

## **An Advanced 4.3BSD 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 4.3BSD 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 Bell Laboratories.

## 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 current release of Berkeley UNIX, 4.3BSD, completes some of the IPC facilities and provides an upward-compatible interface. It is hoped that the interprocess communication facilities included in 4.3BSD will establish a standard for UNIX. From the response to the design, it appears many organizations carrying out work with UNIX are adopting it.

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 4.3BSD have been designed as a totally independent subsystem. The 4.3BSD 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 4.3BSD 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 4.3BSD IPC facilities support three separate communication domains: the UNIX domain, for on-system communication; the Internet domain, which is used by processes which communicate using the the DARPA standard communication protocols; and the NS domain, which is used by processes which communicate using the Xerox standard communication protocols\*. 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.

### 0.1. 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 on this topic can be found in section 5.

---

\* See *Internet Transport Protocols*, Xerox System Integration Standard (XSYS)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.

## 0.2. 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).

## 0.3. 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 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,

---

\* The manifest constants are named `AF_whatever` as they indicate the “address format” to use in interpreting names.

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_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 under 4.3BSD, 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.

#### 0.4. 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,

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

while in the Internet domain,

---

\* 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_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.

### 0.5. 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:

- 11.
- 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.

### 0.6. 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.

## 0.7. 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 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.

## 0.8. 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 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\_sets*. 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 4.3BSD networking facilities support both the DARPA standard 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 be 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 name-server, *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 internet-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 worthwhile.

\* Courier: The Remote Procedure Call Protocol, XSI:038112.

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.

box;	
l l	
l l.	
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. On some architectures, such as the 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 4.3BSD, 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 4.3BSD 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);
    }
    ...
#ifdef 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.

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

```

1 r11111.
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
calderup 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
ingresdown 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. runtime output.

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. 4.3BSD attempts to isolate host-specific information from applications by 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

\* 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.

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

```

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, onalm);
    onalm();
    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.

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 4.3BSD 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.