LINUX / UNIX SOCKET PROGRAMMING

Interprocess Communication: Refers to two processes which will be communicating with each other.

One of the two processes, the *client*, connects to the other process, the *server*, typically to make a request for information.

A good analogy is a person who makes a phone call to another person.

Notice that the client needs to know of the existence of and the address of the server, but the server does not need to know the address of (or even the existence of) the client prior to the connection being established.

Notice also that once a connection is established, both sides can send and receive information.

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*.

A **socket is one end of an inter-process communication channel**. The two processes each establish their own socket.

Socket Types

A socket is an abstraction of a communication endpoint. Just as they would use file descriptors to access a file, applications use socket descriptors to access sockets. Socket descriptors are implemented as file descriptors in the UNIX System. Indeed, many of the functions that deal with file descriptors, such as read and write, will work with a socket descriptor.

During socket creation, the program has to specify

- the *address domain* and

- the *socket type*.

1.address domain

Two processes can communicate with each other only if their *sockets* ***are of the same type*** *and in* ***the same domain***.

There are two widely used address domains, the ***unix domain***, in which two processes which share a common file system communicate, and the ***Internet domain***, in which two processes running on any two hosts on the Internet communicate. Each of these has its own address format.

The address of a socket in the Unix domain is a character string which is basically an entry in the file system.

The address of a socket in the Internet domain consists of the ***Internet address*** of the host machine (every computer on the Internet has a unique 32-bit address, often referred to as its IP address).

In addition, ***each socket needs a port number on that host***. ***Port numbers are 16 bit unsigned integers***.

The lower numbers are reserved in Unix for standard services. For example, the port number for the FTP server is 21. It is important that standard services be at the same port on all computers so that clients will know their addresses. However, ***port numbers above 2000 are generally available***.

2.socket type

There are two widely used socket types, ***stream sockets***, and ***datagram sockets***. **Stream sockets** treat communications as a continuous stream of characters, while datagram sockets have to read entire messages at once. Each uses its own communications protocol.

Stream sockets use ***TCP (Transmission Control Protocol)***, which is a reliable, stream oriented protocol, and datagram sockets use UDP **(Unix Datagram Protocol)**, which is unreliable and message oriented.

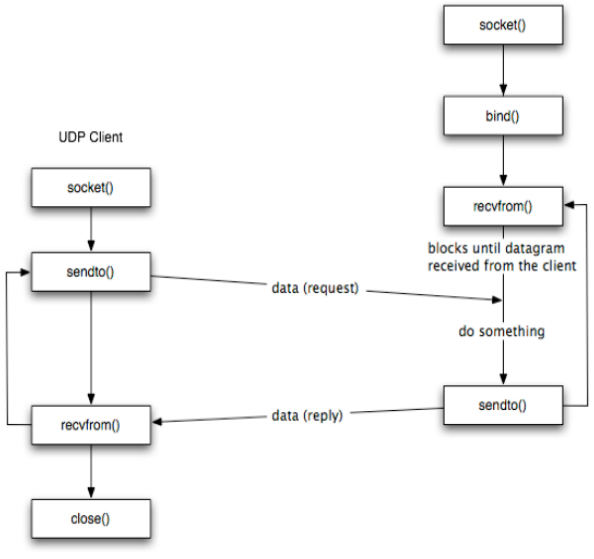
The steps involved in **establishing a socket on the client side** are as follows:

1. Create a socket with the **socket ()** system call
2. Connect the socket to the address of the server using the **connect ()** system call
3. Send and receive data. There are a number of ways to do this, but the simplest is to use the **read ()** and **write ()** system calls (system calls depend on UDP /TCP).

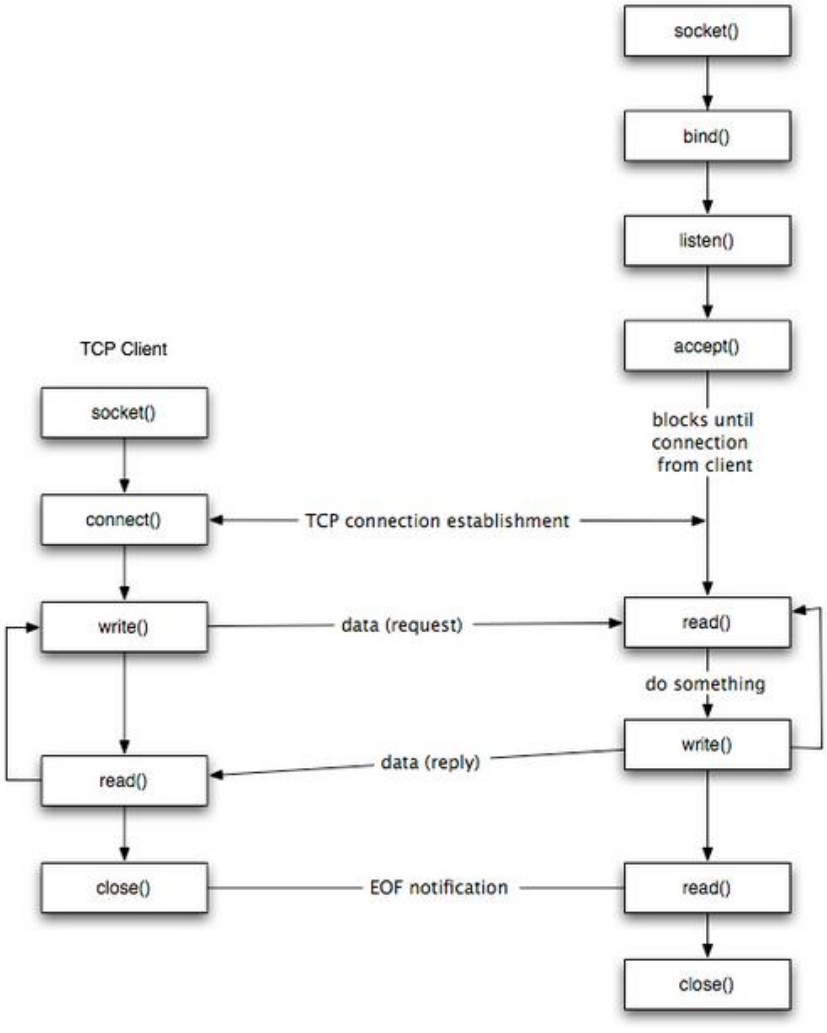
The steps involved in **establishing a socket on the server side** are as follows:

1. Create a socket with the **socket ()** system call
2. 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.
3. Listen for connections with the **listen ()** system call
4. Accept a connection with the **accept ()** system call. This call typically blocks until a client connects with the server.
5. Send and receive data (system calls depend on UDP /TCP).

Connection-less communication (Using UDP)



Connection-oriented communication (Using TCP)



Creating Sockets- socket () used at both Server and Client

To create a socket, we call the socket () function.

#include <sys/socket.h>

int socket(int domain, int type, int protocol);

Returns: *file (socket) descriptor if OK, 1 on error*

Arguments: ***3 arguments are as below***

1. ***domain- Socket communication domains***

Domain	***Description***
AF_INET	IPv4 Internet domain
AF_INET6	IPv6 Internet domain
AF_UNIX	UNIX domain

AF_UNSPEC	unspecified

Most systems define the AF_LOCAL domain also, which is an alias for AF_UNIX. The AF_UNSPEC domain is a wild card that represents "any" domain.

2. Socket types-

Type	Description
SOCK_DGRAM	fixed-length, connectionless, unreliable messages
SOCK_RAW	datagram interface to IP (optional in POSIX.1)
SOCK_SEQPACKET	fixed-length, sequenced, reliable, connection-oriented messages
SOCK_STREAM	sequenced, reliable, bidirectional, connection-oriented byte streams

Calling socket is similar to calling open. In both cases, you get a file descriptor that can be used for I/O. When you are done using the file descriptor, you call close to relinquish access to the file or socket and free up the file descriptor for reuse.

3. Protocol:

The protocol argument is **usually zero, to select the default protocol for the given domain and socket type**. When multiple protocols are supported for the same domain and socket type, we can use the protocol argument to select a particular protocol. The **default protocol for a SOCK_STREAM socket in the AF_INET communication domain is TCP** (Transmission Control Protocol).

The **default *struct sockaddr_in6 {***

sa_family_t *sin6_family;*
in_port_t sin6_port;
uint32_t sin6_flowinfo;
struct in6_addr sin6_addr;
uint32_t sin6_scope_id;
};

struct in6_addr *{*
uint8_t s6_addr[16];
};

protocol for a SOCK_DGRAM socket in the AF_INET communication domain is UDP (User Datagram Protocol).

Example: sockfd = socket(AF_INET, SOCK_STREAM, 0);

shutdown()

Communication on a socket is bidirectional. We can disable I/O on a socket with the shutdown function.

#include <sys/socket.h>

int shutdown (int sockfd, int how);

Returns: *0 if OK, 1 on error*

sockfd **–** is the socket descriptor returned by *socket( )* command.

how- is **SHUT_RD**, then **reading** from the socket is **disabled**. If how is **SHUT_WR**, then we **can't** use the socket for **transmitting data**. We can use **SHUT_RDWR** to **disable both data transmission and reception.**

The **shutdown** function allows us to deactivate a socket independently of the number of active file descriptors referencing it. Second, it is sometimes convenient to shut a socket down in one direction (**like half-close**) only. For example, we can shut a socket down for writing if we want the process we are communicating with to be able to determine when we are done transmitting data, while still allowing us to use the socket to receive data sent to us by the process.

The **close** will deallocate the network endpoint only when the last active reference is closed. This means that if we duplicate the socket (with **dup**, for example), the socket won't be deallocated until we close the last file descriptor referring to it.

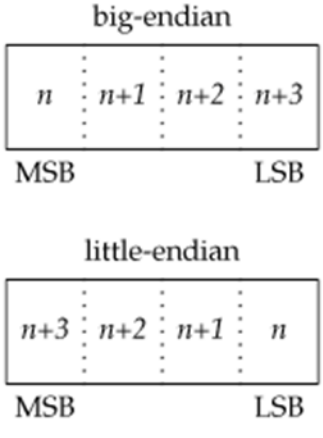
How to identify the process that we want to communicate –Using IP and Port No

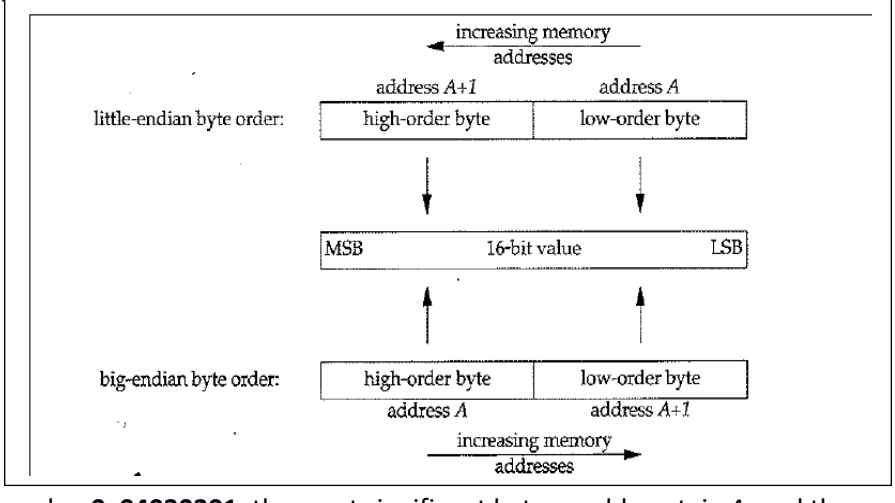
Addressing

In previous section, we learned how to create and destroy a socket. Before we learn to do something useful with a socket, **we need to learn how to identify the process that we want to communicate with**. Identifying the process has two components. The **machine's network address** helps us identify the computer on the network we wish to contact, and the **service** helps us identify the particular process on the computer.

Byte Ordering

The **byte order is a characteristic of the processor architecture**, dictating how bytes are ordered within larger data types, such as integers. Following Figure shows how the bytes within a 32-bit integer are numbered. The **big-endian** and **little-endian** are the two bytes ordering.





If we were to assign a 32-bit integer the value **0x04030201**, the most significant byte would contain 4, and the least significant byte would contain 1, regardless of the byte ordering. If we were then to cast a character pointer (cp) to the address of the integer, we would see a difference from the byte ordering. On a **little-endian processor, cp[0]** would **refer to** the least significant byte and contain **1**; **cp[3]** would refer to the most significant byte and

contain **4**. Compare that to a **big-endian processor,** where **cp[0]** would contain **4**, referring to the most significant byte, and **cp[3]** would contain **1**, referring to the least significant byte.

Four-byte Integer Example

Consider the four-byte integer 0x44332211. The "little" end byte, the lowest or least significant byte, is 0x11, and the "big" end byte, the highest or most significant byte, is 0x44. The two memory storage patterns for the four bytes are:

Four-Byte Integer: 0x44332211		
Memory address	**Big-Endian byte value**	**Little-Endian byte value**
104	11	44
103	22	33
102	33	22
101	44	11

Following Figure summarizes the byte ordering for the four platforms discussed here.

Byte order for test platforms

Operating Processor system architecture	**Byte order**
FreeBSD 5.2.1 Intel Pentium	little-endian
Linux 2.4.22 Intel Pentium	*little-endian*
Mac OS X 10.3 PowerPC	big-endian
Solaris 9 Sun SPARC	big-endian

When communicating with processes running on the same computer, we generally don't have to worry about byte ordering. However if you are communicating with processes running on the different computer then one may be following **big-endian** and another computer may be following **little-endian**, in this case communication is incompatible. So Network protocols specify a byte ordering so that heterogeneous computer systems can exchange protocol information without confusing the byte ordering. The TCP/IP protocol suite uses big-endian byte order. The byte ordering becomes visible to applications when they exchange formatted data. With TCP/IP, addresses are presented in **network byte order,** so applications sometimes need to translate them between the processor's byte order and the network byte order.

Four common functions are provided to convert between the processor byte order and the network byte order for TCP/IP applications.

Integer	Convert to		
	Network Bytes Order	Host Bytes Order	
16 bit (2 byte)	uint16_t **htons**(uint16_t **hostint16**); Returns: 16-bit integer in network byte order	uint16_t **ntohs**(uint16_t **netint16**); Returns: 16-bit integer in host byte order	The **h** is for "host" byte order, and the **n** is for "network" byte order. The **l** is for "long" (i.e., 4-byte) integer, and the **s** is for "short" (i.e., 2-byte) integer. These four functions are **defined in** <**arpa/inet.h**> although some **older systems define them in <netinet/in.h>.**
32-bit (4-byte)	uint32_t **htonl**(uint32_t **hostint32**); Returns: 32-bit integer in network byte order	uint32_t **ntohl**(uint32_t **netint32**); Returns: 32-bit integer in host byte order	

Address Formats

An address identifies a socket endpoint in a particular communication domain. The address format is specific to the particular domain.

Internet addresses are defined in **<netinet/in.h>**.

In the **IPv4 Internet domain (AF_INET)**, a **socket address is** represented by a **sockaddr_in** structure:

```
struct sockaddr_in {
        sa_family_t     sin_family;
        in_port_t       sin_port;
        struct in_addr  sin_addr;
};

        struct in_addr {
        in_addr_t       s_addr;
        };
```

****

Similarly *the IPv6 Internet domain (AF_INET6) socket address is represented by a sockaddr_in6 structure:*

struct sockaddr_in6 *{*
sa_family_t *sin6_family;*
in_port_t sin6_port;
uint32_t sin6_flowinfo;
struct in6_addr sin6_addr;
uint32_t sin6_scope_id;
};

struct in6_addr *{*
uint8_t s6_addr[16];
};

****

##########
on Linux, the sockaddr_in structure is defined as
struct sockaddr_in {
sa_family_t *sin_family;* */* address family */*
in_port_t *sin_port;* */* port number */*
struct in_addr *sin_addr;* */* IPv4 address */*
unsigned char *sin_zero[8];* */*filler */*
};
where the **sin_zero** member is a filler field that should be **set to all-zero values.**
##########
Example:

```
portno = 52632;
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(portno);
```

The value "**INADDR_ANY**" means that we will bind to any/all IP addresses that the local computer currently has.
We can get IP address and assign to **serv_addr.sin_addr.s_addr** in different ways.
Use **inet_addr**

```
in_addr_t inet_addr(const char *cp);
```

The ***inet_addr()*** function shall convert the string pointed to by cp, in the standard IPv4 dotted decimal notation, to an integer value *suitable for use as an Internet address*. Upon successful completion, inet_addr() shall return the Internet address.
Example: char *ch;
ch="127.10.10.20";
serv_addr.sin_addr.s_addr=inet_addr(ch);

Use ***inet_aton()***
inet_aton() converts the Internet host address cp from the IPv4 numbers-and-dots notation into binary form (in network byte order) and stores it in the structure that inp points to.
inet_aton() returns nonzero if the address is valid, zero if not. The address

supplied in *cp* can have one of the following forms:

a.b.c.d Each of the four numeric parts specifies a byte of the address; the bytes are assigned in left-to-right order to produce the binary address.

Example:

struct in_addr addr;

inet_ntoa(addr)

Use **gethostbyname();**

The gethostbyname() function returns a **structure of type *hostent*** for the given host *name*. Here *name* is either a hostname or an IPv4 address in standard dot notation.

The *hostent* structure is defined in *<netdb.h>* as follows:

```
struct hostent {
    char  *h_name;            /* official name of host */
    char **h_aliases;         /* alias list */
    int    h_addrtype;        /* host address type */
    int    h_length;          /* length of address */
    char **h_addr_list;       /* list of addresses */
}
#define h_addr h_addr_list[0] /* for backward compatibility */
```

The members of the *hostent* structure are:

h_name The official name of the host.

h_aliases
An array of alternative names for the host, terminated by a null pointer.

h_addrtype
The type of address; always **AF_INET** or **AF_INET6** at present.

h_length
The length of the address in bytes.

h_addr_list
An array of pointers to network addresses for the host (in network byte order), terminated by a null pointer.

h_addr The first address in *h_addr_list* for backward compatibility.

RETURN VALUE top

The gethostbyname() and gethostbyaddr() functions return the *hostent* structure or a null pointer if an error occurs. On error, the *h_errno* variable holds an error number. When non-NULL, the return value may point at static data

Associating Addresses with Sockets -Bind () used at Server

The address associated with a client's socket is of little interest, and we can let the system choose a default address for us.

For a server, however, we need to associate a **well-known address** with the server's socket on which client requests will arrive. Clients need a way to discover the address to use to contact a server, and the simplest scheme is for a server to reserve an address and register it in **/etc/services** or with a name service.

This function **binds a name to the socket**

#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr, socklen_t len);

Returns: *0 if OK, 1 on error*

Arguments: ***3 arguments are as below***

1. ***Sockfd -*** value returned by ***socket()*** when socket is created.
2. ***Sockaddr -*** In the IPv4 Internet domain (AF_INET),

```
a socket address is represented by a sockaddr_in structure:
struct sockaddr_in {
        sa_family_t     sin_family;
        in_port_t               sin_port;
        struct in_addr  sin_addr;
};

struct in_addr {
        in_addr_t   s_addr;
};
```

(**given in page 7)**

3. ***Len-*** size of ***Sockaddr*** *specifies the size (in number of bytes) of the object (object which is pointed by addr).*

Example:

struct sockaddr_in serv_addr

```
bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr))
```

Listening for any connection request from client - Listen () used at **Server**

A **server announces that it is willing to accept connect requests** by calling the listen function.

#include <sys/socket.h>

int listen(int sockfd, int backlog);

Returns: *0 if OK, 1 on error*

This **function is called in the server to establis**h a connection-based (of type *SOCKET_STREAM)* for communication.

Arguments: ***2 arguments are as below***

1. ***sockfd*** - value returned by ***socket()*** when socket is created.
2. ***backlog*** – Maximum number of connection requests that may be queued for the socket.

Once the queue is full, the system will **reject additional connect requests**, so the **backlog value must** be **chosen based on the expected load of the server and the amount of processing it must do to accept a connect request and start the service**.

Example:

```
listen(sockfd,5);
```

Connection Establishment - connect () used at client

Before we can exchange data, we need **to create a connection between the socket of the process requesting the service (the client) and the process providing the service (the server). i.e. Connecting client socket to server socket.** We use the **connect()** function to create a connection.

#include <sys/socket.h>

int connect (int sockfd, const struct sockaddr *addr, socklen_t len);

Returns: *0 if OK, 1 on error*

Arguments: ***3 arguments are as below***

The address we specify with connect is the address of the server with which we wish to communicate.

1. ***sockfd*** - value returned by ***socket()*** when socket is created.
2. ***const struct sockaddr *addr*** - **addr** is the pointer to the address of ***sockaddr*** type object that holds name of the server socket to be connected. The actual object **depends on the domain**. If domain is Internet domain then it is ***sockaddr_in***
3. ***Len-*** specifies the size (in number of bytes) of the object (object which is pointed by ***addr).***

Note:

- When we try to connect to a server, the connect request might fail for several reasons.
- The machine to which we are trying to connect must be up and running,
- The server must be bound to the address we are trying to contact, and
- There must be room in the server's pending connect queue.

Thus, applications must be able to handle connect error returns.

Note:

The connect function can also be used with a connectionless network service (SOCK_DGRAM). This might **seem like a contradiction, but it is an optimization instead**. If we call connect with a SOCK_DGRAM socket, the destination address of all messages we send is set to the address we specified in the connect call, **relieving us from having to provide the address every time we transmit a message**. In addition, we will receive datagrams only from the address we've specified.

Example:

```
connect(sockfd,(struct sockaddr *) &serv_addr,sizeof(serv_addr))
```

Server accepts connection - accept() used at server

This is **called in the server process** to establish a connection-based socket connection with a client socket. Once a server has called listen, the socket used can receive connect requests. We use the **accept()** function to retrieve a connect request and convert that into a connection.

#include <sys/socket.h>

int accept (int sockfd, struct sockaddr *restrict_addr,socklen_t *restrict_len);

Returns: *file (socket) descriptor if OK, 1 on error*

Arguments: ***3 arguments are as below***

1. sockfd - value returned by ***socket()*** when socket is created.
2. *struct sockaddr *restrict_addr-* pointer to address of **a sockaddr** type object that holds name of client socket where server socket is connected.
3. *restrict_len-* initially set to the maximum size of the object pointed to by *restrict_addr* .
   On **return, it contains size of client socket** name as pointed by *restrict_addr argument.*

Note: If ***restrict_addr*** and ***restrict_len*** are **NULL ,** the function does not pass back client's socket name to the calling process.

The function returns 1 if it fails; otherwise it **returns a new socket descriptor the server process can use to communicate with client exclusively.**

The file(socket) descriptor returned by accept is a socket descriptor that is connected to the client which has called **Connect().** This new socket descriptor has the same socket type and address family as the original socket (sockfd).
The original socket passed to **accept()** command is not associated with the connection, but instead remains available to receive additional connect requests.
If we don't care about the client's identity, we can set the **addr** and **len** parameters to **NULL.**
Otherwise, before calling **accept ()**, we need to set the **addr** parameter to a buffer large enough to hold the address and set the integer pointed to by **len** to the size of the buffer. On return, **accept ()** will fill in the client's address in the buffer and update the integer pointed to by **len** to reflect the size of the address. If no **connect** requests are pending, **accept()** will block until one arrives.
Example:

```
struct sockaddr_in serv_addr, cli_addr;

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

Data Transfer

Since a socket endpoint is represented as a file descriptor, we can use **read** and **write** to communicate with a socket, as long as it is connected. Recall that a datagram socket can be "connected" if we set the default peer address using the **connect** function. Using read and write with socket descriptors is significant, because it means that we can pass socket descriptors to functions that were originally designed to work with local files. We can also arrange to pass the socket descriptors to child processes that execute programs that know nothing about sockets.

Although **we can exchange data using read and write, that is about all we can do with these two functions**. **If we want to specify options, receive packets from multiple clients, or send out-of-band data, we need to use one of the six socket functions** designed for data transfer.

Three functions are available for sending data, and three are available for receiving data. First, we'll look at the ones used to send data. The simplest one is **send**(). It is similar to **write**(), but allows us to specify flags to change how the data we want to transmit is treated.

Sending data

1. send()

#include <sys/socket.h>

ssize_t send(int sockfd, const void *buf, size_t nbytes, int flags);

***Returns**: number of bytes sent if OK, 1 on error*

***Arguments:** 4 arguments are as below*

This function sends a message, contained in **buf** of size **nbytes**, to a socket that is connected to the socket as designated by sockfd.

i. sockfd - value returned by ***socket()*** when socket is created.
ii. buf- pointer to container which holds message to be sent.
iii. nbytes- size of the message.
iv. Flags- The ***flags*** argument is normally assigned a 0 value, it can also be set to MSG_OOB (other flasg ae which means that the message contained in *buf* should be sent as out-of-band message.

Note:

There are two types of messages –*regular* and out-of-band messages. By default, every message transmitted by a socket is a *regular* message unless it is explicitly tagged as out-of-band messages. If there is more than one messages of a given type sent from a socket, these are received by another socket in a FIFO order. The recipient socket may select which type of messages it wishes to receive. Out-of-band messages should be used as emergency messages only.

TCP supports out-of-band data, but **UDP doesn't**. **TCP** refers to out-of-band data as **"urgent" data.**

Note:If **send returns success**, it **doesn't necessarily mean that the process at the other end of the connection receives the data**. All we are guaranteed is that when send succeeds, the data has been delivered to the network drivers without error.

2. sendto()

The *sendto* function is similar to *send*. The difference is that *sendto* allows us to specify a destination address to be used with connectionless sockets.

#include <sys/socket.h>

ssize_t sendto(int sockfd, const void *buf, size_t nbytes, int flags, const struct sockaddr *destaddr, socklen_t destlen);

***Returns:** number of bytes sent if OK, -1 on error.*

***Arguments:** 4arguments are as below*

The function is same as the *send*, except that the calling process also specifies the address of the recipient socket name via ***destaddr*** and ***destlen.***

1. ***2. 3***. **Sockfd, buf, nbytes, flags** arguments are same as ***send()*** .
4. **destaddr** -The ***destaddr*** is a pointer to the object that contains the name of a recipient socket.
5. **destlen-** The ***destlen*** contains the number of bytes in the object pointed to by the ***destaddr.***

NOTE:

With a connection-oriented socket, the destination address is ignored, as the destination is implied by the connection (i.e. see NOTE part in connect()). With a connectionless socket, we can't use send unless the destination address is first set by calling connect, so **sendto()** gives us an alternate way to send(using **send()**) a message.

{ check yourself. **3.sendmsg()**

NOTE: We have one more choice when transmitting data over a socket. We can call sendmsg with a msghdr structure to specify multiple buffers from which to transmit data, similar to the writev function

#include <sys/socket.h>

*ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);*

}

Receiving data

1. recv()

The **recv()** function is similar to read, but allows us to specify some options to control how we receive the data.

#include <sys/socket.h>

ssize_t recv(int sockfd, void *buf, size_t nbytes, int flags);

Returns: *length of message received in bytes, 0 if no messages are available and peer has done an orderly shutdown, or 1 on error*

Arguments: ***4 arguments are as below***

This function sends a message, contained in **buf** of size **nbytes**, to a socket that is connected to the socket as designated by sockfd.

1. **sockfd** - value returned by ***socket()*** when socket is created.
2. **buf-** pointer to container which used to holds message being received.
3. **nbytes-** size of the message.
4. **Flags-** The ***flags*** argument is normally assigned a **0** value, it can also be set to **MSG_OOB** which means that the message received is out-of-band message.

2. recvfrom()

If we are interested in the identity of the sender, we can use **recvfrom ()** to obtain the source address from which the data was sent.

#include <sys/socket.h>

ssize_t recvfrom(int sockfd, void *restrict buf, size_t len, int flags, struct sockaddr *restrict addr,socklen_t *restrict addrlen);

Returns: *length of message received in bytes, 0 if no messages are available and peer has done an orderly shutdown, or 1 on error*

If ***addr*** is non-null, it will contain the **address of the socket endpoint from which the data was sent**.
When calling **recvfrom**, we need to set the ***addrlen*** parameter to point to an integer containing the size in bytes of the socket buffer to which *addr* points. On return, the integer is set to the actual size of the address in bytes.
Because it allows us to retrieve the address of the sender, **recvfrom** is usually used with **connectionless sockets**. Otherwise, **recvfrom** behaves identically to **recv**.

{ check yourself. 3.recvmsg()

#include <sys/socket.h>

*ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);*

Returns: length of message in bytes, 0 if no messages are available and peer has done an orderly shutdown, or 1 on error

}