## Socket Types

Socket types define the communication properties visible to a user. The Internet family sockets provide access to the TCP/IP transport protocols. The Internet family is identified by the value AF\_INET6, for sockets that can communicate over both IPv6 and IPv4.

Three types of sockets are supported:

1. Stream sockets allow processes to communicate using TCP. A stream socket provides bidirectional, reliable, sequenced, and unduplicated flow of data with no record boundaries. After the connection has been established, data can be read from and written to these sockets as a byte stream. The socket type is SOCK\_STREAM.
2. Datagram sockets allow processes to use UDP to communicate. A datagram socket supports bidirectional flow of messages. A process on a datagram socket can receive messages in a different order from the sending sequence and can receive duplicate messages. Record boundaries in the data are preserved. The socket type is SOCK\_DGRAM.
3. Raw sockets provide access to ICMP. These sockets are normally datagram oriented, although their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not for most applications. They are provided to support developing new communication protocols or for access to more esoteric facilities of an existing protocol. Only superuser processes can use raw sockets. The socket type is SOCK\_RAW.

## Interface Sets

SunOS 5.8 provides two sets of socket interfaces.

1. The BSD socket interfaces. 2. XNS 5 (Unix98) Socket interfaces.

## Socket Basics

Socket Creation

The socket (3SOCKET) call creates a socket in the specified family and of the specified type.

s = socket (family, type, protocol);

If the protocol is unspecified (a value of 0), the system selects a protocol that supports the requested socket type. The socket handle (a file descriptor) is returned. The family is specified by one of the constants defined in sys/socket.h. Constants named AF\_*suite* specify the address format to use in interpreting names.

**Protocol Family**

|  |  |
| --- | --- |
| AF\_APPLETALK | Apple Computer Inc. Appletalk network |
| AF\_INET6 | Internet family for IPv6 and IPv4 |
| AF\_INET | Internet family for IPv4 only |
| AF\_PUP | Xerox Corporation PUP internet |
| AF\_UNIX | Unix file system |

Socket types(SOCK\_STREAM, SOCK\_DGRAM, or SOCK\_RAW are supported by AF\_INET6, AF\_INET, and AF\_UNIX) are defined in sys/socket.h. The following creates a stream socket in the Internet family:

s = socket (AF\_INET6, SOCK\_STREAM, 0);

This call results in a stream socket with the TCP protocol providing the underlying communication. Use the default protocol (the *protocol* argument is 0) in most situations.

**Binding Local Names**

A socket is created with no name. A remote process has no way to refer to a socket until an address is bound to it. Communicating processes are connected through addresses. In the Internet family, a connection is composed of local and remote addresses, and local and remote ports.

There can never be duplicate ordered sets, such as: protocol, local address, local port, foreign address, foreign port. In most families, connections must be unique.

The bind(3SOCKET) call allows a process to specify the local address of the socket. This forms the set local address, local port. connect(3SOCKET), and accept(3SOCKET) complete a socket's association by fixing the remote half of the address tuple. The bind(3SOCKET)call is used as follows:

bind (s, name, namelen);

The socket handle is s. The bound name is a byte string that is interpreted by the supporting protocol(s). Internet family names contain an Internet address and port number.

This example demonstrates binding an Internet address:

#include <sys/types.h>

#include <netinet/in.h>

...

struct sockaddr\_in6 sin6;

...

s = socket(AF\_INET6, SOCK\_STREAM, 0);

bzero (&sin6, sizeof (sin6));

sin6.sin6\_family = AF\_INET6;

sin6.sin6\_addr.s6\_addr = in6addr\_arg;

sin6.sin6\_port = htons(MYPORT);

bind(s, (struct sockaddr \*) &sin6, sizeof sin6);

The content of the address sin6 is described in "Address Binding", where Internet address bindings are discussed.

**Connection Establishment**

Connection establishment is usually asymmetric, with one process acting as the client and the other as the server. The server binds a socket to a well-known address associated with the service and blocks on its socket for a connect request. An unrelated process can then connect to the server. The client requests services from the server by initiating a connection to the server's socket. On the client side, the connect(3SOCKET) call initiates a connection. In the Internet family, this might appear as:

struct sockaddr\_in6 server;

...

connect(s, (struct sockaddr \*)&server, sizeof server);

If the client's socket is unbound at the time of the connect call, the system automatically selects and binds a name to the socket. See "Address Binding". This is the usual way that local addresses are bound to a socket on the client side.

To receive a client's connection, a server must perform two steps after binding its socket. The first is to indicate how many connection requests can be queued. The second step is to accept a connection:

struct sockaddr\_in6 from;

...

listen(s, 5); /\* Allow queue of 5 connections \*/

fromlen = sizeof(from);

newsock = accept(s, (struct sockaddr \*) &from, &fromlen);

The socket handle s is the socket bound to the address to which the connection request is sent. The second parameter of listen(3SOCKET)specifies the maximum number of outstanding connections that might be queued. from is a structure that is filled with the address of the client. A NULL pointer might be passed. fromlen is the length of the structure. (In the UNIX family, from is declared a struct sockaddr\_un.)

accept(3SOCKET) normally blocks. accept(3SOCKET) returns a new socket descriptor that is connected to the requesting client. The value of fromlen is changed to the actual size of the address.

A server cannot indicate that it accepts connections only from specific addresses. The server can check the from address returned by accept(3SOCKET) and close a connection with an unacceptable client. A server can accept connections on more than one socket, or avoid blocking on the accept call. These techniques are presented in "Advanced Topics".

**Connection Errors**

An error is returned if the connection is unsuccessful (however, an address bound by the system remains). Otherwise, the socket is associated with the server and data transfer can begin.

Table 2-2 lists some of the more common errors returned when a connection attempt fails.

Table 2-2 Socket Connection Errors

|  |  |
| --- | --- |
| Socket Errors | Error Description |
| ENOBUFS | Lack of memory available to support the call. |
| EPROTONOSUPPORT | Request for an unknown protocol. |
| EPROTOTYPE | Request for an unsupported type of socket. |
| ETIMEDOUT | No connection established in specified time. This happens when the destination host is down or when problems in the network result in lost transmissions. |
| ECONNREFUSED | The host refused service. This happens when a server process is not present at the requested address. |
| ENETDOWN or EHOSTDOWN | These errors are caused by status information delivered by the underlying communication interface. |
| ENETUNREACH or EHOSTUNREACH | These operational errors can occur either because there is no route to the network or host, or because of status information returned by intermediate gateways or switching nodes. The status returned is not always sufficient to distinguish between a network that is down and a host that is down. |

**Data Transfer**

This section describes the functions to send and receive data. You can send or receive a message with the normal read(2) and write(2) interfaces:

write(s, buf, sizeof buf);

read(s, buf, sizeof buf);

Or the calls send(3SOCKET) and recv(3SOCKET) can be used:

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

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

send(3SOCKET) and recv(3SOCKET) are very similar to read(2) and write(2), but the flags argument is important. The flags, defined in sys/socket.h, can be specified as a nonzero value if one or more of the following is required:

MSG\_OOB

Send and receive out-of-band data

MSG\_PEEK

Look at data without reading

MSG\_DONTROUTE

Send data without routing packets

Out-of-band data is specific to stream sockets. When MSG\_PEEK is specified with a recv(3SOCKET) call, any data present is returned to the user but treated as still unread. The next read(2) or recv(3SOCKET) call on the socket returns the same data. The option to send data without routing packets applied to the outgoing packets is currently used only by the routing table management process and is unlikely to be interesting to most users.

**Closing Sockets**

A SOCK\_STREAM socket can be discarded by a close(2) function call. If data is queued to a socket that promises reliable delivery after a close(2), the protocol continues to try to transfer the data. If the data is still undelivered after an arbitrary period, it is discarded.

A shutdown(3SOCKET) closes SOCK\_STREAM sockets gracefully. Both processes can acknowledge that they are no longer sending. This call has the form:

shutdown(s, how);

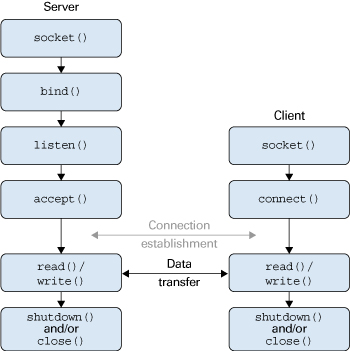
Where how is defined as:

|  |  |
| --- | --- |
| 0 | Disallows further receives |
| 1 | Disallows further sends |
| 2 | Disallows both further sends and receives |

**Connecting Stream Sockets**

Figure 2-1 and the next two examples illustrate initiating and accepting an Internet family stream connection.

Figure 2-1 Connection-Oriented Communication Using Stream Sockets



The program in Example 2-1 is a server. It creates a socket and binds a name to it, then displays the port number. The program calls listen(3SOCKET) to mark the socket ready to accept connection requests and initialize a queue for the requests. The rest of the program is an infinite loop. Each pass of the loop accepts a new connection and removes it from the queue, creating a new socket. The server reads and displays the messages from the socket and closes it. The use of in6addr\_any is explained in "Address Binding".

Example 2-1 Accepting an Internet Stream Connection (Server)

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <netdb.h>

#include <stdio.h>

#define TRUE 1

/\*

\* This program creates a socket and then begins an infinite loop.

\* Each time through the loop it accepts a connection and prints

\* data from it. When the connection breaks, or the client closes

\* the connection, the program accepts a new connection.

\*/

main()

{

int sock, length;

struct sockaddr\_in6 server;

int msgsock;

char buf[1024];

int rval;

/\* Create socket. \*/

sock = socket(AF\_INET6, SOCK\_STREAM, 0);

if (sock == -1) {

perror("opening stream socket");

exit(1);

}

/\* Bind socket using wildcards.\*/

bzero (&server, sizeof(server));

bzero (&sin6, sizeof (sin6));

server.sin6\_family = AF\_INET6;

server.sin6\_addr.s6\_addr = in6addr\_any;

server.sin6\_port = 0;

if (bind(sock, (struct sockaddr \*) &server, sizeof server)

== -1) {

perror("binding stream socket");

exit(1);

}

/\* Find out assigned port number and print it out. \*/

length = sizeof server;

if (getsockname(sock,(struct sockaddr \*) &server,&length)

== -1) {

perror("getting socket name");

exit(1);

}

printf("Socket port #%d\n", ntohs(server.sin6\_port));

/\* Start accepting connections. \*/

listen(sock, 5);

do {

msgsock = accept(sock,(struct sockaddr \*) 0,(int \*) 0);

if (msgsock == -1

perror("accept");

else do {

memset(buf, 0, sizeof buf);

if ((rval = read(msgsock,buf, 1024)) == -1)

perror("reading stream message");

if (rval == 0)

printf("Ending connection\n");

else

/\* assumes the data is printable \*/

printf("-->%s\n", buf);

} while (rval > 0);

close(msgsock);

} while(TRUE);

/\*

\* Since this program has an infinite loop, the socket "sock" is

\* never explicitly closed. However, all sockets will be closed

\* automatically when a process is killed or terminates normally.

\*/

exit(0);

}

/\*

To initiate a connection, the client program in Example 2-2 creates a stream socket and calls connect(3SOCKET), specifying the address of the socket for connection. If the target socket exists and the request is accepted, the connection is complete and the program can send data. Data are delivered in sequence with no message boundaries. The connection is destroyed when either socket is closed. For more information about data representation routines, such as ntohl(3SOCKET), ntohs(3SOCKET), htons(3SOCKET), and htonl(3XNET), in this program, see the byteorder(3SOCKET) man page.

\*/

Example 2-2 Internet family Stream Connection (Client)

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <netdb.h>

#include <stdio.h>

#define DATA "Half a league, half a league . . ."

/\*

\* This program creates a socket and initiates a connection with

\* the socket given in the command line. Some data are sent over the

\* connection and then the socket is closed, ending the connection.

\* The form of the command line is: streamwrite hostname portnumber

\* Usage: pgm host port

\*/

main(argc, argv)

int argc;

char \*argv[];

{

int sock, errnum h\_addr\_index;

struct sockaddr\_in6 server;

struct hostent \*hp;

char buf[1024];

/\* Create socket. \*/

sock = socket( AF\_INET6, SOCK\_STREAM, 0);

if (sock == -1) {

perror("opening stream socket");

exit(1);

}

/\* Connect socket using name specified by command line. \*/

bzero (&sin6, sizeof (sin6));

server.sin6\_family = AF\_INET6;

hp = getipnodebyname(AF\_INET6, argv[1], AI\_DEFAULT, &errnum);

/\*

\* getinodebyname returns a structure including the network address

\* of the specified host.

\*/

if (hp == (struct hostent \*) 0) {

fprintf(stderr, "%s: unknown host\n", argv[1]);

exit(2);

}

h\_addr\_index = 0;

while (hp->h\_addr\_list[h\_addr\_index] != NULL) {

bcopy(hp->h\_addr\_list[h\_addr\_index], &server.sin6\_addr,

hp->h\_length);

server.sin6\_port = htons(atoi(argv[2]));

if (connect(sock, (struct sockaddr \*) &server,

sizeof (server)) == -1) {

if (hp->h\_addr\_list[++h\_addr\_index] != NULL) {

/\* Try next address \*/

continue;

}

perror("connecting stream socket");

freehostent(hp);

exit(1);

}

break;

}

freehostent(hp);

if (write( sock, DATA, sizeof DATA) == -1)

perror("writing on stream socket");

close(sock);

freehostent (hp);

exit(0);

}

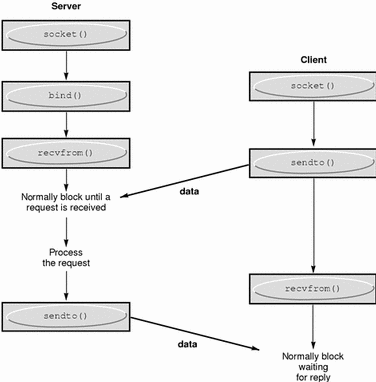
**Datagram Sockets**

A datagram socket provides a symmetric data exchange interface. There is no requirement for connection establishment. Each message carries the destination address. Figure 2-2 shows the flow of communication between server and client.

**Note -**

The bind(3SOCKET) step shown below for the server is optional.

Figure 2-2 Connectionless Communication Using Datagram Sockets



Datagram sockets are created as described in "Socket Creation". If a particular local address is needed, the bind(3SOCKET) operation must precede the first data transmission. Otherwise, the system sets the local address and/or port when data is first sent. To send data, sendto(3SOCKET) is used:

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

The s, buf, buflen, and flags parameters are the same as in connection-oriented sockets. The to and tolen values indicate the address of the intended recipient of the message. A locally detected error condition (such as an unreachable network) causes a return of -1 and errno to be set to the error number.

To receive messages on a datagram socket, recvfrom(3SOCKET) is used:

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

Before the call, fromlen is set to the size of the from buffer. On return, it is set to the size of the address from which the datagram was received.

Datagram sockets can also use the connect(3SOCKET) call to associate a socket with a specific destination address. It can then use the send(3SOCKET) call. Any data sent on the socket without explicitly specifying a destination address is addressed to the connected peer, and only data received from that peer is delivered. Only one connected address is permitted for one socket at a time. A second connect(3SOCKET) call changes the destination address. Connect requests on datagram sockets return immediately. The system records the peer's address. accept(3SOCKET), and listen(3SOCKET) are not used with datagram sockets.

While a datagram socket is connected, errors from previous send(3SOCKET) calls can be returned asynchronously. These errors can be reported on subsequent operations on the socket, or an option of getsockopt(3SOCKET), SO\_ERROR, can be used to interrogate the error status.

Example 2-3 shows how to send an Internet call by creating a socket, binding a name to the socket, and sending the message to the socket.

Example 2-3 Sending an Internet Family Datagram

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <netdb.h>

#include <stdio.h>

#define DATA "The sea is calm, the tide is full . . ."

/\*

\* Here I send a datagram to a receiver whose name I get from

\* the command line arguments. The form of the command line is:

\* dgramsend hostname portnumber

\*/

main(argc, argv)

int argc, errnum;

char \*argv[];

{

int sock;

struct sockaddr\_in6 name;

struct hostent \*hp;

/\* Create socket on which to send. \*/

sock = socket(AF\_INET6,SOCK\_DGRAM, 0);

if (sock == -1) {

perror("opening datagram socket");

exit(1);

}

/\*

\* Construct name, with no wildcards, of the socket to ``send''

\* to. getinodebyname returns a structure including the network

\* address of the specified host. The port number is taken from

\* the command line.

\*/

hp = getipnodebyname(AF\_INET6, argv[1], AI\_DEFAULT, &errnum);

if (hp == (struct hostent \*) 0) {

fprintf(stderr, "%s: unknown host\n", argv[1]);

exit(2);

}

bzero (&sin6, sizeof (sin6));

bzero (&name, sizeof (name));

memcpy((char \*) &name.sin6\_addr, (char \*) hp->h\_addr,

hp->h\_length);

name.sin6\_family = AF\_INET6;

name.sin6\_port = htons(atoi(argv[2]));

/\* Send message. \*/

if (sendto(sock,DATA, sizeof DATA ,0,

(struct sockaddr \*) &name,sizeof name) == -1)

perror("sending datagram message");

close(sock);

exit(0);

}

Example 2-4 shows how to read an Internet call by creating a socket, binding a name to the socket, and then reading from the socket.

Example 2-4 Reading Internet Family Datagrams

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <stdio.h>

/\*

\* This program creates a datagram socket, binds a name to it, then

\* reads from the socket.

\*/

main()

{

int sock, length;

struct sockaddr\_in6 name;

char buf[1024];

/\* Create socket from which to read. \*/

sock = socket(AF\_INET6, SOCK\_DGRAM, 0);

if (sock == -1) {

perror("opening datagram socket");

exit(1);

}

/\* Create name with wildcards. \*/

bzero (&sin6, sizeof (sin6));

name.sin6\_family = AF\_INET6;

name.sin6\_addr.s6\_addr = in6addr\_any;

name.sin6\_port = 0;

if (bind(sock,(struct sockaddr \*)&name, sizeof name) == -1) {

perror("binding datagram socket");

exit(1);

}

/\* Find assigned port value and print it out. \*/

length = sizeof(name);

if (getsockname(sock,(struct sockaddr \*) &name, &length)

== -1) {

perror("getting socket name");

exit(1);

}

printf("Socket port #%d\n", ntohs(name.sin6\_port));

/\* Read from the socket. \*/

if (read(sock, buf, 1024) == -1 )

perror("receiving datagram packet");

/\* Assumes the data is printable \*/

printf("-->%s\n", buf);

close(sock);

exit(0);

}

**Input/Output Multiplexing**

Requests can be multiplexed among multiple sockets or files. Use select(3C) to do this:

#include <sys/time.h>

#include <sys/types.h>

#include <sys/select.h>

...

fd\_set readmask, writemask, exceptmask;

struct timeval timeout;

...

select(nfds, &readmask, &writemask, &exceptmask, &timeout);

The first argument of select(3C) is the number of file descriptors in the lists pointed to by the next three arguments.

The second, third, and fourth arguments of select(3C) point to three sets of file descriptors: a set of descriptors to read on, a set to write on, and a set on which exception conditions are accepted. Out-of-band data is the only exceptional condition. Any of these pointers can be a properly cast null. Each set is a structure containing an array of long integer bit masks. The size of the array is set by FD\_SETSIZE (defined in select.h). The array is long enough to hold one bit for each FD\_SETSIZE file descriptor.

The macros FD\_SET(fd, &mask) and FD\_CLR(fd, &mask) add and delete, respectively, the file descriptor fd in the set mask. The set should be zeroed before use, and the macro FD\_ZERO(&mask) clears the set mask.

The fifth argument of select(3C) allows a time-out value to be specified. If the timeout pointer is NULL, select(3C) blocks until a descriptor is selectable, or until a signal is received. If the fields in timeout are set to 0, select(3C) polls and returns immediately.

select(3C) normally returns the number of file descriptors selected. select(3C) returns a 0 if the time-out has expired. select(3C) returns -1 for an error or interrupt with the error number in errno and the file descriptor masks unchanged. For a successful return, the three sets indicate which file descriptors are ready to be read from, written to, or have exceptional conditions pending.

You should test the status of a file descriptor in a select mask with the FD\_ISSET(fd, &mask) macro. It returns a nonzero value if fd is in the set mask, and 0 if it is not. Use select(3C) followed by a FD\_ISSET(fd, &mask) macro on the read set to check for queued connect requests on a socket.

Example 2-5 shows how to select on a "listening" socket for readability to determine when a new connection can be picked up with a call to accept(3SOCKET). The program accepts connection requests, reads data, and disconnects on a single socket.

Example 2-5 Using select(3C) to Check for Pending Connections

#include <sys/types.h>

#include <sys/socket.h>

#include <sys/time.h>

#include <netinet/in.h>

#include <netdb.h>

#include <stdio.h>

#define TRUE 1

/\*

\* This program uses select to check that someone is

\* trying to connect before calling accept.

\*/

main()

{

int sock, length;

struct sockaddr\_in6 server;

int msgsock;

char buf[1024];

int rval;

fd\_set ready;

struct timeval to;

/\* Open a socket and bind it as in previous examples. \*/

/\* Start accepting connections. \*/

listen(sock, 5);

do {

FD\_ZERO(&ready);

FD\_SET(sock, &ready);

to.tv\_sec = 5;

to.tv\_usec = 0;

if (select(sock + 1, &ready, (fd\_set \*)0, (fd\_set \*)0, &to) == -1) {

perror("select");

continue;

}

if (FD\_ISSET(sock, &ready)) {

msgsock = accept(sock, (struct sockaddr \*)0,

(int \*)0);

if (msgsock == -1)

perror("accept");

else do {

memset(buf, 0, sizeof buf);

if ((rval = read(msgsock, buf, 1024)) == -1)

perror("reading stream message");

else if (rval == 0)

printf("Ending connection\n");

else

printf("-->%s\n", buf);

} while (rval > 0);

close(msgsock);

} else

printf("Do something else\n");

} while (TRUE);

exit(0);

}

In previous versions of the select(3C) routine, its arguments were pointers to integers instead of pointers to fd\_sets. This style of call still works if the number of file descriptors is smaller than the number of bits in an integer.

select(3C) provides a synchronous multiplexing scheme. The SIGIO and SIGURG signals (described in "Advanced Topics") provide asynchronous notification of output completion, input availability, and exceptional conditions.

**Standard Routines**

You might need to locate and construct network addresses. This section describes the routines that manipulate network addresses. Unless otherwise stated, functions presented in this section apply only to the Internet family.

Locating a service on a remote host requires many levels of mapping before client and server communicate. A service has a name for human use. The service and host names must be translated to network addresses. Finally, the address is used to locate and route to the host. The specifics of the mappings can vary between network architectures. Preferably, a network does not require that hosts be named, thus protecting the identity of their physical locations. It is more flexible to discover the location of the host when it is addressed.

Standard routines map 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.

**Host and Service Names**

The interfaces getaddrinfo(3SOCKET), getnameinfo(3SOCKET), and freeaddrinfo(3SOCKET) provide a simplified method of translating between the names and addresses of a service on a host. For IPv6, these interfaces can be used instead of calling getipnodebyname(3SOCKET) and getservbyname(3SOCKET) and then figuring out how to combine the addresses. Similarly, for IPv4, these interfaces can be used instead of gethostbyname(3NSL) and getservbyname(3SOCKET). Both IPv6 and IPv4 addresses are handled transparently.

getaddrinfo(3SOCKET) returns the combined address and port number of the specified host and service names. Since all of the information returned by getaddrinfo(3SOCKET) is dynamically allocated, it must be freed by freeaddrinfo(3SOCKET) to prevent memory leaks.getnameinfo(3SOCKET) returns the host and services names associated with a specified address and port number. To print error messages based on the EAI\_xxx codes returned by getaddrinfo(3SOCKET) and getnameinfo(3SOCKET), call gai\_strerror(3SOCKET).

An example of using getaddrinfo(3SOCKET) follows:

struct addrinfo \*res, \*aip;

struct addrinfo hints;

int sock = -1;

int error;

/\* Get host address. Any type of address will do. \*/

bzero(&hints, sizeof (hints));

hints.ai\_flags = AI\_ALL|AI\_ADDRCONFIG;

hints.ai\_socktype = SOCK\_STREAM;

error = getaddrinfo(hostname, servicename, &hints, &res);

if (error != 0) {

(void) fprintf(stderr, "getaddrinfo: %s for host %s service %s\n",

gai\_strerror(error), hostname, servicename);

return (-1);

}

After processing the information returned by getaddrinfo(3SOCKET) in the structure pointed to by res, the storage should be released by

freeaddrinfo(res);

getnameinfo(3SOCKET) is particularly useful in identifying the cause of an error as in the following example:

struct sockaddr\_storage faddr;

int sock, new\_sock;

socklen\_t faddrlen;

int error;

char hname[NI\_MAXHOST];

char sname[NI\_MAXSERV];

...

faddrlen = sizeof (faddr);

new\_sock = accept(sock, (struct sockaddr \*)&faddr, &faddrlen);

if (new\_sock == -1) {

if (errno != EINTR && errno != ECONNABORTED) {

perror("accept");

}

continue;

}

error = getnameinfo((struct sockaddr \*)&faddr, faddrlen, hname,

sizeof (hname), sname, sizeof (sname), 0);

if (error) {

(void) fprintf(stderr, "getnameinfo: %s\n",

gai\_strerror(error));

} else {

(void) printf("Connection from %s/%s\n", hname, sname);

}

**hostent - 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; /\* hostaddrtype(e.g.,AF\_INET6) \*/

int h\_length; /\* length of address \*/

char \*\*h\_addr\_list; /\* list of addrs, null terminated \*/

};

/\*1st addr, net byte order\*/

#define h\_addr h\_addr\_list[0]

getipnodebyname(3SOCKET) maps an Internet host name to a hostent structure, getipnodebyaddr(3SOCKET) maps an Internet host address to a hostent structure, freehostent(3SOCKET) frees the memory of a hostent structure, and inet\_ntop(3SOCKET) maps an Internet host address to a displayable string.

The routines return a hostent structure containing the name of the host, its aliases, the address type (address family), and a NULL-terminated list of variable length addresses. The list of addresses is required because a host can have many addresses. The h\_addr definition is for backward compatibility, and is the first address in the list of addresses in the hostent structure.

**netent - Network Names**

The routines to map network names to numbers and back return a netent structure:

/\*

\* Assumes that a network number fits in 32 bits.

\*/

struct netent {

char \*n\_name; /\* official name of net \*/

char \*\*n\_aliases; /\* alias list \*/

int n\_addrtype; /\* net address type \*/

int n\_net; /\* net number, host byte order \*/

};

getnetbyname(3SOCKET), getnetbyaddr\_r(3SOCKET), and getnetent(3SOCKET) are the network counterparts to the host routines described above.

**protoent - Protocol Names**

The protoent structure defines the protocol-name mapping used with getprotobyname(3SOCKET), getprotobynumber(3SOCKET), and getprotoent(3SOCKET):

struct protoent {

char \*p\_name; /\* official protocol name \*/

char \*\*p\_aliases /\* alias list \*/

int p\_proto; /\* protocol number \*/

};

**servent - Service Names**

An Internet family service resides at a specific, well-known port and uses a particular protocol. A service-name-to-port-number 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 \*/

};

getservbyname(3SOCKET) maps service names and, optionally, a qualifying protocol to a servent structure. The call:

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

returns the service specification of a telnet server using any protocol. The call:

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

returns the telnet server that uses the TCP protocol. getservbyport(3SOCKET) and getservent(3SOCKET) are also provided. getservbyport(3SOCKET) has an interface similar to that of getservbyname(3SOCKET); an optional protocol name can be specified to qualify lookups.

**Other Routines**

In addition to address-related database routines, there are several other routines that simplify manipulating names and addresses. Table 2-3summarizes the routines for manipulating variable-length byte strings and byte-swapping network addresses and values.

Table 2-3 Runtime Library Routines

|  |  |
| --- | --- |
| Call | Synopsis |
| memcmp(3C) | Compares byte-strings; 0 if same, not 0 otherwise |
| memcpy(3C) | Copies *n* bytes from *s2* to *s1* |
| memset(3C) | Sets *n* bytes to value starting at base |
| htonl(3SOCKET) | 32-bit quantity from host into network byte order |
| htons(3SOCKET) | 16-bit quantity from host into network byte order |
| ntohl(3SOCKET) | 32-bit quantity from network into host byte order |
| ntohs(3SOCKET) | 16-bit quantity from network into host byte order |

The byte-swapping routines are provided because the operating system expects addresses to be supplied in network order. On some architectures, the host byte ordering is different from network byte order, so programs must sometimes byte-swap values. Routines that return network addresses do so in network order. Byte-swapping problems occur only when interpreting network addresses. For example, the following code formats a TCP or UDP port:

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

On certain machines, where these routines are not needed, they are defined as null macros.

**Client-Server Programs**

The most common form of distributed application is the client/server model. In this scheme, client processes request services from a server process.

An alternate scheme is a service server that can eliminate dormant server processes. An example is inetd(1M), the Internet service daemon. inetd(1M) listens at a variety of ports, determined at start up by reading a configuration file. When a connection is requested on an inetd(1M)serviced port, inetd(1M) spawns the appropriate server to serve the client. Clients are unaware that an intermediary has played any part in the connection. inetd(1M) is described in more detail in "inetd(1M) Daemon".

Servers

Most servers are accessed at well-known Internet port numbers or UNIX family names. Example 2-6 illustrates the main loop of a remote-login server.

Example 2-6 Remote Login Server

main(argc, argv)

int argc;

char \*argv[];

{

int f;

struct sockaddr\_in6 from;

struct sockaddr\_in6 sin;

struct servent \*sp;

sp = getservbyname("login", "tcp");

if (sp == (struct servent \*) NULL) {

fprintf(stderr, "rlogind: tcp/login: unknown service");

exit(1);

}

...

#ifndef DEBUG

/\* Disassociate server from controlling terminal. \*/

...

#endif

sin.sin6\_port = sp->s\_port; /\* Restricted port \*/

sin.sin6\_addr.s6\_addr = in6addr\_any;

...

f = socket(AF\_INET6, SOCK\_STREAM, 0);

...

if (bind( f, (struct sockaddr \*) &sin, sizeof sin ) == -1) {

...

}

...

listen(f, 5);

while (TRUE) {

int g, len = sizeof from;

g = accept(f, (struct sockaddr \*) &from, &len);

if (g == -1) {

if (errno != EINTR)

syslog(LOG\_ERR, "rlogind: accept: %m");

continue;

}

if (fork() == 0) {

close(f);

doit(g, &from);

}

close(g);

}

exit(0);

}

Example 2-7 shows how the server gets its service definition.

Example 2-7 Remote Login Server: Step 1

sp = getservbyname("login", "tcp");

if (sp == (struct servent \*) NULL) {

fprintf(stderr, "rlogind: tcp/login: unknown service\n");

exit(1);

}

The result from getservbyname(3SOCKET) is used later to define the Internet port at which the program listens for service requests. Some standard port numbers are in /usr/include/netinet/in.h.

Example 2-8 shows how the server dissociates from the controlling terminal of its invoker in the non-DEBUG mode of operation.

Example 2-8 Dissociating From the Controlling Terminal

(void) close(0);

(void) close(1);

(void) close(2);

(void) open("/", O\_RDONLY);

(void) dup2(0, 1);

(void) dup2(0, 2);

setsid();

This prevents the server from receiving signals from the process group of the controlling terminal. After a server has dissociated itself, it cannot send reports of errors to a terminal and must log errors with syslog(3C).

A server next creates a socket and listens for service requests. bind(3SOCKET) ensures that the server listens at the expected location. (The remote login server listens at a restricted port number, so it runs as superuser.)

Example 2-9 illustrates the main body of the loop.

Example 2-9 Remote Login Server: Main Body

while(TRUE) {

int g, len = sizeof(from);

if (g = accept(f, (struct sockaddr \*) &from, &len) == -1) {

if (errno != EINTR)

syslog(LOG\_ERR, "rlogind: accept: %m");

continue;

}

if (fork() == 0) { /\* Child \*/

close(f);

doit(g, &from);

}

close(g); /\* Parent \*/

}

accept(3SOCKET) blocks messages until a client requests service. accept(3SOCKET) returns a failure indication if it is interrupted by a signal, such as SIGCHLD. The return value from accept(3SOCKET) is checked and an error is logged with syslog(3C) if an error has occurred.

The server then fork(2)s a child process and invokes the main body of the remote login protocol processing. The socket used by the parent to queue connection requests is closed in the child. The socket created by accept(3SOCKET) is closed in the parent. The address of the client is passed to the server application's **doit()** routine, which performs the actual application protocol with the client, for authenticating clients.

Clients

This section describes the steps taken by the client remote login process. As in the server, the first step is to locate the service definition for a remote login:

sp = getservbyname("login", "tcp");

if (sp == (struct servent \*) NULL) {

fprintf(stderr,"rlogin: tcp/login: unknown service");

exit(1);

}

Next, the destination host is looked up by a call togetipnodebyname(3SOCKET):

hp = getipnodebyname (AF\_INET6, argv[1], AI\_DEFAULT, &errnum);

if (hp == (struct hostent \*) NULL) {

fprintf(stderr, "rlogin: %s: unknown host", argv[1]);

exit(2);

}

The next step is to connect to the server at the requested host and start the remote login protocol. The address buffer is cleared and filled with the Internet address of the foreign host and the port number at which the login server listens:

memset((char \*) &server, 0, sizeof server);

bzero (&sin6, sizeof (sin6));

memcpy((char\*) &server.sin6\_addr,hp->h\_addr,hp->h\_length);

server.sin6\_family = hp->h\_addrtype;

server.sin6\_port = sp->s\_port;

A socket is created, and a connection initiated. connect(3SOCKET) implicitly does a bind(3SOCKET), 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);

}

Connectionless Servers

Some services use datagram sockets. The rwho(1) service provides status information on hosts connected to a local area network. (Avoid running in.rwhod(1M) because it causes heavy network traffic.) This service requires the ability to broadcast information to all hosts connected to a particular network. It is an example of datagram socket use.

A user on a host running the rwho(1) server can get the current status of another host with ruptime(1). Typical output is illustrated in Example 2-10.

Example 2-10 Output of ruptime(1) Program

itchy up 9:45, 5 users, load 1.15, 1.39, 1.31

scratchy up 2+12:04, 8 users, load 4.67, 5.13, 4.59

click up 10:10, 0 users, load 0.27, 0.15, 0.14

clack up 2+06:28, 9 users, load 1.04, 1.20, 1.65

ezekiel up 25+09:48, 0 users, load 1.49, 1.43, 1.41

dandy 5+00:05, 0 users, load 1.51, 1.54, 1.56

peninsula down 0:24

wood down 17:04

carpediem down 16:09

chances up 2+15:57, 3 users, load 1.52, 1.81, 1.86

Status information is periodically broadcast by the rwho(1) server processes on each host. The server process also receives the status information and updates a database. This database is interpreted for the status of each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

Use of broadcast is fairly inefficient because a lot of net traffic is generated. Unless the service is used widely and frequently, the expense of periodic broadcasts outweighs the simplicity.

Example 2-11 shows a simplified version of the rwho(1) server. It performs two tasks: receives status information broadcast by other hosts on the network and supplies the status of its host. The first task is done in the main loop of the program: Packets received at the rwho(1) port are checked to be sure they were sent by another rwho(1) server process, and are stamped with the arrival time. They then update a file with the status of the host. When a host has not been heard from for an extended time, the database routines assume the host is down and logs it. This application is prone to error, as a server might be down while a host is up.

Example 2-11 rwho(1) Server

main()

{

...

sp = getservbyname("who", "udp");

net = getnetbyname("localnet");

sin.sin6\_addr = inet\_makeaddr(net->n\_net, in6addr\_any);

sin.sin6\_port = sp->s\_port;

...

s = socket(AF\_INET6, SOCK\_DGRAM, 0);

...

on = 1;

if (setsockopt(s, SOL\_SOCKET, SO\_BROADCAST, &on, sizeof on)

== -1) {

syslog(LOG\_ERR, "setsockopt SO\_BROADCAST: %m");

exit(1);

}

bind(s, (struct sockaddr \*) &sin, sizeof sin);

...

signal(SIGALRM, onalrm);

onalrm();

while(1) {

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 == -1 && errno != EINTR)

syslog(LOG\_ERR, "rwhod: recv: %m");

continue;

}

if (from.sin6\_port != sp->s\_port) {

syslog(LOG\_ERR, "rwhod: %d: bad from port",

ntohs(from.sin6\_port));

continue;

}

...

if (!verify( wd.wd\_hostname)) {

syslog(LOG\_ERR, "rwhod: bad host name from %x",

ntohl(from.sin6\_addr.s6\_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);

}

exit(0);

}

The second server task is to supply the status of its host. This requires periodically acquiring system status information, packaging it in a message, and broadcasting it on the local network for other rwho(1) servers to hear. This task is run by a timer and triggered with a signal. Locating the system status information is involved but uninteresting.

Status information is broadcast on the local network. For networks that do not support broadcast, use another scheme.