CHAPTER IX
DEVELOPING APPLICATIONS OVER TCP/IP
Developing applications over TCP/IP using Internet socket

9.1 Introduction

A socket is an end point for communication between processes. It is a software and entity that provide the basic building block for inter process communication. For Internet address a fully named pair of socket uniquely identify a connection between two communicating sites:

<< node.port > node.port>>

_node_ is the four-byte Internet address, and port is two bytes identify the network interface. The first _node.port_ pair identify the local socket; the second identify the remote (foreign) socket.

TCP/IP support socket in both the Internet and Unix domains. A communication domain is a type of network with particular convention, such as how socket are named and used. Sockets exchange data only with socket in the same domain.

The _netstat_ function displays the status of any existing domain socket connection. The activity status of the user processes can be seen in the active connection display. If both sides of a socket pair or operating on the local machine, each is listed separately.

The Internet address manipulation routines (_inet_addr, inet_aton, inet_network, inet_inaof, inetmakeaddr, inet_netof_) convert between different address formats.

9.2 Socket types (Internet domain)

A socket has a type and or more associated process. Sockets are typed by the communications properties visible to the programmer. Usually a socket type is associated with the particular protocol that the Socket Supports. Process usually communicates between sockets of the same times.

Three types of sockets are available to the programmers that is used in developing the different types of Applications:

- stream socket
- data gram socket
- raw socket
9.2.1 Stream sockets

A stream socket (TYPE SOCK_STREAM) is the most commonly used type. In the AF_INET communication domain, a stream socket takes advantage of the inherent reliability of the transport level byte stream protocol, TCP. It provides bi-directional, sequenced, and unduplicated flow of data without boundaries.

9.2.2 Datagram sockets

A Datagram socket (type SOCK_DGRAM) support bi-directional flow of data in the Datagram model of the network level protocol. Record boundaries are preserved. The receiving process must perform resequencing, elimination of duplicates, and reliability assurance. The Datagram socket can be used in applications where reliability of an individual packet is not essential, for example, in broadcasting messages for the purpose of updating a status table.

9.2.3 Raw Sockets

Using a raw socket (type SOCK_RAW) the programmer has access to the underlying communications protocols which support sockets, such as the IP. Raw socket can be implemented variously depending on the interface provided by the communication protocol chosen.

Raw sockets are not intended for the general user. They are 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.

9.3 Socket Creation

Socket are created by calling the socket (SSC) routine:

\[
S = \text{socket} (\text{domain, type, protocol});
\]

**Domain**  
The domain is specified as a manifest constant named AF_* because it indicates the "address family" to use in interpreting names. For the Internet domain, the constant is always AF_INET (address family _ Internet domain)

**Type**  
Types are:

- SOCK_STREAM
- SOCK_DGRAM
- SOCK_RAW

**Protocol**  
If the protocol is unspecified (a value of 0), the system selects an appropriate protocol from those available to support the requested socket type. The system returns a small integer descriptor, similar to a file descriptor, to use in later system calls which operate on sockets.
The socket call also requires the `<sys/socket.h>` include file. To create a stream socket in the Internet domain with TCP providing the underlying support.

```c
#include <sys/socket.h>
S = socket(AF_INET, SOCK_STREAM, 0)
```

### 9.4 Selecting a protocol (Internet domain)

To obtain a particular protocol, select the protocol number as defined within the communication domain. For the Internet domain, the available protocols are defined in netinet/in.h. You can also use one of the library routines such as `getprotobynames` in `getprotents` (SLIB):

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
/*.... */
PP = getprotobyname("tcp");
S = socket(AF_INET, SOCK_STREAM, PP->P Proto);
```

### 9.4.1 Errors in Socket creation

The most common errors returned during socket creation are:

- **EPROTONOSUPPORT** The protocol type or the specified protocol is not support within this domain.
- **EMFILE** The per-process descriptor table is full.
- **EACCESS** Permission to create a socket of the specified type and/or protocol is denied.
- **ENOBUFS** Insufficient buffer space is available. The socket cannot be created until resources are freed.

If a connection fails at any time, the socket cannot be re-used; it must be closed and a new socket created.

### 9.4.2 Binding socket names

A socket is created without a name, but to be used it must be given a name until a name is bound to a socket, processes have no way to reference the socket, so no message may be received on it. Names can be bound explicitly by a process; otherwise, the system assigns a name (this is called an "implicitbind") during certain calls, such as `connect`.
Communicating processes are bound by an association. In the Internet domain, unlike the UNIX domain, an association is composed of local and foreign ports. 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. (The term "tuple" refers to a joined set, like the term "pair" but applying to more than two elements.)

The bind call is used to assign a name to a socket on the local side of a connection:

\[
\text{bind (s, name, namelen);} 
\]

The argument \text{s} is the socket descriptor returned from the socket call.

The argument \text{name} is a variable length byte string to be interpreted by the supporting protocol or protocols according to the domain type. In the internet domain, \text{name} contains an Internet address and port number, usually formatted in a structure called sockaddr_in. The argument \text{namelen} is the length of the name.

You do not have to specify an address unless you want a certain one.

To bind an Internet address, the call is:

\[
\# \text{include < sys/types.h>}
\# \text{include < netinet/in.h>}
/* .... */
\text{bind (struct sockaddr *) & size of (sin);}
\]

If you set \text{sin} to 0, the system binds to the server and returns it used.

9.4.3 How to make connection

Once a process has bounded a local socket, the process can rendezvous with an unrelated foreign process (a process initiated by a remote source). Usually, the rendezvous takes the form of a client-server relationship. The client completes the other side of the socket pair when it requests services from the server, initiating a connection by issuing a connect call:

\[
\text{Struct sockaddr_in server;}
\text{Connects (s,(Struct sockaddr*) & server, size of (server));}
\]
Server is the Internet address and port number of the server to which the server is the client is connecting. If the client process's socket is unbound when it issues the connect call, the system automatically selects a name and bind the socket. If the socket is successfully associated with the server, data transfer can begin.

Only the active process uses connect. An active process is a process that initiates a connection; a passive process, such as a server process, listen for calls and accepts them, but does not initiate them.

9.5 The server

A completed connection is identified by a unique pair of sockets, each socket being an endpoint associated with one of the reciprocating process. For the server to receive a client's connection, the server must issue two-system call after binding its socket. The first is to indicate a willingness to listen for incoming connection requests. The second is to accept the client's connect.

Consider the following example:

Listen \( s, 5 \)

The second parameter, \( 5 \), indicate the maximum number of outstanding connection which can be queued awaiting the acceptance of the server. This limit prevents processes from tying up system resources. Should a connection, be requested while the queue is full, the server does not refuse the connection, but ignores the messages which comprise the request. This forces the client to retry the connection request and gives the server time to make room in its queue. The listen call itself does not make the server process wait; it only specifies the maximum number of outstanding connection requests that can be queued.

Had the connection been returned with the ECONNREFUSED error, the client would unable to tell if the server was up or not. As it is, now is still possible to get the ETIMEDOUT error back, though this is unlikely. The backlog figure supplied with the listen call is limited by the system to a maximum of five pending connections on any one queue. This avoids the problem of processes tying up system resources by setting an infinite backlog and then ignoring all connection requests.

9.5.1 Wildcard in socket address
A server can use the wildcard symbol "*" to under specify its location in order to service incoming requests from multiple network interfaces. A service such as ftp can program can listen on all the network interfaces:

```
<< * , 21 > << * , * >>
```

This tuple signifies that the local ftp on port 21 is listening on multiple interface addresses for whatever client processes that wish to connect.

To name a socket that listen on all networks interfaces the Internet address INADDR_ANY must be bound. If a listening port is not specified, the system assigns one.

9.5.2 Accepting a connection (Internet domain)

With the socket marked as listening, the server can now accept a connection:

```
formlen = size of (from);
Snew = accept (s, (struct sockaddr*)& from, & fromlen);
```

The server returns a new descriptor to the client 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 address.

The `formlen` argument is a value-result parameter initialized by the server to indicate how much space is associated with `from` (the client). It is modified on return to reflect the true size of the address. Only a passive process uses `accept`. A passive process, such as a several process, can listen for calls and accept. A passive process, such as a server process, can listen for calls and accept them, but cannot initiate them.

The call to `accept` will not return until a connection is available or the system call is interrupted by a signal to the process. This is called "blocking". The `accept` call cannot screen for connections from certain sources only. A user process must provide its own screening, if required, by identifying the source of the connection and closing down the connection if it does not wish to speak to the other process.

Servers often bind multiple sockets. When a server accepts a connection, it usually spins off ("forks") a process that is the
connected socket. The parent then goes back to listening on the same local socket.

9.5.3 Connection errors (Internet domain)

The most common errors that can be returned when a connection fails are:

**ETIMEDOUT** After failing to establish a connection during a period of time, the system decided there was no point in retrying any more. The cause for this error is usually that the remote host is down or that problem in the network resulted in transmission being lost.

**ECONNREUSED** The host service for some reason. This error is usually caused by a server process not being present at the request host.

**ENETDOWN**
**EHOSTDOWN** Status information received by the client host from the underlying communication service indicates the net or the remote host is down.

**ENETUREACH**
**EHOSTUNREACH** These operational errors can occur either because the network or host is unknown (no route to the host or network is present) or because status information to that effect has been delivered to the client host by the underlying communication services.

If a connection fails at any time, the socket cannot be re-used; it must be close and a new socket created.

9.5.4 Data Transfer

When a connection is established, data flow can begin using a number of possible calls. If the peer entity at each end of a connection is anchored (that is, there is a connection), a user can send or receive a message without specifying the peer by using `read` and `write`:

```c
Write (s, buf, sizeof (buf);
read (s, buf, sizeof (buf);
```

The calls `send` and `recv` are virtually identical to `read` and `write`, except that a flags argument is added.

```c
Send (s, buf, size of (buf), flags);
recv (s, buf, size of (buf), flags);
```
The flags can be specified as nonzero values:

**MSG_OOB** Send / receive out-of-band data. Out-of-band data is specific to stream sockets.

**MSG_PEEK** Look at data without reading. When this value is specified in a recv call, any data present is returned to the use but treated as though still "unread". The next read or recv call applied to the socket will return the data previously previewed.

**MSG_DONTROUTE** Send data without routing packets. (Used only by the routing table management process.)

### 9.5.5 Closing sockets and discarding queued data

A process can discard a socket that is no longer of use by calling close (s):

```
Close (s);
```

If data is associated with a stream socket, which promise reliable delivery, the system continues to attempt to transfer the data. Data still undelivered after a long period of time is discarded. A client process can abort cause all data queued for transfer to be discarded immediately by calling **shutdown** (ssc) on the socket before closing it:

```
Shutdown (s, how)
```

The variable how is one of:

- 0 if the user no longer wishes to read data
- 1 if no more data will be sent.
- 2 if no data is to be sent or received.

### 9.6 Connectionless socket (Internet domain)

To this point we have been concerned mostly with sockets which follow a connection-oriented model. However, data can be transmitted without a direct connection by using datagram sockets.
UDP datagram sockets provide only connectionless interactions. When using datagram sockets, the programmer does not have to issue a connect call before sending. A datagram socket provides a symmetric interface to data exchange. Datagram processes are still likely to be client and server, but there is no requirement for connection establishment. Each message includes the destination address.

9.6.1 Sending from datagram sockets (Internet domain)

Datagram sockets are created and bound in the same ways as stream sockets. However, to send data from a datagram socket, the process uses the `sendto` primitive:

```
Sendto (s, buf, buflen, flags, &to, tolen);
```

The parameter are the same as those described for send, except the to and tolen values are used to indicate the intended recipient of the message.

When using an unreliable datagram interface, it is unlikely any errors will be reported to the sender. If information at the sending node indicates that the message cannot be delivered (for example, when a network is unreachable), the call returns -1, and the global variable `errno` contains an error number.

9.6.2 Receiving on datagram socket (Internet domain)

To receive data on an unconnected datagram socket, use `recvfrom`:

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

`fromlen` initially, contains the size of the from buffer, and is modified on return to indicate the actual size of the address from which the datagram was received.

9.6.3 Connecting on a datagram socket

Datagram sockets can also use connect to associate a socket with a specific address. Any data sent on the socket is automatically addresses to the connected peer, and only data received from that peer is delivered to the user.
Only one connected address is permitted for each socket (that is, no multicasting); 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; the system merely records the peer's address, as compared to a stream socket where a connect request initiates establishment of an end to end connection. The accept and listen calls are not used with datagram sockets data.

9.6.4 Input/output multiplexing (Internet domain)

An application system can multiplex I/O request among multiple sockets by using select, as shown for the Internet domain:

The select call takes three I/O descriptor sets as arguments:

- The set of file description on which the caller wishes to be able to read data
- The set of descriptors to which data is to be written
- Any pending exceptional conditions

The descriptor sets are stored as bit fields in arrays of integers. If the user is not interested in certain conditions (read, write, or exceptions), the corresponding argument to the select should be a null pointer.

Bit masks are created by or-ing bits of the form "1<< fd". That is, a descriptor fd is selected if a 1 is present in the fd 'th bit of the mask. The parameter nfds specifies the range of file descriptors (that is, one plus the value of the largest descriptor) specified in a mask.

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 (one plus the value of the largest descriptor) to be examined in a set.

A time out value may be specified if the selection is not to last more than a predetermined period of time. If timeout is 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. (A return takes place only when a descriptor is selectable, or when a signal is received by the caller, interrupting the system call.) The select call normally returns the number of file descriptors selected. If the select call
returns due to the timeout expiring, then a value of -1 is returned along with the error number EINTR.

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 description 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 whether 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 return a non-zero value, indicating permission to read, then a connection is pending on the socket.

9.7 Handling network dependencies (Internet domain)

With the support routines, 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 be some network dependence in a program.

The following is the program for remote login for the client:

Program-1  Client Code showing Network Dependency

To make the remote login program independent of the Internet protocol and addressing scheme, within the limitations of the current organization, it would be necessary to add a layer of routine which mask the network dependent aspects from the mainstream login code.

9.8 Byte Manipulating and swapping

To manipulate variable-length byte string and byte swapping of network addresses and value, see the following run-time library routines:

<table>
<thead>
<tr>
<th>C run-time Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call</td>
</tr>
<tr>
<td>Purpose</td>
</tr>
</tbody>
</table>
Memcpy(s2, sl, n) copy n bytes from sl to s2
Memcpy(s2, sl, 1) copy one byte
Memset(base, '0', 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
Ntoshs(val) convert 16-bit quantity from network to host byte order
Ntohl(val) convert 32-bit quantity from host to network byte order

The byte-swapping routines are provided because the operating system expects addresses to be supplied in network order. Because the hardware architecture of some machine is different from network order, programs are sometimes required to byte swap quantities. The library routines which return network addresses provided them in network order so that they may simply be copied into the structures provided to the system. This implies that users should encounter the byte-swapping problem only when interpreting network addresses. For example, to print out an Internet port:

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

For some architecture these routines are defined as null macros. Nevertheless, using them makes programs more portable.

9.9 Using the client-server model

Distributed applications are usually designed according to the client-server model, where client applications request services from a server process. This implies an asymmetry in establishment communication between the client and server.

Client and server required a shared set of conventions before service may be requested and provided. This set of conventions before service may be requested and provided. This set of conventions comprises a protocol that must be implemented at both ends of a connection. This protocol may be symmetric or asymmetric.

In a symmetric protocol either side may act as master or slave. In an asymmetric protocol, one side is always the master and the other sides 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. In either case, whether the protocol is symmetric or asymmetric, accessing a service involves a client process and a server process.

9.9.1 Server process

A server process normally listens at a well-known address for service requests. "Well-known" means that port assignments for services are usually stable and can be seen in /etc/services. Alternative scheme which use a service server can be used to eliminate a number of server processes clogging the system while remaining dormant most of the time. Such a scheme is used in the inetd program of TCP/IP.

When accessing one of the standard network services provided by TCP/IP, a client program such as ftp (TC), telnet (TC), or rlogin(TC) contacts the corresponding server at the remote host. However, it is inetd that selectively determines and process the incoming request from the client by examining the port number and protocol (either TCP or UDP) used by the requesting client program. inetd then creates the appropriate server process based on a database (/etc/inetd.conf) and "splices" the client and server together, voiding its part in the transaction. This scheme is attractive in that the inetd server process can provide a single contact point for all services, as well as carrying out the initial steps of connection establishment.

While this is an attractive possibility for standardizing access to services, it does introduce a certain amount of overhead due to the intermediate process involved. Implementations, which provide this type of service, should try to minimize the cost of client server rendezvous.

In TCP/IP, most services are accessed at well-known Internet addresses. When inetd is started at boot time, it advertises the services configured services, and listening at well-known port numbers.

A much simpler approach, used in older BSD TCP/IP implementations, is for each server program to be started separately at system boot time and to listen for requests for their own services.

Program-2 Main loop of a remote login server [Annex 9]

Once a server has established a pristine environment, it creates a socket and begins accepting service requests. The bind call is required to ensure that the server listens at its expected location. The main body of the loop is fairly simple:
An accept call block the server until a client requests service. This call could return a failure status if the call is interrupted by a signal. Therefore, the return values from accept is checked to ensure that a connection has actually been and invokes the main body of the remote login protocol processing. The socket used by the parent for queuing connection requests is closed in the child, while the socket created as result of the accept is closed in the parent. The address of the client is also handed to the doit routine because the address is required to authenticate clients.

9.9.2 Client processes

While the server the remote login service is a passive entity, listening for client connections, the client process is an active entity, which initiates a connection when invoked.

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

```c
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.

```c
p = 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.

```c
memset ((char*)&sin, 0, sizeof(sin);
memcpy (&sin.sin_addr, hp->h_addr, hp->_length);
sin.sin_port = sp->s_port;
```

A socket is created, and a connection is initiated.

```c
s = socket(hp->h_addrtype, SOCK_STREAM, 0);
```
if (s < 0) {
    perror("r login: socket");
    exit(3);
}
/*...*/
if (connect(s, (struct sockaddr *)&sin, size of (sain)) < 0) {
    perror("rlogin: connect");
    exit(4);
}

9.10 IPC Programming techniques

For most users, the IPC mechanism already described suffice to construct distributed applications. However, others need:

- out-of band data
- signal and process group
- Pseudo terminals.
- Internet address binding

9.10.1 Out-of-band data (Internet domain)

Out-of-band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out-of-band data is delivered to the user independently of normal data along with a signal. On systems this signal is SIGUSR1. In addition to the information passed, a logical mark is placed in the data stream to indicate the point at which the out-of-band data was sent. The remote logging and remote shell application use this facility to propagate signals from between client and server processes. When a signal is expected to flush any pending output from the remote process(es), all data up to the mark in the data stream is discarded.

The stream abstraction defines that the out-of-band data facilities must support the reliable delivery of at least one out-of-band message at a time. This message may contain at least one byte of data, and at least one message may be pending delivery to the user at any one time. For communications protocols which support only in-band signaling (that is, the urgent data is delivered in sequence with the normal data) the system extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving the urgent data in order and receiving it out of sequence without having to buffer all the intervening data.
To send an out-of-band message, the MSG_OOB flag is supplied to a send or sendto call; to receive out-of-band data, the MSG_OOB flag is supplied to a recv or recvfrom call. To find out if the read pointer is currently pointing at the mark in the data stream, the SIOCATAMARK ioctl is provided: ioctl (s, SIOCATAMARK,&yes);

If yes is an on return, the next read will return data after the mark. Otherwise (assuming out-of-band data has arrived), the next read will provide data sent by the client prior to transmission of the out-of-band signal.

9.10.2 Signal and process group

Each socket has an associated process group, just as a terminal does. The socket's process group of its creator, but may be redefined at a later time with the SIOCSPGRP IOCTL ioctl:

            ioctl (s, SIOCSPGRP, &pgrp);

A similar ioctl, SIOCSPGRP, is available for determining the current process group of a socket.

The presence of out-of-band data is signaled to the socket's process group through the use of the SIGUSR1 SIGNAL.

9.10.3 Pseudo terminals

Many programs do not function properly without a terminal for standard input and output. Because a socket is not a terminal, it is often necessary for processes communicating over the network to use a pseudo terminal. A pseudo terminal is actually a pair of devices, master and slave, which allow a process to serve as an active agent in communication between processes and users. Data written on the slave side of a pseudo terminal is supplied as input to a process reading from the master side. Data written on the master side is given to the slave as input. In this way, the process manipulating the master side of the pseudo terminal has control over the information read and written on the slave side.

The remote login server uses pseudo terminals for remote login sessions. A user logging in to a machine across the network is provided a shell with a slave pseudo terminal as standard input, output, and error. The server process then handle the communication between the programs invoked by the remote shell and the user's local client process. When a user sends an integer rupt or quit signal to a process executing on a remote machine, the client login program traps the signal, sends an out-of-band message to the server process which then uses the signal number (sent as the data value in the out-of-band message) to perform a kill (s) on the appropriate process group.
9.10.4 Internet address binding

Binding addresses to socket Internet domain can be fairly complex. Communicating processes are bound by an "association", which is composed of local and foreign addresses. And local and foreign ports. Port numbers are allocated out of separate spaces, one for each system and one for each domain on that system. Associations are always unique. That is, there may never be duplicate

<Protocol, local address, local port, foreign address, foreign port> tuples.

The bind system call allows a process to specify half of an association,

< Local address, local port>

While the connect and accept primitives are used to complete a socket's association. This method has several potential problems. Creating the association in two steps can generate accidental violations of the association uniqueness rule. In addition, user programs may not always know the proper values to use for the local address and local port because a host may reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

To simplify local address binding, the concept of a "wildcard" address has been provided. When an address is specified as INADDR_ANY (a manifest constant defined in netinet/in.h), the system interprets the address as "any valid address. To bind a specific port number to a socket, but leave the local address unspecified.

Socket with wild-carded local addresses may receive messages directed to the specified port number and addressed to any of the possible addresses assigned to a host. For example, if a host is on network 46 and 10 and a socket is bound as above, then when an accept call is performed, the process will be able to accept connection requests which arrive either from network 46 or network 10 if a server process wished to only allow hosts on a given network to connect to it, it would bind the address of the host on the appropriate network.

Similarly, in the Internet domain a local port may be left unspecified (specified as zero), in which case the system will select an appropriate port number for it:

    sin.sin_addr.s_addr = MYADDRESS;
    Sin.sin_port = 0;
    bind s,(struct sockaddr *) &sin, size of (sin));

The system selects the port number based on two criteria. The first is that Internet ports numbered 0 through 1023 are reserved for privileged users (that is, root).
The second is that the port number is not currently bound to some other socket. The remote shell server finds a free Internet port number in the privileged range.

The restriction on allocating ports was done to allow processes executing in a secure environment to perform authentication based on the originating address and port number.

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

```c
setsockopt (s, SOL_SOCKET, SO_REUSEADDR, (char *) 0, 0);
binder (s, (struct sockaddr *) & sin, size of (sin));
```

This call may bind local addresses, which are already in use. This does not violate the uniqueness requirement, because the system still checks at connect time to be sure any other sockets with the same local address and port do not have the same foreign address and port. (If an association already exists, the call returns the error EADDRINUSE.)

When a host is on multiple networks, the system bind local addresses some what haphazardly. Logically, one would expect the system to bind the local address associated with the network through which a peer was communicating. For instance, if the local host is connected to network 46 and 10 and the foreign host is on network 32, and traffic from network 32 is arriving via network 10, the local address to be bound would be the host's address on network 10, not network 46.

This unfortunately, is not always the case. The local address bound may appear to be chosen at random. This property of local address binding will normally be invisible to users unless the foreign host does not understand how to reach the address selected. For example, if network 46 were unknown to the host on network 46 then even though a route between the two hosts existed through network 10, a connection would fail.

9.11 List of error codes (Internet domain)

In the following list of networking error codes, the protocol or operation during which the Error is likely to occur is given at the beginning of the explanation.

I/o errors
<table>
<thead>
<tr>
<th>ERROR NAME</th>
<th>DESCRIPTION</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>EALREADY</td>
<td>Operation already in progress</td>
<td></td>
</tr>
<tr>
<td>EINPROGRESS</td>
<td>Operation now in progress</td>
<td></td>
</tr>
<tr>
<td>EWOULDBLOCK</td>
<td>Operation would block</td>
<td></td>
</tr>
<tr>
<td>EADDRINUSE</td>
<td>Address already in use</td>
<td>TCP and UDP, an attempt was made to create a socket with a port which has already been allocated</td>
</tr>
<tr>
<td>EADDRNOTAVAIL</td>
<td>Can't assign requested address</td>
<td>TCP and UDP. An attempt was made to create a socket with a network address for which no network interface exists</td>
</tr>
<tr>
<td>EAPNOSUPPORT</td>
<td>Address family not supported by protocol family</td>
<td></td>
</tr>
<tr>
<td>EDESTADDERREQ</td>
<td>Destination address required</td>
<td></td>
</tr>
<tr>
<td>EMSIZE</td>
<td>Message too long</td>
<td></td>
</tr>
<tr>
<td>ENOSTOCK</td>
<td>Socket operation on non-socket</td>
<td></td>
</tr>
<tr>
<td>EOPNOTSUPP</td>
<td>Operation not supported on socket</td>
<td></td>
</tr>
<tr>
<td>EPFNOSUPPORT</td>
<td>Protocol family not supported</td>
<td></td>
</tr>
<tr>
<td>EPROTONOSUPPORT</td>
<td>Protocol not supported</td>
<td>Creating a socket, unknown protocol or protocol not supported</td>
</tr>
<tr>
<td>EPROTOTYPE</td>
<td>Protocol wrong type for socket</td>
<td>Creating a socket. Socket type request has no supporting protocol</td>
</tr>
<tr>
<td>ESOCKTNOSUPPORT</td>
<td>Socket type not supported</td>
<td></td>
</tr>
</tbody>
</table>

9.11.1 Operational errors

<table>
<thead>
<tr>
<th>ERROR NAME</th>
<th>DESCRIPTION</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECONNABORTED</td>
<td>Software caused connection abort</td>
<td>Socket connection. The host refused service for some reason. This error is caused usually caused by a server process not being present at the requested</td>
</tr>
<tr>
<td>ECONNREFUSED</td>
<td>Connection refused</td>
<td></td>
</tr>
</tbody>
</table>

232
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECONNRESET</td>
<td>Connection reset by peer</td>
<td>TCP. The remote peer forced for session to be closed</td>
</tr>
<tr>
<td>EISCONN</td>
<td>Socket is already connected</td>
<td>TCP, IP, and UDP. An attempt was made to establish a connection on a socket that already has one or an attempt was made to send a datagram with the destination address specified and the socket is already connected. TCP, an attempt was made to establish a connection on a socket that already has one.</td>
</tr>
<tr>
<td>ENETDOWN</td>
<td>Network is down</td>
<td>Socket connection. Status information received by the client host from the underlying communication services indicates the network or the remote host is down.</td>
</tr>
<tr>
<td>ENETRESET</td>
<td>Network is down</td>
<td>Socket connection. Status information received by the client host from the underlying communication services indicates the network or the remote host is down.</td>
</tr>
<tr>
<td>ENETUNREACH</td>
<td>Network is unreachable</td>
<td>TCP, IP, and UDP. Any socket operation. The system lacks sufficient memory for an internal data structure.</td>
</tr>
<tr>
<td>ENOBUPS</td>
<td>No buffer space available</td>
<td>TCP, IP, and UDP. Any socket operation. The system lacks sufficient memory for an internal data structure.</td>
</tr>
<tr>
<td>ENOTCONN</td>
<td>Socket is not connected</td>
<td>UDP. An attempt was made to send a data gram, but no destination address is specified, and the socket has not been connected.</td>
</tr>
<tr>
<td>ESHUTDOWN</td>
<td>Cannot send after socket shutdown</td>
<td>Socket connection. After failing to establish a connection during a period of time (excessive retransmissions), the system decided there was no point in retrying any more. The cause for this</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>EHOSTDOWN</td>
<td>Host is down</td>
<td>Socket connection. Status information received by the client host from the underlying communication services indicates the network or the remote host is down.</td>
</tr>
<tr>
<td>EHOSTUNREACH</td>
<td>No route to host</td>
<td>Socket connection. These operational errors can occur either because the network or host is unknown (no route to the host or network is present) or because status information to that effect has been delivered to the client host by the underlying communication services.</td>
</tr>
<tr>
<td>ENOPROTOOPT</td>
<td>Protocol no available</td>
<td></td>
</tr>
</tbody>
</table>