The Sprite Internet Protocol Server

Andrew Richard Cherenson

M.S. Project Report
Computer Science Division
Dept. of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720

Abstract

This report describes the design and implementation of the DARPA Internet protocol suite for the Sprite network operating system†. The Sprite implementation is based on the 4.3BSD kernel implementation, but most of the code is placed in a user-level process called the IP server. Compared to a kernel-level implementation, a user-level implementation is simpler to debug and test but performance is adversely affected. Throughput performance of TCP on Sprite is about 25% of 4.3BSD TCP throughput using the same hardware. TCP latency on Sprite is about 14 times 4.3BSD's latency. 4.3BSD socket compatibility is achieved with a set of library routines that emulate socket system calls.

† This work was supported in part by the Defense Advanced Research Projects Agency under contract N00039-85-C-0269, in part by the National Science Foundation under grant ECS-8351961, and in part by General Motors Corporation.

1. Introduction

This report describes the design and implementation of the DARPA Internet protocol suite for the Sprite operating system†. The suite is a set of standard protocols used on the DARPA Internet, which is a collection of interconnected wide-area and local-area networks. At Berkeley, the machines in the EECS department that run 4.2BSD Unix and its derivatives use the Internet suite for host-to-host communication. For Sprite systems to communicate with BSD Unix hosts on campus (and Internet hosts across the country), it must provide a communication mechanism using the Internet suite.

The code to process the important protocols of the Internet suite could have been placed in either the operating system kernel or in a user-level process. Implementing the Internet suite in a user-level process reduces the complexity of the kernel and greatly simplifies testing and debugging, but at the expense of performance. Network operating systems requiring high throughput between machines usually perform the protocol processing in the kernel. SunOS, a 4.2/4.3BSD derivative, is an example of such an operating system: it uses the Internet suite to transfer data between file servers and client workstations. Sprite also handles protocol processing in the kernel but uses a specialized remote-procedure-call protocol [WELC86] instead of the Internet protocols. The Sprite applications that use the Internet suite are typically low-throughput remote terminal access and non-time-critical file transfers. Since Sprite does not require high throughput for Internet protocols, a kernel-level implementation was deemed unnecessary.

The rest of this report describes the design and implementation of the Sprite Internet Protocol Server. The report is divided into four sections: Section 2 contains background information on the DARPA Internet suite, the 4.3BSD implementation of the protocols, and related work. Section 3 describes the Sprite IP server in detail, and Section 4 analyzes the performance of the server. Finally, Section 5 contains conclusions and a description of possible enhancements to the server. Appendix 1 is a manager's guide to operating the server.

---

† Sprite is a new network operating system [OUST88] being designed for the SPUR multiprocessor workstation [HILL86] and Sun workstations.

2. Background†

2.1. DARPA Internet Protocol Suite

An important milestone in the development of computer networks and protocol design was the development of the ARPANET. In 1968, the Dept. of Defense's Advanced Research Projects Agency initiated the design and implementation of a packet-switched network to link computer science researchers at 4 sites so they could share ''resources''. Since its inception, the ARPANET has grown into a collection of networks (called the ARPA Internet) with over 2000 hosts in 8 countries on 3 continents. Major computer science research centers in universities, industry and even the military are connected to the Internet. Hosts on the Internet are able to communicate with each other because they use a common set of protocols, called the Internet protocol suite.

A protocol in the suite has a specific function and relies on other protocols to provide services it may need. The interaction between protocols is hierarchical and consists of well-defined layers. The layering principle is formalized in the ARPANET Reference Model (ARM), which is shown in Table 1. The ARM is composed of 7 layers:

physical — consists of the hardware that interfaces to the network. Examples: Intel 82501 ethernet‡ serial interface chip or RS232-C.

data link — responsible for ensuring reliable reception and transmission of data through the network interface. Example: Intel 82586 ethernet controller chip.

network — manages the network hardware and provides complete packets to the next layer.

Network Reference Models and Layering				
ISO Reference Model	ARPANET Reference Model	4.3BSD Implementation Layers	Examples of uses of layers in 4.3BSD	
Application	Application	User programs	telnet, ftp, rlogin, rcp	rwho, talk, tftp
Presentation	Utility	Libraries	rcmd	resolv
Session		Sockets	SOCK_STREAM	SOCK_DGRAM
Transport	Transport	Protocols	TCP	UDP
(Global Network)	Internet		IP, ICMP	
Network	Network	Network Interfaces	Ethernet driver and controller	
Data Link	Data Link			
Physical	Physical	Network Hardware	Ethernet interface	

Table 1. Network reference models and layering. Adapted from [QUAR 86] and [CERF 83].

† This chapter is paraphrased from [QUAR86], [CERF83], [TANE81] and the Internet RFC (Request-For-Comment) documents.

‡ ''Ethernet (capital E) is a specific Xerox protocol used for LAN, whereas an ethernet (small e) refers to an Ethernet-like network'' [QUAR86].

internet	deals with packet fragmentation and routing.
transport	deals with multiplexing packets among users on the host. It may also deal with out-of-order packets and missing packets and flow control.
utility	composed of operating system calls to the transport layer, library routine that simplify use of the system calls, and application-specific protocols (e.g., FTP).
application	composed of user-level programs.

Not all applications follow this strict layering. For example, a sophisticated program may request direct access to the internet layer, bypassing the transport layer all together.

2.1.1. The Major Internet Protocols

The internet, transport, and utility layers are the main focus of this report. The other layers are only mentioned in passing.

2.1.1.1. IP

The protocol of the ARM internet layer is called the Internet Protocol (IP) [POST81a]. IP is a simple protocol that treats network packets as self-contained units called datagrams. IP doesn't guarantee reliable delivery of datagrams to its users: IP datagrams may arrive out-of-order or not at all. The latter case may occur if the machine's network interface is busy when the packet arrives or if an intermediate gateway runs out of buffer space. IP packets are not acknowledged nor are they retransmitted if lost; it is up to higher protocols to ensure reliable data delivery to the utility and application layers. The IP header also does not contain information to multiplex datagrams among users in a host; the transport protocol must provide this service.

An IP packet may contain up to 64 kilobytes of data, less space for the packet header. Since many networks, such as the ethernet, limit packet sizes to much smaller values, the IP specification allows datagrams to be fragmented into smaller packets. Upon receipt at the destination, the fragments are reassembled into a packet of the original size. The reassembly process is complicated by the fact that fragments can arrive out-of-order and may contain data that overlaps with other fragments. If all the fragments of a packet awaiting reassembly are not received within a certain amount of time (15 seconds), the fragments are discarded. Fragmentation does not improve performance and, if possible, it is useful for the transport layer to be cognizant of maximum packet sizes to avoid fragmentation [KENT87].

The IP packet header contains a field that specifies the protocol describing the format of the packet data. The most-frequently-used protocols are the transport protocols TCP and UDP. Another important protocol, ICMP, is used for sending error and information messages to hosts. Other, more specialized protocols include EGP and GGP to handle routing information. UDP, TCP and ICMP are described in more detail below.

2.1.1.2. UDP

The User Datagram Protocol (UDP) is an extremely simple protocol [POST80]. Like IP, UDP is a datagram protocol that does not guarantee reliable delivery. The only services it provides are 1) ports to multiplex the network connection among users and 2) a 16-bit checksum for the UDP header and packet data. UDP is designed for applications that require low overhead and can tolerate unreliable delivery.

2.1.1.3. TCP

The Transmission Control Protocol (TCP) is the main transport protocol of the Internet suite [POST81c]. Like UDP, TCP provides ports to multiplex the network connection among users, but unlike UDP, TCP provides a reliable, in-order, stream of data to applications. TCP also provides a secondary channel to send out-of-band (‘‘urgent’’) data over the connection.

Implementation of the TCP protocol is vastly more complicated than the UDP implementation. For each data stream, TCP must keep state information (called a transmission control block or TCB) in order to recover from lost, duplicated, damaged, or out-of-order datagrams and to throttle the amount of packets from a sender. A 12-state finite state machine is used to keep track of connection establishment and shutdown. A reassembly queue has to be maintained to reorder out-of-order data. Data sent to a remote host are retained by the sender until explicitly acknowledged by the recipient. A timer is used to calculate when to retransmit data if an acknowledgement has not been received within a certain amount of time.

Two processes that use TCP must agree to communicate with each other. To establish a connection, they exchange a series of packets in what is called a three-way handshake: the ‘‘active’’ process sends a packet with the SYN (‘‘synchronize’’) flag on to the ‘‘passive’’ process to start the handshake. The passive process returns a packet with both the SYN and acknowledgement flags on. The connection is considered established when the active process acknowledges the passive process’s SYN packet. Each SYN packet contains the initial sequence number of the first byte of data sent in the connection. TCP uses this value during connection establishment to make sure the other packets in the handshake are valid.

Like connection establishment, connection shutdown requires both sides to recognize that the other side has closed the connection. The first host (either active or passive) to close a connection sends a packet with the FIN flag set. The other host must acknowledge (ACK) the FIN and also send a FIN to indicate it, too, is closing the connection. The first host must ACK the other host’s FIN before it can delete the state information for the connection.

For reliability, each byte of data in every packet is assigned a sequence number to identify it for acknowledgements and duplicate and out-of-order detection. When data are sent, the TCP retains the data until the remote peer sends an ACK packet with the sequence number of the last byte of data received. The ACK of byte N implicitly acknowledges all bytes numbered less than N.

Since most implementations of TCP have a limited amount of buffer space for incoming packets, TCP uses a windowing strategy to prevent buffer overflow. A receiver of data tells the sender how much buffer space is available for receiving data. The sender uses this information to determine how much data to send before the buffer is filled. Acknowledgements from the receiver indicate that buffer space has been freed so the sender can send more packets.

TCP has a mechanism to tag some data as special and not belonging to the normal data stream. These ‘‘urgent’’ data (called ‘‘out-of-band’’ data in 4.3BSD), can be sent with regular data in a packet; a special value in the TCP packet header indicates the starting location of the urgent data in the packet. The TCP implementation is supposed to notify the user process when urgent data arrive.

2.1.1.4. ICMP

The Internet Control Message Protocol (ICMP) is considered ‘‘an integral part’’ of the Internet Protocol [POST81b]. It is used to return error conditions to the internet and transport layers. Error conditions include malformed IP header options, unreachable destinations, and flow control messages. Other types of ICMP packets are used to get the current time, to echo packets back to the sender, and to determine broadcast addresses and subnet masks. Very few 4.3BSD applications use this protocol directly (*ping* is an example.)

2.2. The 4.3BSD Implementation

The DARPA Internet protocol suite was first implemented in Berkeley Unix in the 4.1c distribution. The system call interface implementation was refined in 4.2BSD. The 4.3BSD implementation [LEFF86] has the same system call interface but has been tuned for much better performance [CABR87].

2.2.1. Sockets

In 4.3BSD, sockets are an abstraction for interprocess communication provided by the kernel. In the ARPANET reference model, sockets are part of the utility layer and are used to hide the details of the underlying transport layer. Sockets are not specific to any protocol family; currently the DARPA Internet and the Xerox Network Services protocol suites are supported. Within a protocol family, there are different types of sockets. The Internet suite allows three types: stream, datagram and raw. A stream socket uses TCP as the transport protocol so it provides a reliable, in-order, bi-directional stream of data. UDP is used for the datagram sockets so data delivery is unreliable. Raw sockets are used by privileged programs to gain direct access to the IP and ICMP protocols.

Table 2 lists the 16 socket-specific and 4 file-system kernel calls that operate on sockets. The *socket* call creates a socket using a specific protocol family (e.g., PF_INET) and socket type (e.g., SOCK_STREAM, SOCK_DGRAM, SOCK_RAW). The call returns a descriptor that is used with other socket system calls. To communicate with a remote peer, a stream socket must have the local and remote Internet addresses and ports assigned. The *bind* call explicitly assigns the local address and port, though the *connect* call implicitly defines them if they are missing. Active stream sockets must establish a

4.3BSD Socket-related System Calls	
socket	creates a socket
read	read data from a connected socket
recv	read regular or out-of-band data
recvfrom	like recv, but can get sender’s address
recvmsg	read data into several buffers
write	write data to a connected socket
send	write regular or out-of-band data
sendto	like send, but send to a specific address
sendmsg	send data from several buffers
select	wait until the socket is readable, writable or has out-of-band data to be read
bind	set the socket’s local address
getsockname	get socket’s local address
getpeername	get socket’s remote address
getsockopt	get a value of a socket option
setsockopt	change a value of a socket option
listen	allow a socket to accept connections
accept	waits for connection requests
connect	make a connection request
ioctl	change a socket state
shutdown	shut down a connection
close	destroys a socket

Table 2. The 4.3BSD system calls that manipulate sockets. Each call is described in more detail in Section 2 of the Berkeley Unix manual.

connection with a passive socket on the remote peer using the *connect* call. For the connection attempt to succeed, the remote peer must accept the request (using the *accept* call on BSD systems).

4.3BSD provides four types of system calls to send and receive data over sockets. The simplest calls, *read* and *write,* assume the socket is connected to a remote peer and just send normal data. The *recv* and *send* calls are like read and write but can process both normal and out-of-band data. *Recvfrom* and *sendto* are useful for datagram sockets when they need to obtain the source or specify the destination of a packet, respectively. The *recvmsg* and *sendmsg* calls are like *recvfrom* and *sendto* but can do scatter/gather I/O.

Several C library routines simplify the use of sockets. There are database lookup routines to find the IP address of a host and the number of a protocol. IP address routines can decompose an address into the net and host parts and create an address from those parts. The *rcmd* routine is used by certain daemon programs to create and bind a socket with a unique privileged port.

2.2.2. Kernel Implementation Structure

The major modules of the 4.3BSD implementation are the network, protocols, socket and system call modules, corresponding to the network through utility layers in the ARPANET reference module. The network module hides the details of different network interfaces and handles packet I/O for the protocol module. The protocol module implements the IP, ICMP, TCP and UDP protocols. Other protocols that use IP are implemented by user-level processes using raw sockets. The transport protocols interact with the socket layer through routines and direct manipulation of socket data structures (unfortunately, principles of information hiding are not always applied). The socket layer interacts with the process management: processes waiting for data from sockets need to be awoken when data arrives. User-level processes manipulate sockets via the system calls in Table 2.

Dynamically-allocated memory for socket data structures, packet data and headers is maintained in *mbufs* (memory buffers). A small mbuf is a 128-byte structure that contains up to 112 bytes of data, the amount of data in use, and the offset to the start of the data from the beginning of the structure. Large mbufs can store 1024 bytes of data: a special pool of pages is used for the data. Structures larger than 112 or 1024 bytes are stored in a chain of mbufs. A set of routines is used to abstract the details of mbuf management. The routines perform such operations as allocate, free, copy, adjust size, and consolidate. For efficiency, mbuf copy operations try to increment the underlying page's reference count instead of doing a true copy. The effect of mbuf sizes and chaining on TCP and UDP performance are shown in Section 4.

2.3. Related Work

The idea of implementing the Internet suite at user-level is not new. At MIT, the original TCP implementation for MULTICS used a simple demultiplexor in the kernel and the protocol code ran in the user ring. The Laboratory for Computer Science added a powerful packet filter to the Unix version 6 kernel. The filter was changed by recompiling the module and relinking the kernel. The protocols were handled at user-level, which was suitable for client telnet programs. The server telnet program required virtual terminal support in the kernel to get data to the clients [CLAR87].

The Stanford V Kernel also implements the Internet suite with a user-level process [THEI87]. The Internet server consists of one address space shared by several threads of control. Important threads include the packet reader, which reads packets from the network device, a timer thread to handle retransmission timeouts, and a dispatcher thread to listen for connection requests from clients. A new thread is created when a client establishes a connection to the server. This thread obtains packets from the reader and calls the IP and TCP protocol routines to process the packet. The connection thread and clients communicate using the standard V IPC mechanisms. On Sun-2 hardware running the V kernel, FTP could achieve 50 kilobytes/sec when using a large TCP window size.

The CMU/Stanford packet filter for BSD Unix was designed to allow efficient user-level implementations of protocols [MOGU87]. The packet filter is a flexible mechanism in the kernel to direct incoming packets to the appropriate user-level process; it avoids much of the overhead incurred by user-level demultiplexing. Each incoming packet is evaluated using filters specified by processes interested in specific types of packets. The first filter to accept the packet causes it to be placed in the receive queue of the corresponding process. A filter is a sequence of boolean operations specified in a simple, non-protocol-specific stack-oriented language interpreted by the kernel. The language lacks arithmetic operations so fields with varying offsets within a packet header cannot be examined. For example, IP packets containing header options displace the TCP and UDP fields from their usual offsets. Demultiplexing by the packet filter is about two times faster than user-level demultiplexing but two to three times slower than kernel protocol implementations.

3. Design and Implementation of the Sprite IP Server

The Sprite implementation of the Internet suite was developed with several goals in mind:

1) Short development time: the ability to transfer files between Sprite and Unix machines was needed as soon as possible.

2) Minimal modifications to the Sprite kernel: to reduce the complexity of the kernel, we have tried to keep unessential code out of the Sprite kernel.

3) Compatible Berkeley Unix socket interface: we wanted to use 4.3BSD network utilities like *rcp* and *rlogin* with as few modifications as possible.

4) ‘‘Reasonable’’ performance: No explicit performance goals were made, but a throughput rate within an order of magnitude of the rate for Sun Unix was desirable.

We achieved the first goal by using the existing 4.3BSD Internet implementation as the basis of the Sprite implementation. We used the BSD implementation for several reasons: it had been in use for a year and had proven to be reliable, it gave very good performance, and it was easily obtainable and in the public domain. Only the protocol-processing code in the BSD implementation was ported to Sprite though it was extensively reworked to conform to the rigorous Sprite coding style. The 4.3BSD kernel code to handle sockets was extensively reworked and simplified to fit in the Sprite framework.

To achieve the second goal, all of the code to handle the internet and transport layers and some of the code to handle the session layer were placed in a user-level process instead of in the Sprite kernel. This process, called the IP server, does, however, require some assistance by the kernel to transfer data between itself and its client programs. We designed a general-purpose interprocess communication (IPC) mechanism in the file system, called pseudo-devices, to facilitate such data transfers. (Pseudo-devices are discussed in more detail below.)

Goal 3 required that we support 4.3BSD’s socket programming interface, including 16 system calls and numerous library calls. The IP server uses pseudo-devices to implement sockets: the BSD socket system calls are implemented as library routines that make Sprite system calls on pseudo-devices. The other 4.3BSD network library routines were ported to Sprite without difficulty.

3.1. The IP Server Process

The IP Server is a regular user-level program that is started during the Sprite boot sequence. Source code for the server consists of 14,000 lines† of C and is structured as four modules: network I/O and routing, protocol processing, sockets, and client I/O. The overall relationship between the modules is shown in Figure 1. The network I/O-routing module deals with packet I/O between the protocol module and the kernel. This module

† The binary image of the IP server contains 93,304 bytes of code: the server routines account for 42,968 bytes and the remainder are from the Sprite C library. The Sprite kernel contains 349,156 bytes of code. Adding the IP server into the kernel would probably increase its size by 50,000 bytes or about 15%.

also handles packet routing, which is the task of determining a host's ethernet address from its Internet address‡. The protocol module consists of routines to process packets from the network and data from programs using the IP, TCP, UDP and ICMP protocols.

A socket is an abstraction for application programs (''clients'') provided by the IP server to hide the details of the state information required by the protocols. The socket module maintains this information in per-client and per-socket data structures. (One or more clients can access a socket as a result of the Proc_Fork or Fs_GetNewID system calls.) Per-client information includes process, user and host IDs, as well as information used to emulate the BSD recv and send family of system calls. Shared state information consists of protocol-dependent data, the state of the socket (e.g., created, connected, closed), send and receive buffers, and the local Internet address and port and the remote peer's address and port. The protocol-dependent data for stream sockets store the TCP transmission control blocks. Datagram sockets (which use UDP) and raw sockets (which use IP and ICMP) do not keep additional data.

The client I/O module handles the details of interprocess communication between the server and client programs. Clients communicate with the IP server using pseudo-

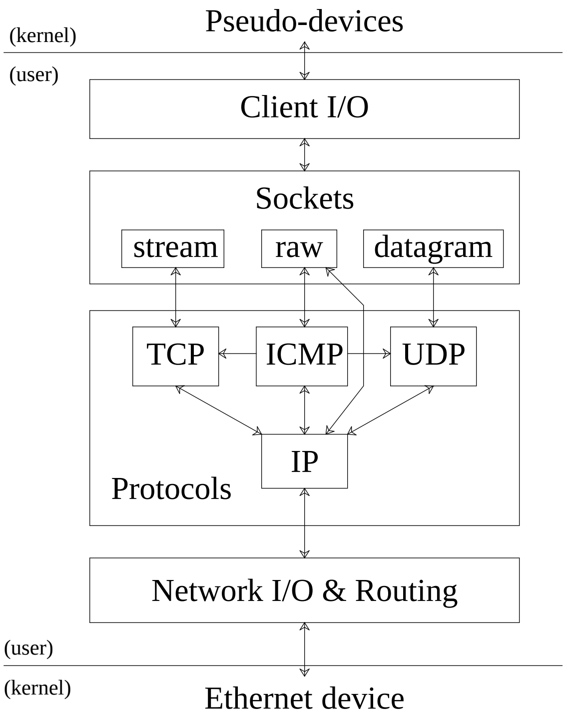

Figure 1. The four major modules of the IP server. Pseudo-devices and the ethernet device are accessed with file system kernel calls.

‡ The routing routines provide this mapping function for the IP processing routine using a static database of <ethernet, Internet> address pairs. The result of the last route request is cached in case the next request is for the same host, which is common during file transfers.

devices. Like regular devices, pseudo-devices appear as files in the Sprite file system and they are accessed by programs using the standard file system kernel calls. Unlike a regular device file, operations on a pseudo-device file do not manipulate any underlying hardware device (such as a tape drive) but cause a message to be sent to the user-level process that controls the pseudo-device. This ‘‘master’’ process responds to the message in any way that it wishes, and returns data and status information to clients using systems calls on the pseudo-device. For example, when a client tries to read data from a pseudo-device with the Fs_Read call, the kernel sends a read message to the master and suspends the client. If data are available, the master will supply the data to the pseudo-device using the Fs_Write call. If no data are available, the master returns an FS_WOULD_BLOCK status to the device. After the master responds, the kernel wakes the client to complete the system call.

Clients manipulate sockets via three pseudo-devices controlled by the IP server. The files /hosts/*machine*/netTCP, /hosts/*machine*/netUDP and /hosts/*machine*/netIP, correspond to the stream, datagram and raw sockets on a particular machine. When a client opens a stream socket, for example, the IP server receives an open message for the netTCP pseudo-device. The message contains a stream ID of a new private communication path between itself and the client. Future client requests to read and write data will arrive on this new stream. Access to the pseudo-devices is usually hidden from clients by a set of library routines.

3.1.1. Flow of Control

At a high level, the IP server is an event-driven program that executes a loop waiting for three types of events: 1) pseudo-device requests arriving from client programs, 2) IP packets arriving from the network, and 3) timeouts. When one of these event occurs, the appropriate handler routine is called. The following paragraphs describe the actions taken by the server when handling an event. Figure 2 diagrams the actions in terms of the flow of data for a file transfer between two Sprite hosts.

When a client writes data to the socket pseudo-device, the IP server is notified and reads the data into a buffer. This buffer is given to a protocol-dependent socket routine that handles output. For UDP, the routine adds the UDP header and sends the buffer to the IP layer for output. For TCP, the data are appended to the socket’s send buffer and the TCP output routine is called to create a packet for output. This extra routine is necessary for TCP because the packet may need to be retransmitted if not acknowledged within a certain amount of time. Once the IP module receives a packet, it calls on the routing module to determine a route for the packet: if a route cannot be found, an error status is returned. IP also fragments the packet into smaller ones if the packet is larger than the network’s maximum packet size. The packets are then given to the routing module for output, which writes them to the /dev/etherIP device after prepending the ethernet header.

The second type of event occurs when an IP packet is received by the network interface. The kernel copies the packet into a pre-allocated buffer, which is added to the /dev/etherIP device’s input queue. The device is made readable, causing the server’s event dispatcher to call its packet handler. This handler reads the packet from the device into a buffer using the Fs_Read kernel call and passes the buffer to the IP protocol’s

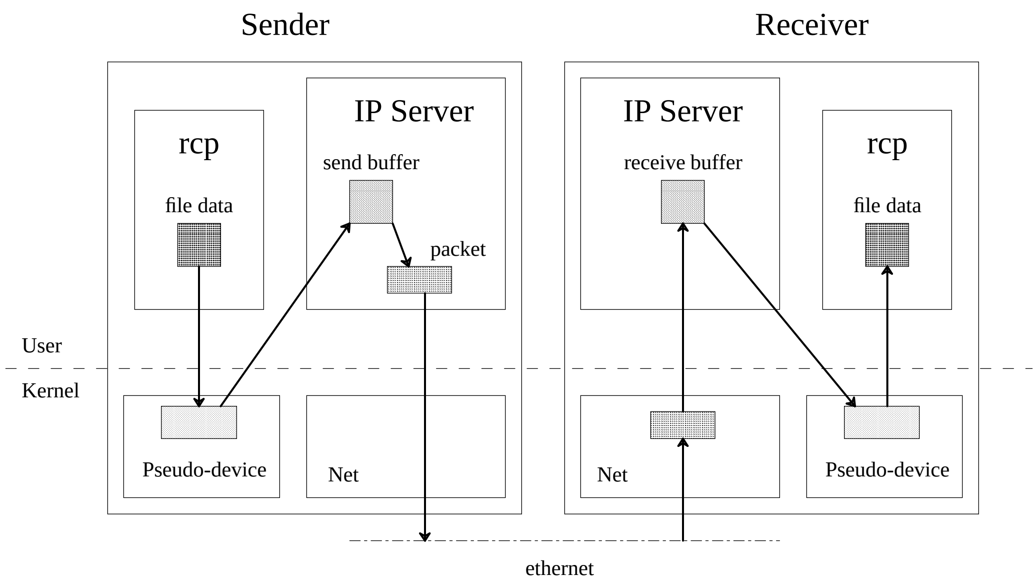

Figure 2. Flow of data during a file transfer between two Sprite hosts using the *rcp* remote copy utility, which uses TCP streams. Each arrow indicates a buffer copy operation. Packet output from and input to an IP server also include a checksum calculation. Acknowledgement packets returned to the sender's IP server by the receiver's server are not shown in the figure.

input routine. The IP input routine validates the packet by checking that the header is not corrupted and contains consistent information. For example, the header length in the packet must be at least as large as the standard IP header and no larger than the whole packet. Once these checks are passed, the destination address in the header is examined to make sure the packet is addressed to the host. If not, the packet is discarded (in the current implementation); if the host was acting as a router between networks, the packet would be forwarded to the appropriate network. Since IP packets can arrive in fragments, the packet is checked to see if it is a fragment. Fragments of a packet are saved until all of them have been received. Once a complete packet is available, the protocol field is examined to determine which protocol input routine should be called to handle the packet. If a protocol does not have an input routine, the packet is given to the ‘‘raw’’ socket handler, which makes such packets available to privileged programs via raw sockets. The packet is dropped if no raw sockets are active.

The UDP and TCP input routines validate the values in the header and compute the header/data checksum to make sure the packet is not corrupted. The destination port in the header is used to associate the packet with a socket data structure. For UDP, if a socket doesn't exist, the packet is dropped. If it does exist, the data in the packet are appended to the socket's receive buffer and the server notifies the client that the socket is now readable. For TCP, if the socket doesn't exist, the packet is dropped and a RST (reset) packet is returned to the sender. If the socket exists, additional processing is required to make sure the packet contains valid data and/or control flags like SYN, ACK, or FIN before the data are appended to the socket's receive buffer and the client is notified.

The third type of event handled by the server is timeouts. Timeout events are used by IP to dispose of expired packet fragments. The TCP protocol relies on timers for packet retransmission, sending keep-alive packets to maintain connections, deleting state information after closing a connection, and for maintaining the sending window size.

3.1.2. Memory Buffer Management

The IP server does not use a memory buffer scheme like 4.3BSD's mbufs. Memory for packets, socket buffers and dynamic data structures are allocated from the heap segment using the Mem_Alloc routine in the Sprite C library. The server tries to be smart when allocating a buffer to hold data that might be sent to the network: the allocation size includes space for all protocol headers that might be required to form a complete packet. This technique avoids additional allocations and copies by each protocol module as the packet is constructed.

3.1.3. Client Interface and 4.3BSD Emulation

A client program interacts with the IP server using sockets, which are implemented with pseudo-devices. A socket is manipulated by using the file system kernel calls Fs_Open, Fs_Close, Fs_Read, Fs_Write, Fs_Select and Fs_IOControl on the server's pseudo-devices. The open and close calls create and destroy an IPC channel to the server. Data are read from the server using Fs_Read and written to it using Fs_Write. Socket functions such as assigning local and remote addresses, allowing a socket to accept remote connections, and sending and receiving out-of-band data are performed

Socket I/O Controls Supported by the Sprite IP Server	
IOC_NET_LISTEN	make into a passive socket
IOC_NET_ACCEPT_CONN_1	accept a pending connection request
IOC_NET_ACCEPT_CONN_2	associate the new connection with a socket
IOC_NET_CONNECT	try to connect to a remote host
IOC_NET_GET_LOCAL_ADDR	get the local <address, port> of the socket
IOC_NET_SET_LOCAL_ADDR	set the local <address, port> of the socket
IOC_NET_GET_REMOTE_ADDR	get the remote <address,port> of the socket
IOC_NET_GET_OPTION	get the current value of an option
IOC_NET_SET_OPTION	set the value of an option
IOC_NET_RECV_FLAGS	flags to modify behavior of next read, i.e., peek, get out-of-band data
IOC_NET_RECV_FROM	get <address,port> of last packet
IOC_NET_SEND_INFO	flags to modify behavior of next write, i.e., send out-of-band data, specify destination
IOC_NET_SHUTDOWN	prevent further sending and receiving
IOC_NET_SET_PROTOCOL	specify the protocol for a socket
IOC_NET_IS_OOB_DATA_NEXT	returns TRUE if out-of-band data can be read by the next read call

Table 3. Sprite I/O controls to support 4.3BSD-style sockets.

using I/O controls. Table 3 lists all the socket-specific I/O controls supported by the IP server.

4.3BSD socket system calls are emulated with a set of library routines that use native Sprite file system calls (see Table 4). Some routines, such as *getsockname*, are emulated with one Fs_IOControl call to the IP server. More complicated routines, such as *recvfrom*, *sendto*, *accept* and *connect*, require several calls. For example, the *recvfrom* call can be used to perform a non-destructive read (‘‘peek’’) of data in the socket and to find out who sent the data. The library routine for *recvfrom* first makes an I/O control to tell the server that the next Fs_Read call should not discard the datagram. The data is read with Fs_Read and the sender’s address is obtained with another I/O control.

Socket operations like *recvfrom*, which require several operations on the pseudo-device, complicate the server. Related processes can share a socket and the operations by the processes on the socket may be interspersed. In order to correlate related pseudo-device operations, the server keeps some state for each client process. An example clarifies the need for state: two processes, A and B, share a socket and each makes a *sendto* call specifying different destination addresses. If no private state is kept and the

Sprite Emulation of 4.3BSD Socket System Calls	
Unix Call	Sprite Call(s)
socket	Fs_Open, [Fs_IOControl(SET_PROTOCOL)]
read	Fs_Read
recv	[Fs_IOControl(RECV_FLAGS)], Fs_Read
recvfrom	[Fs_IOControl(RECV_FLAGS)], Fs_Read, [Fs_IOControl(RECV_FROM)]
recvmsg	[Fs_IOControl(RECV_FLAGS)], Fs_Read, [Fs_IOControl(RECV_FROM)]
write	Fs_Write
send	[Fs_IOControl(SEND_INFO)], Fs_Write
sendto	[Fs_IOControl(SEND_INFO)], Fs_Write
sendmsg	[Fs_IOControl(SEND_INFO)], Fs_Write
select	Fs_Select
bind	Fs_IOControl(SET_LOCAL_ADDR)
getsockname	Fs_IOControl(GET_LOCAL_ADDR)
getpeername	Fs_IOControl(GET_REMOTE_ADDR)
getsockopt	Fs_IOControl(GET_OPTION)
setsockopt	Fs_IOControl(SET_OPTION)
listen	Fs_IOControl(LISTEN)
accept	Fs_IOControl(ACCEPT_CONN1), Fs_Open, Fs_IOControl(ACCEPT_CONN2)
connect	Fs_IOControl(CONNECT)
ioctl	Fs_IOControl
shutdown	Fs_IOControl(SHUTDOWN)
close	Fs_Close

Table 4. Summary of Sprite emulation of the 4.3BSD socket system calls. Calls in brackets (such as [Fs_IOControl]) are only made if necessary. For example, Fs_IOControl(SET_PROTOCOL) is made if the protocol argument to the socket routine is non-zero.

operations are ordered as:

A: Fs_IOControl(IOC_NET_SEND_INFO)
B: Fs_IOControl(IOC_NET_SEND_INFO)
A: Fs_Write()
B: Fs_Write()

then both packets are sent to B's destination.

3.2. Differences from the 4.3BSD Implementation

The following functions present in the 4.3BSD implementation are not implemented in the current version of the IP server:

- Getting and setting all protocol-specific and some socket-specific options: for example, IP and TCP protocol header options cannot be specified. Only a few test programs use these functions.
- Packet forwarding: if given a packet not destined for the host, the IP server cannot forward the packet to the appropriate host. This means that a Sprite host cannot act as an IP router between networks.
- Dynamic routing: currently, the server uses a static routing table that is initialized from a file at start-up time. When a new host is added to the Sprite network, the server's configuration file must be updated.
- Network device independence: the routing module is specific to the ethernet.
- Availability of routing information: it is not propagated to the protocol layer nor is it used by the protocols. For instance, 4.3BSD TCP uses the route to determine the receive window size. Lack of this feature only affects performance of long-distance TCP connections.
- ICMP: the input routine does not yet handle routing redirects nor does it process source quench messages. Lack of source quench handling only affects performance of long-distance TCP connections. Redirect packets are sent when there is a shorter route to a remote host; not handling redirects only affects performance.

These functions were not implemented in the initial version because they are not essential for the IP server to work in our environment. They will be implemented in future revisions of the server.

3.3. Debugging and Testing

The IP server was debugged and tested in several stages. The packet I/O and IP/ICMP processing routines were implemented first. They were tested using the *ping* utility from 4.3BSD, which sends an ICMP echo packet to a specific host every second. *Ping* was especially useful in finding bugs in the IP input fragment reassembly and output fragmentation routines. Since the IP server is a user-level program, it was debugged using the standard source-level debugger, *dbx*. Once the pseudo-device interface and socket emulation routines were implemented, they were tested using several small programs that exercised a particular function of the IP server. The programs were written using the socket calls so they could be run on both 4.3BSD and Sprite to verify that they produced the same results.

Another valuable tool in debugging the IP server was the SunOS *etherfind* utility. *Etherfind* can receive and display any packet on the ethernet. Various options allow the user to display packets that are, for example, from a certain host or have a specific protocol. I enhanced the program to decode the packet header fields and print them in a human-readable form. *Etherfind* verified that the IP server was sending properly formatted packets on the network.

A bug in the checksum routine was debugged using a modified SunOS kernel. We replaced the standard 4.2-derived Internet code with the code from 4.3BSD, and modified the TCP input routine to print an error message whenever a TCP packet with a bad checksum arrived. This same kernel was also used in performance testing, which is described in the next section.

The IP server contains many counters to collect statistics about the server's behavior. The protocol-processing code from 4.3BSD had an extensive number of counters; I added counters to the socket and event handler code. The counters have proven to be extremely useful in debugging and analyzing the server's performance.

4. Performance Analysis

I measured the performance of the Sprite IP server for the TCP and UDP protocols using the *ttcp* benchmark, which was written by Mike Muuss of the U. S. Army Ballistic Research Laboratory. *Ttcp* sends data from one host to another and reports the elapsed time and transmission rate. Various data buffer sizes and repetition counts can be specified. Four sets of host-to-host performance measurements were taken: Sprite to Sprite, Sprite to Unix, Unix to Sprite and Unix to Unix. The tests were run on two Sun-3/75 workstations, one with 8 Mbytes of memory, the other with 16 Mbytes. The SunOS 3.2 Unix kernel was modified to use the 4.3BSD Internet code (the Internet code in a standard 3.2 kernel is derived from 4.2BSD, which has worse performance than 4.3BSD). Since the Sprite server is derived from 4.3BSD, the tests are comparing a user-level implementation versus a kernel implementation of the same protocol-processing code. The elapsed times reported by the senders were used to compute throughput rates between the two machines. This user-level performance analysis was inspired by the studies for BSD Unix as described in [CABR84] and [CABR87].

Several factors affect the benchmark results that are presented below. CPU scheduling vagrancies by the kernel may affect the elapsed time reported by *ttcp*. The benchmarks were run on quiescent systems to minimize the effect: for SunOS, no active programs were running (except for the benchmark); for Sprite, just the IP server, the benchmark and a shell were running. (When other programs are using sockets, the IP server's performance is degraded due to overhead in the Fs_Select system call.) Network activity by other workstations connected to the same ethernet as the two test machines may have also affected the results. The timings were made during off-peak hours to avoid high network load but the performance numbers presented here have been measured to vary by approximately ± 5%. Finally, the TCP benchmark results on Sprite may be slightly optimistic due to a ‘feature’ of the current pseudo-device implementation. When a client closes a socket after writing data to it, the client should be blocked until all data have been reliably sent. The current pseudo-device implementation does not allow the IP server to block a client that closes a socket, hence the elapsed time calculation does not include the time to send the last chunk of data. This effect is reduced by having the benchmark write a large number of buffers to the socket before closing it.

The sizes of the socket send and receive buffers affect the benchmark performance. For TCP, the receive buffer size affects how large a window is presented to the sender. A larger window encourages the sender to ship more packets before waiting for acknowledgements, thus improving performance. Because TCP retains data written to the socket until they are acknowledged, the send buffer size affects how much data can be written by the benchmark before it blocks. For UDP, the receive buffer size affects how many packets can be saved before additional incoming packets are dropped. The send buffer size affects the maximum datagram size. The benchmark used the default send and receive socket buffer sizes of 4096 bytes on both Sprite and Unix.

4.1. TCP Performance

TCP throughput rates on Sprite and Unix were measured with *ttcp* for a range of data sizes; the results are shown in Figure 3. Two features of the TCP protocol affect the performance of the benchmark. Since TCP packets must be acknowledged, the benchmark results are affected by the receiver's speed in returning acknowledgement packets to the sender. Also, the TCP output routine delays sending data until a minimum-sized packet can be generated, which affects the throughput rate for small data buffer sizes.

Throughput in the Unix implementation is about four times faster than the Sprite implementation for large buffer sizes (265 KB/sec vs. 61 KB/sec for a 2048-byte buffer). The Unix-to-Sprite rate of 60 KB/sec is fairly constant for buffer sizes between 128 and 4096 bytes, indicating that the rate of ACKs by Sprite is limiting the Unix side from sending data into the window. For buffer sizes larger than 512 bytes, the Sprite-to-Unix rates range from 60 KB/sec to 86 KB/sec and Sprite-to-Sprite rates range from 41 KB/sec to 61 KB/sec. In both instances, the highest rate is obtained using 2048-byte buffers.

The write buffer size has a major effect in throughput performance. The Unix-to-Unix rates in Figure 3 show an interesting saw-tooth line from 1KB to 4KB and an especially large jump in performance from 1023-byte buffers to 1024-byte buffers. These anomalies are due to the *mbuf* memory buffer management scheme described in Section

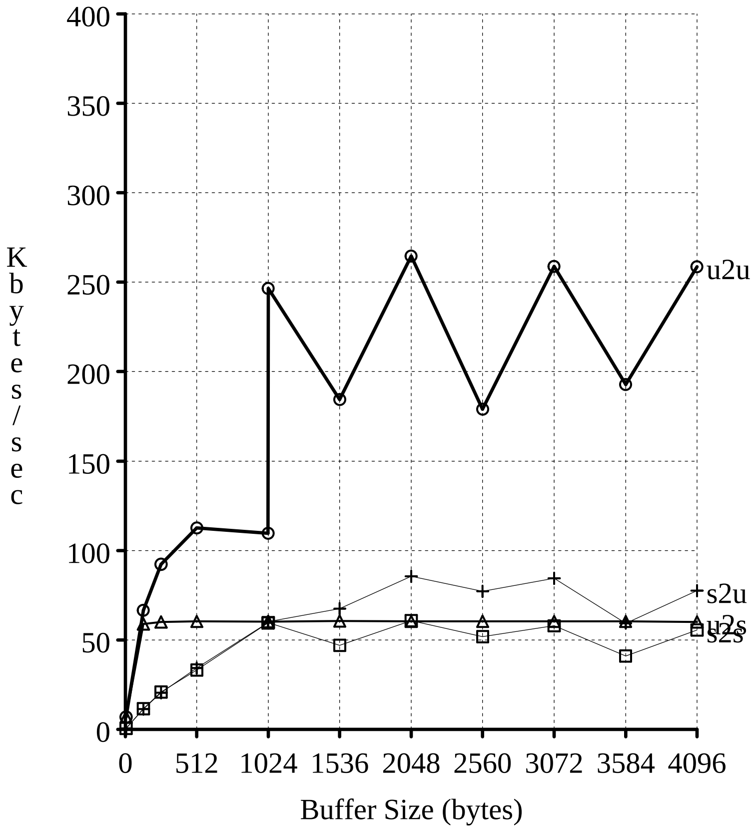

Figure 3. Comparison of Sprite and Unix TCP throughput rates. The following amounts of data were written to a stream socket by *ttcp*: 5, 128, 256, 512, 1023, 1024, 1536, 2048, 2560, 3072, 3584, and 4096 bytes. Legend: u2s – Unix-to-Unix, u2s – Unix-to-Sprite, s2u – Sprite-to-Unix, s2s – Sprite-to-Sprite.

2. The networking code is optimized for 128- and 1024-byte mbufs; sending a 1023-byte buffer has a lot of overhead because the buffer is really composed of a chain of 128-byte mbufs. When a Sprite host is the sender, the throughput rate plots show a similar saw-tooth pattern, but for a different reason. Analysis of the packet traffic for the Sprite-to-Sprite run shows that when sending buffer with sizes of integer multiples of 1024, Sprite sends packets that are 1024 bytes long. With buffer sizes of 1536, 2560, and 3584, Sprite sends a mixture of 1024 and 512 byte packets. The worst throughput rate occurs with 3584-byte buffers because the majority of packets (57%) are 512 bytes long. The smaller packets do not drain the socket send buffer as quickly as 1K packets. As a result, the *ttcp* benchmark is more likely to encounter a full socket buffer when writing the data, causing *ttcp* to block waiting for space to become available. When the write is retried, additional overhead is incurred due to extra interactions with server. This overhead is considerable: the IP server's CPU usage increased by 48% (26 seconds) for 3.5KB buffers over 3KB buffers.

Figure 4 shows the elapsed times for the *ttcp* benchmark to send data buffer of various sizes on Unix and Sprite. The intersection of the lines at the *x* origin measures the fixed overhead or latency incurred by *ttcp* when writing to a socket to send data a remote

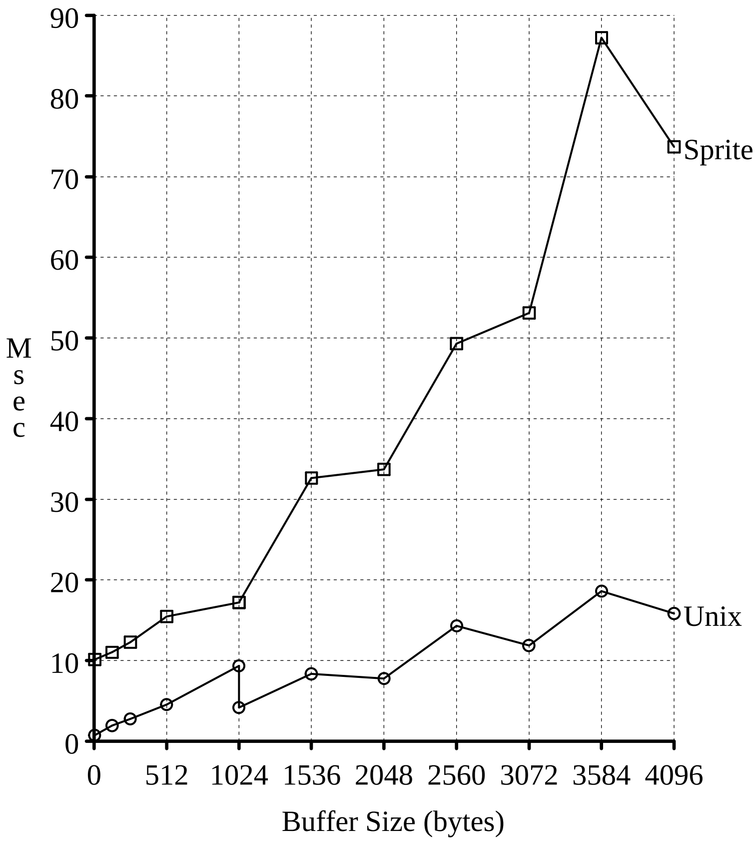

Figure 4. Elapsed time to write one data buffer to a stream socket from Sprite to Sprite and from Unix to Unix. The data buffer sizes are the same as in Figure 3.

host. This value includes the time it takes the remote TCP to ACK the data but does not include the time it takes the receiving process on the remote host to obtain the data from its socket. Latencies for Unix and Sprite when sending to a Unix host differ by about 14 times: 0.72 ms vs. 10.14 ms. The Sprite latency is mostly due to pseudo-devices (see below).

4.2. UDP Performance

Figure 5 shows the throughput rates for sending UDP packets from Sprite and Unix hosts. On both systems, throughput increases as the buffer size increases. The maximum throughput for Unix is almost 400 KB/sec using 4096-byte buffers, while Sprite's maximum is only 164 KB/sec. The Unix UDP throughput rate, like the TCP rate, is noticeably affected by mbuf sizes: the difference of 512 bytes in the buffer size can mean a difference of at least 50 KB/sec in throughput. The Sprite rate begins to level off at large buffer sizes, indicating that the CPU is becoming a bottleneck; the IP server is a very compute-intensive program.

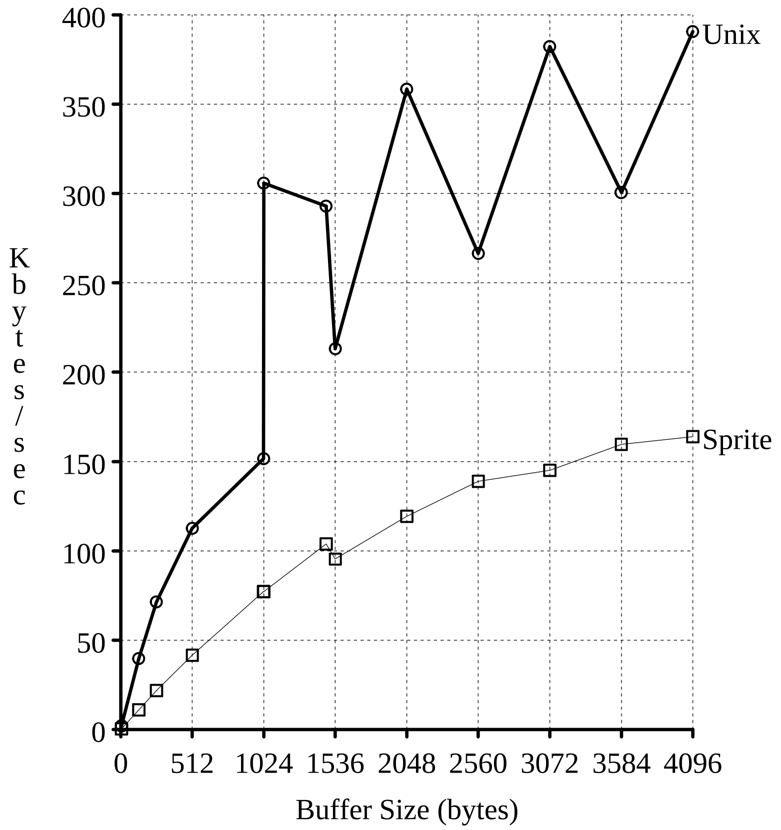

Figure 5. Comparison of Sprite-to-Sprite and Unix-to-Unix UDP throughput rates. The following amounts of data were written to a datagram socket by *ttcp*: 5, 128, 256, 512, 1023, 1024, 1470, 1536, 2048, 2560, 3072, 3584, and 4096 bytes.

Several aspects of the UDP and IP protocols affect *ttcp* performance. UDP does not buffer data, so any data written to the socket by *ttcp* is sent directly to the network by the IP server. Hence UDP has lower throughput than TCP for small packets. UDP does not guarantee delivery of datagrams and the benchmark does not account for dropped packets. Because of the user-level overhead, a Sprite host is only able to receive about 60% of 4096-byte packets sent by another Sprite host, compared to 98–99% for Unix†. IP packet fragmentation also affects the UDP throughput rate. The ethernet's maximum packet size is 1500 bytes so UDP packets can contain up to 1472 bytes of data (the IP and UDP headers require a total of 28 bytes). The effect of fragmentation is visible for the 1470-byte buffer size on both the Unix and Sprite plots. The Sprite rate for a 3072-byte buffer also shows the effect: it is sent in three packets of 1472, 1472 and 128 bytes.

Figure 6 shows the elapsed times for sending various amounts of data using UDP.

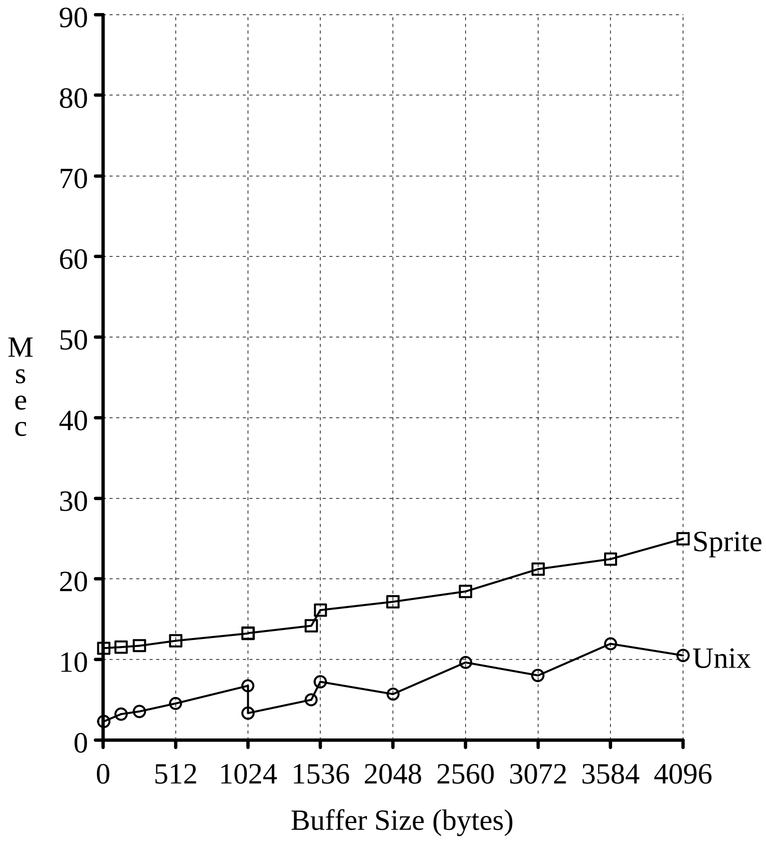

Figure 6. Elapsed time to write one data buffer to a datagram socket from Sprite to Sprite and from Unix to Unix. The data buffer sizes are the same as in Figure 5.

---

† Since the socket receive buffer size used in the benchmarks is 4096 bytes, a datagram socket can buffer only one 4096-byte UDP packet. If the benchmark can't read the socket at a rate faster than the rate of arriving packets, packets will be dropped. Also, the /dev/etherIP packet queue in the kernel has 16 buffers – it can overflow if the IP server doesn't read it fast enough.

The latencies seen by *ttcp* for Unix and Sprite differ by 5 times: 2.27 ms vs. 11.39 ms. Buffering by TCP has an effect on latency: Unix TCP latency is one-third of the UDP latency; on Sprite, the ratio for the two protocols is about 0.9.

4.3. User-level Implementation Overhead

The Sprite user-level implementation incurs more overhead than a kernel-level implementation for two reasons: client-server IPC and the ethernet device interface. Interprocess communication between clients and the IP server results in extra context switches and buffer copies. The server's access to the ethernet is through a device file, resulting in extra buffer copies. Of the two, the IPC is more costly. This was determined by replacing the IP server with another program that satisfied client requests on the server's pseudo-devices but did not process the requests. Figure 7 is a combination of Figures 4 and 6 with the elapsed times using the fake IP server. For UDP and TCP, 75-80% of the latency is due to pseudo-devices. Also, the Fs_Select system call is used by the server to wait for requests and packets; about a quarter of the latency is due to this system call.

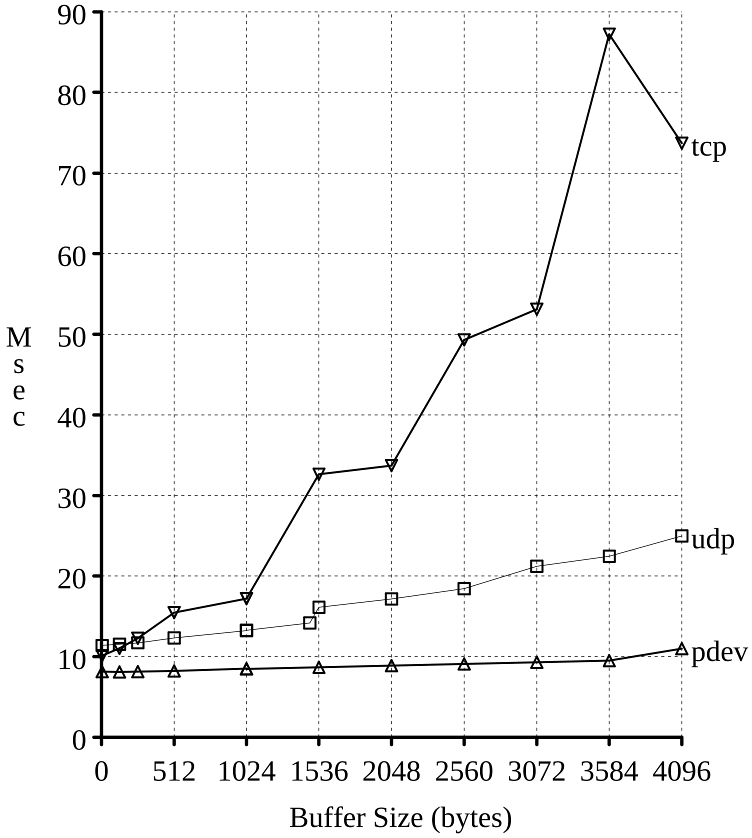

Figure 7. Times to send various buffer sizes using TCP and UDP on Sprite. The pdev line show the amount of time spent in using the UDP or TCP pseudo-devices.

Measurements of the CPU time used by the IP server and *ttcp* benchmark show that they are CPU-intensive. Elapsed time for *ttcp* to send 2000 1KB buffers to a stream socket was 33.2 seconds. For *ttcp*, user and system cpu time were 0.1 and 6.6 seconds. For the IP server, they were 7.2 and 16.3 seconds. The sum of these times is 30.2 seconds, which is close to the elapsed time. This means that almost 70% of the time is spent executing in the kernel and much of the remainder in the server. Preliminary profiling of the kernel did not highlight any routines as bottlenecks – this needs to be investigated in more depth.

5. Summary and Conclusions

The Sprite IP server has met its design goals: working versions of the server and 4.3BSD network utilities such as rcp were completed in about 5 man-months, the programming interface is 4.3BSD-compatible, and the server provides adequate performance. Two changes in the Sprite kernel were required to support the server: direct access to the ethernet and a flexible file system-based IPC mechanism (pseudo-devices). The former was trivial to implement, the latter was not. The initial pseudo-device implementation was buggy and slow; it was reimplemented several times to improve performance and to integrate it into the file system code more cleanly. Pseudo-devices were designed to be general-purpose so other programs could use them effectively. They are used by the X window system server and by terminal emulator routines for their IPC needs.

The IP server's performance is not spectacular but is adequate for the tasks that currently use it. Typically, we use the Internet protocols for file transfers and remote logins between Sprite and Unix. Most file transfers are performed by a daemon process that backups Sprite files to Unix each night. Remote logins usually do not require high throughput but do require low latency; the server's performance in this area seems to be acceptable.

The current user-level implementation of the Internet protocols is not adequate for frequently-used, high-performance applications. To base Sun's Network File System, which uses TCP and UDP, as a client of the current server will condemn NFS to poor performance. Also, a user-level implementation of the IP suite precludes use of the protocols by the Sprite kernel. In the future, we would like to layer the Sprite kernel-to-kernel RPC protocol on top of the Internet Protocol to allow the RPC protocol to work in an internet environment. Since performance of the RPC protocol is critical to overall system performance, the IP routines must be placed in the kernel in order to reduce overhead. With IP in the kernel, we should investigate whether the UDP and TCP code need to be in the kernel or can remain at user-level. For example, each process using UDP could have library routines to process the protocol if the kernel handled packet demultiplexing. The optimal kernel-user boundary needs to be researched.

5.1. Suggestions for Future Work

Pseudo-devices have been recently reimplemented to acheive greater performance and the IP server should be updated to use them. Also, additional performance tuning of the server and the Sprite kernel is definitely necessary. The first step is to profile the server and kernel to determine the bottlenecks. Sprite does not currently support profiling of user-level processes; this support must be added. The TCP output routines should be tuned to reduce the output of small packets. UDP socket receive buffers and the kernel packet buffers have to be increased to reduce the number of dropped UDP packets.

For completeness, the following functions should be added to the server:

- Ability to forward packets. This would allow a Sprite host to act as a gateway between networks.
- Dynamic updating of the routing database. The 4.3BSD implementation uses the Address Resolution Protocol (ARP) [PLUM82] to map IP addresses to ethernet

addresses. A routing daemon process listens for routes broadcast on UDP port 520 and updates the kernel's gateway route table using I/O controls. For Sprite, the server should be modified to use ARP and obtain the route data dynamically.

- Processing of ICMP source quench messages. Currently, the IP server is used for local operations and these messages are not seen. If remote operations through congested internets become frequent, supporting this message will be essential.

6. Acknowledgements

I would like to thank John Ousterhout, Brent Welch, Michael Nelson, Fred Douglis, Adam de Boor, and Luis-Felipe Cabrera for their excellent advice and help. Brent has continually strived to improve the pseudo-device implementation. Keith Sklower was kind to provide a SunOS kernel with the 4.3BSD network code. John Ousterhout, David Patterson, Fred Douglis and Michael Nelson provided helpful comments on early drafts of this report.

7. References

[CABR84]

Cabrera, L-F., Hunter, E., Karels, M., and Mosher, D. ‘‘A User-Process Oriented Performance Study of Ethernet Networking under Berkeley Unix 4.2BSD’’, U.C. Berkeley Computer Science Division Technical Report #84/217, December 1984.

[CABR87]

Cabrera, L-F. ‘‘Improving Network Subsystem Performance in a Distributed Environment. A Berkeley Unix Case Study,’’ Research Report, IBM Almaden Research Center, San Jose, CA, June 1987.

[CERF83]

Cerf, V.G. and Cain, E. ‘‘The DoD Internet Architecture Model,’’ *Computer Networks* **7**, pp. 307-318, 1983.

[CLAR87]

Clark, D. Personal communication.

[HILL86]

Hill, M.D., *et al*. ‘‘Design Decisions in SPUR: a VLSI Multiprocessor,’’ *IEEE Computer* **19**, November 1986.

[KENT87]

Kent, C.A., and Mogul, J.C. ‘‘Fragmentation Considered Harmful,’’ in *Proceedings of ACM SIGCOMM ’87 Workshop on Frontiers in Computer Communications Technology*, Stowe, Vt., August 11-13, 1987. To be published in *Computer Communications Review*, 1988.

[LEFF86]

Leffler, S.J., Joy, W.N., Fabry, R.S, and Karels, M.J. ‘‘Network Implementation Notes, 4.3BSD edition,’’ System Manager’s Manual (SMM 15), Computer Systems Research Group, University of California, Berkeley, CA, June 1986.

[MOGU87]

Mogul, J., Rashid, R., and Accetta, M. ‘‘The Packet Filter: An Efficient Mechanism for User-level Network Code,’’ in *Proceedings of the 11th Symposium of Operating Systems Principles*, Austin, TX, November 1987.

[OUST88]

Ousterhout, J.K., Cherenson, A.R., Douglis, F., Nelson, M.N. and Welch, B.B. ‘‘The Sprite Network Operating System,’’ *IEEE Computer* **21**, to appear, 1988.

[PLUM82]

Plummer, D.C. ‘‘An Ethernet Address Resolution Protocol’’, RFC 826, Symbolics, Inc., Cambridge, MA, November 1982. Available from the Network Information Center at SRI International, Menlo Park, CA.

[POST80]

Postel, J. ‘‘User Datagram Protocol’’, RFC 768, USC Information Sciences Institute, Marina Del Ray, CA, August 1980 Available from the Network Information Center at SRI International, Menlo Park, CA.

[POST81a]

Postel, J. ‘‘Internet Protocol’’, RFC 791, USC Information Sciences Institute, Marina Del Ray, CA, September 1981. Available from the Network Information Center at SRI International, Menlo Park, CA.

[POST81b]
Postel, J. ‘‘Internet Control Message Protocol’’, RFC 792, USC Information Sciences Institute, Marina Del Ray, CA, September 1981. Available from the Network Information Center at SRI International, Menlo Park, CA.

[POST81c]
Postel, J. ‘‘Transmission Control Protocol’’, RFC 793, USC Information Sciences Institute, Marina Del Ray, CA, September 1981. Available from the Network Information Center at SRI International, Menlo Park, CA.

[QUAR86]
Quarterman, J.S. and Hoskins, J.C. ‘‘Notable Computer Networks,’’ *Communications of the ACM* **29**, October 1986.

[TANE81]
Tanenbaum, A.S. ‘‘Network Protocols,’’ *ACM Computing Surveys* **13**, no. 4, December 1981.

[THEI87]
Theimer, M.M. Personal communication.

[WELC86]
Welch, B.B. ‘‘The Sprite Remote Procedure Call System,’’ Master’s Report, Computer Science Division, U.C. Berkeley. Published as UCB/CSD Technical Report #86/302, June 1986.

Appendix 1: Using the Sprite IP Server

The Sprite IP server, /sprite/daemons/ipServer, is started from the /bootcmds file during system initialization. Information and error messages are written to standard error and should be redirected to a file or to the system log. The program understands the following command-line options:

–b Will not automatically detach itself and run in background. This is useful when debugging the server with dbx.

–d Turns on debug output. Signal 25 can be sent to toggle the state of this flag. Implies the use of the –b flag.

–c *file* Specifies the name of a configuration file. Default is /sprite/daemons/ipServer.config. The file format is described below.

–i *address* Specifies the Internet address of the server. All packets sent to this address will be processed. The default address is obtained from the configuration file.

The server also handles the following signal numbers. A signal is sent to the server using the ‘‘kill -*NUM*’’ command.

25 Toggles the debug flag, which enables or disables debugging messages.

26 Prints cumulative statistics counters.

27 Prints information about active sockets.

28 Prints memory allocator statistics.

29 Prints the number of times each system call has been called since the server started.

30 Equivalent to sending signals 26, 27 and 25 in that order.

Configuration File

The IP server uses the configuration file to initialize various facts about the network. A simple config file is shown below. Lines that begin with # are ignored. The server must be restarted if the config file is changed. The first line must begin with the string ‘‘version:’’. This version ID is printed in the log when the server starts. Ideally, the ID should be updated whenever the file is updated. The second line describes the network interface: the filename of the device, the IP subnet mask and the maximum packet size for the network. The subnet mask is used to obtain the host ID of an IP address if subnetting is used. If subnetting is turned off, the mask should be the same as the network mask for the host’s IP address. Using the example file below, the subnet mask is 0xFFFFFF00. Since Berkeley’s IP addresses are class B addresses (16 bits for network part, 16 bits for host part), this mask says that the upper 8 bits of the host part is the subnet. If subnet is not used, the mask would be 0xFFFF0000.

The rest of the configuration file contains variable length data. The first such section contains the aliases for the host, i.e., IP addresses that should be recognized as belonging to the current host. Currently, the server does not use aliases because it cannot serve as a gateway. The alias list is delimited by two lines: Start_Aliases and End_Aliases. Each token must start at the beginning of a line. The next section contains the routing database and delimited by the tokens Start_Neighbors and End_Neighbors.

```
#
# Sprite Internet Protocol Server configuration file
version:  IPS  8/6/87
#
# net device          subnet mask         max #bytes/packet
/dev/etherIP          0xffffff00          1500

# not used yet
Start_Aliases
End_Aliases

# the first address is assumed to be the gateway for the net.
Start_Neighbors
csgw                  128.32.150.73       8:0:2b:3:bc:d6          gateway
cayenne               128.32.150.25       8:0:20:1:20:F6          neighbor
ginger                128.32.150.28       8:0:20:1:49:48          neighbor
lust                  128.32.150.11       8:0:20:1:2:c6           neighbor
mint                  128.32.150.52       8:0:20:1:5c:ce          neighbor
murder                128.32.150.09       8:0:20:1:1D:73          neighbor
paprika               128.32.150.08       8:0:20:1:48:2E          neighbor
pride                 128.32.150.13       8:0:20:1:13:E1          neighbor
sage                  128.32.150.06       8:0:20:1:be:34          neighbor
thyme                 128.32.150.17       8:0:20:1:48:51          neighbor
End_Neighbors
```

Figure App-1. Sample configuration file for the IP server.

The format of the line is: hostname, IP address, ethernet address and one of the tokens ‘‘gateway’’ or ‘‘neighbor’’. A host acting as a gateway should be the first host in the list. Note that this list must be updated whenever the ethernet address of host changes or a new host is added to the network. (If a host is not in the list, then the IP server can’t communicate with it!) The current host must be in the list because the server scans the list during initialization to determine its IP address.