the socket. Then it determines the amount of space available in the send buffer, and compares it with the length of the request. If there not enough space is available the following strategy is used to decide whether to put the process to sleep: if data in the user request is more than a page long (1024 bytes in this implementation) and more than a page of data is sitting in the buffer and the socket is not marked as non-blocking, then the buffer is unlocked and the process is put to sleep. Otherwise the network interrupts are unmasked and sosend() proceeds to send as much data as it can.

The next step after determining how much data to send, is to copy the data from user space into an mbuf chain. The size of data will determine whether small or cluster mbufs will be used. Small mbufs are used when the data to be copied is less than half a page (which in this implementation is defined by the constant MCLBYTES = 1024). Otherwise a cluster mbuf is allocated. If cluster mbufs are exhausted, the remaining data goes into small mbufs. For each allocated mbuf the function uiomove() is called to perform the actual copy. The mbuf chain completes when either data or buffer space is exhausted. The mbuf chain is then passed to the socket's protocol user request function with the PRU_SEND flag.

Finally, if there is more data remaining in the uio structure the sending loop is repeated. Otherwise the loop is terminated, the socket's send buffer is unlocked, and any possible errors are returned. Note that the mbuf chain is not yet appended to the socket send buffer.

TCP layer function tcp_usrreq (): This is the TCP protocol interface to the socket abstraction. There are several requests serviced by this function. Data transfer is accomplished via the PRU_RCVD and PRU_SEND requests. The former request is made when data is received by the protocol and copied to the application space. The PRU_RCVD will be discussed in more detail later when examining the data reception path. For sending, the tcp_usrreq function is called with the PRU_SEND request.

The operation the user request function is described as follows: Tcp_usrreq() first masks all network interrupts. Then if the request is a control request, another function (in_control) is called to handle it and tcp_usrreq returns. If the request is a data request, tcp_usrreq searches for the socket's control block data structure (by calling the function sotoinpcb). If the search fails an error is returned. Otherwise, the connection's protocol control block[1] is located from the socket control block (by calling intotcpcb).

There are only a couple of steps left for tcp_usrreq to send data. The first is to append the mbuf chain to the socket send buffer and the second is to call the tcp_output() function.

TCP layer function tcp_output (): Function tcp_output() handles the TCP output processing. The processing includes transmission of data and acknowledgments. All TCP output state information is kept and maintained by this function. The operations of tcp_output include avoidance of silly window syndrome, congestion control, decision on when to send a window update, and handling retransmissions. Since TCP is a reliable stream protocol, a

1. The protocol control block is a data structure created by the protocol for each connection. The protocol keeps information relevant to the connection in this block.

copy of the data is kept in the socket buffer, meaning that `tcp_output()` needs to make a copy of the outgoing data. Note that if cluster mbufs are used this is not a real copy, but an increment to the page reference count of the mbuf. For small mbufs an actual byte copy takes place.

The actions taken in tcp_output are as follows:

1. Determine the length of the data to go out as the MIN (data length, current window). The current window is set as the MIN (space in receive socket buffer, congestion window). If this length is greater than the maximum segment size for this connection, then a portion equal to the maximum segment size is used to form a packet (i.e. fragmentation occurs). For local connections the value of the maximum segment size is 1024 bytes to allow efficient use of cluster mbufs (true for SunOS 4.0.3. SunOS 4.1 uses 1460 bytes).
2. Set the packet sequence number from snd_next field in the TCP control block, and the acknowledgment number from the rcv_next field.
3. Set the window to be advertized to be equal to the space remaining in the socket's receive buffer.
4. Perform silly window avoidance checks.

The sending cycle consists of the following: first a new mbuf for the TCP/IP header is allocated. Then, data to be sent (if any) are copied to a separate chain of mbufs using m_copy(). Next, the header template is copied to the header mbuf, and the appropriate fields are filled out. If options are present they are placed in a separate mbuf and inserted in the chain. The receive window is then checked and set to zero if it is less than 25% of the socket's hi-water mark and less than the maximum segment size. Checksuming is performed next with a call to in_cksum(). The snd_next value is updated, the transmission is timed to get the round trip time, the retransmission timer is set, and finally a call to ip_output() is made passing it the new mbuf chain. Upon return from ip_output() the advertised window is recorded if it was larger than any previous advertised window. If there is more data waiting to be transmitted, the above operations are repeated.

IP layer function ip_output (): Ip_output() performs all IP pertinent processing. This consists of inserting IP options (if any), filling in the remaining header fields (IP version, IP header length and a new packet identification), and determining the interface the packet needs to go out. The packet may be fragmented if it is bigger than the maximum packet size for the outgoing interface. When the packet is ready the IP header checksum is computed, and the packet is passed to the network interface routine.

In general IP output processing is minimal for packets coming from TCP. Most of the work (like allocation of the header mbuf) was done by TCP. There are no data copies or any complicated checks, and there is minimal state information that needs to be maintained.

Network layer function leoutput (): Function leoutput() simply calls ether_output() passing all the input parameters it has received, in addition to the address of the specific function to be called which starts the network device driver. For the AMD LANCE Ethernet controller this function is lestart().

Network layer function ether_output (): Ether_output() is the Ethernet generic output routine. The routine deals with the Ethernet in general and not with the specifics of any particular Ethernet controller. Ether_output() encapsulates the mbuf chain into an ethernet packet which is then queued for transmission. Most of the code in ether_output() deals with broadcasting. If the packet is not destined for broadcast, then ether_output() appends the ethernet header (either in the first mbuf of the chain if there is space, or by allocating a new mbuf), and attempts to enqueue the packet to the interface queue. If the queue is full, random_drop()[1] is called to drop a packet. Otherwise the packet is enqueued and a call is made to the device driver.

Network layer function lestart (): The lestart() function is part of the AMD7990 Lance Ethernet controller driver. This particular routine starts or restarts output on an interface. If the interface is already active nothing happens. if the interface is idle, a packet is dequeued and sent out to the chip for transmission. It is interesting to note that during each iteration only one packet is queued in the transmit descriptor ring of the controller. This essentially prevents the controller from achieving back-to-back transmission of packets. The impact of this on IPC performance is examined in Chapter 5.

4.4.2 Data Reception

Section 4.4.1 described the events that take place during data transmission. In turn this section will now describe the events that take place during data reception. Recall that an application wishing to receive data performs a read() system call. The application receives immediately any data already queued in the receive buffer, otherwise it blocks until data become available.

The discussion describes how data move from the socket buffer to the application buffer. What remains is the discussion of how data get into the socket buffer after arriving from the network. Following the data flow the description begins at the network layer and moves up to the socket layer.

User level function read (): When an application wishes to receive data from a socket it calls read(). The parameters passed to read are the socket from which to read the data, the address of the buffer where data will be placed, and the maximum size of the data the

1. As implied by its name, random_drop() picks a random packet from the interface queue and drops it. The random drop policy is employed for fairness considerations.

application is willing to receive. The operating system traps to the kernel and the kernel function read() is called.

System level function read (): The kernel function read() like its sibling write(), takes no direct arguments but expects to find them on the stack. Its operation is similar to write(), except that it calls rwuio() with the request UIO_READ. The buffer length now denotes the maximum amount of data the application is willing to receive.

System level function rwuio (): This function was described in earlier in this section when describing data transmission. It distinguishes between real file descriptors and socket descriptors, calling the appropriate function for each case.

Socket layer function soo_rw (): As mentioned earlier, this function switches between reads and writes, calling the appropriate function, which now is soreceive().

Socket layer function soreceive (): Soreceive() is responsible for moving data from the socket buffer to the user space and notifying the protocol of data reception.

The operation of soreceive is described as follows: Soreceive() first locks the socket receive buffer and checks for any pending data already in the buffer. If no data exist, a series of tests are performed to detect any errors. If all tests fail, the socket's receive buffer is unlocked and the process is put to sleep waiting for data to arrive. When data arrive, the process is awakened by the protocol with a call to sorwakeup(). The previous tests are then repeated. The data is copied to the user buffer via a call to uiomove() and the protocol is notified of data delivery via a PRU_RCVD call to tcp_usrreq(). Finally the buffer is unlocked and soreceive() returns.

Network layer function leintr (): This function is called when a packet destined for the current host is received. Leintr() performs the controller specific operations to initiate packet reception. Then it calls leread() which performs the actual packet reading from the network interface.

Network layer function leread (): Leread() implements the controller specific operations to read a packet into kernel memory. To avoid copying the packet twice (one copy into the controller's designated memory buffer and a second into mbufs), buffer loaning occurs if the packet is larger that a plain mbuf. Buffer loaning is the operation where the buffer used by the controller to read the data is converted into a cluster mbuf and loaned to the higher layers. Finally the packet is passed to do_protocol().

Network layer function do_protocol (): This routine distributes packets received from the network to the appropriate protocol input queue. The pointer of the interface where the packet was received from is placed in front of the packet. After a check of queue overflow (in which case the packet is dropped), the packet is appended to the appropriate protocol queue. Finally, a software interrupt is scheduled for the appropriate protocol interrupt routine, by a call to the network interrupt scheduler.

IP layer function ipintr (): This routine is invoked via a software interrupt when packets arrive for IP. Ipintr() performs all IP protocol processing, including option handling and reassembly of fragments. It also performs packet forwarding if the host is allowed to act as a gateway. The operation of ipintr is summarized as follows:

1. Dequeue a packet from the queue.
2. Make the header contiguous via a call to m_pullup().
3. Determine the actual header length, and if it is less than the IP header size discard the packet.
4. Checksum the IP packet header via a call to in_cksum().
5. Search of the list of addresses for this host to determine whether the packet is

indeed destined for this host. If there is no address match the packet is checked to determine if it is a broadcast packet. If that also fails and the host is doing forwarding, ip_forward() is called to handle it. Otherwise the packet is dropped.

6. If the packet belongs to this host attempt reassembly. Reassembly involves a search to find if a queue of fragments of this packet already exists. If the packet is a fragment, call ip_reass().

When a fully reassembled datagram is returned by the ip_reass() routine it is passed to the next level protocol input function, which in the case of TCP is tcp_input().

TCP layer function tcp_input (): This is the function that implements TCP receive operations. The TCP input function is quite complicated. Reception operations include checksuming the header and the data, locating the protocol control block for the connection, ensuring that the packet falls within the current window, and appending the data to the receive socket buffer. Packet reassembly is also handled by this function by a call to tcp_reass(). Tcp_input() will also call tcp_output() if a message needs to go out (acknowledgment or error). The last operation in tcp_input() is to append the packet to the socket's receive buffer and call sorwakeup() to wake up the application.

4.5 QUEUEING MODEL

Figure 4.4 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 queued at the socket buffer until the protocol is ready to transmit. In the case of TCP or any other protocol providing reliable delivery, data are queued in this buffer until acknowledged. The protocol retrieves data equal to the size of the available socket buffer minus unacknowledged data, breaks data in packets, adds the

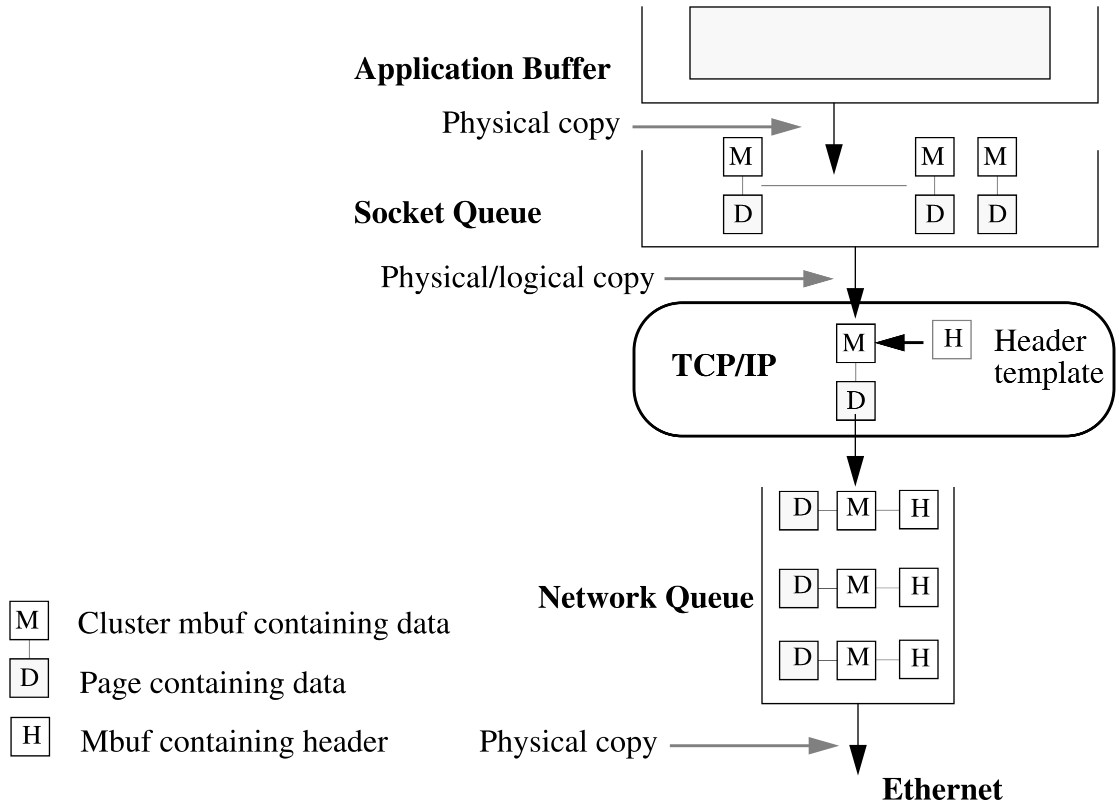

Figure 4.4: Data distribution into Mbufs

appropriate headers and passes packets to the network interface for transmission. Packets in the network interface are queued until the driver is able to transmit them. On the receiving side, 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, notified via a software interrupt, wakes up and processes packets, passing them to higher layer protocols if necessary. The top layer protocol after processing the packets appends them to the socket receive queue and wakes up the application. The four queueing points identified above are depicted in Figure 4.5.

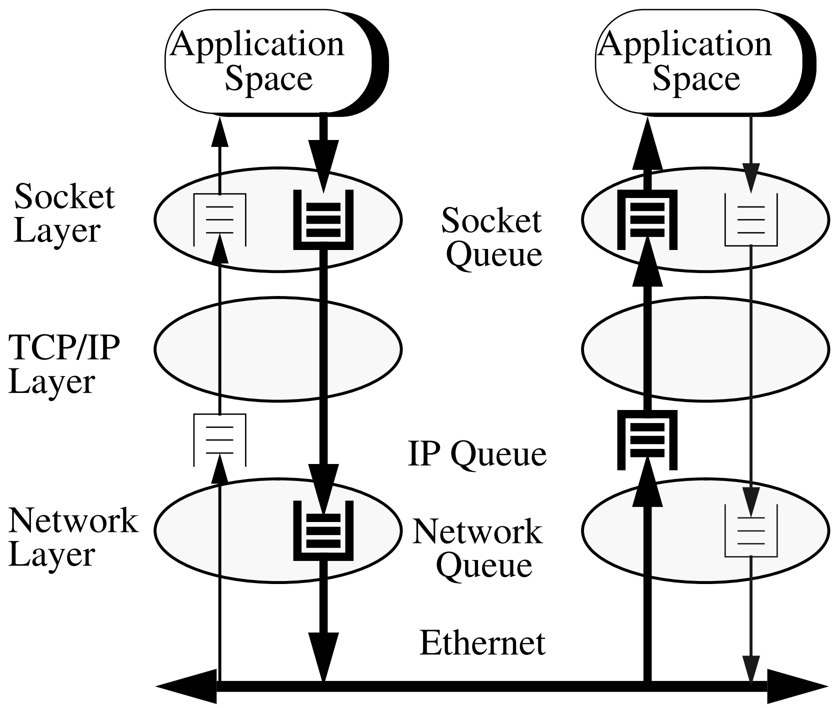

Figure 4.5: Queueing Model

4.6 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, followed by 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, followed by probe activation issues. Finally, the actual code of the probes is presented.

4.6.1 Probed Parameters

Each probe when activated records a timestamp and the amount of data passing through its checkpoint. It also fills a type field to identify the probe that produced the record. The structure of a record is shown in Figure 4.6.

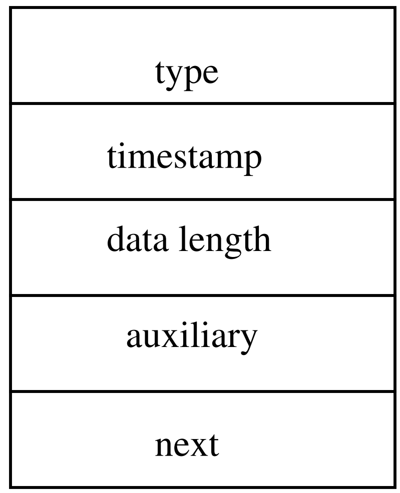

Figure 4.6: Structure of a record produced by a probe

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.

4.6.2 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 Figure 4.7 and listed next:

- **Probe 1:** This probe monitors the transmission requests made by the application to the socket layer. This probe produces one record every time the application performs a write(). It is located in the function soo_rw(). The data specified by the write request may not be copied entirely into mbufs if the amount exceeds the size of the socket buffer. More data are copied into mbufs as space becomes

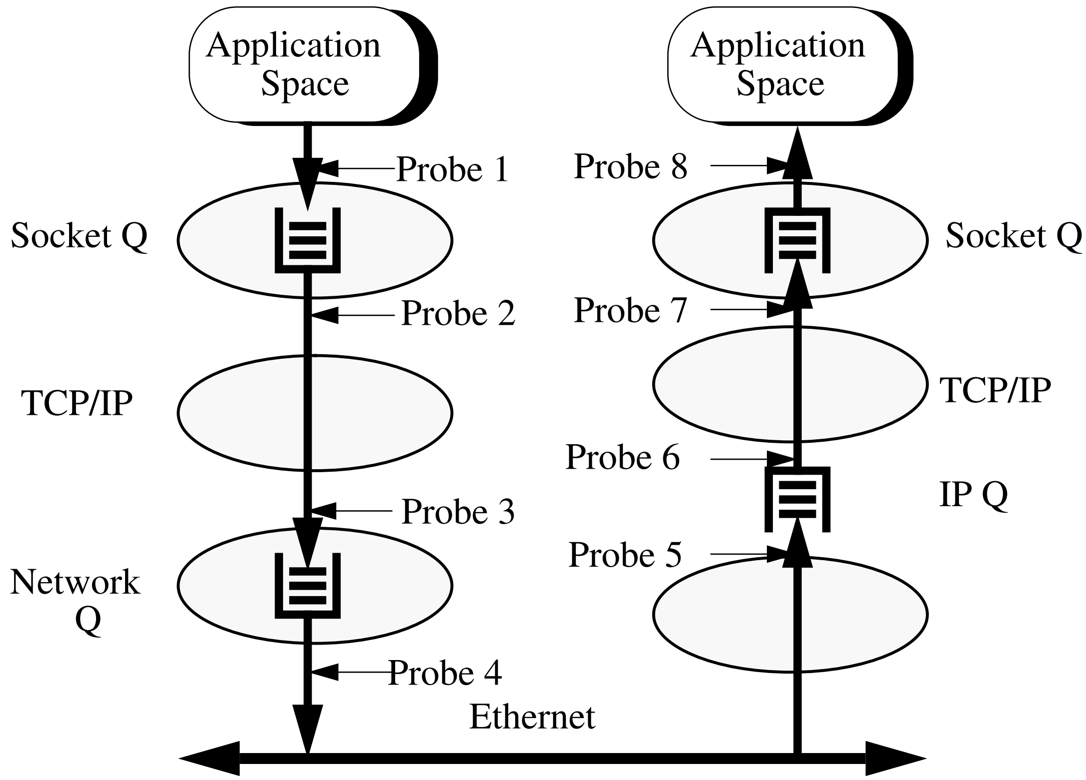

Figure 4.7: Probe Location

available.

- **Probe 2:** Probe 2 monitors transmit requests made by the socket layer to the protocol layer. The size of each request is the minimum of user data and available space in the socket buffer. This probe resides in routine tcp_usrreq() and produces a record after the mbuf chain is appended to the socket queue.
- **Probe 3:** Probe 3 monitors packets passed by the protocol to the network for transmission. The probe resides in ether_output(). Packets are logged after protocol processing has finished and the packets are appended to the network queue.
- **Probe 4:** Probe 4 resides in lestart(). A packet is logged as the network driver

removes it from the network queue and attempts transmission.

- **Probe 5:** After a packet is received by the driver, it is passed to routine do_protocol(), where the destination protocol is identified and the packet is appended on the protocol's input queue. Probe 5 resides in do_protocol and logs each packet right after it is appended to the protocol queue.
- **Probe 6:** Probe 6 logs packets right after they are removed from the protocol queue. This happens at the top of routine ipintr().
- **Probe 7:** Probe 7 monitors packets after protocol processing is complete and data are appended to the socket receive queue. The probe resides in tcp_input(), right before the call to sbappend().
- **Probe 8:** This probe monitors the transfer of data from mbufs in the socket receive queue to user space. The probe resides in routine soreceive(). A record is produced every time data is moved from an mbuf to user space.

Monitoring The 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.

Monitoring The 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.

Monitoring The 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.

Monitoring The 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.

Monitoring the Receive Protocol Processing Delay

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

Monitoring the Receive Socket Queue

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

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.

4.6.3 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; and (3) overhead due to probe activation. The probes were designed to address all three areas of overhead.

To address the overhead of probe execution time, any other kind of processing besides logging a few important parameters in the probes was precluded. To address the overhead from recording the measurements, it was decided that probes should not write records to a file since this would introduce unacceptable delays, but rather should store records in the kernel virtual space. To accomplish this, a circular list residing in kernel space is used as depicted in Figure 4.8. The list is initialized at system start-up by the TCP initialization rou-

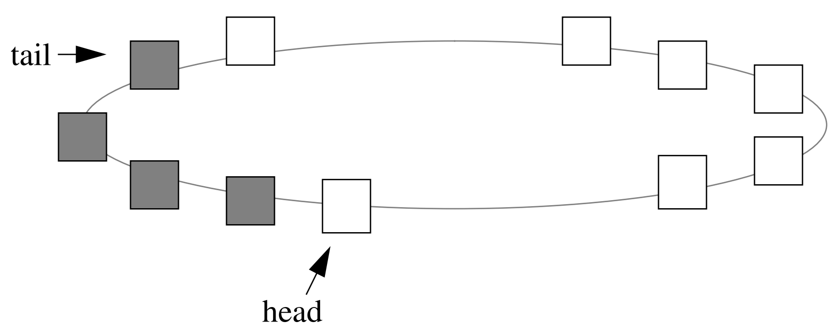

Figure 4.8: Kernel Circular List holding records

tine. 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. Each probe when activated, checks the head and tail pointers, and if they are different grabs the record pointed by the head pointer. If the head and tail pointers coincide, there are no free records and no logging takes place. Otherwise the head pointer is advanced to the next record. At the end of record collection a user process is employed to retrieve the records from kernel space to the disk, where post processing routines can access the records later to extract the desired measurements.

The third source of overhead, probe activation, refers to the steps required to determine if a packet should be logged. It is critical that probe activation should interfere as little as possible with normal operation since every packet needs to be examined to determine whether it should be logged or not. Thus if the activation mechanism is not implemented efficiently it could add significant overhead. The developed mechanism is described next.

4.6.4 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 to, 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 unable to activate its probes before the arrival of some of the initial packets, leading to the logging of incomplete data. To overcome this problem the kernel was modified to set the socket logging option automatically if the SYN packet sent for connection establishment has the logging flag set.

```
#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 4.9: Code for a typical probe

4.6.5 Probe Internal Structure

The code for a typical probe is shown in Figure 4.9, and 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. A probe having no access to the socket structure would test the IP header “tos” field here. Then the probe masks processor interrupts and ensures that the next record on the list is indeed free by comparing head and tail pointers. If the record 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.

4.7 COMPARISON WITH OTHER MONITORING TOOLS

Currently we have examined 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, NETMON 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.

4.7.1 TCPDUMP

Tcpdump[1] 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 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 4.10.

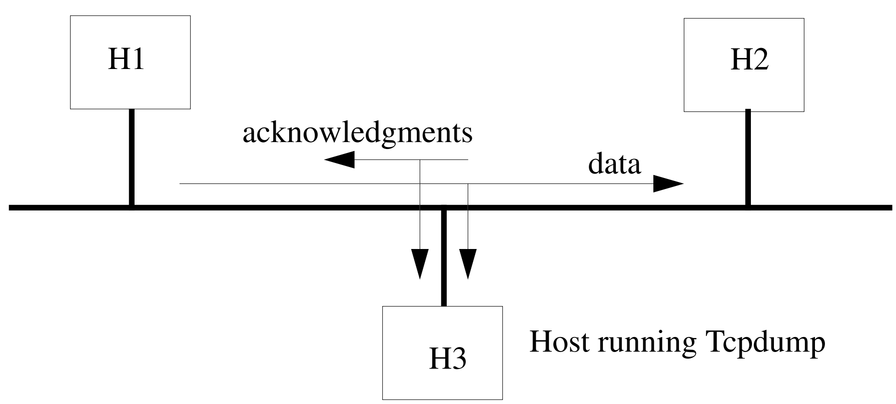

Figure 4.10: Capturing packets with Tcpdump

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

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
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.

4.7.2 An Example: Packet and Acknowledgment Trace

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. Such a plot is produced next from data collected with tcpdump during data transfer between two Sparcstation 1 workstations. The machine running tcpdump was a Sparcstation 2 workstation. The packet trace produced was used to construct the graph shown in Figure 4.11. The acknowledgments were also extracted from

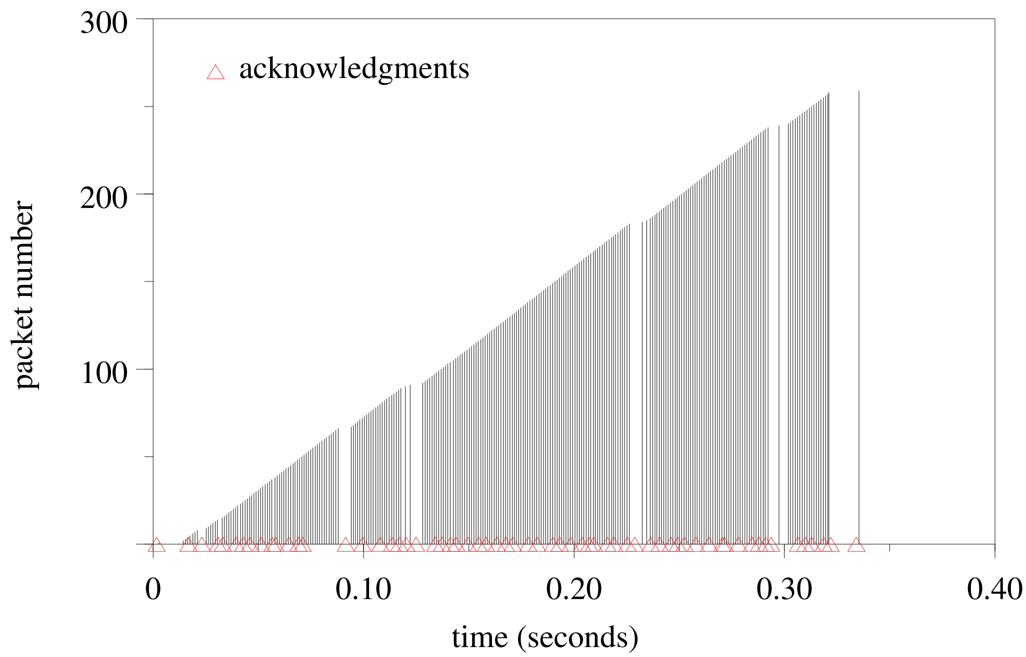

Figure 4.11: Packet and acknowledgment trace produced using Tcpdump

the trace and are shown in the same figure. The x-axis shows the elapsed time and the y-axis the packet number (all the data packets were the same size). Each packet is shown as a vertical bar and each acknowledgment is represented by a triangle. Since this is a unidirectional connection, each bar represents a packet in the forward direction (data), and each triangle represents a packet in the reverse direction (acknowledgments). The data transferred during this experiment was 256 kilobytes.

The graph shows that there is a smooth flow of packets from the sender to the receiver, a sign of a well behaved connection. However, no information about performance of the

IPC mechanism can be extracted from this graph. 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 are observed on the graph, no information is apparent about when they were actually generated by the protocol[1]. Information such as window burstiness and acknowledgment generation is necessary when trying to evaluate the performance of the IPC implementation.

4.7.3 NETMON

NETMON was written by Allison Mankin at MITRE Corporation. It is a system for recording network protocol information for monitoring and/or 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
```

1. 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.

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.

4.8 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.