Sockets allow communication between two different processes on the same or different machines. To be more precise, it's a way to talk to other computers using standard Unix file descriptors. In Unix, every I/O action is done by writing or reading a file descriptor. A file descriptor is just an integer associated with an open file and it can be a network connection, a text file, a terminal, or something else.

To a programmer, a socket looks and behaves much like a low-level file descriptor. This is because commands such as read() and write() work with sockets in the same way they do with files and pipes.

Sockets were first introduced in 2.1BSD and subsequently refined into their current form with 4.2BSD. The sockets feature is now available with most current UNIX system releases.

Where is Socket Used?

A Unix Socket is used in a client-server application framework. A server is a process that performs some functions on request from a client. Most of the application-level protocols like FTP, SMTP, and POP3 make use of sockets to establish connection between client and server and then for exchanging data.

Socket Types

There are four types of sockets available to the users. The first two are most commonly used and the last two are rarely used.

Processes are presumed to communicate only between sockets of the same type but there is no restriction that prevents communication between sockets of different types.

- **Stream Sockets** – Delivery in a networked environment is guaranteed. If you send through the stream socket three items "A, B, C", they will arrive in the same order – "A, B, C". These sockets use TCP (Transmission Control Protocol) for data transmission. If delivery is impossible, the sender receives an error indicator. Data records do not have any boundaries.

- **Datagram Sockets** – Delivery in a networked environment is not guaranteed. They're connectionless because you don't need to have an open connection as in Stream Sockets – you build a packet with the destination information and send it out. They use UDP (User Datagram Protocol).

- **Raw Sockets** – These provide 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 cryptic facilities of an existing protocol.

- **Sequenced Packet Sockets** – They are similar to a stream socket, with the exception that record boundaries are preserved. This interface is provided only as a part of the Network Systems (NS) socket abstraction, and is very important in most serious NS applications. Sequenced-packet sockets allow the user to manipulate the Sequence Packet Protocol (SPP) or Internet Datagram Protocol (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.

Before we proceed with the actual stuff, let us discuss a bit about the Network Addresses – the IP Address.

The IP host address, or more commonly just IP address, is used to identify hosts connected to the Internet. IP stands for Internet Protocol and refers to the Internet Layer of the overall network architecture of the Internet.

An IP address is a 32-bit quantity interpreted as 48-bit numbers or octets. Each IP address uniquely identifies the participating user network, the host on the network, and the class of the user network.

An IP address is usually written in a dotted-decimal notation of the form N1.N2.N3.N4, where each Ni is a decimal number between 0 and 255 decimal (00 through FF hexadecimal).

Address Classes

IP addresses are managed and created by the *Internet Assigned Numbers Authority* (IANA). There are five different address classes. You can determine which class an IP address is in by examining the first four bits of the IP address.

- **Class A** addresses begin with **0xxx**, or **1 to 126** decimal.

- **Class B** addresses begin with **10xx**, or **128 to 191** decimal.

- **Class C** addresses begin with **110x**, or **192 to 223** decimal.
- **Class D** addresses begin with **1110**, or **224 to 239** decimal.
- **Class E** addresses begin with **1111**, or **240 to 254** decimal.

Addresses beginning with **01111111**, or **127** decimal, are reserved for loopback and for internal testing on a local machine [You can test this: you should always be able to ping **127.0.0.1**, which points to yourself]; Class D addresses are reserved for multicasting; Class E addresses are reserved for future use. They should not be used for host addresses.

Example

Class	Leftmost bits	Start address	Finish address
A	0xxx	0.0.0.0	127.255.255.255
B	10xx	128.0.0.0	191.255.255.255
C	110x	192.0.0.0	223.255.255.255
D	1110	224.0.0.0	239.255.255.255
E	1111	240.0.0.0	255.255.255.255

Subnetting

Subnetting or subnetworking basically means to branch off a network. It can be done for a variety of reasons like network in an organization, use of different physical media (such as Ethernet, FDDI, WAN, etc.), preservation of address space, and security. The most common reason is to control network traffic.

The basic idea in subnetting is to partition the host identifier portion of the IP address into two parts –

- A subnet address within the network address itself; and
- A host address on the subnet.

For example, a common Class B address format is N1.N2.S.H, where N1.N2 identifies the Class B network, the 8-bit S field identifies the subnet, and the 8-bit H field identifies the host on the subnet.

Host names in terms of numbers are difficult to remember and hence they are termed by ordinary names such as Takshila or Nalanda. We write software applications to find out the dotted IP address corresponding to a given name.

The process of finding out dotted IP address based on the given alphanumeric host name is known as **hostname resolution**.

A hostname resolution is done by special software residing on high-capacity systems. These systems are called Domain Name Systems (DNS), which keep the mapping of IP addresses and the corresponding ordinary names.

The /etc/hosts File

The correspondence between host names and IP addresses is maintained in a file called *hosts*. On most of the systems, this file is found in ***/etc*** directory.

Entries in this file look like the following –

```
# This represents a comments in /etc/hosts file.
127.0.0.1       localhost
192.217.44.207  nalanda metro
153.110.31.18   netserve
153.110.31.19   mainserver centeral
153.110.31.20   samsonite
64.202.167.10   ns3.secureserver.net
64.202.167.97   ns4.secureserver.net
66.249.89.104   www.google.com
68.178.157.132  services.amrood.com
```

Note that more than one name may be associated with a given IP address. This file is used while converting from IP address to host name and vice versa.

You would not have access to edit this file, so if you want to put any host name along with IP address, then you would need to have root permission.

Unix Socket - Client Server Model

Most of the Net Applications use the Client-Server architecture, which refers to two processes or two applications that communicate with each other to

exchange some information. One of the two processes acts as a client process, and another process acts as a server.

Client Process

This is the process, which typically makes a request for information. After getting the response, this process may terminate or may do some other processing.

Example, Internet Browser works as a client application, which sends a request to the Web Server to get one HTML webpage.

Server Process

This is the process which takes a request from the clients. After getting a request from the client, this process will perform the required processing, gather the requested information, and send it to the requestor client. Once done, it becomes ready to serve another client. Server processes are always alert and ready to serve incoming requests.

Example – Web Server keeps waiting for requests from Internet Browsers and as soon as it gets any request from a browser, it picks up a requested HTML page and sends it back to that Browser.

Note that the client needs to know the address of the server, but the server does not need to know the address or even the existence of the client prior to the connection being established. Once a connection is established, both sides can send and receive information.

2-tier and 3-tier architectures

There are two types of client-server architectures –

- **2-tier architecture** – In this architecture, the client directly interacts with the server. This type of architecture may have some security holes and performance problems. Internet Explorer and Web Server work on two-tier architecture. Here security problems are resolved using Secure Socket Layer (SSL).

- **3-tier architectures** – In this architecture, one more software sits in between the client and the server. This middle software is called 'middleware'. Middleware are used to perform all the security checks and load balancing in case of heavy load. A middleware takes all requests from the client and after performing the required authentication, it

passes that request to the server. Then the server does the required processing and sends the response back to the middleware and finally the middleware passes this response back to the client. If you want to implement a 3-tier architecture, then you can keep any middleware like Web Logic or WebSphere software in between your Web Server and Web Browser.

Types of Server

There are two types of servers you can have –

- **Iterative Server** – This is the simplest form of server where a server process serves one client and after completing the first request, it takes request from another client. Meanwhile, another client keeps waiting.

- **Concurrent Servers** – This type of server runs multiple concurrent processes to serve many requests at a time because one process may take longer and another client cannot wait for so long. The simplest way to write a concurrent server under Unix is to *fork* a child process to handle each client separately.

How to Make Client

The system calls for establishing a connection are somewhat different for the client and the server, but both involve the basic construct of a socket. Both the processes establish their own sockets.

The steps involved in establishing a socket on the client side are as follows –

- Create a socket with the **socket()** system call.

- Connect the socket to the address of the server using the **connect()** system call.

- Send and receive data. There are a number of ways to do this, but the simplest way is to use the **read()** and **write()** system calls.

How to make a Server

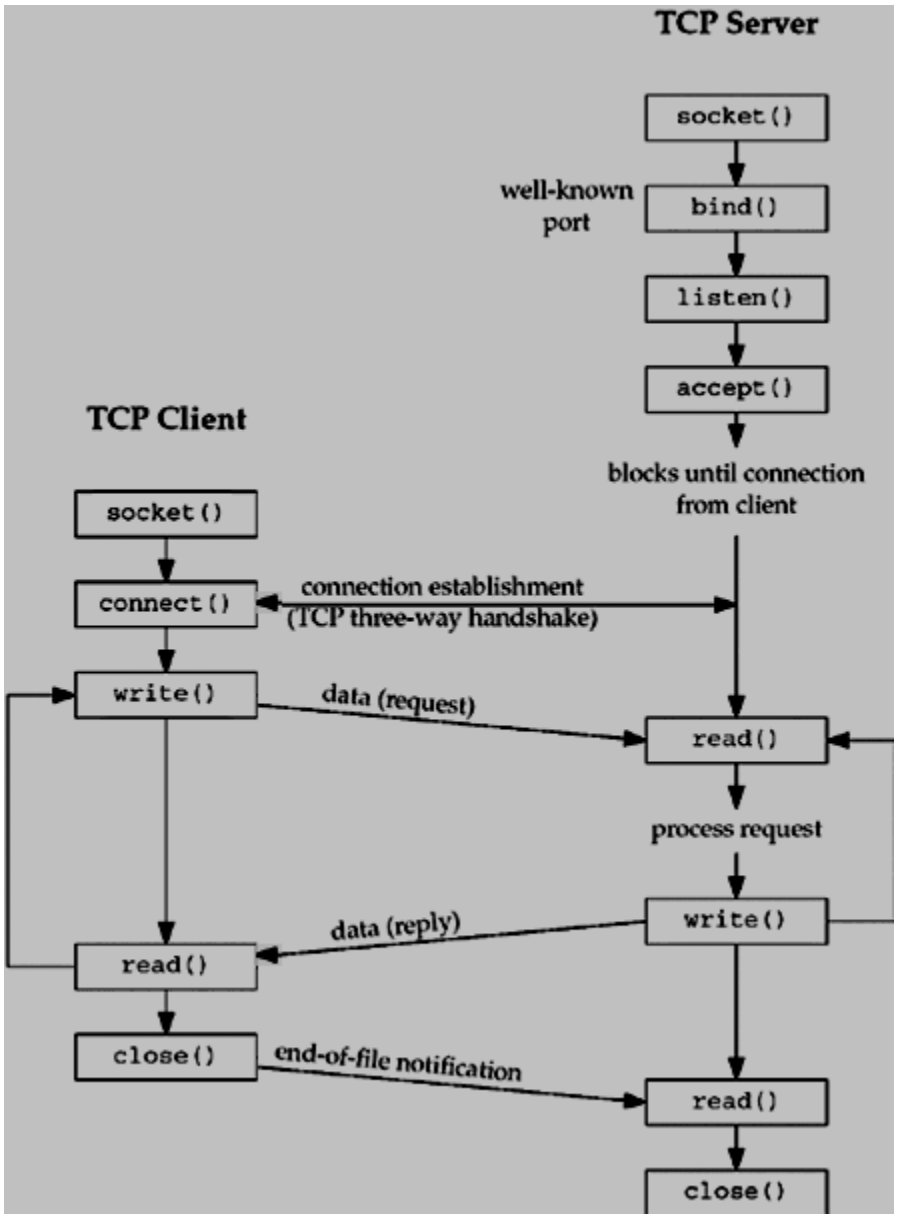
The steps involved in establishing a socket on the server side are as follows –

- Create a socket with the **socket()** system call.

- Bind the socket to an address using the **bind()** system call. For a server socket on the Internet, an address consists of a port number on the host machine.
- Listen for connections with the **listen()** system call.
- Accept a connection with the **accept()** system call. This call typically blocks the connection until a client connects with the server.
- Send and receive data using the **read()** and **write()** system calls.

Client and Server Interaction

Following is the diagram showing the complete Client and Server interaction –



Unix Socket - Structures

Various structures are used in Unix Socket Programming to hold information about the address and port, and other information. Most socket functions require a pointer to a socket address structure as an argument. Structures defined in this chapter are related to Internet Protocol Family.

sockaddr

The first structure is *sockaddr* that holds the socket information –

```
struct sockaddr {
   unsigned short   sa_family;
   char             sa_data[14];
};
```

This is a generic socket address structure, which will be passed in most of the socket function calls. The following table provides a description of the member fields –

Attribute	Values	Description
sa_family	AF_INET AF_UNIX AF_NS AF_IMPLINK	It represents an address family. In most of the Internet-based applications, we use AF_INET.
sa_data	Protocol-specific Address	The content of the 14 bytes of protocol specific address are interpreted according to the type of address. For the Internet family, we will use port number IP address, which is represented by *sockaddr_in* structure defined below.

sockaddr in

The second structure that helps you to reference to the socket's elements is as follows –

```
struct sockaddr_in {
   short int            sin_family;
   unsigned short int   sin_port;
   struct in_addr       sin_addr;
   unsigned char        sin_zero[8];
};
```

Here is the description of the member fields –

Attribute	Values	Description

sa_family	AF_INET AF_UNIX AF_NS AF_IMPLINK	It represents an address family. In most of the Internet-based applications, we use AF_INET.
sin_port	Service Port	A 16-bit port number in Network Byte Order.
sin_addr	IP Address	A 32-bit IP address in Network Byte Order.
sin_zero	Not Used	You just set this value to NULL as this is not being used.

in addr

This structure is used only in the above structure as a structure field and holds 32 bit netid/hostid.

```
struct in_addr {
   unsigned long s_addr;
};
```

Here is the description of the member fields –

Attribute	Values	Description
s_addr	service port	A 32-bit IP address in Network Byte Order.

hostent

This structure is used to keep information related to host.

```
struct hostent {
   char *h_name;
   char **h_aliases;
   int h_addrtype;
   int h_length;
   char **h_addr_list

#define h_addr  h_addr_list[0]
```

```
};
```

Here is the description of the member fields –

Attribute	Values	Description
h_name	ti.com etc.	It is the official name of the host. For example, tutorialspoint.com, google.com, etc.
h_aliases	TI	It holds a list of host name aliases.
h_addrtype	AF_INET	It contains the address family and in case of Internet based application, it will always be AF_INET.
h_length	4	It holds the length of the IP address, which is 4 for Internet Address.
h_addr_list	in_addr	For Internet addresses, the array of pointers h_addr_list[0], h_addr_list[1], and so on, are points to structure in_addr.

NOTE – h_addr is defined as h_addr_list[0] to keep backward compatibility.

servent

This particular structure is used to keep information related to service and associated ports.

```
struct servent {
   char  *s_name;
   char  **s_aliases;
   int   s_port;
   char  *s_proto;
};
```

Here is the description of the member fields –

Attribute	Values	Description

s_name	http	This is the official name of the service. For example, SMTP, FTP POP3, etc.
s_aliases	ALIAS	It holds the list of service aliases. Most of the time this will be set to NULL.
s_port	80	It will have associated port number. For example, for HTTP, this will be 80.
s_proto	TCP UDP	It is set to the protocol used. Internet services are provided using either TCP or UDP.

Tips on Socket Structures

Socket address structures are an integral part of every network program. We allocate them, fill them in, and pass pointers to them to various socket functions. Sometimes we pass a pointer to one of these structures to a socket function and it fills in the contents.

We always pass these structures by reference (i.e., we pass a pointer to the structure, not the structure itself), and we always pass the size of the structure as another argument.

When a socket function fills in a structure, the length is also passed by reference, so that its value can be updated by the function. We call these value-result arguments.

Always, set the structure variables to NULL (i.e., '\0') by using memset() for bzero() functions, otherwise it may get unexpected junk values in your structure.

Unix Socket - Ports and Services

When a client process wants to a connect a server, the client must have a way of identifying the server that it wants to connect. If the client knows the 32-bit Internet address of the host on which the server resides, it can contact that host. But how does the client identify the particular server process running on that host?

To resolve the problem of identifying a particular server process running on a host, both TCP and UDP have defined a group of well-known ports.

For our purpose, a port will be defined as an integer number between 1024 and 65535. This is because all port numbers smaller than 1024 are considered *well-known* -- for example, telnet uses port 23, http uses 80, ftp uses 21, and so on.

The port assignments to network services can be found in the file /etc/services. If you are writing your own server then care must be taken to assign a port to your server. You should make sure that this port should not be assigned to any other server.

Normally it is a practice to assign any port number more than 5000. But there are many organizations who have written servers having port numbers more than 5000. For example, Yahoo Messenger runs on 5050, SIP Server runs on 5060, etc.

Example Ports and Services

Here is a small list of services and associated ports. You can find the most updated list of internet ports and associated service at IANA - TCP/IP Port Assignments.

Service	**Port Number**	**Service Description**
echo	7	UDP/TCP sends back what it receives.
discard	9	UDP/TCP throws away input.
daytime	13	UDP/TCP returns ASCII time.
chargen	19	UDP/TCP returns characters.
ftp	21	TCP file transfer.
telnet	23	TCP remote login.

smtp	25	TCP email.
daytime	37	UDP/TCP returns binary time.
tftp	69	UDP trivial file transfer.
finger	79	TCP info on users.
http	80	TCP World Wide Web.
login	513	TCP remote login.
who	513	UDP different info on users.
Xserver	6000	TCP X windows (N.B. >1023).

Port and Service Functions

Unix provides the following functions to fetch service name from the /etc/services file.

- **struct servent *getservbyname(char *name, char *proto)** – This call takes service name and protocol name, and returns the corresponding port number for that service.
- **struct servent *getservbyport(int port, char *proto)** – This call takes port number and protocol name, and returns the corresponding service name.

The return value for each function is a pointer to a structure with the following form –

```
struct servent {
   char  *s_name;
   char  **s_aliases;
   int   s_port;
   char  *s_proto;
};
```

Here is the description of the member fields –

Attribute	Values	Description
s_name	http	It is the official name of the service. For example, SMTP, FTP POP3, etc.
s_aliases	ALIAS	It holds the list of service aliases. Most of the time, it will be set to NULL.
s_port	80	It will have the associated port number. For example, for HTTP, it will be 80.
s_proto	TCP UDP	It is set to the protocol used. Internet services are provided using either TCP or UDP.

Unfortunately, not all computers store the bytes that comprise a multibyte value in the same order. Consider a 16-bit internet that is made up of 2 bytes. There are two ways to store this value.

- **Little Endian** − In this scheme, low-order byte is stored on the starting address (A) and high-order byte is stored on the next address (A + 1).
- **Big Endian** − In this scheme, high-order byte is stored on the starting address (A) and low-order byte is stored on the next address (A + 1).

To allow machines with different byte order conventions communicate with each other, the Internet protocols specify a canonical byte order convention for data transmitted over the network. This is known as Network Byte Order.

While establishing an Internet socket connection, you must make sure that the data in the sin_port and sin_addr members of the sockaddr_in structure are represented in Network Byte Order.

Byte Ordering Functions

Routines for converting data between a host's internal representation and Network Byte Order are as follows −

Function	Description
htons()	Host to Network Short
htonl()	Host to Network Long
ntohl()	Network to Host Long
ntohs()	Network to Host Short

Listed below are some more detail about these functions –

- **unsigned short htons(unsigned short hostshort)** – This function converts 16-bit (2-byte) quantities from host byte order to network byte order.
- **unsigned long htonl(unsigned long hostlong)** – This function converts 32-bit (4-byte) quantities from host byte order to network byte order.
- **unsigned short ntohs(unsigned short netshort)** – This function converts 16-bit (2-byte) quantities from network byte order to host byte order.
- **unsigned long ntohl(unsigned long netlong)** – This function converts 32-bit quantities from network byte order to host byte order.

These functions are macros and result in the insertion of conversion source code into the calling program. On little-endian machines, the code will change the values around to network byte order. On big-endian machines, no code is inserted since none is needed; the functions are defined as null.

Program to Determine Host Byte Order

Keep the following code in a file *byteorder.c* and then compile it and run it over your machine.

In this example, we store the two-byte value 0x0102 in the short integer and then look at the two consecutive bytes, c[0] (the address A) and c[1] (the address A + 1) to determine the byte order.

```
#include <stdio.h>
```

```
int main(int argc, char **argv) {

   union {
      short s;
      char c[sizeof(short)];
   }un;

   un.s = 0x0102;

   if (sizeof(short) == 2) {
      if (un.c[0] == 1 && un.c[1] == 2)
         printf("big-endian\n");

      else if (un.c[0] == 2 && un.c[1] == 1)
         printf("little-endian\n");

      else
         printf("unknown\n");
   }
   else {
      printf("sizeof(short) = %d\n", sizeof(short));
   }

   exit(0);
}
```

An output generated by this program on a Pentium machine is as follows –

```
$> gcc byteorder.c
$> ./a.out
little-endian
$>
```

Unix Socket - IP Address Functions

Unix provides various function calls to help you manipulate IP addresses. These functions convert Internet addresses between ASCII strings (what humans prefer to use) and network byte ordered binary values (values that are stored in socket address structures).

The following three function calls are used for IPv4 addressing –

- int inet_aton(const char *strptr, struct in_addr *addrptr)
- in_addr_t inet_addr(const char *strptr)
- char *inet_ntoa(struct in_addr inaddr)

int inet_aton(const char *strptr, struct in_addr *addrptr)

This function call converts the specified string in the Internet standard dot notation to a network address, and stores the address in the structure provided. The converted address will be in Network Byte Order (bytes ordered from left to right). It returns 1 if the string was valid and 0 on error.

Following is the usage example –

```
#include <arpa/inet.h>

(...)

   int retval;
   struct in_addr addrptr

   memset(&addrptr, '\0', sizeof(addrptr));
   retval = inet_aton("68.178.157.132", &addrptr);

(...)
```

in_addr_t inet_addr(const char *strptr)

This function call converts the specified string in the Internet standard dot notation to an integer value suitable for use as an Internet address. The converted address will be in Network Byte Order (bytes ordered from left to right). It returns a 32-bit binary network byte ordered IPv4 address and INADDR_NONE on error.

Following is the usage example –

```
#include <arpa/inet.h>
```

```
(...)

   struct sockaddr_in dest;

   memset(&dest, '\0', sizeof(dest));
   dest.sin_addr.s_addr = inet_addr("68.178.157.132");

(...)
```

char *inet_ntoa(struct in_addr inaddr)

This function call converts the specified Internet host address to a string in the Internet standard dot notation.

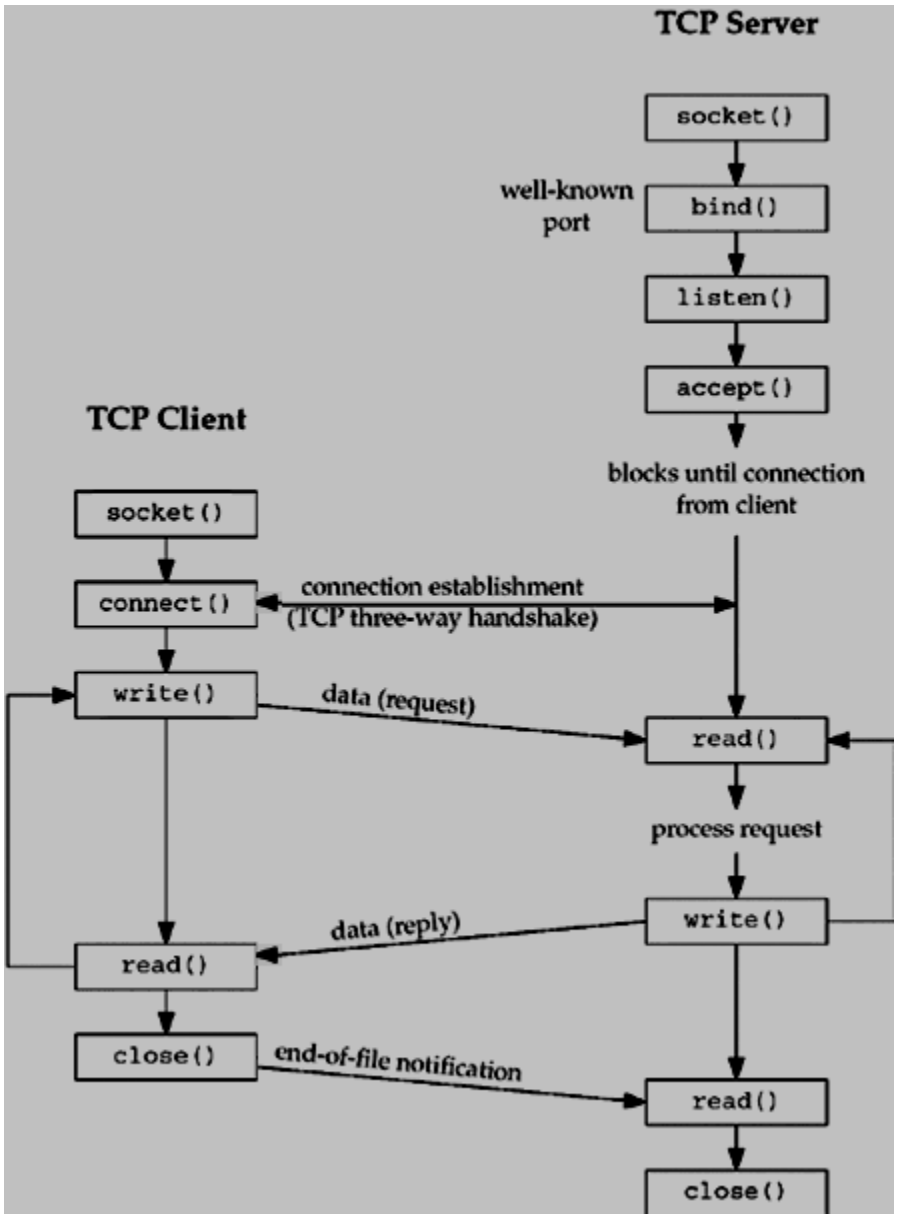
Following is the usage example –

```
#include <arpa/inet.h>

(...)

   char *ip;

   ip = inet_ntoa(dest.sin_addr);

   printf("IP Address is: %s\n",ip);

(...)
```

This chapter describes the core socket functions required to write a complete TCP client and server.

Unix Socket - Core Functions

The following diagram shows the complete Client and Server interaction –



The socket Function

To perform network I/O, the first thing a process must do is, call the socket function, specifying the type of communication protocol desired and protocol family, etc.

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


int socket (int family, int type, int protocol);
```

This call returns a socket descriptor that you can use in later system calls or -1 on error.

Parameters

family – It specifies the protocol family and is one of the constants shown below –

Family	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols
AF_ROUTE	Routing Sockets
AF_KEY	Ket socket

This chapter does not cover other protocols except IPv4.

type – It specifies the kind of socket you want. It can take one of the following values –

Type	Description
SOCK_STREAM	Stream socket
SOCK_DGRAM	Datagram socket
SOCK_SEQPACKET	Sequenced packet socket
SOCK_RAW	Raw socket

protocol – The argument should be set to the specific protocol type given below, or 0 to select the system's default for the given combination of family and type –

Protocol	Description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

The *connect* Function

The *connect* function is used by a TCP client to establish a connection with a TCP server.

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


int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

This call returns 0 if it successfully connects to the server, otherwise it returns -1 on error.

Parameters

- **sockfd** – It is a socket descriptor returned by the socket function.
- **serv_addr** – It is a pointer to struct sockaddr that contains destination IP address and port.
- **addrlen** – Set it to sizeof(struct sockaddr).

The *bind* Function

The *bind* function assigns a local protocol address to a socket. With the Internet protocols, the protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number. This function is called by TCP server only.

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


int bind(int sockfd, struct sockaddr *my_addr,int addrlen);
```

This call returns 0 if it successfully binds to the address, otherwise it returns -1 on error.

Parameters

- **sockfd** – It is a socket descriptor returned by the socket function.
- **my_addr** – It is a pointer to struct sockaddr that contains the local IP address and port.
- **addrlen** – Set it to sizeof(struct sockaddr).

You can put your IP address and your port automatically

A 0 value for port number means that the system will choose a random port, and *INADDR_ANY* value for IP address means the server's IP address will be assigned automatically.

```
server.sin_port = 0;
server.sin_addr.s_addr = INADDR_ANY;
```

NOTE – All ports below 1024 are reserved. You can set a port above 1024 and below 65535 unless they are the ones being used by other programs.

The *listen* Function

The *listen* function is called only by a TCP server and it performs two actions –

- The listen function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket.
- The second argument to this function specifies the maximum number of connections the kernel should queue for this socket.

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


int listen(int sockfd,int backlog);
```

This call returns 0 on success, otherwise it returns -1 on error.

Parameters

- **sockfd** – It is a socket descriptor returned by the socket function.
- **backlog** – It is the number of allowed connections.

The *accept* Function

The *accept* function is called by a TCP server to return the next completed connection from the front of the completed connection queue. The signature of the call is as follows –

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


int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

This call returns a non-negative descriptor on success, otherwise it returns -1 on error. The returned descriptor is assumed to be a client socket descriptor and all read-write operations will be done on this descriptor to communicate with the client.

Parameters

- **sockfd** – It is a socket descriptor returned by the socket function.
- **cliaddr** – It is a pointer to struct sockaddr that contains client IP address and port.
- **addrlen** – Set it to sizeof(struct sockaddr).

The *send* Function

The *send* function is used to send data over stream sockets or CONNECTED datagram sockets. If you want to send data over UNCONNECTED datagram sockets, you must use sendto() function.

You can use *write()* system call to send data. Its signature is as follows –

```
int send(int sockfd, const void *msg, int len, int flags);
```

This call returns the number of bytes sent out, otherwise it will return -1 on error.

Parameters

- **sockfd** – It is a socket descriptor returned by the socket function.

- **msg** – It is a pointer to the data you want to send.
- **len** – It is the length of the data you want to send (in bytes).
- **flags** – It is set to 0.

The *recv* Function

The *recv* function is used to receive data over stream sockets or CONNECTED datagram sockets. If you want to receive data over UNCONNECTED datagram sockets you must use recvfrom().

You can use *read()* system call to read the data. This call is explained in helper functions chapter.

```
int recv(int sockfd, void *buf, int len, unsigned int flags);
```

This call returns the number of bytes read into the buffer, otherwise it will return -1 on error.

Parameters

- **sockfd** – It is a socket descriptor returned by the socket function.
- **buf** – It is the buffer to read the information into.
- **len** – It is the maximum length of the buffer.
- **flags** – It is set to 0.

The *sendto* Function

The *sendto* function is used to send data over UNCONNECTED datagram sockets. Its signature is as follows –

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int
tolen);
```

This call returns the number of bytes sent, otherwise it returns -1 on error.

Parameters

- **sockfd** – It is a socket descriptor returned by the socket function.
- **msg** – It is a pointer to the data you want to send.
- **len** – It is the length of the data you want to send (in bytes).

- **flags** – It is set to 0.
- **to** – It is a pointer to struct sockaddr for the host where data has to be sent.
- **tolen** – It is set it to sizeof(struct sockaddr).

The *recvfrom* Function

The *recvfrom* function is used to receive data from UNCONNECTED datagram sockets.

```
int recvfrom(int sockfd, void *buf, int len, unsigned int flags struct sockaddr *from, int *fromlen);
```

This call returns the number of bytes read into the buffer, otherwise it returns -1 on error.

Parameters

- **sockfd** – It is a socket descriptor returned by the socket function.
- **buf** – It is the buffer to read the information into.
- **len** – It is the maximum length of the buffer.
- **flags** – It is set to 0.
- **from** – It is a pointer to struct sockaddr for the host where data has to be read.
- **fromlen** – It is set it to sizeof(struct sockaddr).

The *close* Function

The *close* function is used to close the communication between the client and the server. Its syntax is as follows –

```
int close( int sockfd );
```

This call returns 0 on success, otherwise it returns -1 on error.

Parameters

- **sockfd** – It is a socket descriptor returned by the socket function.

The *shutdown* Function

The *shutdown* function is used to gracefully close the communication between the client and the server. This function gives more control in comparison to the *close* function. Given below is the syntax of *shutdown* −

```
int shutdown(int sockfd, int how);
```

This call returns 0 on success, otherwise it returns -1 on error.

Parameters

- **sockfd** − It is a socket descriptor returned by the socket function.
- **how** − Put one of the numbers −
    - **0** − indicates that receiving is not allowed,
    - **1** − indicates that sending is not allowed, and
    - **2** − indicates that both sending and receiving are not allowed. When *how* is set to 2, it's the same thing as close().

The *select* Function

The *select* function indicates which of the specified file descriptors is ready for reading, ready for writing, or has an error condition pending.

When an application calls *recv or recvfrom*, it is blocked until data arrives for that socket. An application could be doing other useful processing while the incoming data stream is empty. Another situation is when an application receives data from multiple sockets.

Calling *recv or recvfrom* on a socket that has no data in its input queue prevents immediate reception of data from other sockets. The select function call solves this problem by allowing the program to poll all the socket handles to see if they are available for non-blocking reading and writing operations.

Given below is the syntax of *select* −

```
int select(int  nfds, fd_set  *readfds, fd_set  *writefds, fd_set *errorfds, struct timeval *timeout);
```

This call returns 0 on success, otherwise it returns -1 on error.

Parameters

- **nfds** – It specifies the range of file descriptors to be tested. The select() function tests file descriptors in the range of 0 to nfds-1

- **readfds** – It points to an object of type *fd_set* that on input, specifies the file descriptors to be checked for being ready to read, and on output, indicates which file descriptors are ready to read. It can be NULL to indicate an empty set.

- **writefds** – It points to an object of type *fd_set* that on input, specifies the file descriptors to be checked for being ready to write, and on output, indicates which file descriptors are ready to write. It can be NULL to indicate an empty set.

- **exceptfds** – It points to an object of type *fd_set* that on input, specifies the file descriptors to be checked for error conditions pending, and on output indicates, which file descriptors have error conditions pending. It can be NULL to indicate an empty set.

- **timeout** – It points to a timeval struct that specifies how long the select call should poll the descriptors for an available I/O operation. If the timeout value is 0, then select will return immediately. If the timeout argument is NULL, then select will block until at least one file/socket handle is ready for an available I/O operation. Otherwise *select* will return after the amount of time in the timeout has elapsed OR when at least one file/socket descriptor is ready for an I/O operation.

The return value from select is the number of handles specified in the file descriptor sets that are ready for I/O. If the time limit specified by the timeout field is reached, select return 0. The following macros exist for manipulating a file descriptor set –

- **FD_CLR(fd, &fdset)** – Clears the bit for the file descriptor fd in the file descriptor set *fdset.*

- **FD_ISSET(fd, &fdset)** – Returns a non-zero value if the bit for the file descriptor *fd* is set in the file descriptor set pointed to by *fdset*, and 0 otherwise.

- **FD_SET(fd, &fdset)** – Sets the bit for the file descriptor fd in the file descriptor set fdset.

- **FD_ZERO(&fdset)** – Initializes the file descriptor set fdset to have zero bits for all file descriptors.

The behavior of these macros is undefined if the fd argument is less than 0 or greater than or equal to FD_SETSIZE.

Example

```
fd_set fds;

struct timeval tv;

/* do socket initialization etc.
tv.tv_sec = 1;
tv.tv_usec = 500000;

/* tv now represents 1.5 seconds */
FD_ZERO(&fds);

/* adds sock to the file descriptor set */
FD_SET(sock, &fds);

/* wait 1.5 seconds for any data to be read from any single socket */
select(sock+1, &fds, NULL, NULL, &tv);

if (FD_ISSET(sock, &fds)) {
   recvfrom(s, buffer, buffer_len, 0, &sa, &sa_len);
   /* do something */
}
else {
   /* do something else */
}
```

Unix Socket - Helper Functions

This chapter describes all the helper functions, which are used while doing socket programming. Other helper functions are described in the chapters −**Ports and Services**, and Network **Byte Orders**.

The *write* Function

The *write* function attempts to write nbyte bytes from the buffer pointed by *buf* to the file associated with the open file descriptor, *fildes*.

You can also use *send()* function to send data to another process.

```
#include <unistd.h>


int write(int fildes, const void *buf, int nbyte);
```

Upon successful completion, write() returns the number of bytes actually written to the file associated with fildes. This number is never greater than nbyte. Otherwise, -1 is returned.

Parameters

- **fildes** – It is a socket descriptor returned by the socket function.
- **buf** – It is a pointer to the data you want to send.
- **nbyte** – It is the number of bytes to be written. If nbyte is 0, write() will return 0 and have no other results if the file is a regular file; otherwise, the results are unspecified.

The *read* Function

The *read* function attempts to read nbyte bytes from the file associated with the buffer, fildes, into the buffer pointed to by buf.

You can also use *recv()* function to read data to another process.

```
#include <unistd.h>


int read(int fildes, const void *buf, int nbyte);
```

Upon successful completion, write() returns the number of bytes actually written to the file associated with fildes. This number is never greater than nbyte. Otherwise, -1 is returned.

Parameters

- **fildes** – It is a socket descriptor returned by the socket function.
- **buf** – It is the buffer to read the information into.
- **nbyte** – It is the number of bytes to read.

The *fork* Function

The *fork* function creates a new process. The new process called the child process will be an exact copy of the calling process (parent process). The child process inherits many attributes from the parent process.

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


int fork(void);
```

Upon successful completion, fork() returns 0 to the child process and the process ID of the child process to the parent process. Otherwise -1 is returned to the parent process, no child process is created and errno is set to indicate the error.

Parameters

- **void** – It means no parameter is required.

The *bzero* Function

The *bzero* function places *nbyte* null bytes in the string *s*. This function is used to set all the socket structures with null values.

```
void bzero(void *s, int nbyte);
```

This function does not return anything.

Parameters

- **s** – It specifies the string which has to be filled with null bytes. This will be a point to socket structure variable.

- **nbyte** – It specifies the number of bytes to be filled with null values. This will be the size of the socket structure.

The *bcmp* Function

The *bcmp* function compares byte string s1 against byte string s2. Both strings are assumed to be nbyte bytes long.

```
int bcmp(const void *s1, const void *s2, int nbyte);
```

This function returns 0 if both strings are identical, 1 otherwise. The bcmp() function always returns 0 when nbyte is 0.

Parameters

- **s1** – It specifies the first string to be compared.
- **s2** – It specifies the second string to be compared.
- **nbyte** – It specifies the number of bytes to be compared.

The *bcopy* Function

The *bcopy* function copies nbyte bytes from string s1 to the string s2. Overlapping strings are handled correctly.

```
void bcopy(const void *s1, void *s2, int nbyte);
```

This function does not return anything.

Parameters

- **s1** – It specifies the source string.
- **s2v** – It specifies the destination string.
- **nbyte** – It specifies the number of bytes to be copied.

The *memset* Function

The *memset* function is also used to set structure variables in the same way as **bzero**. Take a look at its syntax, given below.

```
void *memset(void *s, int c, int nbyte);
```

This function returns a pointer to void; in fact, a pointer to the set memory and you need to caste it accordingly.

Parameters

- **s** – It specifies the source to be set.
- **c** – It specifies the character to set on nbyte places.
- **nbyte** – It specifies the number of bytes to be set.

Unix Socket - Server Examples

To make a process a TCP server, you need to follow the steps given below –

- Create a socket with the *socket()* system call.
- Bind the socket to an address using the *bind()* system call. For a server socket on the Internet, an address consists of a port number on the host machine.
- Listen for connections with the *listen()* system call.
- Accept a connection with the *accept()* system call. This call typically blocks until a client connects with the server.
- Send and receive data using the *read()* and *write()* system calls.

Now let us put these steps in the form of source code. Put this code into the file *server.c* and compile it with *gcc* compiler.

```
#include <stdio.h>
#include <stdlib.h>

#include <netdb.h>
#include <netinet/in.h>

#include <string.h>

int main( int argc, char *argv[] ) {
   int sockfd, newsockfd, portno, clilen;
   char buffer[256];
   struct sockaddr_in serv_addr, cli_addr;
   int  n;

   /* First call to socket() function */
   sockfd = socket(AF_INET, SOCK_STREAM, 0);

   if (sockfd < 0) {
      perror("ERROR opening socket");
      exit(1);
   }

   /* Initialize socket structure */
   bzero((char *) &serv_addr, sizeof(serv_addr));
```

```
portno = 5001;

serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(portno);

/* Now bind the host address using bind() call.*/
if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
   perror("ERROR on binding");
   exit(1);
}

/* Now start listening for the clients, here process will
   * go in sleep mode and will wait for the incoming connection
*/

listen(sockfd,5);
clilen = sizeof(cli_addr);

/* Accept actual connection from the client */
newsockfd = accept(sockfd, (struct sockaddr *)&cli_addr, &clilen);

if (newsockfd < 0) {
   perror("ERROR on accept");
   exit(1);
}

/* If connection is established then start communicating */
bzero(buffer,256);
n = read( newsockfd,buffer,255 );

if (n < 0) {
   perror("ERROR reading from socket");
   exit(1);
}
```

```
   printf("Here is the message: %s\n",buffer);

   /* Write a response to the client */
   n = write(newsockfd,"I got your message",18);

   if (n < 0) {
      perror("ERROR writing to socket");
      exit(1);
   }

   return 0;
}
```

Handle Multiple Connections

To allow the server to handle multiple simultaneous connections, we make the following changes in the above code −

- Put the *accept* statement and the following code in an infinite loop.
- After a connection is established, call *fork()* to create a new process.
- The child process will close *sockfd* and call *doprocessing* function, passing the new socket file descriptor as an argument. When the two processes have completed their conversation, as indicated by *doprocessing()* returning, this process simply exits.
- The parent process closes *newsockfd*. As all of this code is in an infinite loop, it will return to the accept statement to wait for the next connection.

```
#include <stdio.h>
#include <stdlib.h>

#include <netdb.h>
#include <netinet/in.h>

#include <string.h>

void doprocessing (int sock);

int main( int argc, char *argv[] ) {
```

```c
int sockfd, newsockfd, portno, clilen;
char buffer[256];
struct sockaddr_in serv_addr, cli_addr;
int n, pid;

/* First call to socket() function */
sockfd = socket(AF_INET, SOCK_STREAM, 0);

if (sockfd < 0) {
   perror("ERROR opening socket");
   exit(1);
}

/* Initialize socket structure */
bzero((char *) &serv_addr, sizeof(serv_addr));
portno = 5001;

serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(portno);

/* Now bind the host address using bind() call.*/
if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
   perror("ERROR on binding");
   exit(1);
}

/* Now start listening for the clients, here
   * process will go in sleep mode and will wait
   * for the incoming connection
*/

listen(sockfd,5);
clilen = sizeof(cli_addr);

while (1) {
```

```
      newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);

      if (newsockfd < 0) {
         perror("ERROR on accept");
         exit(1);
      }

      /* Create child process */
      pid = fork();

      if (pid < 0) {
         perror("ERROR on fork");
         exit(1);
      }

      if (pid == 0) {
         /* This is the client process */
         close(sockfd);
         doprocessing(newsockfd);
         exit(0);
      }
      else {
         close(newsockfd);
      }

   } /* end of while */
}
```

The following code seqment shows a simple implementation of *doprocessing* function.

```
void doprocessing (int sock) {
   int n;
   char buffer[256];
   bzero(buffer,256);
   n = read(sock,buffer,255);
```

```
   if (n < 0) {
      perror("ERROR reading from socket");
      exit(1);
   }

   printf("Here is the message: %s\n",buffer);
   n = write(sock,"I got your message",18);

   if (n < 0) {
      perror("ERROR writing to socket");
      exit(1);
   }

}
```

Unix Socket - Client Examples

To make a process a TCP client, you need to follow the steps given below &minus ;

- Create a socket with the *socket()* system call.
- Connect the socket to the address of the server using the *connect()* system call.
- Send and receive data. There are a number of ways to do this, but the simplest way is to use the *read()* and *write()* system calls.

Now let us put these steps in the form of source code. Put this code into the file **client.c** and compile it with **gcc** compiler.

Run this program and pass *hostname* and *port number* of the server, to connect to the server, which you already must have run in another Unix window.

```
#include <stdio.h>
#include <stdlib.h>

#include <netdb.h>
#include <netinet/in.h>
```

```c
#include <string.h>

int main(int argc, char *argv[]) {
   int sockfd, portno, n;
   struct sockaddr_in serv_addr;
   struct hostent *server;

   char buffer[256];

   if (argc < 3) {
      fprintf(stderr,"usage %s hostname port\n", argv[0]);
      exit(0);
   }

   portno = atoi(argv[2]);

   /* Create a socket point */
   sockfd = socket(AF_INET, SOCK_STREAM, 0);

   if (sockfd < 0) {
      perror("ERROR opening socket");
      exit(1);
   }

   server = gethostbyname(argv[1]);

   if (server == NULL) {
      fprintf(stderr,"ERROR, no such host\n");
      exit(0);
   }

   bzero((char *) &serv_addr, sizeof(serv_addr));
   serv_addr.sin_family = AF_INET;
   bcopy((char *)server->h_addr, (char *)&serv_addr.sin_addr.s_addr, server->h_length);
   serv_addr.sin_port = htons(portno);
```

```c
/* Now connect to the server */
if (connect(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0) {
   perror("ERROR connecting");
   exit(1);
}

/* Now ask for a message from the user, this message
   * will be read by server
*/

printf("Please enter the message: ");
bzero(buffer,256);
fgets(buffer,255,stdin);

/* Send message to the server */
n = write(sockfd, buffer, strlen(buffer));

if (n < 0) {
   perror("ERROR writing to socket");
   exit(1);
}

/* Now read server response */
bzero(buffer,256);
n = read(sockfd, buffer, 255);

if (n < 0) {
   perror("ERROR reading from socket");
   exit(1);
}

printf("%s\n",buffer);
return 0;
```

```
}
```

Unix Socket - Summary

Here is a list of all the functions related to socket programming.

Port and Service Functions

Unix provides the following functions to fetch service name from the /etc/services file.

- **struct servent *getservbyname(char *name, char *proto)** – This call takes a service name and a protocol name and returns the corresponding port number for that service.
- **struct servent *getservbyport(int port, char *proto)** – This call takes a port number and a protocol name and returns the corresponding service name.

Byte Ordering Functions

- **unsigned short htons (unsigned short hostshort)** – This function converts 16-bit (2-byte) quantities from host byte order to network byte order.
- **unsigned long htonl (unsigned long hostlong)** – This function converts 32-bit (4-byte) quantities from host byte order to network byte order.
- **unsigned short ntohs (unsigned short netshort)** – This function converts 16-bit (2-byte) quantities from network byte order to host byte order.
- **unsigned long ntohl (unsigned long netlong)** – This function converts 32-bit quantities from network byte order to host byte order.

IP Address Functions

- **int inet_aton (const char *strptr, struct in_addr *addrptr)** – This function call converts the specified string, in the Internet standard dot notation, to a network address, and stores the address in the structure provided. The converted address will be in Network Byte Order (bytes ordered from left to right). It returns 1 if the string is valid and 0 on error.

- **in_addr_t inet_addr (const char *strptr)** – This function call converts the specified string, in the Internet standard dot notation, to an integer value suitable for use as an Internet address. The converted address will be in Network Byte Order (bytes ordered from left to right). It returns a 32-bit binary network byte ordered IPv4 address and INADDR_NONE on error.

- **char *inet_ntoa (struct in_addr inaddr)** – This function call converts the specified Internet host address to a string in the Internet standard dot notation.

Socket Core Functions

- **int socket (int family, int type, int protocol)** – This call returns a socket descriptor that you can use in later system calls or it gives you -1 on error.

- **int connect (int sockfd, struct sockaddr *serv_addr, int addrlen)** – The connect function is used by a TCP client to establish a connection with a TCP server. This call returns 0 if it successfully connects to the server, otherwise it returns -1.

- **int bind(int sockfd, struct sockaddr *my_addr,int addrlen)** – The bind function assigns a local protocol address to a socket. This call returns 0 if it successfully binds to the address, otherwise it returns -1.

- **int listen(int sockfd, int backlog)** – The listen function is called only by a TCP server to listen for the client request. This call returns 0 on success, otherwise it returns -1.

- **int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen)** – The accept function is called by a TCP server to accept client requests and to establish actual connection. This call returns a non-negative descriptor on success, otherwise it returns -1.

- **int send(int sockfd, const void *msg, int len, int flags)** – The send function is used to send data over stream sockets or CONNECTED datagram sockets. This call returns the number of bytes sent out, otherwise it returns -1.

- **int recv (int sockfd, void *buf, int len, unsigned int flags)** – The recv function is used to receive data over stream sockets or CONNECTED datagram sockets. This call returns the number of bytes read into the buffer, otherwise it returns -1 on error.

- **int sendto (int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen)** – The sendto function is used to send data over UNCONNECTED datagram sockets. This call returns the number of bytes sent, otherwise it returns -1 on error.

- **int recvfrom (int sockfd, void *buf, int len, unsigned int flags struct sockaddr *from, int *fromlen)** – The recvfrom function is used to receive data from UNCONNECTED datagram sockets. This call returns the number of bytes read into the buffer, otherwise it returns -1 on error.

- **int close (int sockfd)** – The close function is used to close a communication between the client and the server. This call returns 0 on success, otherwise it returns -1.

- **int shutdown (int sockfd, int how)** – The shutdown function is used to gracefully close a communication between the client and the server. This function gives more control in comparison to close function. It returns 0 on success, -1 otherwise.

- **int select (int nfds, fd_set *readfds, fd_set *writefds, fd_set *errorfds, struct timeval *timeout)** – This function is used to read or write multiple sockets.

Socket Helper Functions

- **int write (int fildes, const void *buf, int nbyte)** – The write function attempts to write nbyte bytes from the buffer pointed to by buf to the file associated with the open file descriptor, fildes. Upon successful completion, write() returns the number of bytes actually written to the file associated with fildes. This number is never greater than nbyte. Otherwise, -1 is returned.

- **int read (int fildes, const void *buf, int nbyte)** – The read function attempts to read nbyte bytes from the file associated with the open file descriptor, fildes, into the buffer pointed to by buf. Upon successful completion, write() returns the number of bytes actually written to the file associated with fildes. This number is never greater than nbyte. Otherwise, -1 is returned.

- **int fork (void)** – The fork function creates a new process. The new process, called the child process, will be an exact copy of the calling process (parent process).

- **void bzero (void *s, int nbyte)** – The bzero function places nbyte null bytes in the string s. This function will be used to set all the socket structures with null values.

- **int bcmp (const void *s1, const void *s2, int nbyte)** – The bcmp function compares the byte string s1 against the byte string s2. Both the strings are assumed to be nbyte bytes long.

- **void bcopy (const void *s1, void *s2, int nbyte)** – The bcopy function copies nbyte bytes from the string s1 to the string s2. Overlapping strings are handled correctly.

- **void *memset(void *s, int c, int nbyte)** – The memset function is also used to set structure variables in the same way as bzero.

Advanced SOCKET Introduction

This chapter is adapted from *An Advanced 4.3 BSD Interprocess Communication Tutorial*. It provides a high-level description of the socket facilities and is designed to complement the reference material found in the TCP/IP Libraries chapter.

In this chapter, we look at:

- socket-related functions and the basic model of communication
- some of the supporting library functions you may find useful in constructing distributed applications
- the client/server model used in developing applications, including examples of the two major types of servers
- issues that sophisticated users are likely to encounter when using the socket facilities.

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.

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.

Several types of sockets are currently available:

Stream socket

Provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from the bidirectional nature of the dataflow, a pair of connected stream sockets provides an interface nearly identical to that of pipes.

Datagram socket

Supports bidirectional flow of data that isn't guaranteed 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 other than the one in which they were 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 (e.g. Ethernet).

Raw socket

Provides users access to the underlying communication protocols that support socket abstractions. These sockets are normally datagram-oriented, though their exact characteristics depend on the interface provided by the protocol. Raw sockets aren't intended for the general user; they've been provided mainly for anyone interested in developing new communication protocols or in gaining access to some of the more esoteric facilities of an existing protocol. Using raw sockets is discussed in the ``Advanced topics'' section in this chapter.

Creating sockets

To create a socket, you use the *socket()* function:

```
sock_fd = socket( domain, type, protocol);
```

This function requests that a socket be created in the specified domain and of the specified type. A particular protocol may also be requested. If you leave *protocol* unspecified (a value of 0), the system will select an appropriate protocol from those protocols that make up the communication domain and that can be used to support the requested socket type. A descriptor is returned that may be used in later functions that operate on sockets.

The domain is specified as one of the manifest constants defined in the file `<sys/socket.h>`. For the Internet domain, this constant is AF_INET. The socket types are also defined in this file; one of SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW must be specified.

To create a stream socket in the Internet domain, you could use the following call:

```
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.

The default protocol, which is chosen when the *protocol* argument to the socket call is 0, should be correct for most situations. Still, it's possible to specify a protocol other than the default (see the ``Advanced topics'' section).

> TCP is the protocol used by default to support the SOCK_STREAM abstraction, while UDP is used by default to support the SOCK_DGRAM abstraction.

A socket call may fail for any of several reasons, including:

- lack of memory (ENOBUFS)
- request for an unknown protocol (EPROTONOSUPPORT)
- request for a type of socket for which there's no supporting protocol (EPROTOTYPE).

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 domain, an association is composed of local and remote addresses, and local and remote ports. In most domains, associations must be unique; in the Internet domain, there may never be duplicate <*local address*, *local port*, *remote address*, *remote port*> tuples.

The *bind()* function allows a process to specify half of an association (<*local_address*, *local_port*>) while the *connect()* and *accept()* functions are used to complete a socket's association.

Binding names to sockets can be fairly complex. Fortunately, you don't usually have to explicitly bind an address and port number to a socket, since the *connect()* and *send()* calls will automatically bind an appropriate address when they're used with an unbound socket.

The *bind()* function has this form:

```
bind( s, name, namelen );
```

The bound name is a variable-length byte string that's interpreted by the supporting protocols. Its interpretation may vary from communication domain to communication domain (this is one of the properties that constitute the domain). As mentioned earlier, names in the Internet domain contain an Internet address and port number.

In binding an Internet address, you use the following sequence:

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

Deciding what you should place in the address requires some discussion. We'll return to the issue of formulating Internet addresses in the ``Network address functions'' section, which covers name resolution.

Establishing connections

Establishing a connection is usually asymmetric; one process is a client and the other is 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. An unrelated process can then rendezvous with the server.

The client requests services from the server by initiating a connection to the server's socket. To initiate the connection, the client uses a *connect()* call. This might appear as:

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

where *server* would contain the Internet address and the port number that the client wishes to speak to. If the client'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 (this is usually how local addresses are bound to a socket).

Errors returned when establishing a connection

An error is returned if the connection was unsuccessful (but any name automatically bound by the operating system remains). If successful, the socket is associated with the server and data transfer may begin.

Here are some of the more common errors returned when a connection attempt fails:

ETIMEDOUT

> After failing to establish a connection for a period of time, the system decided there was no point in retrying. This usually occurs because the destination host is down or because problems in the network resulted in the loss of transmissions.

ECONNREFUSED

> The host refused service for some reason (usually because a server process isn't being presented 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. Often the status returned isn't sufficient to distinguish a network's being down from a host's being down, in which case the system indicates the entire network is unreachable.

Listening on a socket

For the server to receive a client's connection, it must, after binding its socket:

- Indicate a willingness to listen for incoming connection requests.
- Actually accept the connection.

To indicate a willingness to listen for connection requests, the server uses a *listen()* call:

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

The *backlog* parameter to the *listen()* call specifies the maximum number of outstanding connections that may be queued awaiting acceptance by the server process. There is a system-defined number of maximum connections on any one queue. This prevents processes from hogging system resources by setting the *backlog* value to be very large and then ignoring all connection requests.

If a connection is requested while the server's queue is full, the connection won't be refused. Instead, the individual messages that make up the request will be ignored, forcing the client to retry. While the client retries the connection request, the server has time to make room in its pending connection queue.

If the connection is returned with the ECONNREFUSED error, the client won't be able to tell whether the server is up or not. By having the server ignore the connection request, it's still possible to get the ETIMEDOUT error back.

Accepting a connection

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);
```

When a connection is accepted, a new descriptor is returned (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*, and is then modified on return to reflect the true size of the name. If the client's name isn't of interest, the second parameter may be a NULL pointer.

An *accept()* call normally blocks. It won't return until a connection is available or until the call is interrupted by a signal to the process.

Furthermore, a process can't indicate that it will accept connections from only a specific individual or individuals. The user process takes care of considering who the connection is from and of closing down the connection if it doesn't wish to speak to the process. If the server process wants to accept connections on more than one socket or to avoid blocking on the accept call, it can do so in several ways (which are discussed in the ``Advanced topics'' section).

Data transfer

With a connection established, data may begin to flow. To send and receive data, you can choose from several calls.

If the peer entity at each end of a connection is anchored, you can send or receive a message without specifying the peer. In this case, you can use the normal *read()* and *write()* functions:

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

In addition to *read()* and *write()*, you can use the new *recv()* and *send()* calls:

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

Although *recv()* and *send()* are virtually identical to *read()* and *write()*, the extra *flags* argument is important (the flag values are defined in `<sys/socket.h>`). One or more of the following flags may be specified:

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; we won't immediately consider it here. 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.

On the other hand, the ability to preview data can be quite useful. When MSG_PEEK is specified with a *recv()* call, any data present is returned, but treated as still unread. That is, the next *read()* or *recv()* call applied to the socket will return the data previously viewed.

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

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

where *how* is 0 if you're no longer interested in reading data, 1 if no more data is to be sent, or 2 if no data is to be sent or received.

Connectionless (datagram) sockets

Up to now, we've looked mostly at sockets that follow a connection-oriented model. But there's also support for connectionless interactions that are typical of the datagram facilities found in contemporary packet-switched networks. A datagram socket provides a symmetric interface to data exchange. Although processes are still likely to be clients and servers, there's no requirement that connections be established. 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, you use the *sendto()* function:

```
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 indicate the address of the intended recipient of the message.

When an unreliable datagram interface is being used, it's unlikely that any errors will be reported to the sender. When information is present locally to recognize a message that can't be delivered (e.g. a network is unreachable), the *sendto()* call will return -1 and the global variable *errno* will contain an error number.

To receive messages on an unconnected datagram socket, you use the *recvfrom()* function:

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

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

Using *connect()* with datagrams

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, since the peer's address is simply recorded. Compare this to stream socket connections, where a *connect()* request would actually initiate the establishment of an end-to-end connection. The *accept()* and *listen()* functions aren't used with datagram sockets.

Detecting errors

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 returns true when an error indication has been received. The error is returned by the next operation, and the error status is then cleared.

Input/Output multiplexing

Many applications use the facility for multiplexing I/O requests among multiple sockets and/or files. This is done with the *select()* function (see the *C Library Reference*).

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

The *select()* function provides a *synchronous* multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions can be achieved by using the SIGIO and SIGURG signals, which are described in the ``Advanced topics'' section.

Network address functions

In the previous section, we looked at the possible need to locate and construct network addresses when using the interprocess communication facilities in a distributed environment. In this section we'll look at the C functions used to manipulate network addresses.

Mapping levels

Locating a service on a remote host requires up to three levels of mapping before the client and server may communicate:

1. The service is assigned a human-readable name (e.g. ``telnet'').
2. This name, and that of the peer host (e.g. ``sun''), must each be translated into an IP address.
3. These addresses are used to select an interface and route to the desired service.

The specifics of these three mappings tend to vary among network architectures. For instance, a network shouldn't require that hosts be named in such a way that their

physical location is known by the client host. Instead, underlying services in the network should discover the actual location of the host when the client host wishes to communicate.

Although naming hosts in a location-independent manner may induce overhead in establishing connections (because a discovery process must take place), it lets a host be physically mobile without the need to notify its potential clients of its current location. We've provided functions for mapping:

- hostnames to network addresses
- network names to network numbers
- protocol names to protocol numbers
- service names to port numbers and appropriate protocol for the server process.

When using any of these functions, you must include the `<netdb.h>` file.

Hostnames

An Internet hostname to address mapping is represented by the `hostent` structure:

```
struct hostent {
     char *h_name;
     char **h_aliases;
     int  h_addrtype;
     int  h_length;
     char **h_addr_list;
};

#define h_addr h_addr_list[0]
```

The *gethostbyname()* function takes an Internet hostname and returns a `hostent` structure, while the function *gethostbyaddr()* maps Internet host addresses into a `hostent` structure.

The official name and the public aliases of the host are returned by these functions, along with the address type (family) and a null-terminated list of variable-length addresses. This list is required because a host may have many addresses, all with 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 `/etc/hosts` file or by use of a name server (as specified in `/etc/resolv.conf`). Because of the differences between these databases and their access protocols, the information returned may differ. When

the host-table version of *gethostbyname()* is used, only one address will be returned, but all listed aliases will be included. The name-server version may return alternate addresses, but won't provide any aliases other than the one given as an argument.

Network names

As in the case of hostnames, we've provided functions for mapping network names to numbers, and numbers to names. These functions return a pointer to a `netent` structure:

```
struct netent {
     char *n_name;          /* official name of net */
     char **n_aliases;      /* alias list */
     int n_addrtype;        /* net address type */
     unsigned long n_net;   /* network number, host byte order */
};
```

The network counterparts to the host functions are *getnetbyname()*, *getnetbyaddr()*, and *getnetent()*; these network functions extract their information from the `/etc/networks` file.

Protocol names

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

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

Service names

A well-known 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. Furthermore, a service may reside on multiple ports. If this occurs, the higher-level library functions will have to be bypassed or extended.

Services available are contained in the `/etc/services` file. 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 *getservbyname()* function 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, whereas the call:

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

returns only the `telnet` server that uses the TCP protocol.

We've also provided the functions *getservbyport()* and *getservent()*.
The *getservbyport()* function has an interface similar to that provided by *getservbyname()*; an optional protocol name may be specified to qualify lookups.

Miscellaneous

With the support functions 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. Yet it's clear that purging all network dependencies is very difficult. As long as you have to supply network addresses when naming services and sockets, there'll be some network dependency in your programs.

If you wanted to make the remote login program independent of the Internet protocols and addressing scheme, you'd have to add a layer of functions that masked the network-dependent aspects from the mainstream login code. For the current facilities available in the system, this probably isn't worthwhile.

Manipulation functions

Aside from the address-related database functions, there are several other functions available in the library that you may find useful. These are intended mostly to simplify manipulation of addresses. The following table summarizes the functions for handling byte-swapping of network addresses and values.

Function	Synopsis
htonl()	Convert a 32-bit value from host byte order to network byte order
htons()	Convert a 16-bit value from host byte order to network byte order
ntohl()	Convert a 32-bit value from network byte order to host byte order
ntohs()	Convert a 16-bit value from network byte order to host byte order

We've provided the byte-swapping functions because the operating system expects addresses to be supplied in network order. On some architectures (e.g. ix86), host byte ordering differs from network byte ordering. Consequently, programs sometimes have to byte-swap quantities. The library functions that return network addresses provide them in network order so that the addresses may simply be copied into the structures provided to the system. This implies that you should encounter the byte-swapping problem only when interpreting network addresses. For example, to print out an Internet port, the following code would be required:

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

These functions are defined as null macros on machines where they aren't required.

In the QNX implementation, the host byte ordering is different from the network byte ordering.

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 asymmetry in establishing communication between the client and server, which has been examined in the ``Basics" section. In this section, we'll look more closely at the interactions between client and server, and consider some of the issues in developing client and server applications.

The client and server require a well-known set of conventions before a service may be rendered (and accepted). This set of conventions comprises a protocol that 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 role. In an asymmetric protocol, one side is immutably recognized as the master, 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 FTP, the Internet File Transfer Protocol. No matter whether the specific protocol used in obtaining a service is symmetric or asymmetric, when accessing a service there's always a client process and a server process. We'll first consider the properties of server processes, then client processes.

Servers

A server process normally listens at a well-known port 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 employing 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, this scheme has been implemented via `inetd`, the so-called ``Internet superserver.''

The `inetd` server reads a configuration file at startup and listens to a variety of ports based on the contents of the file. When a connection is requested to a port that `inetd` is listening at, `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. The `inetd` server is described in more detail in the ``Advanced topics'' section.

Remote login server

For example, the remote login server's main loop is of the following form:

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

   sp = getservbyname("login", "tcp");
   if (sp == NULL) {
       fprintf(stderr,
```

```
                "rlogind: tcp/login: unknown service\n");
        exit(1);
    }
    ...
#ifndef DEBUG
    /* Disassociate  server from  controlling terminal */
    setsid();
    ...
#endif

    /* Restricted port--see section 5 */
    sin.sin_port = sp->s_port;
    ...
    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,
                       "accept failed with errno = %d\n",
                       errno);
            continue;
        }
        if (fork() == 0) {
            close(f);
            doit(g, &from);
        }
        close(g);
    }
}
```

Look up service definition

We'll now examine the design of a server in more detail. The first step taken by the server is to 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()* function is used in later portions of the code to define the Internet port that the server will listen at for service requests (indicated by a connection).

Disassociate from controlling terminal

The next step the server takes is to disassociate itself from the controlling terminal of its invoker:

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

This step is important as the server probably won't 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. (In such situations, the *syslog()* or *Trace...* functions are typically used.)

Once a server has established its environment, it creates a socket and begins accepting service requests. The *bind()* function is required to ensure that the server listens at its expected location. You should note that the remote login server listens at a restricted port number and therefore must be run with a user ID of `root`. This concept of a restricted port number is described in the ``Advanced topics'' section.

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, "accept failed with errno = %d\n",
                   errno);
        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 (discussed in the ``Advanced topics''section). Therefore, the return value from *accept()* is checked

to ensure that a connection has actually been established and that an error report is logged via *syslog()* if an error has occurred.

With a connection in hand, the server 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 to the *doit()* function because it requires the address in authenticating clients.

Clients

Remote login client code

The normal code included in client programs, such as the remote login program, is of the following form:

```
#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);
       }
       memset( (char *)&server, 0, sizeof (server));
       memcpy( (char *)&server.sin_addr, hp->h_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);
        }
        ...
}
```

You 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.

Locate service definition

Let's take a closer look at 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);
}
```

Locate host

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);
}
```

Establish connection

With this accomplished, all that's 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 remote host and with the port number where the login process resides on the remote host:

```
memset( (char *)&server, 0, sizeof (server));
memcpy( (char *)&server.sin_addr, hp->h_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 aren't discussed here.

Connectionless (datagram) servers

Although connection-based services are the norm, some services are based on the use of datagram sockets. One in particular, the `rwhod` service, provides users with status information for hosts connected to a local area network. While predicated on the ability to broadcast information to all hosts connected to a particular network, this service is of interest as an example of how datagram sockets can be used.

Example - `rwhod`

A user on any machine running the `rwhod` server may find out the current status of a machine with the `ruptime` program. An example of the generated output is shown below.

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

Status information for each host is periodically broadcast by the `rwhod` server processes on each machine. The `rwhod` server also receives status information from other hosts 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, because all hosts must process each message, whether or not they're using an `rwhod` server. Unless such a service is sufficiently universal and is frequently used, the expense of using periodic broadcasts outweighs their simplicity.

Simplified `rwhod` server

The following is a simplified form of the `rwhod` server:

This server uses one of the *Trace...* functions instead of *syslog()*. The *syslog()* function is the historical logging facility in UNIX. The *Trace...* functions perform high-speed event logging in QNX. Choose whichever function suits your needs.

```
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) {
            Trace1(_TRACE_TCP_SERVER | SO_BROADCAST,
                   _TRACE_EXPECTED, errno);
            exit(1);
      }
      bind(s, (struct sockaddr *) &sin, sizeof (sin));
      ...
      signal(SIGALRM, onalrm);
      onalrm();
      for (;;) {
            struct whod wd;
            int cc, whod, len = sizeof (from);

            cc = recvfrom(s, (char *)&wd,
                          sizeof (struct whod), 0,
```

```
                (struct sockaddr *)&from, &len);
        if (cc <= 0) {
                if (cc < 0 && errno != EINTR)
                        Trace1(_TRACE_TCP_SERVER | RWHOD_RECV,
                                _TRACE_EXPECTED, errno);
                continue;
        }
        if (from.sin_port != sp->s_port) {
                Trace1(_TRACE_TCP_SERVER | RWHOD_BAD_FROM_PORT,
                        _TRACE_EXPECTED,
                        ntohs(from.sin_port));
                continue;
        }
        ...
        if (!verify(wd.wd_hostname)) {
                Trace1(_TRACE_TCP_SERVER |
                        RWHOD_MALFORMED_HOST_NAME,
                        _TRACE_EXPECTED,
                        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);
}
```

The `rwhod` server performs two separate tasks. The first is to receive status information broadcasts by other hosts on the network (this job is carried out in the main loop of the program); the second is to transmit information regarding the status of those hosts.

Receiving status packets

Packets received at the `rwhod` port are interrogated to ensure they've been sent by another `rwhod` server. The packets are then time-stamped with their arrival time and used to update a file indicating the status of the host. When a host hasn't been heard from for an extended period of time, the database interpretation functions assume that 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.

Transmitting status packets

The second task performed by the `rwhod` server is to transmit information regarding the status of its host. This involves periodically acquiring system-status information, packaging it up in a message, and broadcasting the message on the local network for other `rwhod` 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 can be a problem, however.

Status information must be broadcast on the local network. For networks that don't support the notion of broadcasting, another scheme must be used to simulate or replace it.

One scheme is to enumerate the known neighbors, based on the status messages received from other `rwhod` servers. Unfortunately, this requires some bootstrapping information, since a server has 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 problem is identical to that faced by the routing table management process in propagating routing status information.

The standard solution to this problem - unsatisfactory as it may be - is to inform one or more servers of known neighbors and request that they always communicate with those neighbors. If each server has at least one neighbor supplied to it, status information may then propagate through a neighbor to hosts that aren't directly neighbors. If the server can support networks that provide a broadcast capability, as well as those that don't, then networks with an arbitrary topology may share status information.

When using this standard solution, you must 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.

Any software operating in a distributed environment shouldn't contain site-dependent information. Otherwise, you need separate copies of the server at each host, making maintenance a real headache. TCP/IP for QNX attempts to isolate host-specific information from applications by providing functions that return the necessary information (e.g. the *gethostname()* function, which returns the host's official name).

There is a mechanism (using the *ioctl()* function) for finding the collection of networks that a host is directly connected to. Also, a local network broadcasting mechanism has been implemented at the socket level. By combining these two

features, a server process can broadcast in a site-independent manner on any directly connected local network that supports the notion of broadcasting.

With these mechanisms, TCP/IP for QNX can solve the problem of deciding how to propagate status information in the case of `rwhod` (or of broadcasting in general). 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 and are covered in the next section.

Advanced topics

For most users, the IPC mechanisms already described will suffice in constructing distributed applications. However, depending on your needs, you may want to take advantage of some of the features described in this section.

Out-of-band data

The stream socket abstraction includes the notion of out-of-band data. Out-of-band data is a logically independent transmission channel associated with each pair of connected stream sockets. It's delivered to the user independently of normal data.

The abstraction defines that the out-of-band data facilities must support the reliable delivery of at least one out-of-band message at a time. This message may contain at least one byte of data. At least one message may be pending delivery to the user at any one time.

Programs can choose between receiving the urgent data *in order* or receiving it *out of sequence* without having to buffer all the intervening data. Consider cases where the communications protocol supports only in-band signaling (e.g. TCP). Although the urgent data is delivered in sequence with the normal data, once urgent data has arrived, it is extracted from the normal data stream and stored separately. In this way, client programs can choose to receive the urgent data in order or out of sequence with the normal data, regardless of whether the communications protocol supports in-band or out-of-band signaling.

Peeking at out-of-band data

Using MSG_PEEK, you can ``peek'' at out-of-band data. If the socket has a process group, a SIGURG signal is generated when the protocol is notified of the out-of-band data's existence. A process can set the process group or process ID to be informed by the SIGURG signal via the appropriate *ioctl()* call, as described below for SIGIO.

If multiple sockets may have out-of-band data awaiting delivery, a *select()* call for exceptional conditions may be used to determine those sockets with such data pending. Neither the SIGURG signal nor the *select()* function indicate the actual arrival of the out-of-band data, but only give notification that it is pending.

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

For an out-of-band message to be sent, the MSG_OOB flag must be supplied to a *send()* or *sendto()* call; for out-of-band data to be received, MSG_OOB should be indicated when performing a *recvfrom()* or *recv()* call. To find out if the read pointer is currently pointing at the mark in the data stream, the SIOCATMARK *ioctl()* is provided:

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

If *yes* has a value of 1 upon 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.

Flushing terminal I/O on receipt of out-of-band data

The function used in the remote login process to flush output on receipt of an interrupt or SIGQUIT signal is shown in the following example, which reads the normal data up to the mark (to discard it), then reads the out-of-band byte:

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

        /* flush local terminal output */
        tcflush (1, TCIOFLUSH);
        for (;;) {
                if (ioctl(rem, SIOCATMARK, &mark) < 0) {
                        perror("ioctl");
                        break;
                }
                if (mark)
                        break;
```

```
                (void) read(rem, waste, sizeof (waste));
        }
        if (recv(rem, &mark, 1, MSG_OOB) < 0) {
                perror("recv");
                ...
        }
        ...
}
```

A process may also read or peek at the out-of-band data without first reading up to the mark.

This is more difficult when the underlying protocol delivers the urgent data in-band with the normal data, and only sends notification of its presence ahead of time (for example, the TCP protocol used to implement streams in the Internet domain). With such protocols, the out-of-band byte may not yet have arrived when a *recv()* is done with the MSG_OOB flag. In that case, the call will return an error of EWOULDBLOCK. What's worse, there may be enough in-band data in the input buffer that normal flow control prevents the peer from sending the urgent data until the buffer is cleared. The process must then read enough of the queued data that the urgent data may be delivered.

Some programs (e.g. `telnet`) support multiple bytes of urgent data and must handle multiple urgent signals. Such programs need to retain the position of urgent data within the stream. This treatment is available as a socket-level option, SO_OOBINLINE (see *getsockopt()* for usage). With this option, the position of urgent data (the mark) is retained, but the urgent data immediately follows the mark within the normal data stream returned without the MSG_OOB flag. Reception of multiple urgent indications causes the mark to move, but no out-of-band data is lost.

Nonblocking sockets

It's convenient to use sockets that don't block in cases where an I/O request might not complete immediately, causing the process to be suspended awaiting completion of the request. Once a socket has been created via the *socket()* call, it may be marked as nonblocking by *ioctl()* as follows:

```
#include <ioctl.h>
...
int s, on;
...
on=1;
s = socket(AF_INET, SOCK_STREAM, 0);
...
if (ioctl(s, FIONBIO, &on) < 0) {
        perror("ioctl F_SETFL, FNDELAY");
```

```
        exit(1);
}
...
```

When performing nonblocking I/O on sockets, you must be careful to check for the error codes (stored in the global variable *errno*) that occur when an operation would normally block, but the socket it was performed on is marked as nonblocking. In particular:

These functions:	**Can set *errno* to:**
accept(), send(), recv()	EWOULDBLOCK
connect()	EINPROGRESS or EALREADY
read(), write()	EAGAIN

Processes calling these functions should be prepared to deal with such return codes. If an operation such as a *send()* can't be done in its entirety, but partial writes are sensible (e.g. when using a stream socket), the data that can be sent immediately will be processed; the return value will indicate the amount actually sent.

Interrupt-driven socket I/O

The SIGIO signal allows a process to be notified via a signal when a socket (or more generally, a file descriptor) has data waiting to be read. Use of the SIGIO facility requires these steps:

1. The process must set up a SIGIO signal handler by use of the *signal()* or *sigaction()* calls.
2. The process must set the process ID or process group ID that is to receive notification of pending input to its own process ID or to the process group ID of its process group (note that the default process group of a socket is group 0). This is accomplished by use of an *ioctl()* call.
3. The process must enable asynchronous notification of pending I/O requests with another *ioctl()* call.

Use of asynchronous notification of I/O requests

The following sample code shows how to allow a process to receive information on pending I/O requests as they occur for a socket. With the addition of a handler for SIGURG, this code can also be used to prepare for receipt of SIGURG signals.

```
#include <ioctl.h>
#include <sys/types.h>
...
int io_handler(), on;
pid_t pgrp;
...
on=1;
signal(SIGIO, io_handler);

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

pgrp=getpid();
if (ioctl(s, SIOCSPGRP, &pgrp) < 0) {
        perror("ioctl F_SETOWN");
        exit(1);
}

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

if (ioctl(s, FIOASYNC, &on) < 0) {
        perror("ioctl F_SETFL, FASYNC");
        exit(1);
}
```

Signals and process groups

Because of the existence of the SIGURG and SIGIO signals, each socket has an associated process number, just as is done for terminals. This value is initialized to zero, but may be redefined at a later time with the SIOCSPGRP *ioctl()* (as was done in the above code for SIGIO).

To set the socket's process ID for signals, *positive* arguments should be given to the *ioctl()* call. To set the socket's process group for signals, *negative* arguments should be passed to *ioctl()*. Note that the process number indicates either the associated process ID or the associated process group (you can't specify both at the same time). A similar *ioctl()*, SIOCGPGRP, is available for determining the current process number of a socket.

Pseudo terminals

Many programs don't function properly without a terminal for standard input and output. Since sockets don't provide the semantics of terminals, it's often necessary to have a process communicating over the network do so through a pseudo-terminal.

The pseudo-tty driver in QNX is Dev.pty; see the QNX *Utilities Reference*.

A pseudo-terminal is actually two devices, master and slave, that let a process serve as an active agent in communication between processes. Data written on the slave side of a pseudo-terminal is supplied as input to a process reading from the master side, while data written on the master side is processed as terminal input for the slave.

In this way, the process manipulating the master side of the pseudo-terminal has control over the information read and written on the slave side as if it were manipulating the keyboard and reading the screen on a real terminal. The purpose of this abstraction is to preserve terminal semantics over a network connection. That is, the slave side of the pseudo-terminal appears as a normal terminal to any process reading from or writing to it.

For example, the remote login server uses pseudo-terminals for remote login sessions. A user logging in to a machine across the network is given a shell with a slave pseudo-terminal as its standard input, output, and error.

If the user sends a character that generates an interrupt on the remote machine that flushes terminal output, the pseudo-terminal generates a control message for the server process. The server then sends an out-of-band message to the client process to signal a flush of data at the real terminal and on the intervening data buffered in the network.

In general, the method of obtaining a pair of master and slave pseudo-terminals is to find a pseudo-terminal that isn't currently in use. The master half of a pseudo-terminal is a single open device; thus, each master may be opened in turn until an open succeeds.

The slave side of the pseudo-terminal is then opened and set to the proper terminal modes if necessary. The process then forks; the child closes the master side of the pseudo-terminal and *execs*the appropriate program. Meanwhile, the parent closes the slave side of the pseudo-terminal and begins reading and writing from the master side.

Acquiring a pseudo terminal

The following sample code illustrates how to use pseudo-terminals:

```
gotpty = 0;
for (c = 'p'; !gotpty && c <= 's'; c++) {
        line = "/dev/ptypXX";
```

```
        line[sizeof("/dev/pty")-1] = c;
        line[sizeof("/dev/ptyp")-1] = '0';
        if (stat(line, &statbuf) < 0)
                break;
        for (i = 0; i < 16; i++) {
                line[sizeof("/dev/ptyp")-1] =
                     "0123456789abcdef"[i];
                master = open(line, O_RDWR);
                if (master > 0) {
                        gotpty = 1;
                        break;
                }
        }
}
if (!gotpty) {
    Trace0(_TRACE_TCP_SERVER | NO_FREE_PORTS, _TRACE_EXPECTED);
    exit(1);
}

line[sizeof("/dev/")-1] = 't';
slave = open(line,  O_RDWR);    /*  slave is now slave side */
if (slave < 0) {
    Trace0(_TRACE_TCP_SERVER | CANT_OPEN_SLAVE_PTY,
            _TRACE_EXPECTED);
    exit(1);
}
```

In QNX, the name of the slave side of a pseudo-terminal is of the form `/dev/ttyp`*n*, where *n* is a hex digit (i.e. a single character in the range 0 through 9 or `a` through `f`). The name of the master side of a pseudo-terminal is `/dev/ptyp`*n*.

For applications that need many pseudo-terminals, there is an agreed-upon way to start `Dev.pty` that provides more banks of pseudo-terminals. For more information about `Dev.pty`, see the QNX *Utilities Reference*.

Selecting specific protocols

If the *protocol* argument to the *socket()* call is 0, *socket()* will select a default protocol to use with the returned socket of the type requested. The default protocol is usually correct; alternate choices aren't usually available.

However, when you're using ``raw'' sockets to communicate directly with lower-level protocols or hardware interfaces, the protocol argument may be important for setting up demultiplexing.

For example, raw sockets in the Internet family may be used to implement a new protocol above IP. The socket will receive packets only for the specified protocol. To

obtain a particular protocol, you determine the protocol number defined within the communication domain, using the *getprotobyname()* function, for example:

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

This would result in a socket *s* that uses a stream-based connection, but with protocol type of `newtcp` instead of the default `tcp`.

Address binding

As we mentioned in the ``Basics" section, binding addresses to sockets can be fairly complex. As a brief reminder, these associations are composed of local and remote host addresses, and local and remote port numbers.

Port numbers are allocated out of separate spaces, one for each system and one for each domain on that system. Through the *bind()* function, a process may specify half of an association (the <*local_address*,*local_port*> part), while the *connect()* and *accept()* functions are used to complete a socket's association by specifying the <*remote_address*,*remote_port*> part.

Because the association is created in two steps, the required uniqueness of the association could be violated unless care is taken. Furthermore, it's unrealistic to always expect user processes to know the proper values to use for the local address and local port (a host may reside on multiple networks and the set of allocated port numbers isn't directly accessible to a user).

To simplify local-address binding, we've provided the notion of a wildcard address. When an address is specified as INADDR_ANY, (defined in `<netinet/in.h>`), the address is interpreted as ``any valid address." For example, to bind a specific port number to a socket (but leave the local address unspecified), you might use the following code:

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in sin;
...
s = socket(AF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

Wildcard addresses

Sockets with wildcard local addresses may receive messages directed to the specified port number (sent to any of the possible addresses assigned to a host - regardless of which address was used).

For example, if a host has two IP addresses, `128.32.0.4` and `10.0.0.78`, and a socket is bound as above, the above process would be able to accept connection requests that were addressed to `128.32.0.4` or `10.0.0.78`. If a server process wanted to allow hosts only on a given network to connect to it, the server would bind the address of the host on the appropriate network.

Unspecified ports

Similarly, a process may leave the local port unspecified (by specifying it as zero), in which case a port number will be assigned. For example, to bind a specific local address to a socket, but to leave the local port number unspecified, you could use this:

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

Two criteria determine which local port number is selected. The first is that Internet ports below IPPORT_RESERVED (1024) are reserved for privileged users (i.e. the superuser); Internet ports above IPPORT_USERRESERVED (50000) are reserved for nonprivileged servers.

The second criterion is that the port number isn't currently bound to some other socket. In order to find a free Internet port number in the privileged range, you can use the *rresvport()* function as follows. It will return a stream socket in with a privileged port number:

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

```
        ...
}
```

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

Example - `rlogin`

For example, the `rlogin` command lets a user log in across a network without being asked for a password, if two conditions hold:

- The name of the system from which the user is logging in is in the `/etc/hosts.equiv` file on the system that the user is logging in to. Alternatively, the system name and the username are in the `.rhosts` file in the user's home directory.
- The user's `rlogin` process is coming from a privileged port on the machine from which the user is logging in. The port number and network address of the machine from which the user is logging in can be determined either by the *addr* result of the *accept()* call or by the *getpeername()* call.

Multiple binds to same local port

With certain applications, the algorithm used by the Socket Manager to select port numbers may be unsuitable. For example, the Internet file transfer protocol, FTP, specifies that data connections must always originate from the same local port (i.e. local from the server's point of view).

A server (e.g. `ftpd`) avoids duplicate associations because the initiating programs (e.g. `ftp`) use different remote ports (i.e. remote from the server's point of view), even though the server is accessed from the same local port (i.e. local from the server's point of view).

In this situation, the Socket Manager would typically disallow the server's binding the same local address and port number if a previous data connection's socket still existed on that port. (This would be a bad thing for servers such as `ftpd`, which always want to listen to the same well-known local port).

To override the default port selection algorithm, an option call must be performed prior to address binding:

```
...

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

With the above call, local addresses already in use may be bound. This doesn't 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 don't have the same remote address and port. If the association already exists, the error EADDRINUSE is returned.

Broadcasting and determining network configuration

By using a datagram socket, you can send broadcast packets on many networks supported by the system. The network itself must support broadcasting - the system provides no broadcast simulation in software.

Broadcast messages can place a high load on a network since they force every host on the network to service them. Consequently, the ability to send broadcast packets has been limited to sockets explicitly marked as allowing broadcasting. Broadcasting is typically used for one of two reasons:

- to find a resource on a local network without prior knowledge of its address
- for functions such as routing that require information to be sent to all accessible neighbors

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

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

The socket is marked as allowing broadcasting:

```
int on = 1;

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

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

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

The destination address of the message to be broadcast depends on the networks that the message is to be broadcast on. The Internet domain supports a shorthand notation for broadcast on the local network: the address INADDR_BROADCAST (defined in `<netinet/in.h>`).

Determining the list of addresses for all reachable neighbors requires knowledge of the networks that the host is connected to. Since this information should be obtained in a host-independent fashion and may be impossible to derive, QNX provides a method of retrieving this information from the system data structures.

The SIOCGIFCONF *ioctl()* call returns the interface configuration of a host in the form of a single `ifconf` structure. This structure contains a ``data area," which is made up of an array of `ifreq`structures, one for each network interface that the host is connected to. These structures are defined in `<net/if.h>`.

To get the base interface information, use the SIOCGIFCONF *ioctl()*. After this call, one `ifreq` structure for each network the host is connected to will be returned and *ifc.ifc_len* will have been modified to reflect the number of bytes used by the `ifreq` structures (see example below).

For each structure, there's a set of ``interface flags" that tell whether the network corresponding to that interface is up or down, point-to-point or broadcast, etc. The SIOCGIFFLAGS *ioctl()*retrieves these flags for an interface specified by an `ifreq` structure (see example below).

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

Here's an example showing how to use *ioctl()*:

if_example.c

```
#include <ioctl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <sys/socket.h>
#include <arpa/inet.h>
#include <net/if.h>
#include <netinet/in.h>

int
main (int argc, char *argv[])
{
```

```
int sock;
struct ifconf d;
struct ifreq *ifr, *end, *cur, *temp;
char buffer[128];

if ((sock= socket (AF_INET, SOCK_DGRAM, 0)) == -1) {
    perror ("socket");
    exit (EXIT_FAILURE);
}

/* temporary storage for getting broadcast address */
temp= (struct ifreq *)buffer;

d.ifc_len= 4096;
d.ifc_buf= (char *)malloc (d.ifc_len);
if (ioctl (sock, SIOCGIFCONF, &d) == -1) {
    perror ("ioctl (SIOCGIFCONF)");
    exit (EXIT_FAILURE);
}

/*
 * Note that the ifreq structure is variable length so
 * we have to step along the structure by the size of
 * the previous structure
 */
ifr= (struct ifreq *) d.ifc_req;
end= (struct ifreq *) ((char *) ifr + d.ifc_len);
while (ifr < end) {
    cur= ifr;
    /* step along the array by the size of the
       current structure */
    ifr=(struct ifreq *)((char *)ifr +
        ifr->ifr_addr.sa_len + IFNAMSIZ);

    /* if this isn't in the INET address family ignore */
    if (cur->ifr_addr.sa_family != AF_INET)
        continue;

    /* save aside the ifr structure to get the
       broadcast addr */
    memcpy(temp, cur, cur->ifr_addr.sa_len + IFNAMSIZ);

    printf ("%s (%s) ", cur->ifr_name,
            inet_ntoa(((struct sockaddr_in *)
                       &cur->ifr_addr)->sin_addr));

    /* get the flags for this interface */
    if (ioctl (sock, SIOCGIFFLAGS, (char *) cur) < 0) {
        perror ("ioctl (SIOCGIFFLAGS)");
        exit (EXIT_FAILURE);
    }

    /* if the interface is up, print out some information
       about it */
    if (cur->ifr_flags & IFF_UP) {
        if (cur->ifr_flags & IFF_BROADCAST) {
            printf ("broadcast ");
```

```
            if (ioctl(sock, SIOCGIFBRDADDR,
                      (char *)temp) != -1)
                printf("%s ",
                       inet_ntoa(((struct sockaddr_in *)
                                 &temp->ifr_addr)->sin_addr));
            else {
                perror("ioctl (SIOCGIFBRDADDR)");
                exit(EXIT_FAILURE);
            }
        }

        if (cur->ifr_flags & IFF_POINTOPOINT) {
            printf ("point-to-point dst ");
            if (ioctl(sock, SIOCGIFDSTADDR,
                      (char *)temp) != -1)
                printf("%s ",
                       inet_ntoa(((struct sockaddr_in *)
                                  &cur->ifr_addr)->sin_addr));
            else {
                perror("ioctl (SIOCGIFDSTADDR)");
                exit(EXIT_FAILURE);
            }
        }
    }
    putc ('\n', stdout);
  }

  return EXIT_SUCCESS;
}
```

To broadcast on each network, the appropriate broadcast address should be used as an argument to *sendto()* (or related function).

Received broadcast messages contain the sender's address and port, since datagram sockets are bound before a message is allowed to go out.

Socket options

You can set and get a number of options on sockets via the *setsockopt()* and *getsockopt()* functions. These options include such things as marking a socket for broadcasting, not to route, to linger on close, etc. The general forms of the functions are:

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

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

Here are the parameters to these functions:

s

the socket that the option is to be applied on

level

specifies the protocol layer that the option is to be applied to (in most cases, this is the ``socket level,'' indicated by the symbolic constant SOL_SOCKET, defined in `<sys/socket.h>`)

optname

specifies the actual option (it's a symbolic constant defined in `<sys/socket.h>`). For descriptions of the available options, see *getsockopt()* in the TCP/IP Libraries chapter.

optval

points to the value of the option (in most cases, whether the option is to be turned on or off)

optlen

points to the length of the value of the option

For *getsockopt()*, *optlen* is a value-result parameter, initially set to the size of the storage area pointed to by *optval* and modified upon return to indicate the actual amount of storage used.

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

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

int type, size;

size = sizeof (int);

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

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

`inetd` daemon

Usually invoked during the boot procedure, the `inetd` daemon determines from the `/etc/inetd.conf` file what ports to listen to (and their respective server processes). Having read this information, `inetd` proceeds to create one socket for each service it's to listen for, binding the appropriate port number to each socket.

The `inetd` daemon then performs a *select()* on all these sockets for read availability, waiting for somebody who wishes a connection to the service that corresponds to a socket. The daemon then:

1. Performs an *accept()* on the socket in question.
2. Forks.
3. *dups* the new socket to file descriptors 0 and 1 (*stdin* and *stdout*).
4. Closes other open file descriptors.
5. *execs* the appropriate server.

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

One function that may be of interest to anyone writing servers under `inetd` is the *getpeername()* function. This function returns the address of the peer (process) connected on the other end of the socket. For example, to log the Internet address in ``dot notation'' (e.g. `128.32.0.4`) of a client connected to a server under `inetd`, you might use the following code:

```
struct sockaddr_in name;
int namelen = sizeof (name);
...
if (getpeername(0, (struct sockaddr  *)&name,
                    &namelen) < 0) {
        Trace1(_TRACE_TCP_SERVER | GETPEERNAME_FAILED,
               _TRACE_EXPECTED, errno);
        exit(1);
} else
        Trace0(_TRACE_TCP_SERVER | CONNECTION_OK,
```

```
            _TRACE_EXPECTED );
    ...
```

While the *getpeername()* function is especially useful when writing programs to run with `inetd`, it can be used under other circumstances as well.