Unix Socket Programming

What is a Socket?

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.

Unix Socket - Network Addresses

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 four 8-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.

Unix Socket - Network Host Names

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.

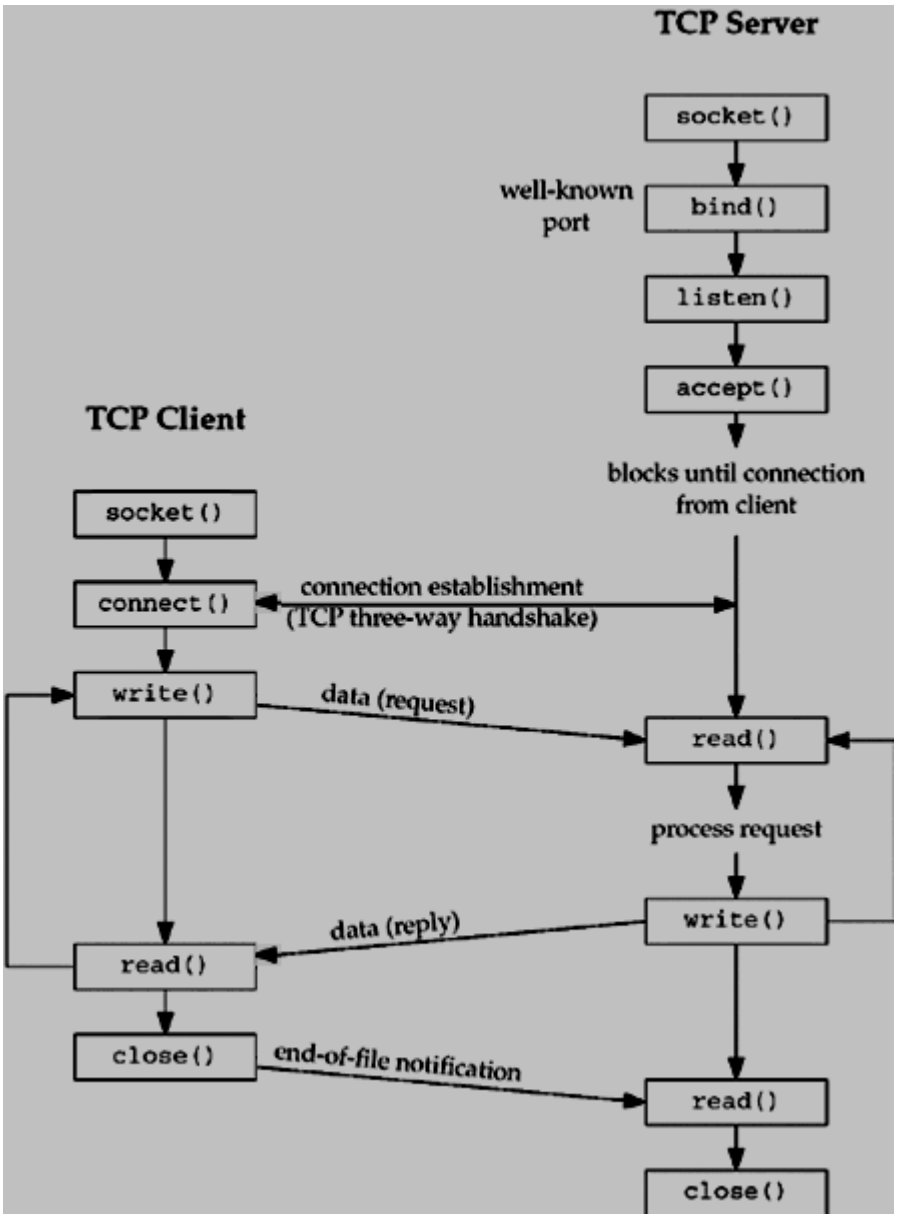
How to make a Server

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.

Unix Socket - Network Byte Orders

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.

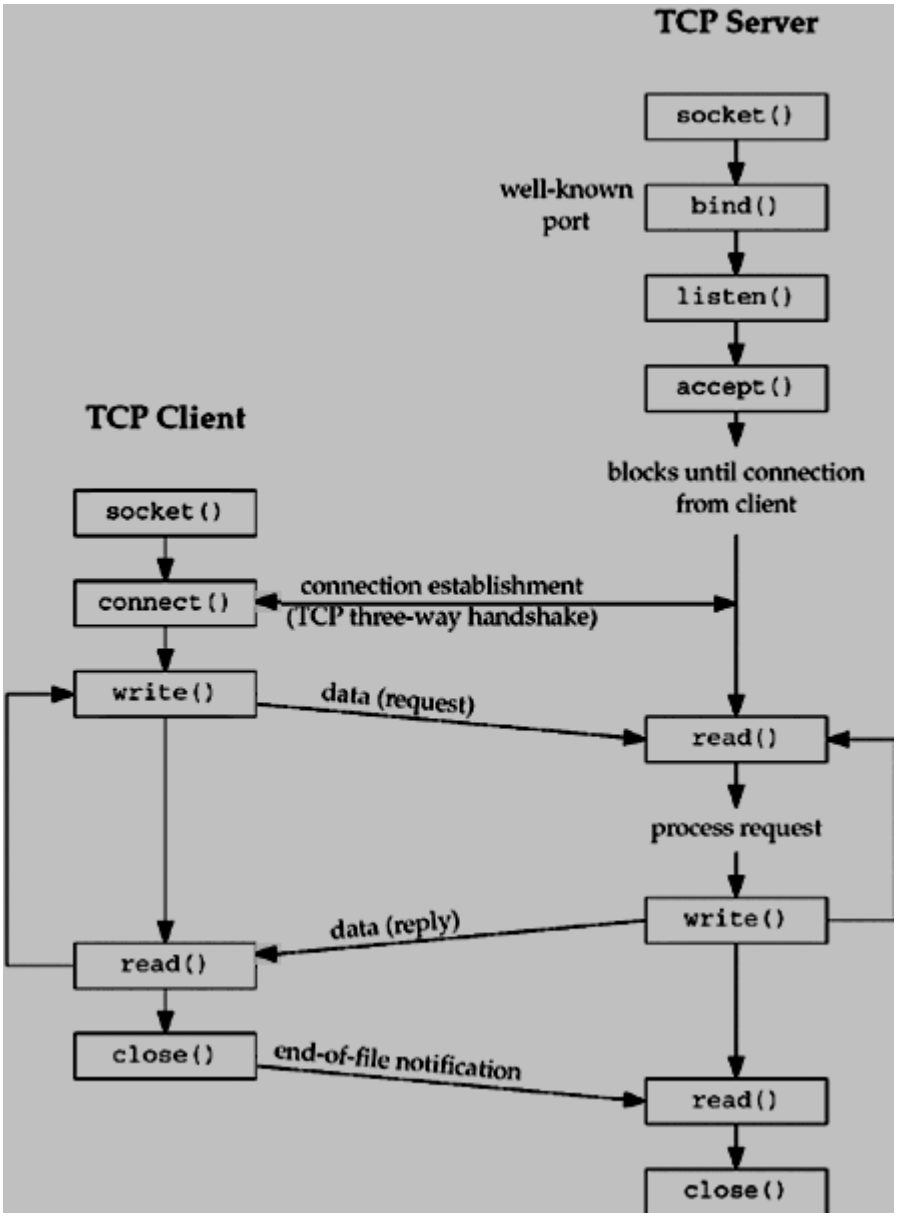
Following is the usage example −

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

(...)

  char *ip;

  ip = inet_ntoa(dest.sin_addr);

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

(...)
```

Unix Socket - Core Functions

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

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[] ) {
  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>

#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);

/* 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.