An Advanced 4.4BSD Interprocess Communication Tutorial

Samuel J. Leffler

Robert S. Fabry

William N. Joy

Phil Lapsley

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

Steve Miller

Chris Torek

Heterogeneous Systems Laboratory
Department of Computer Science
University of Maryland, College Park
College Park, Maryland 20742

ABSTRACT

This document provides an introduction to the interprocess communication facilities included in the 4.4BSD release of the UNIX* system.

It discusses the overall model for interprocess communication and introduces the interprocess communication primitives which have been added to the system. The majority of the document considers the use of these primitives in developing applications. The reader is expected to be familiar with the C programming language as all examples are written in C.

---

* UNIX is a trademark of UNIX System Laboratories, Inc. in the US and some other countries.

1. INTRODUCTION

One of the most important additions to UNIX in 4.2BSD was interprocess communication. These facilities were the result of more than two years of discussion and research. The facilities provided in 4.2BSD incorporated many of the ideas from current research, while trying to maintain the UNIX philosophy of simplicity and conciseness. The 4.3BSD release of Berkeley UNIX improved upon some of the IPC facilities while providing an upward-compatible interface. 4.4BSD adds support for ISO protocols and IP multicasting. The BSD interprocess communication facilities have become a defacto standard for UNIX.

UNIX has previously been very weak in the area of interprocess communication. Prior to the 4BSD facilities, the only standard mechanism which allowed two processes to communicate were pipes (the mpx files which were part of Version 7 were experimental). Unfortunately, pipes are very restrictive in that the two communicating processes must be related through a common ancestor. Further, the semantics of pipes makes them almost impossible to maintain in a distributed environment.

Earlier attempts at extending the IPC facilities of UNIX have met with mixed reaction. The majority of the problems have been related to the fact that these facilities have been tied to the UNIX file system, either through naming or implementation. Consequently, the IPC facilities provided in 4.2BSD were designed as a totally independent subsystem. The BSD IPC allows processes to rendezvous in many ways. Processes may rendezvous through a UNIX file system-like name space (a space where all names are path names) as well as through a network name space. In fact, new name spaces may be added at a future time with only minor changes visible to users. Further, the communication facilities have been extended to include more than the simple byte stream provided by a pipe. These extensions have resulted in a completely new part of the system which users will need time to familiarize themselves with. It is likely that as more use is made of these facilities they will be refined; only time will tell.

This document provides a high-level description of the IPC facilities in 4.4BSD and their use. It is designed to complement the manual pages for the IPC primitives by examples of their use. The remainder of this document is organized in four sections. Section 2 introduces the IPC-related system calls and the basic model of communication. Section 3 describes some of the supporting library routines users may find useful in constructing distributed applications. Section 4 is concerned with the client/server model used in developing applications and includes examples of the two major types of servers. Section 5 delves into advanced topics which sophisticated users are likely to encounter when using the IPC facilities.

2. BASICS

The basic building block for communication is the *socket*. A socket is an endpoint of communication to which a name may be *bound*. Each socket in use has a *type* and one or more associated processes. Sockets exist within *communication domains*. A communication domain is an abstraction introduced to bundle common properties of processes communicating through sockets. One such property is the scheme used to name sockets. For example, in the UNIX communication domain sockets are named with UNIX path names; e.g. a socket may be named ‘‘/dev/foo’’. Sockets normally exchange data only with sockets in the same domain (it may be possible to cross domain boundaries, but only if some translation process is performed). The 4.4BSD IPC facilities support four separate communication domains: the UNIX domain, for on-system communication; the Internet domain, which is used by processes which communicate using the Internet standard communication protocols; the NS domain, which is used by processes which communicate using the Xerox standard communication protocols*; and the ISO OSI protocols, which are not documented in this tutorial. The underlying communication facilities provided by these domains have a significant influence on the internal system implementation as well as the interface to socket facilities available to a user. An example of the latter is that a socket ‘‘operating’’ in the UNIX domain sees a subset of the error conditions which are possible when operating in the Internet (or NS) domain.

2.1. Socket types

Sockets are typed according to the communication properties visible to a user. Processes are presumed to communicate only between sockets of the same type, although there is nothing that prevents communication between sockets of different types should the underlying communication protocols support this.

Four types of sockets currently are available to a user. A *stream* socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from the bidirectionality of data flow, a pair of connected stream sockets provides an interface nearly identical to that of pipes†.

A *datagram* socket supports bidirectional flow of data which is not promised to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated, and, possibly, in an order different from the order in which it was sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model the facilities found in many contemporary packet switched networks such as the Ethernet.

A *raw* socket provides users access to the underlying communication protocols which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more esoteric facilities of an existing protocol. The use of raw sockets is considered in section 5.

A *sequenced packet* socket is similar to a stream socket, with the exception that record boundaries are preserved. This interface is provided only as part of the NS socket abstraction, and is very important in most serious NS applications. Sequenced-packet sockets allow the user to manipulate the SPP or IDP headers on a packet or a group of packets either by writing a prototype header along with whatever data is to be sent, or by specifying a default header to be used with all outgoing data, and allows the user to receive the headers on incoming packets. The use of these options is considered in section 5.

Another potential socket type which has interesting properties is the *reliably delivered message* socket. The reliably delivered message socket has similar properties to a datagram socket, but with reliable delivery. There is currently no support for this type of socket, but a reliably delivered message protocol similar to Xerox's Packet Exchange Protocol (PEX) may be simulated at the user level. More information

---

* See *Internet Transport Protocols*, Xerox System Integration Standard (XSIS)028112 for more information. This document is almost a necessity for one trying to write NS applications.

† In the UNIX domain, in fact, the semantics are identical and, as one might expect, pipes have been implemented internally as simply a pair of connected stream sockets.

on this topic can be found in section 5.

2.2. Socket creation

To create a socket the *socket* system call is used:

s = socket(domain, type, protocol);

This call requests that the system create a socket in the specified *domain* and of the specified *type*. A particular protocol may also be requested. If the protocol is left unspecified (a value of 0), the system will select an appropriate protocol from those protocols which comprise the communication domain and which may be used to support the requested socket type. The user is returned a descriptor (a small integer number) which may be used in later system calls which operate on sockets. The domain is specified as one of the manifest constants defined in the file <*sys/socket.h*>. For the UNIX domain the constant is AF_UNIX*; for the Internet domain AF_INET; and for the NS domain, AF_NS. The socket types are also defined in this file and one of SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, or SOCK_SEQPACKET must be specified. To create a stream socket in the Internet domain the following call might be used:

s = socket(AF_INET, SOCK_STREAM, 0);

This call would result in a stream socket being created with the TCP protocol providing the underlying communication support. To create a datagram socket for on-machine use the call might be:

s = socket(AF_UNIX, SOCK_DGRAM, 0);

The default protocol (used when the *protocol* argument to the *socket* call is 0) should be correct for most every situation. However, it is possible to specify a protocol other than the default; this will be covered in section 5.

There are several reasons a socket call may fail. Aside from the rare occurrence of lack of memory (ENOBUFS), a socket request may fail due to a request for an unknown protocol (EPROTONOSUPPORT), or a request for a type of socket for which there is no supporting protocol (EPROTOTYPE).

2.3. Binding local names

A socket is created without a name. Until a name is bound to a socket, processes have no way to reference it and, consequently, no messages may be received on it. Communicating processes are bound by an *association*. In the Internet and NS domains, an association is composed of local and foreign addresses, and local and foreign ports, while in the UNIX domain, an association is composed of local and foreign path names (the phrase ‘‘foreign pathname’’ means a pathname created by a foreign process, not a pathname on a foreign system). In most domains, associations must be unique. In the Internet domain there may never be duplicate <protocol, local address, local port, foreign address, foreign port> tuples. UNIX domain sockets need not always be bound to a name, but when bound there may never be duplicate <protocol, local pathname, foreign pathname> tuples. The pathnames may not refer to files already existing on the system in 4.3; the situation may change in future releases.

The *bind* system call allows a process to specify half of an association, <local address, local port> (or <local pathname>), while the *connect* and *accept* primitives are used to complete a socket's association.

In the Internet domain, binding names to sockets can be fairly complex. Fortunately, it is usually not necessary to specifically bind an address and port number to a socket, because the *connect* and *send* calls will automatically bind an appropriate address if they are used with an unbound socket. The process of binding names to NS sockets is similar in most ways to that of binding names to Internet sockets.

The *bind* system call is used as follows:

bind(s, name, namelen);

The bound name is a variable length byte string which is interpreted by the supporting protocol(s). Its interpretation may vary from communication domain to communication domain (this is one of the

* The manifest constants are named AF_whatever as they indicate the ‘‘address format’’ to use in interpreting names.

properties which comprise the ‘‘domain’’). As mentioned, in the Internet domain names contain an Internet address and port number. NS domain names contain an NS address and port number. In the UNIX domain, names contain a path name and a family, which is always AF_UNIX. If one wanted to bind the name ‘‘/tmp/foo’’ to a UNIX domain socket, the following code would be used*:

```
#include <sys/un.h>
 ...
struct sockaddr_un addr;
 ...
strcpy(addr.sun_path, "/tmp/foo");
addr.sun_family = AF_UNIX;
bind(s, (struct sockaddr *) &addr, strlen(addr.sun_path) +
   sizeof (addr.sun_len) + sizeof (addr.sun_family));
```

Note that in determining the size of a UNIX domain address null bytes are not counted, which is why *strlen* is used. In the current implementation of UNIX domain IPC, the file name referred to in *addr.sun_path* is created as a socket in the system file space. The caller must, therefore, have write permission in the directory where *addr.sun_path* is to reside, and this file should be deleted by the caller when it is no longer needed. Future versions of 4BSD may not create this file.

In binding an Internet address things become more complicated. The actual call is similar,

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

but the selection of what to place in the address *sin* requires some discussion. We will come back to the problem of formulating Internet addresses in section 3 when the library routines used in name resolution are discussed.

Binding an NS address to a socket is even more difficult, especially since the Internet library routines do not work with NS hostnames. The actual call is again similar:

```
#include <sys/types.h>
#include <netns/ns.h>
 ...
struct sockaddr_ns sns;
 ...
bind(s, (struct sockaddr *) &sns, sizeof (sns));
```

Again, discussion of what to place in a ‘‘struct sockaddr_ns’’ will be deferred to section 3.

2.4. Connection establishment

Connection establishment is usually asymmetric, with one process a ‘‘client’’ and the other a ‘‘server’’. The server, when willing to offer its advertised services, binds a socket to a well-known address associated with the service and then passively ‘‘listens’’ on its socket. It is then possible for an unrelated process to rendezvous with the server. The client requests services from the server by initiating a ‘‘connection’’ to the server’s socket. On the client side the *connect* call is used to initiate a connection. Using the UNIX domain, this might appear as,

---

* Note that, although the tendency here is to call the ‘‘addr’’ structure ‘‘sun’’, doing so would cause problems if the code were ever ported to a Sun workstation.

```
struct sockaddr_un server;
 ...
connect(s, (struct sockaddr *)&server, strlen(server.sun_path) +
   sizeof (server.sun_family));
```

while in the Internet domain,

```
struct sockaddr_in server;
 ...
connect(s, (struct sockaddr *)&server, sizeof (server));
```

and in the NS domain,

```
struct sockaddr_ns server;
 ...
connect(s, (struct sockaddr *)&server, sizeof (server));
```

where *server* in the example above would contain either the UNIX pathname, Internet address and port number, or NS address and port number of the server to which the client process wishes to speak. If the client process's socket is unbound at the time of the connect call, the system will automatically select and bind a name to the socket if necessary; c.f. section 5.4. This is the usual way that local addresses are bound to a socket.

An error is returned if the connection was unsuccessful (any name automatically bound by the system, however, remains). Otherwise, the socket is associated with the server and data transfer may begin. Some of the more common errors returned when a connection attempt fails are:

ETIMEDOUT
: After failing to establish a connection for a period of time, the system decided there was no point in retrying the connection attempt any more. This usually occurs because the destination host is down, or because problems in the network resulted in transmissions being lost.

ECONNREFUSED
: The host refused service for some reason. This is usually due to a server process not being present at the requested name.

ENETDOWN or EHOSTDOWN
: These operational errors are returned based on status information delivered to the client host by the underlying communication services.

ENETUNREACH or EHOSTUNREACH
: These operational errors can occur either because the network or host is unknown (no route to the network or host is present), or because of status information returned by intermediate gateways or switching nodes. Many times the status returned is not sufficient to distinguish a network being down from a host being down, in which case the system indicates the entire network is unreachable.

For the server to receive a client's connection it must perform two steps after binding its socket. The first is to indicate a willingness to listen for incoming connection requests:

```
listen(s, 5);
```

The second parameter to the *listen* call specifies the maximum number of outstanding connections which may be queued awaiting acceptance by the server process; this number may be limited by the system. Should a connection be requested while the queue is full, the connection will not be refused, but rather the individual messages which comprise the request will be ignored. This gives a harried server time to make room in its pending connection queue while the client retries the connection request. Had the connection been returned with the ECONNREFUSED error, the client would be unable to tell if the server was up or not. As it is now it is still possible to get the ETIMEDOUT error back, though this is unlikely. The backlog figure supplied with the listen call is currently limited by the system to a maximum of 5 pending connections on any one queue. This avoids the problem of processes hogging system resources by setting an infinite backlog, then ignoring all connection requests.

With a socket marked as listening, a server may *accept* a connection:

```
struct sockaddr_in from;
 ...
fromlen = sizeof (from);
newsock = accept(s, (struct sockaddr *)&from, &fromlen);
```

(For the UNIX domain, *from* would be declared as a *struct sockaddr_un*, and for the NS domain, *from* would be declared as a *struct sockaddr_ns*, but nothing different would need to be done as far as *fromlen* is concerned. In the examples which follow, only Internet routines will be discussed.) A new descriptor is returned on receipt of a connection (along with a new socket). If the server wishes to find out who its client is, it may supply a buffer for the client socket's name. The value-result parameter *fromlen* is initialized by the server to indicate how much space is associated with *from*, then modified on return to reflect the true size of the name. If the client's name is not of interest, the second parameter may be a null pointer.

Accept normally blocks. That is, *accept* will not return until a connection is available or the system call is interrupted by a signal to the process. Further, there is no way for a process to indicate it will accept connections from only a specific individual, or individuals. It is up to the user process to consider who the connection is from and close down the connection if it does not wish to speak to the process. If the server process wants to accept connections on more than one socket, or wants to avoid blocking on the accept call, there are alternatives; they will be considered in section 5.

2.5. Data transfer

With a connection established, data may begin to flow. To send and receive data there are a number of possible calls. With the peer entity at each end of a connection anchored, a user can send or receive a message without specifying the peer. As one might expect, in this case, then the normal *read* and *write* system calls are usable,

```
write(s, buf, sizeof (buf));
read(s, buf, sizeof (buf));
```

In addition to *read* and *write*, the new calls *send* and *recv* may be used:

```
send(s, buf, sizeof (buf), flags);
recv(s, buf, sizeof (buf), flags);
```

While *send* and *recv* are virtually identical to *read* and *write*, the extra *flags* argument is important. The flags, defined in <*sys/socket.h*>, may be specified as a non-zero value if one or more of the following is required:

MSG_OOB	send/receive out of band data
MSG_PEEK	look at data without reading
MSG_DONTROUTE	send data without routing packets

Out of band data is a notion specific to stream sockets, and one which we will not immediately consider. The option to have data sent without routing applied to the outgoing packets is currently used only by the routing table management process, and is unlikely to be of interest to the casual user. The ability to preview data is, however, of interest. When MSG_PEEK is specified with a *recv* call, any data present is returned to the user, but treated as still ‘‘unread’’. That is, the next *read* or *recv* call applied to the socket will return the data previously previewed.

2.6. Discarding sockets

Once a socket is no longer of interest, it may be discarded by applying a *close* to the descriptor,

```
close(s);
```

If data is associated with a socket which promises reliable delivery (e.g. a stream socket) when a close takes place, the system will continue to attempt to transfer the data. However, after a fairly long period of time, if the data is still undelivered, it will be discarded. Should a user have no use for any pending data, it may perform a *shutdown* on the socket prior to closing it. This call is of the form:

```
shutdown(s, how);
```

where *how* is 0 if the user is no longer interested in reading data, 1 if no more data will be sent, or 2 if no data is to be sent or received.

2.7. Connectionless sockets

To this point we have been concerned mostly with sockets which follow a connection oriented model. However, there is also support for connectionless interactions typical of the datagram facilities found in contemporary packet switched networks. A datagram socket provides a symmetric interface to data exchange. While processes are still likely to be client and server, there is no requirement for connection establishment. Instead, each message includes the destination address.

Datagram sockets are created as before. If a particular local address is needed, the *bind* operation must precede the first data transmission. Otherwise, the system will set the local address and/or port when data is first sent. To send data, the *sendto* primitive is used,

```
sendto(s, buf, buflen, flags, (struct sockaddr *)&to, tolen);
```

The *s*, *buf*, *buflen*, and *flags* parameters are used as before. The *to* and *tolen* values are used to indicate the address of the intended recipient of the message. When using an unreliable datagram interface, it is unlikely that any errors will be reported to the sender. When information is present locally to recognize a message that can not be delivered (for instance when a network is unreachable), the call will return −1 and the global value *errno* will contain an error number.

To receive messages on an unconnected datagram socket, the *recvfrom* primitive is provided:

```
recvfrom(s, buf, buflen, flags, (struct sockaddr *)&from, &fromlen);
```

Once again, the *fromlen* parameter is handled in a value-result fashion, initially containing the size of the *from* buffer, and modified on return to indicate the actual size of the address from which the datagram was received.

In addition to the two calls mentioned above, datagram sockets may also use the *connect* call to associate a socket with a specific destination address. In this case, any data sent on the socket will automatically be addressed to the connected peer, and only data received from that peer will be delivered to the user. Only one connected address is permitted for each socket at one time; a second connect will change the destination address, and a connect to a null address (family AF_UNSPEC) will disconnect. Connect requests on datagram sockets return immediately, as this simply results in the system recording the peer's address (as compared to a stream socket, where a connect request initiates establishment of an end to end connection). *Accept* and *listen* are not used with datagram sockets.

While a datagram socket socket is connected, errors from recent *send* calls may be returned asynchronously. These errors may be reported on subsequent operations on the socket, or a special socket option used with *getsockopt*, SO_ERROR, may be used to interrogate the error status. A *select* for reading or writing will return true when an error indication has been received. The next operation will return the error, and the error status is cleared. Other of the less important details of datagram sockets are described in section 5.

2.8. Input/Output multiplexing

One last facility often used in developing applications is the ability to multiplex i/o requests among multiple sockets and/or files. This is done using the *select* call:

```
#include <sys/time.h>
#include <sys/types.h>
 ...

fd_set readmask, writemask, exceptmask;
struct timeval timeout;
 ...
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

Select takes as arguments pointers to three sets, one for the set of file descriptors for which the caller wishes to be able to read data on, one for those descriptors to which data is to be written, and one for which exceptional conditions are pending; out-of-band data is the only exceptional condition currently implemented by the socket If the user is not interested in certain conditions (i.e., read, write, or exceptions), the corresponding argument to the *select* should be a null pointer.

Each set is actually a structure containing an array of long integer bit masks; the size of the array is set by the definition FD_SETSIZE. The array is be long enough to hold one bit for each of FD_SETSIZE file descriptors.

The macros FD_SET(*fd, &mask*) and FD_CLR(*fd, &mask*) have been provided for adding and removing file descriptor *fd* in the set *mask*. The set should be zeroed before use, and the macro FD_ZERO(*&mask*) has been provided to clear the set *mask*. The parameter *nfds* in the *select* call specifies the range of file descriptors (i.e. one plus the value of the largest descriptor) to be examined in a set.

A timeout value may be specified if the selection is not to last more than a predetermined period of time. If the fields in *timeout* are set to 0, the selection takes the form of a *poll*, returning immediately. If the last parameter is a null pointer, the selection will block indefinitely*. *Select* normally returns the number of file descriptors selected; if the *select* call returns due to the timeout expiring, then the value 0 is returned. If the *select* terminates because of an error or interruption, a −1 is returned with the error number in *errno*, and with the file descriptor masks unchanged.

Assuming a successful return, the three sets will indicate which file descriptors are ready to be read from, written to, or have exceptional conditions pending. The status of a file descriptor in a select mask may be tested with the *FD_ISSET(fd, &mask)* macro, which returns a non-zero value if *fd* is a member of the set *mask*, and 0 if it is not.

To determine if there are connections waiting on a socket to be used with an *accept* call, *select* can be used, followed by a *FD_ISSET(fd, &mask)* macro to check for read readiness on the appropriate socket. If *FD_ISSET* returns a non-zero value, indicating permission to read, then a connection is pending on the socket.

As an example, to read data from two sockets, *s1* and *s2* as it is available from each and with a one-second timeout, the following code might be used:

* To be more specific, a return takes place only when a descriptor is selectable, or when a signal is received by the caller, interrupting the system call.

```
#include <sys/time.h>
#include <sys/types.h>
 ...
fd_set read_template;
struct timeval wait;
 ...
for (;;) {
	wait.tv_sec = 1;		/* one second */
	wait.tv_usec = 0;

	FD_ZERO(&read_template);

	FD_SET(s1, &read_template);
	FD_SET(s2, &read_template);

	nb = select(FD_SETSIZE, &read_template, (fd_set *) 0, (fd_set *) 0, &wait);
	if (nb <= 0) {
		An error occurred during the select, or
		the select timed out.
	}

	if (FD_ISSET(s1, &read_template)) {
		Socket #1 is ready to be read from.
	}

	if (FD_ISSET(s2, &read_template)) {
		Socket #2 is ready to be read from.
	}
}
```

In 4.2, the arguments to *select* were pointers to integers instead of pointers to *fd_set*s. This type of call will still work as long as the number of file descriptors being examined is less than the number of bits in an integer; however, the methods illustrated above should be used in all current programs.

Select provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions is possible through use of the SIGIO and SIGURG signals described in section 5.

3. NETWORK LIBRARY ROUTINES

The discussion in section 2 indicated the possible need to locate and construct network addresses when using the interprocess communication facilities in a distributed environment. To aid in this task a number of routines have been added to the standard C run-time library. In this section we will consider the new routines provided to manipulate network addresses. While the 4.4BSD networking facilities support the Internet protocols and the Xerox NS protocols, most of the routines presented in this section do not apply to the NS domain. Unless otherwise stated, it should be assumed that the routines presented in this section do not apply to the NS domain.

Locating a service on a remote host requires many levels of mapping before client and server may communicate. A service is assigned a name which is intended for human consumption; e.g. ‘‘the *login server* on host monet’’. This name, and the name of the peer host, must then be translated into network *addresses* which are not necessarily suitable for human consumption. Finally, the address must then used in locating a physical *location* and *route* to the service. The specifics of these three mappings are likely to vary between network architectures. For instance, it is desirable for a network to not require hosts to be named in such a way that their physical location is known by the client host. Instead, underlying services in the network may discover the actual location of the host at the time a client host wishes to communicate. This ability to have hosts named in a location independent manner may induce overhead in connection establishment, as a discovery process must take place, but allows a host to be physically mobile without requiring it to notify its clientele of its current location.

Standard routines are provided for: mapping host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers and the appropriate protocol to use in communicating with the server process. The file <*netdb.h*> must be included when using any of these routines.

3.1. Host names

An Internet host name to address mapping is represented by the *hostent* structure:

```
struct	hostent {
	char	*h_name;		/* official name of host */
	char	**h_aliases;		/* alias list */
	int	h_addrtype;		/* host address type (e.g., AF_INET) */
	int	h_length;		/* length of address */
	char	**h_addr_list;		/* list of addresses, null terminated */
};

#define	h_addr	h_addr_list[0]		/* first address, network byte order */
```

The routine *gethostbyname*(3N) takes an Internet host name and returns a *hostent* structure, while the routine *gethostbyaddr*(3N) maps Internet host addresses into a *hostent* structure.

The official name of the host and its public aliases are returned by these routines, along with the address type (family) and a null terminated list of variable length address. This list of addresses is required because it is possible for a host to have many addresses, all having the same name. The *h_addr* definition is provided for backward compatibility, and is defined to be the first address in the list of addresses in the *hostent* structure.

The database for these calls is provided either by the file */etc/hosts* (*hosts* (5)), or by use of a nameserver, *named* (8). Because of the differences in these databases and their access protocols, the information returned may differ. When using the host table version of *gethostbyname*, only one address will be returned, but all listed aliases will be included. The nameserver version may return alternate addresses, but will not provide any aliases other than one given as argument.

Unlike Internet names, NS names are always mapped into host addresses by the use of a standard NS *Clearinghouse service*, a distributed name and authentication server. The algorithms for mapping NS names to addresses via a Clearinghouse are rather complicated, and the routines are not part of the standard libraries. The user-contributed Courier (Xerox remote procedure call protocol) compiler contains routines to accomplish this mapping; see the documentation and examples provided therein for more information. It is expected that almost all software that has to communicate using NS will need to use the facilities of the Courier compiler.

An NS host address is represented by the following:

```
union ns_host {
        u_char      c_host[6];
        u_short     s_host[3];
};

union ns_net {
        u_char      c_net[4];
        u_short     s_net[2];
};

struct ns_addr {
        union ns_net        x_net;
        union ns_host       x_host;
        u_short     x_port;
};
```

The following code fragment inserts a known NS address into a *ns_addr*:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
 ...
u_long netnum;
struct sockaddr_ns dst;
 ...
bzero((char *)&dst, sizeof(dst));

/*
 * There is no convenient way to assign a long
 * integer to a ``union ns_net'' at present; in
 * the future, something will hopefully be provided,
 * but this is the portable way to go for now.
 * The network number below is the one for the NS net
 * that the desired host (gyre) is on.
 */
netnum = htonl(2266);
dst.sns_addr.x_net = *(union ns_net *) &netnum;
dst.sns_family = AF_NS;

/*
 * host 2.7.1.0.2a.18 == "gyre:Computer Science:UofMaryland"
 */
dst.sns_addr.x_host.c_host[0] = 0x02;
dst.sns_addr.x_host.c_host[1] = 0x07;
dst.sns_addr.x_host.c_host[2] = 0x01;
dst.sns_addr.x_host.c_host[3] = 0x00;
dst.sns_addr.x_host.c_host[4] = 0x2a;
dst.sns_addr.x_host.c_host[5] = 0x18;
dst.sns_addr.x_port = htons(75);
```

3.2. Network names

As for host names, routines for mapping network names to numbers, and back, are provided. These routines return a *netent* structure:

```
/*
 * Assumption here is that a network number
 * fits in 32 bits -- probably a poor one.
 */
struct	netent {
	char	*n_name;	/* official name of net */
	char	**n_aliases;	/* alias list */
	int	n_addrtype;	/* net address type */
	int	n_net;	/* network number, host byte order */
};
```

The routines *getnetbyname*(3N), *getnetbynumber*(3N), and *getnetent*(3N) are the network counterparts to the host routines described above. The routines extract their information from */etc/networks*.

NS network numbers are determined either by asking your local Xerox Network Administrator (and hardcoding the information into your code), or by querying the Clearinghouse for addresses. The internetwork router is the only process that needs to manipulate network numbers on a regular basis; if a process wishes to communicate with a machine, it should ask the Clearinghouse for that machine's address (which will include the net number).

3.3. Protocol names

For protocols, which are defined in */etc/protocols*, the *protoent* structure defines the protocol-name mapping used with the routines *getprotobyname*(3N), *getprotobynumber*(3N), and *getprotoent*(3N):

```
struct  protoent {
        char    *p_name;        /* official protocol name */
        char    **p_aliases;    /* alias list */
        int     p_proto;        /* protocol number */
};
```

In the NS domain, protocols are indicated by the "client type" field of a IDP header. No protocol database exists; see section 5 for more information.

3.4. Service names

Information regarding services is a bit more complicated. A service is expected to reside at a specific ‘‘port’’ and employ a particular communication protocol. This view is consistent with the Internet domain, but inconsistent with other network architectures. Further, a service may reside on multiple ports. If this occurs, the higher level library routines will have to be bypassed or extended. Services available are contained in the file */etc/services*. A service mapping is described by the *servent* structure,

```
struct  servent {
        char    *s_name;        /* official service name */
        char    **s_aliases;    /* alias list */
        int     s_port;         /* port number, network byte order */
        char    *s_proto;       /* protocol to use */
};
```

The routine *getservbyname*(3N) maps service names to a servent structure by specifying a service name and, optionally, a qualifying protocol. Thus the call

```
sp = getservbyname("telnet", (char *) 0);
```

returns the service specification for a telnet server using any protocol, while the call

```
sp = getservbyname("telnet", "tcp");
```

returns only that telnet server which uses the TCP protocol. The routines *getservbyport*(3N) and *getservent*(3N) are also provided. The *getservbyport* routine has an interface similar to that provided by *getservbyname*; an optional protocol name may be specified to qualify lookups.

In the NS domain, services are handled by a central dispatcher provided as part of the Courier remote procedure call facilities. Again, the reader is referred to the Courier compiler documentation and to the Xerox standard* for further details.

3.5. Miscellaneous

With the support routines described above, an Internet application program should rarely have to deal directly with addresses. This allows services to be developed as much as possible in a network independent fashion. It is clear, however, that purging all network dependencies is very difficult. So long as the user is required to supply network addresses when naming services and sockets there will always some network dependency in a program. For example, the normal code included in client programs, such as the remote login program, is of the form shown in Figure 1. (This example will be considered in more detail in section 4.)

If we wanted to make the remote login program independent of the Internet protocols and addressing scheme we would be forced to add a layer of routines which masked the network dependent aspects from the mainstream login code. For the current facilities available in the system this does not appear to be

* *Courier: The Remote Procedure Call Protocol*, XSIS 038112.

worthwhile.

Aside from the address-related data base routines, there are several other routines available in the run-time library which are of interest to users. These are intended mostly to simplify manipulation of names and addresses. Table 1 summarizes the routines for manipulating variable length byte strings and handling byte swapping of network addresses and values.

Call	Synopsis
bcmp(s1, s2, n)	compare byte-strings; 0 if same, not 0 otherwise
bcopy(s1, s2, n)	copy n bytes from s1 to s2
bzero(base, n)	zero-fill n bytes starting at base
htonl(val)	convert 32-bit quantity from host to network byte order
htons(val)	convert 16-bit quantity from host to network byte order
ntohl(val)	convert 32-bit quantity from network to host byte order
ntohs(val)	convert 16-bit quantity from network to host byte order

Table 1. C run-time routines.

The byte swapping routines are provided because the operating system expects addresses to be supplied in network order (aka “big-endian” order). On “little-endian” architectures, such as Intel x86 and VAX, host byte ordering is different than network byte ordering. Consequently, programs are sometimes required to byte swap quantities. The library routines which return network addresses provide them in network order so that they may simply be copied into the structures provided to the system. This implies users should encounter the byte swapping problem only when *interpreting* network addresses. For example, if an Internet port is to be printed out the following code would be required:

```
printf("port number %d\n", ntohs(sp->s_port));
```

On machines where unneeded these routines are defined as null macros.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
 ...
main(argc, argv)
	int argc;
	char *argv[];
{
	struct sockaddr_in server;
	struct servent *sp;
	struct hostent *hp;
	int s;
	...
	sp = getservbyname("login", "tcp");
	if (sp == NULL) {
		fprintf(stderr, "rlogin: tcp/login: unknown service\n");
		exit(1);
	}
	hp = gethostbyname(argv[1]);
	if (hp == NULL) {
		fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
		exit(2);
	}
	bzero((char *)&server, sizeof (server));
	bcopy(hp->h_addr, (char *)&server.sin_addr, hp->h_length);
	server.sin_family = hp->h_addrtype;
	server.sin_port = sp->s_port;
	s = socket(AF_INET, SOCK_STREAM, 0);
	if (s < 0) {
		perror("rlogin: socket");
		exit(3);
	}
	...
	/* Connect does the bind() for us */

	if (connect(s, (char *)&server, sizeof (server)) < 0) {
		perror("rlogin: connect");
		exit(5);
	}
	...
}
```

Figure 1. Remote login client code.

4. CLIENT/SERVER MODEL

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme client applications request services from a server process. This implies an asymmetry in establishing communication between the client and server which has been examined in section 2. In this section we will look more closely at the interactions between client and server, and consider some of the problems in developing client and server applications.

The client and server require a well known set of conventions before service may be rendered (and accepted). This set of conventions comprises a protocol which must be implemented at both ends of a connection. Depending on the situation, the protocol may be symmetric or asymmetric. In a symmetric protocol, either side may play the master or slave roles. In an asymmetric protocol, one side is immutably recognized as the master, with the other as the slave. An example of a symmetric protocol is the TELNET protocol used in the Internet for remote terminal emulation. An example of an asymmetric protocol is the Internet file transfer protocol, FTP. No matter whether the specific protocol used in obtaining a service is symmetric or asymmetric, when accessing a service there is a ‘‘client process’’ and a ‘‘server process’’. We will first consider the properties of server processes, then client processes.

A server process normally listens at a well known address for service requests. That is, the server process remains dormant until a connection is requested by a client’s connection to the server’s address. At such a time the server process ‘‘wakes up’’ and services the client, performing whatever appropriate actions the client requests of it.

Alternative schemes which use a service server may be used to eliminate a flock of server processes clogging the system while remaining dormant most of the time. For Internet servers in 4.4BSD, this scheme has been implemented via *inetd*, the so called ‘‘internet super-server.’’ *Inetd* listens at a variety of ports, determined at start-up by reading a configuration file. When a connection is requested to a port on which *inetd* is listening, *inetd* executes the appropriate server program to handle the client. With this method, clients are unaware that an intermediary such as *inetd* has played any part in the connection. *Inetd* will be described in more detail in section 5.

A similar alternative scheme is used by most Xerox services. In general, the Courier dispatch process (if used) accepts connections from processes requesting services of some sort or another. The client processes request a particular <program number, version number, procedure number> triple. If the dispatcher knows of such a program, it is started to handle the request; if not, an error is reported to the client. In this way, only one port is required to service a large variety of different requests. Again, the Courier facilities are not available without the use and installation of the Courier compiler. The information presented in this section applies only to NS clients and services that do not use Courier.

4.1. Servers

In 4.4BSD most servers are accessed at well known Internet addresses or UNIX domain names. For example, the remote login server’s main loop is of the form shown in Figure 2.

The first step taken by the server is look up its service definition:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogind: tcp/login: unknown service\n");
        exit(1);
}
```

The result of the *getservbyname* call is used in later portions of the code to define the Internet port at which it listens for service requests (indicated by a connection).

```
main(argc, argv)
	int argc;
	char *argv[];
{
	int f;
	struct sockaddr_in from;
	struct servent *sp;

	sp = getservbyname("login", "tcp");
	if (sp == NULL) {
		fprintf(stderr, "rlogind: tcp/login: unknown service\n");
		exit(1);
	}
	...
#ifndef DEBUG
	/* Disassociate server from controlling terminal */
	...
#endif

	sin.sin_port = sp->s_port;  /* Restricted port -- see section 5 */
	...
	f = socket(AF_INET, SOCK_STREAM, 0);
	...
	if (bind(f, (struct sockaddr *) &sin, sizeof (sin)) < 0) {
		...
	}
	...
	listen(f, 5);
	for (;;) {
		int g, len = sizeof (from);

		g = accept(f, (struct sockaddr *) &from, &len);
		if (g < 0) {
			if (errno != EINTR)
				syslog(LOG_ERR, "rlogind: accept: %m");
			continue;
		}
		if (fork() == 0) {
			close(f);
			doit(g, &from);
		}
		close(g);
	}
}
```

Figure 2. Remote login server.

Step two is to disassociate the server from the controlling terminal of its invoker:

```
for (i = 0; i < 3; ++i)
        close(i);

open("/", O_RDONLY);
dup2(0, 1);
dup2(0, 2);

i = open("/dev/tty", O_RDWR);
if (i >= 0) {
        ioctl(i, TIOCNOTTY, 0);
        close(i);
}
```

This step is important as the server will likely not want to receive signals delivered to the process group of the controlling terminal. Note, however, that once a server has disassociated itself it can no longer send reports of errors to a terminal, and must log errors via *syslog*.

Once a server has established a pristine environment, it creates a socket and begins accepting service requests. The *bind* call is required to insure the server listens at its expected location. It should be noted that the remote login server listens at a restricted port number, and must therefore be run with a user-id of root. This concept of a ‘‘restricted port number’’ is 4BSD specific, and is covered in section 5.

The main body of the loop is fairly simple:

```
for (;;) {
        int g, len = sizeof (from);

        g = accept(f, (struct sockaddr *)&from, &len);
        if (g < 0) {
                if (errno != EINTR)
                        syslog(LOG_ERR, "rlogind: accept: %m");
                continue;
        }
        if (fork() == 0) {  /* Child */
                close(f);
                doit(g, &from);
        }
        close(g);           /* Parent */
}
```

An *accept* call blocks the server until a client requests service. This call could return a failure status if the call is interrupted by a signal such as SIGCHLD (to be discussed in section 5). Therefore, the return value from *accept* is checked to insure a connection has actually been established, and an error report is logged via *syslog* if an error has occurred.

With a connection in hand, the server then forks a child process and invokes the main body of the remote login protocol processing. Note how the socket used by the parent for queuing connection requests is closed in the child, while the socket created as a result of the *accept* is closed in the parent. The address of the client is also handed the *doit* routine because it requires it in authenticating clients.

4.2. Clients

The client side of the remote login service was shown earlier in Figure 1. One can see the separate, asymmetric roles of the client and server clearly in the code. The server is a passive entity, listening for client connections, while the client process is an active entity, initiating a connection when invoked.

Let us consider more closely the steps taken by the client remote login process. As in the server process, the first step is to locate the service definition for a remote login:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogin: tcp/login: unknown service\n");
        exit(1);
}
```

Next the destination host is looked up with a *gethostbyname* call:

```
hp = gethostbyname(argv[1]);
if (hp == NULL) {
        fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
        exit(2);
}
```

With this accomplished, all that is required is to establish a connection to the server at the requested host and start up the remote login protocol. The address buffer is cleared, then filled in with the Internet address of the foreign host and the port number at which the login process resides on the foreign host:

```
bzero((char *)&server, sizeof (server));
bcopy(hp->h_addr, (char *) &server.sin_addr, hp->h_length);
server.sin_family = hp->h_addrtype;
server.sin_port = sp->s_port;
```

A socket is created, and a connection initiated. Note that *connect* implicitly performs a *bind* call, since *s* is unbound.

```
s = socket(hp->h_addrtype, SOCK_STREAM, 0);
if (s < 0) {
        perror("rlogin: socket");
        exit(3);
}
 ...
if (connect(s, (struct sockaddr *) &server, sizeof (server)) < 0) {
        perror("rlogin: connect");
        exit(4);
}
```

The details of the remote login protocol will not be considered here.

4.3. Connectionless servers

While connection-based services are the norm, some services are based on the use of datagram sockets. One, in particular, is the ‘‘rwho’’ service which provides users with status information for hosts connected to a local area network. This service, while predicated on the ability to *broadcast* information to all hosts connected to a particular network, is of interest as an example usage of datagram sockets.

A user on any machine running the rwho server may find out the current status of a machine with the *ruptime*(1) program. The output generated is illustrated in Figure 3.

Status information for each host is periodically broadcast by rwho server processes on each machine. The same server process also receives the status information and uses it to update a database. This database is then interpreted to generate the status information for each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

Note that the use of broadcast for such a task is fairly inefficient, as all hosts must process each message, whether or not using an rwho server. Unless such a service is sufficiently universal and is frequently used, the expense of periodic broadcasts outweighs the simplicity.

Multicasting is an alternative to broadcasting. Setting up multicast sockets is described in Section 5.10.

arpa	up	9:45,	5 users, load	1.15,	1.39,	1.31
cad	up	2+12:04,	8 users, load	4.67,	5.13,	4.59
calder	up	10:10,	0 users, load	0.27,	0.15,	0.14
dali	up	2+06:28,	9 users, load	1.04,	1.20,	1.65
degas	up	25+09:48,	0 users, load	1.49,	1.43,	1.41
ear	up	5+00:05,	0 users, load	1.51,	1.54,	1.56
ernie	down	0:24				
esvax	down	17:04				
ingres	down	0:26				
kim	up	3+09:16,	8 users, load	2.03,	2.46,	3.11
matisse	up	3+06:18,	0 users, load	0.03,	0.03,	0.05
medea	up	3+09:39,	2 users, load	0.35,	0.37,	0.50
merlin	down	19+15:37				
miro	up	1+07:20,	7 users, load	4.59,	3.28,	2.12
monet	up	1+00:43,	2 users, load	0.22,	0.09,	0.07
oz	down	16:09				
statvax	up	2+15:57,	3 users, load	1.52,	1.81,	1.86
ucbvax	up	9:34,	2 users, load	6.08,	5.16,	3.28

Figure 3. ruptime output.

The rwho server, in a simplified form, is pictured in Figure 4. There are two separate tasks performed by the server. The first task is to act as a receiver of status information broadcast by other hosts on the network. This job is carried out in the main loop of the program. Packets received at the rwho port are interrogated to insure they've been sent by another rwho server process, then are time stamped with their arrival time and used to update a file indicating the status of the host. When a host has not been heard from for an extended period of time, the database interpretation routines assume the host is down and indicate such on the status reports. This algorithm is prone to error as a server may be down while a host is actually up, but serves our current needs.

The second task performed by the server is to supply information regarding the status of its host. This involves periodically acquiring system status information, packaging it up in a message and broadcasting it on the local network for other rwho servers to hear. The supply function is triggered by a timer and runs off a signal. Locating the system status information is somewhat involved, but uninteresting. Deciding where to transmit the resultant packet is somewhat problematical, however.

Status information must be broadcast on the local network. For networks which do not support the notion of broadcast another scheme must be used to simulate or replace broadcasting. One possibility is to enumerate the known neighbors (based on the status messages received from other rwho servers). This, unfortunately, requires some bootstrapping information, for a server will have no idea what machines are its neighbors until it receives status messages from them. Therefore, if all machines on a net are freshly booted, no machine will have any known neighbors and thus never receive, or send, any status information. This is the identical problem faced by the routing table management process in propagating routing status information. The standard solution, unsatisfactory as it may be, is to inform one or more servers of known neighbors and request that they always communicate with these neighbors. If each server has at least one neighbor supplied to it, status information may then propagate through a neighbor to hosts which are not (possibly) directly neighbors. If the server is able to support networks which provide a broadcast capability, as well as those which do not, then networks with an arbitrary topology may share status information*.

It is important that software operating in a distributed environment not have any site-dependent information compiled into it. This would require a separate copy of the server at each host and make maintenance a severe headache. 4.4BSD attempts to isolate host-specific information from applications by

* One must, however, be concerned about "loops". That is, if a host is connected to multiple networks, it will receive status information from itself. This can lead to an endless, wasteful, exchange of information.

```
main()
{
        ...
        sp = getservbyname("who", "udp");
        net = getnetbyname("localnet");
        sin.sin_addr = inet_makeaddr(INADDR_ANY, net);
        sin.sin_port = sp->s_port;
        ...
        s = socket(AF_INET, SOCK_DGRAM, 0);
        ...
        on = 1;
        if (setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on)) < 0) {
                syslog(LOG_ERR, "setsockopt SO_BROADCAST: %m");
                exit(1);
        }
        bind(s, (struct sockaddr *) &sin, sizeof (sin));
        ...
        signal(SIGALRM, onalrm);
        onalrm();
        for (;;) {
                struct whod wd;
                int cc, whod, len = sizeof (from);

                cc = recvfrom(s, (char *)&wd, sizeof (struct whod), 0,
                    (struct sockaddr *)&from, &len);
                if (cc <= 0) {
                        if (cc < 0 && errno != EINTR)
                                syslog(LOG_ERR, "rwhod: recv: %m");
                        continue;
                }
                if (from.sin_port != sp->s_port) {
                        syslog(LOG_ERR, "rwhod: %d: bad from port",
                                ntohs(from.sin_port));
                        continue;
                }
                ...
                if (!verify(wd.wd_hostname)) {
                        syslog(LOG_ERR, "rwhod: malformed host name from %x",
                                ntohl(from.sin_addr.s_addr));
                        continue;
                }
                (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
                whod = open(path, O_WRONLY | O_CREAT | O_TRUNC, 0666);
                ...
                (void) time(&wd.wd_recvtime);
                (void) write(whod, (char *)&wd, cc);
                (void) close(whod);
        }
}
```

Figure 4. rwho server.

providing system calls which return the necessary information*. A mechanism exists, in the form of an *ioctl* call, for finding the collection of networks to which a host is directly connected. Further, a local network broadcasting mechanism has been implemented at the socket level. Combining these two features allows a process to broadcast on any directly connected local network which supports the notion of broadcasting in a site independent manner. This allows 4.4BSD to solve the problem of deciding how to propagate status information in the case of *rwho*, or more generally in broadcasting: Such status information is broadcast to connected networks at the socket level, where the connected networks have been obtained via the appropriate *ioctl* calls. The specifics of such broadcastings are complex, however, and will be covered in section 5.

* An example of such a system call is the *gethostname*(2) call which returns the host's ''official'' name.

5. ADVANCED TOPICS

A number of facilities have yet to be discussed. For most users of the IPC the mechanisms already described will suffice in constructing distributed applications. However, others will find the need to utilize some of the features which we consider in this section.

5.1. Out of band data

The stream socket abstraction includes the notion of "out of band" data. Out of band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out of band data is delivered to the user independently of normal data. The abstraction defines that the out of band data facilities must support the reliable delivery of at least one out of band message at a time. This message may contain at least one byte of data, and at least one message may be pending delivery to the user at any one time. For communications protocols which support only in-band signaling (i.e. the urgent data is delivered in sequence with the normal data), the system normally extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving the urgent data in order and receiving it out of sequence without having to buffer all the intervening data. It is possible to "peek" (via MSG_PEEK) at out of band data. If the socket has a process group, a SIGURG signal is generated when the protocol is notified of its existence. A process can set the process group or process id to be informed by the SIGURG signal via the appropriate *fcntl* call, as described below for SIGIO. If multiple sockets may have out of band data awaiting delivery, a *select* call for exceptional conditions may be used to determine those sockets with such data pending. Neither the signal nor the select indicate the actual arrival of the out-of-band data, but only notification that it is pending.

In addition to the information passed, a logical mark is placed in the data stream to indicate the point at which the out of band data was sent. The remote login and remote shell applications use this facility to propagate signals between client and server processes. When a signal flushs any pending output from the remote process(es), all data up to the mark in the data stream is discarded.

To send an out of band message the MSG_OOB flag is supplied to a *send* or *sendto* calls, while to receive out of band data MSG_OOB should be indicated when performing a *recvfrom* or *recv* call. To find out if the read pointer is currently pointing at the mark in the data stream, the SIOCATMARK ioctl is provided:

```
ioctl(s, SIOCATMARK, &yes);
```

If *yes* is a 1 on return, the next read will return data after the mark. Otherwise (assuming out of band data has arrived), the next read will provide data sent by the client prior to transmission of the out of band signal. The routine used in the remote login process to flush output on receipt of an interrupt or quit signal is shown in Figure 5. It reads the normal data up to the mark (to discard it), then reads the out-of-band byte.

A process may also read or peek at the out-of-band data without first reading up to the mark. This is more difficult when the underlying protocol delivers the urgent data in-band with the normal data, and only sends notification of its presence ahead of time (e.g., the TCP protocol used to implement streams in the Internet domain). With such protocols, the out-of-band byte may not yet have arrived when a *recv* is done with the MSG_OOB flag. In that case, the call will return an error of EWOULDBLOCK. Worse, there may be enough in-band data in the input buffer that normal flow control prevents the peer from sending the urgent data until the buffer is cleared. The process must then read enough of the queued data that the urgent data may be delivered.

Certain programs that use multiple bytes of urgent data and must handle multiple urgent signals (e.g., *telnet*(1C)) need to retain the position of urgent data within the stream. This treatment is available as a socket-level option, SO_OOBINLINE; see *setsockopt*(2) for usage. With this option, the position of urgent data (the "mark") is retained, but the urgent data immediately follows the mark within the normal data stream returned without the MSG_OOB flag. Reception of multiple urgent indications causes the mark to move, but no out-of-band data are lost.

```
#include <sys/ioctl.h>
#include <sys/file.h>
 ...
oob()
{
        int out = FWRITE, mark;
        char waste[BUFSIZ];

        /* flush local terminal output */
        ioctl(1, TIOCFLUSH, (char *)&out);
        for (;;) {
                if (ioctl(rem, SIOCATMARK, &mark) < 0) {
                        perror("ioctl");
                        break;
                }
                if (mark)
                        break;
                (void) read(rem, waste, sizeof (waste));
        }
        if (recv(rem, &mark, 1, MSG_OOB) < 0) {
                perror("recv");
                ...
        }
        ...
}
```

Figure 5. Flushing terminal I/O on receipt of out of band data.

5.2. Non-Blocking Sockets

It is occasionally convenient to make use of sockets which do not block; that is, I/O requests which cannot complete immediately and would therefore cause the process to be suspended awaiting completion are not executed, and an error code is returned. Once a socket has been created via the *socket* call, it may be marked as non-blocking by *fcntl* as follows:

```
#include <fcntl.h>
 ...
int     s;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
 ...
if (fcntl(s, F_SETFL, FNDELAY) < 0)
        perror("fcntl F_SETFL, FNDELAY");
        exit(1);
}
 ...
```

When performing non-blocking I/O on sockets, one must be careful to check for the error EWOULDBLOCK (stored in the global variable *errno*), which occurs when an operation would normally block, but the socket it was performed on is marked as non-blocking. In particular, *accept*, *connect*, *send*, *recv*, *read*, and *write* can all return EWOULDBLOCK, and processes should be prepared to deal with such return codes. If an operation such as a *send* cannot be done in its entirety, but partial writes are sensible (for example, when using a stream socket), the data that can be sent immediately will be processed, and the return value will indicate the amount actually sent.

5.3. Interrupt driven socket I/O

The SIGIO signal allows a process to be notified via a signal when a socket (or more generally, a file descriptor) has data waiting to be read. Use of the SIGIO facility requires three steps: First, the process must set up a SIGIO signal handler by use of the *signal* or *sigvec* calls. Second, it must set the process id or process group id which is to receive notification of pending input to its own process id, or the process group id of its process group (note that the default process group of a socket is group zero). This is accomplished by use of an *fcntl* call. Third, it must enable asynchronous notification of pending I/O requests with another *fcntl* call. Sample code to allow a given process to receive information on pending I/O requests as they occur for a socket *s* is given in Figure 6. With the addition of a handler for SIGURG, this code can also be used to prepare for receipt of SIGURG signals.

```
#include <fcntl.h>
 ...
int     io_handler();
 ...
signal(SIGIO, io_handler);

/* Set the process receiving SIGIO/SIGURG signals to us */

if (fcntl(s, F_SETOWN, getpid()) < 0) {
        perror("fcntl F_SETOWN");
        exit(1);
}

/* Allow receipt of asynchronous I/O signals */

if (fcntl(s, F_SETFL, FASYNC) < 0) {
        perror("fcntl F_SETFL, FASYNC");
        exit(1);
}
```

Figure 6. Use of asynchronous notification of I/O requests.

5.4. Signals and process groups

Due to the existence of the SIGURG and SIGIO signals each socket has an associated process number, just as is done for terminals. This value is initialized to zero, but may be redefined at a later time with the F_SETOWN *fcntl*, such as was done in the code above for SIGIO. To set the socket's process id for signals, positive arguments should be given to the *fcntl* call. To set the socket's process group for signals, negative arguments should be passed to *fcntl*. Note that the process number indicates either the associated process id or the associated process group; it is impossible to specify both at the same time. A similar *fcntl*, F_GETOWN, is available for determining the current process number of a socket.

Another signal which is useful when constructing server processes is SIGCHLD. This signal is delivered to a process when any child processes have changed state. Normally servers use the signal to ‘‘reap’’ child processes that have exited without explicitly awaiting their termination or periodic polling for exit status. For example, the remote login server loop shown in Figure 2 may be augmented as shown in Figure 7.

If the parent server process fails to reap its children, a large number of ‘‘zombie’’ processes may be created.

5.5. Pseudo terminals

Many programs will not function properly without a terminal for standard input and output. Since sockets do not provide the semantics of terminals, it is often necessary to have a process communicating over the network do so through a *pseudo-terminal*. A pseudo- terminal is actually a pair of devices, master

```
int reaper();
 ...
signal(SIGCHLD, reaper);
listen(f, 5);
for (;;) {
        int g, len = sizeof (from);

        g = accept(f, (struct sockaddr *)&from, &len,);
        if (g < 0) {
                if (errno != EINTR)
                        syslog(LOG_ERR, "rlogind: accept: %m");
                continue;
        }
        ...
}
 ...
#include <wait.h>
reaper()
{
        union wait status;

        while (wait3(&status, WNOHANG, 0) > 0)
                ;
}
```

Figure 7. Use of the SIGCHLD signal.

and slave, which allow a process to serve as an active agent in communication between processes and users. Data written on the slave side of a pseudo-terminal is supplied as input to a process reading from the master side, while data written on the master side are processed as terminal input for the slave. In this way, the process manipulating the master side of the pseudo-terminal has control over the information read and written on the slave side as if it were manipulating the keyboard and reading the screen on a real terminal. The purpose of this abstraction is to preserve terminal semantics over a network connection— that is, the slave side appears as a normal terminal to any process reading from or writing to it.

For example, the remote login server uses pseudo-terminals for remote login sessions. A user logging in to a machine across the network is provided a shell with a slave pseudo-terminal as standard input, output, and error. The server process then handles the communication between the programs invoked by the remote shell and the user's local client process. When a user sends a character that generates an interrupt on the remote machine that flushes terminal output, the pseudo-terminal generates a control message for the server process. The server then sends an out of band message to the client process to signal a flush of data at the real terminal and on the intervening data buffered in the network.

Under 4.4BSD, the name of the slave side of a pseudo-terminal is of the form */dev/ttyxy*, where *x* is a single letter starting at 'p' and continuing to 't'. *y* is a hexadecimal digit (i.e., a single character in the range 0 through 9 or 'a' through 'f'). The master side of a pseudo-terminal is */dev/ptyxy*, where *x* and *y* correspond to the slave side of the pseudo-terminal.

In general, the method of obtaining a pair of master and slave pseudo-terminals is to find a pseudo-terminal which is not currently in use. The master half of a pseudo-terminal is a single-open device; thus, each master may be opened in turn until an open succeeds. The slave side of the pseudo-terminal is then opened, and is set to the proper terminal modes if necessary. The process then *fork*s; the child closes the master side of the pseudo-terminal, and *exec*s the appropriate program. Meanwhile, the parent closes the slave side of the pseudo-terminal and begins reading and writing from the master side. Sample code making use of pseudo-terminals is given in Figure 8; this code assumes that a connection on a socket *s* exists, connected to a peer who wants a service of some kind, and that the process has disassociated itself from any previous controlling terminal.

```
gotpty = 0;
for (c = 'p'; !gotpty && c <= 's'; c++) {
        line = "/dev/ptyXX";
        line[sizeof("/dev/pty")-1] = c;
        line[sizeof("/dev/ptyp")-1] = '0';
        if (stat(line, &statbuf) < 0)
                break;
        for (i = 0; i < 16; i++) {
                line[sizeof("/dev/ptyp")-1] = "0123456789abcdef"[i];
                master = open(line, O_RDWR);
                if (master > 0) {
                        gotpty = 1;
                        break;
                }
        }
}
if (!gotpty) {
        syslog(LOG_ERR, "All network ports in use");
        exit(1);
}

line[sizeof("/dev/")-1] = 't';
slave = open(line, O_RDWR); /* slave is now slave side */
if (slave < 0) {
        syslog(LOG_ERR, "Cannot open slave pty %s", line);
        exit(1);
}

ioctl(slave, TIOCGETP, &b);   /* Set slave tty modes */
b.sg_flags = CRMOD|XTABS|ANYP;
ioctl(slave, TIOCSETP, &b);

i = fork();
if (i < 0) {
        syslog(LOG_ERR, "fork: %m");
        exit(1);
} else if (i) {                 /* Parent */
        close(slave);
        ...
} else {                 /* Child */
        (void) close(s);
        (void) close(master);
        dup2(slave, 0);
        dup2(slave, 1);
        dup2(slave, 2);
        if (slave > 2)
                (void) close(slave);
        ...
}
```

Figure 8. Creation and use of a pseudo terminal

5.6. Selecting specific protocols

If the third argument to the *socket* call is 0, *socket* will select a default protocol to use with the returned socket of the type requested. The default protocol is usually correct, and alternate choices are not usually available. However, when using "raw" sockets to communicate directly with lower-level protocols or hardware interfaces, the protocol argument may be important for setting up demultiplexing. For example, raw sockets in the Internet family may be used to implement a new protocol above IP, and the socket will receive packets only for the protocol specified. To obtain a particular protocol one determines the protocol number as defined within the communication domain. For the Internet domain one may use one of the library routines discussed in section 3, such as *getprotobyname*:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
 ...
pp = getprotobyname("newtcp");
s = socket(AF_INET, SOCK_STREAM, pp->p_proto);
```

This would result in a socket *s* using a stream based connection, but with protocol type of "newtcp" instead of the default "tcp."

In the NS domain, the available socket protocols are defined in <*netns/ns.h*>. To create a raw socket for Xerox Error Protocol messages, one might use:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
 ...
s = socket(AF_NS, SOCK_RAW, NSPROTO_ERROR);
```

5.7. Address binding

As was mentioned in section 2, binding addresses to sockets in the Internet and NS domains can be fairly complex. As a brief reminder, these associations are composed of local and foreign addresses, and local and foreign ports. Port numbers are allocated out of separate spaces, one for each system and one for each domain on that system. Through the *bind* system call, a process may specify half of an association, the <local address, local port> part, while the *connect* and *accept* primitives are used to complete a socket's association by specifying the <foreign address, foreign port> part. Since the association is created in two steps the association uniqueness requirement indicated previously could be violated unless care is taken. Further, it is unrealistic to expect user programs to always know proper values to use for the local address and local port since a host may reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

To simplify local address binding in the Internet domain the notion of a "wildcard" address has been provided. When an address is specified as INADDR_ANY (a manifest constant defined in <netinet/in.h>), the system interprets the address as "any valid address". For example, to bind a specific port number to a socket, but leave the local address unspecified, the following code might be used:

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

Sockets with wildcarded local addresses may receive messages directed to the specified port number, and sent to any of the possible addresses assigned to a host. For example, if a host has addresses 128.32.0.4 and 10.0.0.78, and a socket is bound as above, the process will be able to accept connection requests which are addressed to 128.32.0.4 or 10.0.0.78. If a server process wished to only allow hosts on a given network connect to it, it would bind the address of the host on the appropriate network.

In a similar fashion, a local port may be left unspecified (specified as zero), in which case the system will select an appropriate port number for it. This shortcut will work both in the Internet and NS domains. For example, to bind a specific local address to a socket, but to leave the local port number unspecified:

```
hp = gethostbyname(hostname);
if (hp == NULL) {
        ...
}
bcopy(hp->h_addr, (char *) sin.sin_addr, hp->h_length);
sin.sin_port = htons(0);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

The system selects the local port number based on two criteria. The first is that on 4BSD systems, Internet ports below IPPORT_RESERVED (1024) (for the Xerox domain, 0 through 3000) are reserved for privileged users (i.e., the super user); Internet ports above IPPORT_USERRESERVED (50000) are reserved for non-privileged servers. The second is that the port number is not currently bound to some other socket. In order to find a free Internet port number in the privileged range the *rresvport* library routine may be used as follows to return a stream socket in with a privileged port number:

```
int lport = IPPORT_RESERVED − 1;
int s;
...
s = rresvport(&lport);
if (s < 0) {
        if (errno == EAGAIN)
                fprintf(stderr, "socket: all ports in use\n");
        else
                perror("rresvport: socket");
        ...
}
```

The restriction on allocating ports was done to allow processes executing in a ‘‘secure’’ environment to perform authentication based on the originating address and port number. For example, the *rlogin*(1) command allows users to log in across a network without being asked for a password, if two conditions hold: First, the name of the system the user is logging in from is in the file */etc/hosts.equiv* on the system he is logging in to (or the system name and the user name are in the user’s *.rhosts* file in the user’s home directory), and second, that the user’s rlogin process is coming from a privileged port on the machine from which he is logging. The port number and network address of the machine from which the user is logging in can be determined either by the *from* result of the *accept* call, or from the *getpeername* call.

In certain cases the algorithm used by the system in selecting port numbers is unsuitable for an application. This is because associations are created in a two step process. For example, the Internet file transfer protocol, FTP, specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. In this situation the system would disallow binding the same local address and port number to a socket if a previous data connection's socket still existed. To override the default port selection algorithm, an option call must be performed prior to address binding:

```
 ...
int     on = 1;
 ...
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

With the above call, local addresses may be bound which are already in use. This does not violate the uniqueness requirement as the system still checks at connect time to be sure any other sockets with the same local address and port do not have the same foreign address and port. If the association already exists, the error EADDRINUSE is returned. A related socket option, SO_REUSEPORT, which allows completely duplicate bindings, is described in the IP multicasting section.

5.8. Socket Options

It is possible to set and get a number of options on sockets via the *setsockopt* and *getsockopt* system calls. These options include such things as marking a socket for broadcasting, not to route, to linger on close, etc. In addition, there are protocol-specific options for IP and TCP, as described in *ip*(4), *tcp*(4), and in the section on multicasting below.

The general forms of the calls are:

```
setsockopt(s, level, optname, optval, optlen);
```

and

```
getsockopt(s, level, optname, optval, optlen);
```

The parameters to the calls are as follows: *s* is the socket on which the option is to be applied. *Level* specifies the protocol layer on which the option is to be applied; in most cases this is the ‘‘socket level’’, indicated by the symbolic constant SOL_SOCKET, defined in *<sys/socket.h>*. The actual option is specified in *optname*, and is a symbolic constant also defined in *<sys/socket.h>*. *Optval* and *Optlen* point to the value of the option (in most cases, whether the option is to be turned on or off), and the length of the value of the option, respectively. For *getsockopt*, *optlen* is a value-result parameter, initially set to the size of the storage area pointed to by *optval*, and modified upon return to indicate the actual amount of storage used.

An example should help clarify things. It is sometimes useful to determine the type (e.g., stream, datagram, etc.) of an existing socket; programs under *inetd* (described below) may need to perform this task. This can be accomplished as follows via the SO_TYPE socket option and the *getsockopt* call:

```
#include <sys/types.h>
#include <sys/socket.h>

int type, size;

size = sizeof (int);

if (getsockopt(s, SOL_SOCKET, SO_TYPE, (char *) &type, &size) < 0) {
        ...
}
```

After the *getsockopt* call, *type* will be set to the value of the socket type, as defined in *<sys/socket.h>*. If, for example, the socket were a datagram socket, *type* would have the value corresponding to

SOCK_DGRAM.

5.9. Broadcasting and determining network configuration

By using a datagram socket, it is possible to send broadcast packets on many networks supported by the system. The network itself must support broadcast; the system provides no simulation of broadcast in software. Broadcast messages can place a high load on a network since they force every host on the network to service them. Consequently, the ability to send broadcast packets has been limited to sockets which are explicitly marked as allowing broadcasting. Broadcast is typically used for one of two reasons: it is desired to find a resource on a local network without prior knowledge of its address, or important functions such as routing require that information be sent to all accessible neighbors.

Multicasting is an alternative to broadcasting. Setting up IP multicast sockets is described in the next section.

To send a broadcast message, a datagram socket should be created:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

or

```
s = socket(AF_NS, SOCK_DGRAM, 0);
```

The socket is marked as allowing broadcasting,

```
int     on = 1;

setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof (on));
```

and at least a port number should be bound to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

or, for the NS domain,

```
sns.sns_family = AF_NS;
netnum = htonl(net);
sns.sns_addr.x_net = *(union ns_net *) &netnum; /* insert net number */
sns.sns_addr.x_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sns, sizeof (sns));
```

The destination address of the message to be broadcast depends on the network(s) on which the message is to be broadcast. The Internet domain supports a shorthand notation for broadcast on the local network, the address INADDR_BROADCAST (defined in <*netinet/in.h*>. To determine the list of addresses for all reachable neighbors requires knowledge of the networks to which the host is connected. Since this information should be obtained in a host-independent fashion and may be impossible to derive, 4.4BSD provides a method of retrieving this information from the system data structures. The SIOCGIFCONF *ioctl* call returns the interface configuration of a host in the form of a single *ifconf* structure; this structure contains a ''data area'' which is made up of an array of of *ifreq* structures, one for each network interface to which the host is connected. These structures are defined in <*net/if.h*> as follows:

```
struct ifconf {
        int       ifc_len;                      /* size of associated buffer */
        union {
                caddr_t  ifcu_buf;
                struct   ifreq *ifcu_req;
        } ifc_ifcu;
};

#define  ifc_buf   ifc_ifcu.ifcu_buf            /* buffer address */
#define  ifc_req   ifc_ifcu.ifcu_req            /* array of structures returned */

#define  IFNAMSIZ       16

struct ifreq {
        char      ifr_name[IFNAMSIZ];           /* if name, e.g. "en0" */
        union {
                struct   sockaddr ifru_addr;
                struct   sockaddr ifru_dstaddr;
                struct   sockaddr ifru_broadaddr;
                short    ifru_flags;
                caddr_t  ifru_data;
        } ifr_ifru;
};

#define  ifr_addr        ifr_ifru.ifru_addr        /* address */
#define  ifr_dstaddr     ifr_ifru.ifru_dstaddr     /* other end of p-to-p link */
#define  ifr_broadaddr   ifr_ifru.ifru_broadaddr   /* broadcast address */
#define  ifr_flags       ifr_ifru.ifru_flags       /* flags */
#define  ifr_data        ifr_ifru.ifru_data        /* for use by interface */
```

The actual call which obtains the interface configuration is

```
struct ifconf ifc;
char buf[BUFSIZ];

ifc.ifc_len = sizeof (buf);
ifc.ifc_buf = buf;
if (ioctl(s, SIOCGIFCONF, (char *) &ifc) < 0) {
        ...
}
```

After this call *buf* will contain one *ifreq* structure for each network to which the host is connected, and *ifc.ifc_len* will have been modified to reflect the number of bytes used by the *ifreq* structures.

For each structure there exists a set of ‘‘interface flags’’ which tell whether the network corresponding to that interface is up or down, point to point or broadcast, etc. The SIOCGIFFLAGS *ioctl* retrieves these flags for an interface specified by an *ifreq* structure as follows:

```
struct ifreq *ifr;

ifr = ifc.ifc_req;

for (n = ifc.ifc_len / sizeof (struct ifreq); --n >= 0; ifr++) {
        /*
         * We must be careful that we don't use an interface
         * devoted to an address family other than those intended;
         * if we were interested in NS interfaces, the
         * AF_INET would be AF_NS.
         */
        if (ifr->ifr_addr.sa_family != AF_INET)
                continue;
        if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
                ...
        }
        /*
         * Skip boring cases.
         */
        if ((ifr->ifr_flags & IFF_UP) == 0 ||
            (ifr->ifr_flags & IFF_LOOPBACK) ||
            (ifr->ifr_flags & (IFF_BROADCAST | IFF_POINTTOPOINT)) == 0)
                continue;
```

Once the flags have been obtained, the broadcast address must be obtained. In the case of broadcast networks this is done via the SIOCGIFBRDADDR *ioctl*, while for point-to-point networks the address of the destination host is obtained with SIOCGIFDSTADDR.

```
struct sockaddr dst;

if (ifr->ifr_flags & IFF_POINTTOPOINT) {
        if (ioctl(s, SIOCGIFDSTADDR, (char *) ifr) < 0) {
                ...
        }
        bcopy((char *) ifr->ifr_dstaddr, (char *) &dst, sizeof (ifr->ifr_dstaddr));
} else if (ifr->ifr_flags & IFF_BROADCAST) {
        if (ioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
                ...
        }
        bcopy((char *) ifr->ifr_broadaddr, (char *) &dst, sizeof (ifr->ifr_broadaddr));
}
```

After the appropriate *ioctl*'s have obtained the broadcast or destination address (now in *dst*), the *sendto* call may be used:

```
        sendto(s, buf, buflen, 0, (struct sockaddr *)&dst, sizeof (dst));
}
```

In the above loop one *sendto* occurs for every interface to which the host is connected that supports the notion of broadcast or point-to-point addressing. If a process only wished to send broadcast messages on a given network, code similar to that outlined above would be used, but the loop would need to find the correct destination address.

Received broadcast messages contain the senders address and port, as datagram sockets are bound before a message is allowed to go out.

5.10. IP Multicasting

IP multicasting is the transmission of an IP datagram to a "host group", a set of zero or more hosts identified by a single IP destination address. A multicast datagram is delivered to all members of its destination host group with the same "best-efforts" reliability as regular unicast IP datagrams, i.e., the datagram is not guaranteed to arrive intact at all members of the destination group or in the same order relative to other datagrams.

The membership of a host group is dynamic; that is, hosts may join and leave groups at any time. There is no restriction on the location or number of members in a host group. A host may be a member of more than one group at a time. A host need not be a member of a group to send datagrams to it.

A host group may be permanent or transient. A permanent group has a well-known, administratively assigned IP address. It is the address, not the membership of the group, that is permanent; at any time a permanent group may have any number of members, even zero. Those IP multicast addresses that are not reserved for permanent groups are available for dynamic assignment to transient groups which exist only as long as they have members.

In general, a host cannot assume that datagrams sent to any host group address will reach only the intended hosts, or that datagrams received as a member of a transient host group are intended for the recipient. Misdelivery must be detected at a level above IP, using higher-level identifiers or authentication tokens. Information transmitted to a host group address should be encrypted or governed by administrative routing controls if the sender is concerned about unwanted listeners.

IP multicasting is currently supported only on AF_INET sockets of type SOCK_DGRAM and SOCK_RAW, and only on subnetworks for which the interface driver has been modified to support multicasting.

The next subsections describe how to send and receive multicast datagrams.

5.10.1. Sending IP Multicast Datagrams

To send a multicast datagram, specify an IP multicast address in the range 224.0.0.0 to 239.255.255.255 as the destination address in a *sendto*(2) call.

The definitions required for the multicast-related socket options are found in *<netinet/in.h>*. All IP addresses are passed in network byte-order.

By default, IP multicast datagrams are sent with a time-to-live (TTL) of 1, which prevents them from being forwarded beyond a single subnetwork. A new socket option allows the TTL for subsequent multicast datagrams to be set to any value from 0 to 255, in order to control the scope of the multicasts:

```
u_char ttl;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, &ttl, sizeof(ttl));
```

Multicast datagrams with a TTL of 0 will not be transmitted on any subnet, but may be delivered locally if the sending host belongs to the destination group and if multicast loopback has not been disabled on the sending socket (see below). Multicast datagrams with TTL greater than one may be delivered to more than one subnet if there are one or more multicast routers attached to the first-hop subnet. To provide meaningful scope control, the multicast routers support the notion of TTL "thresholds", which prevent datagrams with less than a certain TTL from traversing certain subnets. The thresholds enforce the following convention:

Scope	Initial TTL
restricted to the same host	0
restricted to the same subnet	1
restricted to the same site	32
restricted to the same region	64
restricted to the same continent	128
unrestricted	255

"Sites" and "regions" are not strictly defined, and sites may be further subdivided into smaller administrative units, as a local matter.

An application may choose an initial TTL other than the ones listed above. For example, an application might perform an "expanding-ring search" for a network resource by sending a multicast query, first with a TTL of 0, and then with larger and larger TTLs, until a reply is received, perhaps using the TTL sequence 0, 1, 2, 4, 8, 16, 32.

The multicast router *mrouted*(8), refuses to forward any multicast datagram with a destination address between 224.0.0.0 and 224.0.0.255, inclusive, regardless of its TTL. This range of addresses is reserved for the use of routing protocols and other low-level topology discovery or maintenance protocols, such as gateway discovery and group membership reporting.

The address 224.0.0.0 is guaranteed not to be assigned to any group, and 224.0.0.1 is assigned to the permanent group of all IP hosts (including gateways). This is used to address all multicast hosts on the directly connected network. There is no multicast address (or any other IP address) for all hosts on the total Internet. The addresses of other well-known, permanent groups are published in the "Assigned Numbers" RFC, which is available from the InterNIC.

Each multicast transmission is sent from a single network interface, even if the host has more than one multicast-capable interface. (If the host is also serving as a multicast router, a multicast may be *forwarded* to interfaces other than originating interface, provided that the TTL is greater than 1.) The default interface to be used for multicasting is the primary network interface on the system. A socket option is available to override the default for subsequent transmissions from a given socket:

```
struct in_addr addr;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_IF, &addr, sizeof(addr));
```

where "addr" is the local IP address of the desired outgoing interface. An address of INADDR_ANY may be used to revert to the default interface. The local IP address of an interface can be obtained via the SIOCGIFCONF ioctl. To determine if an interface supports multicasting, fetch the interface flags via the SIOCGIFFLAGS ioctl and see if the IFF_MULTICAST flag is set. (Normal applications should not need to use this option; it is intended primarily for multicast routers and other system services specifically concerned with internet topology.) The SIOCGIFCONF and SIOCGIFFLAGS ioctls are described in the previous section.

If a multicast datagram is sent to a group to which the sending host itself belongs (on the outgoing interface), a copy of the datagram is, by default, looped back by the IP layer for local delivery. Another socket option gives the sender explicit control over whether or not subsequent datagrams are looped back:

```
u_char loop;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, sizeof(loop));
```

where *loop* is set to 0 to disable loopback, and set to 1 to enable loopback. This option improves performance for applications that may have no more than one instance on a single host (such as a router demon), by eliminating the overhead of receiving their own transmissions. It should generally not be used by applications for which there may be more than one instance on a single host (such as a conferencing program) or for which the sender does not belong to the destination group (such as a time querying program).

A multicast datagram sent with an initial TTL greater than 1 may be delivered to the sending host on a different interface from that on which it was sent, if the host belongs to the destination group on that other interface. The loopback control option has no effect on such delivery.

5.10.2. Receiving IP Multicast Datagrams

Before a host can receive IP multicast datagrams, it must become a member of one or more IP multicast groups. A process can ask the host to join a multicast group by using the following socket option:

```
struct ip_mreq mreq;
setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq))
```

where "mreq" is the following structure:

```
struct ip_mreq {
    struct in_addr imr_multiaddr; /* multicast group to join */
    struct in_addr imr_interface; /* interface to join on */
}
```

Every membership is associated with a single interface, and it is possible to join the same group on more than one interface. "imr_interface" should be INADDR_ANY to choose the default multicast interface, or one of the host's local addresses to choose a particular (multicast-capable) interface. Up to IP_MAX_MEMBERSHIPS (currently 20) memberships may be added on a single socket.

To drop a membership, use:

```
struct ip_mreq mreq;
setsockopt(sock, IPPROTO_IP, IP_DROP_MEMBERSHIP, &mreq, sizeof(mreq));
```

where "mreq" contains the same values as used to add the membership. The memberships associated with a socket are also dropped when the socket is closed or the process holding the socket is killed. However, more than one socket may claim a membership in a particular group, and the host will remain a member of that group until the last claim is dropped.

The memberships associated with a socket do not necessarily determine which datagrams are received on that socket. Incoming multicast packets are accepted by the kernel IP layer if any socket has claimed a membership in the destination group of the datagram; however, delivery of a multicast datagram to a particular socket is based on the destination port (or protocol type, for raw sockets), just as with unicast datagrams. To receive multicast datagrams sent to a particular port, it is necessary to bind to that local port, leaving the local address unspecified (i.e., INADDR_ANY). To receive multicast datagrams sent to a particular group and port, bind to the local port, with the local address set to the multicast group address. Once bound to a multicast address, the socket cannot be used for sending data.

More than one process may bind to the same SOCK_DGRAM UDP port or the same multicast group and port if the *bind* call is preceded by:

```
int on = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEPORT, &on, sizeof(on));
```

All processes sharing the port must enable this option. Every incoming multicast or broadcast UDP datagram destined to the shared port is delivered to all sockets bound to the port. For backwards compatibility reasons, this does not apply to incoming unicast datagrams. Unicast datagrams are never delivered to more than one socket, regardless of how many sockets are bound to the datagram's destination port.

A final multicast-related extension is independent of IP: two new ioctls, SIOCADDMULTI and SIOCDELMULTI, are available to add or delete link-level (e.g., Ethernet) multicast addresses accepted by a particular interface. The address to be added or deleted is passed as a sockaddr structure of family AF_UNSPEC, within the standard ifreq structure.

These ioctls are for the use of protocols other than IP, and require superuser privileges. A link-level multicast address added via SIOCADDMULTI is not automatically deleted when the socket used to add it goes away; it must be explicitly deleted. It is inadvisable to delete a link-level address that may be in use by IP.

5.10.3. Sample Multicast Program

The following program sends or receives multicast packets. If invoked with one argument, it sends a packet containing the current time to an arbitrarily-chosen multicast group and UDP port. If invoked with no arguments, it receives and prints these packets. Start it as a sender on just one host and as a receiver on all the other hosts.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <time.h>
#include <stdio.h>

#define EXAMPLE_PORT    60123
#define EXAMPLE_GROUP   "224.0.0.250"

main(argc)
    int argc;
{
    struct sockaddr_in addr;
    int addrlen, fd, cnt;
    struct ip_mreq mreq;
    char message[50];

    fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (fd < 0) {
        perror("socket");
        exit(1);
    }

    bzero(&addr, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    addr.sin_port = htons(EXAMPLE_PORT);
    addrlen = sizeof(addr);

    if (argc > 1) {     /* Send */
        addr.sin_addr.s_addr = inet_addr(EXAMPLE_GROUP);
        while (1) {
            time_t t = time(0);
            sprintf(message, "time is %-24.24s", ctime(&t));
            cnt = sendto(fd, message, sizeof(message), 0,
                    (struct sockaddr *)&addr, addrlen);
            if (cnt < 0) {
                perror("sendto");
                exit(1);
            }
            sleep(5);
        }
    } else {            /* Receive */
        if (bind(fd, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
            perror("bind");
            exit(1);
        }

        mreq.imr_multiaddr.s_addr = inet_addr(EXAMPLE_GROUP);
        mreq.imr_interface.s_addr = htonl(INADDR_ANY);
        if (setsockopt(fd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                &mreq, sizeof(mreq)) < 0) {
            perror("setsockopt mreq");
```

```
                exit(1);
            }

            while (1) {
                cnt = recvfrom(fd, message, sizeof(message), 0,
                        (struct sockaddr *)&addr, &addrlen);
                if (cnt <= 0) {
                        if (cnt == 0) {
                            break;
                        }
                    perror("recvfrom");
                    exit(1);
                }
                printf("%s: message = \"%s\"\n",
                    inet_ntoa(addr.sin_addr), message);
            }
        }
    }
```

5.11. NS Packet Sequences

The semantics of NS connections demand that the user both be able to look inside the network header associated with any incoming packet and be able to specify what should go in certain fields of an outgoing packet. Using different calls to *setsockopt*, it is possible to indicate whether prototype headers will be associated by the user with each outgoing packet (SO_HEADERS_ON_OUTPUT), to indicate whether the headers received by the system should be delivered to the user (SO_HEADERS_ON_INPUT), or to indicate default information that should be associated with all outgoing packets on a given socket (SO_DEFAULT_HEADERS).

The contents of a SPP header (minus the IDP header) are:

```
struct sphdr {
        u_char   sp_cc;          /* connection control */
#define  SP_SP   0x80            /* system packet */
#define  SP_SA   0x40            /* send acknowledgement */
#define  SP_OB   0x20            /* attention (out of band data) */
#define  SP_EM   0x10            /* end of message */
        u_char   sp_dt;          /* datastream type */
        u_short  sp_sid;         /* source connection identifier */
        u_short  sp_did;         /* destination connection identifier */
        u_short  sp_seq;         /* sequence number */
        u_short  sp_ack;         /* acknowledge number */
        u_short  sp_alo;         /* allocation number */
};
```

Here, the items of interest are the *datastream type* and the *connection control* fields. The semantics of the datastream type are defined by the application(s) in question; the value of this field is, by default, zero, but it can be used to indicate things such as Xerox's Bulk Data Transfer Protocol (in which case it is set to one). The connection control field is a mask of the flags defined just below it. The user may set or clear the end-of-message bit to indicate that a given message is the last of a given substream type, or may set/clear the attention bit as an alternate way to indicate that a packet should be sent out-of-band. As an example, to associate prototype headers with outgoing SPP packets, consider:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/sp.h>
 ...
struct sockaddr_ns sns, to;
int s, on = 1;
struct databuf {
        struct sphdr proto_spp;   /* prototype header */
        char buf[534];            /* max. possible data by Xerox std. */
} buf;
 ...
s = socket(AF_NS, SOCK_SEQPACKET, 0);
 ...
bind(s, (struct sockaddr *) &sns, sizeof (sns));
setsockopt(s, NSPROTO_SPP, SO_HEADERS_ON_OUTPUT, &on, sizeof(on));
 ...
buf.proto_spp.sp_dt = 1; /* bulk data */
buf.proto_spp.sp_cc = SP_EM;        /* end-of-message */
strcpy(buf.buf, "hello world\n");
sendto(s, (char *) &buf, sizeof(struct sphdr) + strlen("hello world\n"),
   (struct sockaddr *) &to, sizeof(to));
 ...
```

Note that one must be careful when writing headers; if the prototype header is not written with the data with which it is to be associated, the kernel will treat the first few bytes of the data as the header, with unpredictable results. To turn off the above association, and to indicate that packet headers received by the system should be passed up to the user, one might use:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/sp.h>
 ...
struct sockaddr sns;
int s, on = 1, off = 0;
 ...
s = socket(AF_NS, SOCK_SEQPACKET, 0);
 ...
bind(s, (struct sockaddr *) &sns, sizeof (sns));
setsockopt(s, NSPROTO_SPP, SO_HEADERS_ON_OUTPUT, &off, sizeof(off));
setsockopt(s, NSPROTO_SPP, SO_HEADERS_ON_INPUT, &on, sizeof(on));
 ...
```

Output is handled somewhat differently in the IDP world. The header of an IDP-level packet looks like:

```
struct idp {
        u_short         idp_sum;        /* Checksum */
        u_short         idp_len;        /* Length, in bytes, including header */
        u_char          idp_tc;         /* Transport Control (i.e., hop count) */
        u_char          idp_pt;         /* Packet Type (i.e., level 2 protocol) */
        struct ns_addr  idp_dna;        /* Destination Network Address */
        struct ns_addr  idp_sna;        /* Source Network Address */
};
```

The primary field of interest in an IDP header is the *packet type* field. The standard values for this field are (as defined in *<netns/ns.h>*):

```
#define NSPROTO_RI       1        /* Routing Information */
#define NSPROTO_ECHO     2        /* Echo Protocol */
#define NSPROTO_ERROR    3        /* Error Protocol */
#define NSPROTO_PE       4        /* Packet Exchange */
#define NSPROTO_SPP      5        /* Sequenced Packet */
```

For SPP connections, the contents of this field are automatically set to NSPROTO_SPP; for IDP packets, this value defaults to zero, which means ‘‘unknown’’.

Setting the value of that field with SO_DEFAULT_HEADERS is easy:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/idp.h>
 ...
struct sockaddr sns;
struct idp proto_idp;          /* prototype header */
int s, on = 1;
 ...
s = socket(AF_NS, SOCK_DGRAM, 0);
 ...
bind(s, (struct sockaddr *) &sns, sizeof (sns));
proto_idp.idp_pt = NSPROTO_PE;   /* packet exchange */
setsockopt(s, NSPROTO_IDP, SO_DEFAULT_HEADERS, (char *) &proto_idp,
   sizeof(proto_idp));
 ...
```

Using SO_HEADERS_ON_OUTPUT is somewhat more difficult. When SO_HEADERS_ON_OUTPUT is turned on for an IDP socket, the socket becomes (for all intents and purposes) a raw socket. In this case, all the fields of the prototype header (except the length and checksum fields, which are computed by the kernel) must be filled in correctly in order for the socket to send and receive data in a sensible manner. To be more specific, the source address must be set to that of the host sending the data; the destination address must be set to that of the host for whom the data is intended; the packet type must be set to whatever value is desired; and the hopcount must be set to some reasonable value (almost always zero). It should also be noted that simply sending data using *write* will not work unless a *connect* or *sendto* call is used, in spite of the fact that it is the destination address in the prototype header that is used, not the one given in either of those calls. For almost all IDP applications , using SO_DEFAULT_HEADERS is easier and more desirable than writing headers.

5.12. Three-way Handshake

The semantics of SPP connections indicates that a three-way handshake, involving changes in the datastream type, should — but is not absolutely required to — take place before a SPP connection is closed. Almost all SPP connections are ‘‘well-behaved’’ in this manner; when communicating with any process, it is best to assume that the three-way handshake is required unless it is known for certain that it is not required. In a three-way close, the closing process indicates that it wishes to close the connection by sending a zero-length packet with end-of-message set and with datastream type 254. The other side of the connection indicates that it is OK to close by sending a zero-length packet with end-of-message set and datastream type 255. Finally, the closing process replies with a zero-length packet with substream type 255; at this point, the connection is considered closed. The following code fragments are simplified examples of how one might handle this three-way handshake at the user level; in the future, support for this type of close will probably be provided as part of the C library or as part of the kernel. The first code fragment below illustrates how a process might handle three-way handshake if it sees that the process it is communicating with wants to close the connection:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/sp.h>
 ...
#ifndef SPPSST_END
#define SPPSST_END 254
#define SPPSST_ENDREPLY 255
#endif
struct sphdr proto_sp;
int s;
 ...
read(s, buf, BUFSIZE);
if (((struct sphdr *)buf)->sp_dt == SPPSST_END) {
	/*
	 * SPPSST_END indicates that the other side wants to
	 * close.
	 */
	proto_sp.sp_dt = SPPSST_ENDREPLY;
	proto_sp.sp_cc = SP_EM;
	setsockopt(s, NSPROTO_SPP, SO_DEFAULT_HEADERS, (char *)&proto_sp,
	    sizeof(proto_sp));
	write(s, buf, 0);
	/*
	 * Write a zero-length packet with datastream type = SPPSST_ENDREPLY
	 * to indicate that the close is OK with us.  The packet that we
	 * don't see (because we don't look for it) is another packet
	 * from the other side of the connection, with SPPSST_ENDREPLY
	 * on it it, too.  Once that packet is sent, the connection is
	 * considered closed; note that we really ought to retransmit
	 * the close for some time if we do not get a reply.
	 */
	close(s);
}
 ...
```

To indicate to another process that we would like to close the connection, the following code would suffice:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/sp.h>
 ...
#ifndef SPPSST_END
#define SPPSST_END 254
#define SPPSST_ENDREPLY 255
#endif
struct sphdr proto_sp;
int s;
 ...
proto_sp.sp_dt = SPPSST_END;
proto_sp.sp_cc = SP_EM;
setsockopt(s, NSPROTO_SPP, SO_DEFAULT_HEADERS, (char *)&proto_sp,
   sizeof(proto_sp));
write(s, buf, 0);     /* send the end request */
proto_sp.sp_dt = SPPSST_ENDREPLY;
setsockopt(s, NSPROTO_SPP, SO_DEFAULT_HEADERS, (char *)&proto_sp,
   sizeof(proto_sp));
/*
 * We assume (perhaps unwisely)
 * that the other side will send the
 * ENDREPLY, so we'll just send our final ENDREPLY
 * as if we'd seen theirs already.
 */
write(s, buf, 0);
close(s);
 ...
```

5.13. Packet Exchange

The Xerox standard protocols include a protocol that is both reliable and datagram-oriented. This protocol is known as Packet Exchange (PEX or PE) and, like SPP, is layered on top of IDP. PEX is important for a number of things: Courier remote procedure calls may be expedited through the use of PEX, and many Xerox servers are located by doing a PEX ‘‘BroadcastForServers’’ operation. Although there is no implementation of PEX in the kernel, it may be simulated at the user level with some clever coding and the use of one peculiar *getsockopt*. A PEX packet looks like:

```
/*
 * The packet-exchange header shown here is not defined
 * as part of any of the system include files.
 */
struct pex {
        struct idp  p_idp;                  /* idp header */
        u_short     ph_id[2];               /* unique transaction ID for pex */
        u_short     ph_client;              /* client type field for pex */
};
```

The *ph_id* field is used to hold a ‘‘unique id’’ that is used in duplicate suppression; the *ph_client* field indicates the PEX client type (similar to the packet type field in the IDP header). PEX reliability stems from the fact that it is an idempotent (‘‘I send a packet to you, you send a packet to me’’) protocol. Processes on each side of the connection may use the unique id to determine if they have seen a given packet before (the unique id field differs on each packet sent) so that duplicates may be detected, and to indicate which message a given packet is in response to. If a packet with a given unique id is sent and no response is received in a given amount of time, the packet is retransmitted until it is decided that no response will ever be received. To simulate PEX, one must be able to generate unique ids -- something that is hard to do at the user level with any real guarantee that the id is really unique. Therefore, a means (via *getsockopt*) has been provided for getting unique ids from the kernel. The following code fragment indicates how to get a unique id:

```
long uniqueid;
int s, idsize = sizeof(uniqueid);
 ...
s = socket(AF_NS, SOCK_DGRAM, 0);
 ...
/* get id from the kernel -- only on IDP sockets */
getsockopt(s, NSPROTO_PE, SO_SEQNO, (char *)&uniqueid, &idsize);
 ...
```

The retransmission and duplicate suppression code required to simulate PEX fully is left as an exercise for the reader.

5.14. Inetd

One of the daemons provided with 4.4BSD is *inetd*, the so called ‘‘internet super-server.’’ Having one daemon listen for requests for many daemons instead of having each daemon listen for its own requests reduces the number of idle daemons and simplies their implementation. *Inetd* handles two types of services: standard and TCPMUX. A standard service has a well-known port assigned to it and is listed in */etc/services* (see *services*(5)); it may be a service that implements an official Internet standard or is a BSD-specific service. TCPMUX services are nonstandard and do not have a well-known port assigned to them. They are invoked from *inetd* when a program connects to the "tcpmux" well-known port and specifies the service name. This is useful for adding locally-developed servers.

Inetd is invoked at boot time, and determines from the file */etc/inetd.conf* the servers for which it is to listen. Once this information has been read and a pristine environment created, *inetd* proceeds to create one socket for each service it is to listen for, binding the appropriate port number to each socket.

Inetd then performs a *select* on all these sockets for read availability, waiting for somebody wishing a connection to the service corresponding to that socket. *Inetd* then performs an *accept* on the socket in question, *fork*s, *dup*s the new socket to file descriptors 0 and 1 (stdin and stdout), closes other open file descriptors, and *exec*s the appropriate server.

Servers making use of *inetd* are considerably simplified, as *inetd* takes care of the majority of the IPC work required in establishing a connection. The server invoked by *inetd* expects the socket connected to its client on file descriptors 0 and 1, and may immediately perform any operations such as *read*, *write*, *send*, or *recv*. Indeed, servers may use buffered I/O as provided by the ‘‘stdio’’ conventions, as long as as they remember to use *fflush* when appropriate.

One call which may be of interest to individuals writing servers under *inetd* is the *getpeername* call, which returns the address of the peer (process) connected on the other end of the socket. For example, to log the Internet address in ‘‘dot notation’’ (e.g., ‘‘128.32.0.4’’) of a client connected to a server under *inetd*, the following code might be used:

```
struct sockaddr_in name;
int namelen = sizeof (name);
 ...
if (getpeername(0, (struct sockaddr *)&name, &namelen) < 0) {
	syslog(LOG_ERR, "getpeername: %m");
	exit(1);
} else
	syslog(LOG_INFO, "Connection from %s", inet_ntoa(name.sin_addr));
 ...
```

While the *getpeername* call is especially useful when writing programs to run with *inetd*, it can be used under other circumstances. Be warned, however, that *getpeername* will fail on UNIX domain sockets.

Standard TCP services are assigned unique well-known port numbers in the range of 0 to 1023 by the Internet Assigned Numbers Authority (IANA@ISI.EDU). The limited number of ports in this range are assigned to official Internet protocols. The TCPMUX service allows you to add locally-developed protocols without needing an official TCP port assignment. The TCPMUX protocol described in RFC-1078 is simple:

> “A TCP client connects to a foreign host on TCP port 1. It sends the service name followed by a carriage-return line-feed <CRLF>. The service name is never case sensitive. The server replies with a single character indicating positive ("+") or negative ("−") acknowledgment, immediately followed by an optional message of explanation, terminated with a <CRLF>. If the reply was positive, the selected protocol begins; otherwise the connection is closed.”

In 4.4BSD, the TCPMUX service is built into *inetd*, that is, *inetd* listens on TCP port 1 for requests for TCPMUX services listed in *inetd.conf*. *inetd*(8) describes the format of TCPMUX entries for *inetd.conf*.

The following is an example TCPMUX server and its *inetd.conf* entry. More sophisticated servers may want to do additional processing before returning the positive or negative acknowledgement.

```
#include <sys/types.h>
#include <stdio.h>

main()
{
	time_t t;

	printf("+Go\r\n");
	fflush(stdout);
	time(&t);
	printf("%d = %s", t, ctime(&t));
	fflush(stdout);
}
```

The *inetd.conf* entry is:

```
tcpmux/current_time stream tcp nowait nobody /d/curtime curtime
```

Here’s the portion of the client code that handles the TCPMUX handshake:

```
char line[BUFSIZ];
FILE *fp;
 ...

/* Use stdio for reading data from the server */
fp = fdopen(sock, "r");
if (fp == NULL) {
   fprintf(stderr, "Can’t create file pointer\n");
   exit(1);
}

/* Send service request */
sprintf(line, "%s\r\n", "current_time");
if (write(sock, line, strlen(line)) < 0) {
   perror("write");
   exit(1);
}

/* Get ACK/NAK response from the server */
if (fgets(line, sizeof(line), fp) == NULL) {
   if (feof(fp)) {
      die();
   } else {
      fprintf(stderr, "Error reading response\n");
      exit(1);
   }
}

/* Delete <CR> */
if ((lp = index(line, ' ')) != NULL) {
   *lp = ' ';
}

switch (line[0]) {
   case ’+’:
         printf("Got ACK: %s\n", &line[1]);
         break;
   case ’-’:
         printf("Got NAK: %s\n", &line[1]);
         exit(0);
   default:
         printf("Got unknown response: %s\n", line);
         exit(1);
}

/* Get rest of data from the server */
while ((fgets(line, sizeof(line), fp)) != NULL) {
   fputs(line, stdout);
}
```