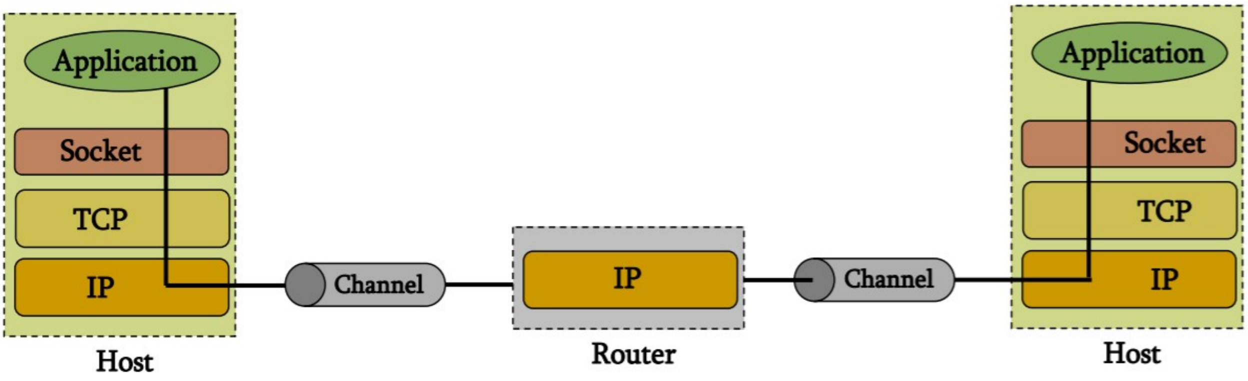
# Socket Programming in C

## 1 Introduction Socket

### 1.1 Definition

- A **socket** is an abstraction **through which an application may send and receive data**. It provide generic access to interprocess communication services.
- In otherwords, we can say, a **socket is an interface between an application process and transport layer.**
- The application process can send/receive messages to/from another application process (local or remote) via a socket.
- In Unix, **a socket is a file descripter.**
- When Unix programs do any sort of I/O, they do it by reading or writing to a file descriptor. **A file descriptor is simply an integer associated with an open file.** But, that file can be a network connection, a FIFO, a pipe, a terminal, a real on-the-disk file, or just about anything else. Everything in Unix is a file! So when you want to communicate with another program over the Internet you're gonna do it through a file descriptor, the socket.
- **Socket** is the **standard API for networking.**



- A **socket** is **uniquely identified** by
  - an **internet address** (32 bit)
  - an end-to-end **protocol** (e.g. TCP or UDP)
  - a **port number** (16 bit)

### 1.2 Socket Address Domain & Its Type

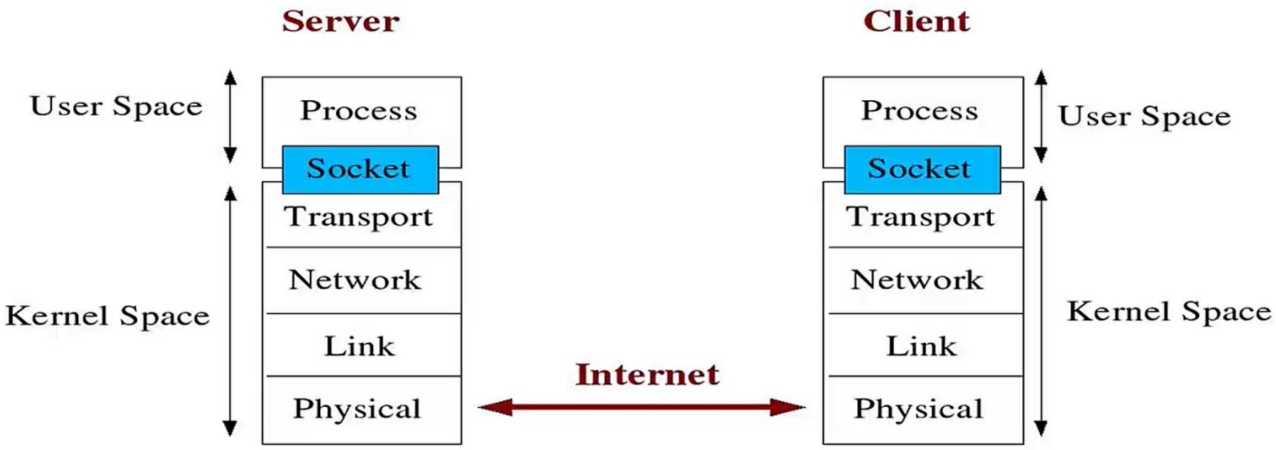
- When a socket is created, the program has to specify the address domain and the socket type.
- Two processes can communicate with each other only if their sockets are of the same type and in the same domain.
- There are **two widely used address domains,** they are ***unix domain*** & ***internet domain.***
  a) ***Unix Domain:*** The domain, in which two processes which share a common file system communicate. The address of a socket in the Unix domain is a character string which is basically an entry in the file system.
  b) ***Internet Domain:*** The domain, in which two processes running on any two hosts on the Internet communicate. 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. In programming we refer this as **AF_INET** (IPv4 protocol) , **AF_INET6** (IPv6 protocol), **PF_UNIX** used for Local communication, File addresses.

- There are **two types** of (TCP/IP) **sockets used in Internet domain.**
  - a) ***Stream sockets:*** It uses **TCP**. It is some times called connectionoriented socket. In programming we refer this as **"SOCK_STREAM"**.
  - b) ***Datagram sockets:*** It uses **UDP**. It is sometimes called connectionless socket. In programming we refer this as s **"SOCK_DGRAM"**.

## 1.3 Client-Server communication

- **Server**
  - passively waits for and responds to clients
  - **passive socket or server socket**
- **Client**
  - initiates the communication
  - must know the address and the port of the server
  - **active socket or client socket**

**(Socket Description)**



## 1.4 Byte Ordering

- **There are two types of byte ordering**
  - a) **Big-Endian (Network Byte Order) :** Higher order byte of the number is stored in memory at the lowest address.
  - b) **Litte-Endian :** Lower order byte of the number is stored in memory at the lowest address. Some hosts use this ordering. Network stack (TCP/IP) excepts Network Byte Order.
- **Port numbers** and **IP Addresses** are represented by multi-byte data types which are placed in packets for the purpose of routing and multiplexing.
- **Port numbers** are **two bytes** (16 bits) and **IP4 addresses** are **4 bytes (**32 bits).
- A problem arises when transferring multi-byte data types between different architectures. Say Host A uses a “big-endian” architecture and sends a packet across the network to Host B which uses a “little-endian” architecture. If Host B looks at the address to see if the packet is for him/her (choose a gender!), it will interpret the bytes in the opposite order and will wrongly conclude that it is not his/her packet.
- The **Internet uses big-endian** and we call it the **network-byte-order.**
- We have the following functions to convert host-byte-ordered values into network-byte-ordered values and vice versa:

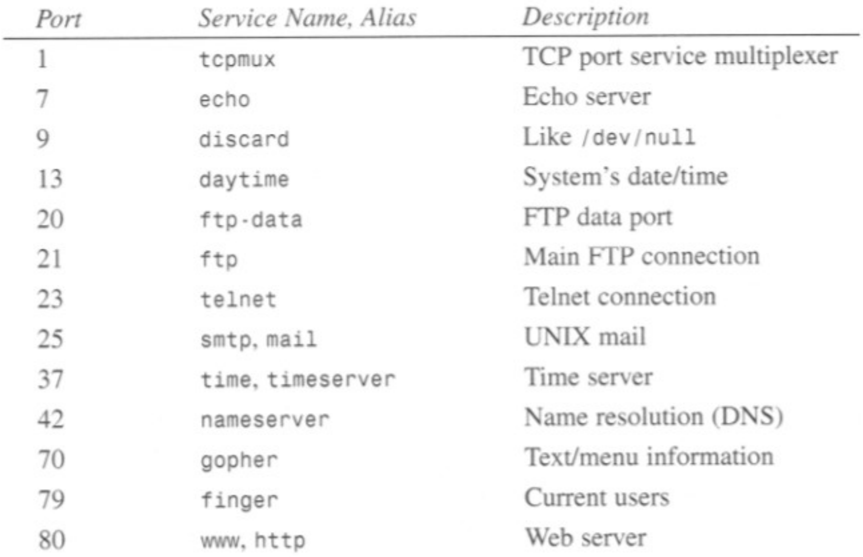
| To convert port numbers (16 bits) | To convert IP4 Addresses (32 bits) |
|---|---|

| | |
|---|---|
| Host -> Network<br>unit16_t **htons** ( uint16_t hostportnumber )<br>Network -> Host<br>unit16_t **ntohs** ( uint16_t netportnumber ) | Host -> Network<br>unit32_t **htonl** ( uint32_t hostportnumber )<br>Network -> Host<br>Unit32_t **ntohl** ( uint32_t netportnumber ) |

## 1.5 Port Numbers

**https://en.wikipedia.org/wiki/Port_%28computer_networking%29**

- In computer networking, a port is an endpoint of communication. Physical as well as wireless connections are terminated at ports of hardware devices. At the software level, within an operating system, a port is a logical construct that identifies a specific process or a type of network service.
- The software port is always associated with an IP address of a host and the protocol type of the communication. It completes the destination or origination network address of a message. Ports are identified for each protocol and address combination by 16-bit unsigned numbers, commonly known as the port number.
- Ports provide a multiplexing service for multiple services or multiple communication sessions at one network address. Specific port numbers are commonly reserved to identify specific services.
- Sockets are UNIQUELY identified by Internet address, end-to-end protocol, and port number. That is why when a socket is first created it is vital to match it with a valid IP address and a port number.
- Ports are software objects to multiplex data between different applications. When a host receives a packet, it travels up the protocol stack and finally reaches the application layer. Now consider a user running an ftp client, a telnet client, and a web browser concurrently. To which application should the packet be delivered? Well part of the packet contains a value holding a port number, and it is this number which determines to which application the packet should be delivered.
- So when a client first tries to contact a server, which port number should the client specify? For many common services, standard port numbers are defined

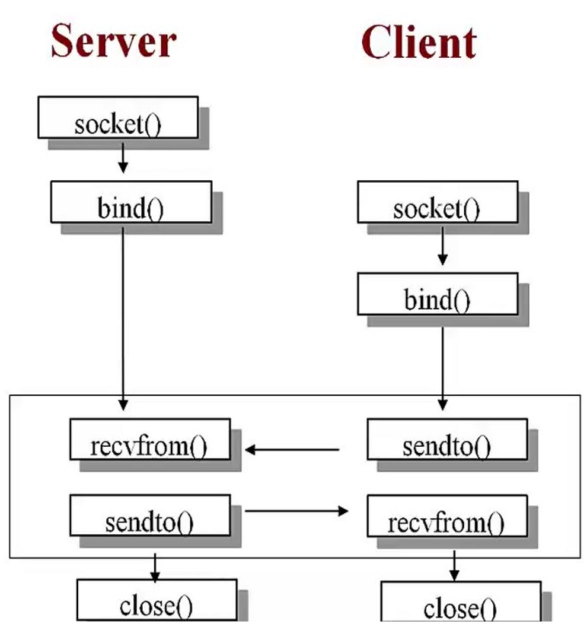
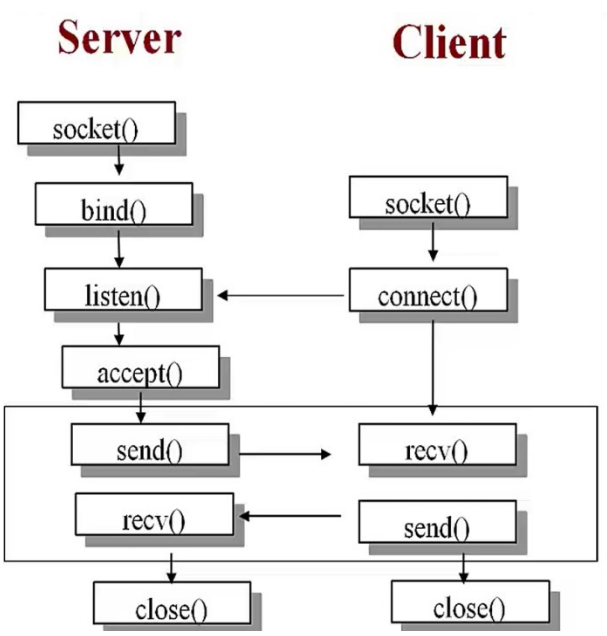


| *Port* | *Service Name, Alias* | *Description* |
|---|---|---|
| 1 | tcpmux | TCP port service multiplexer |
| 7 | echo | Echo server |
| 9 | discard | Like /dev/null |
| 13 | daytime | System's date/time |
| 20 | ftp-data | FTP data port |
| 21 | ftp | Main FTP connection |
| 23 | telnet | Telnet connection |
| 25 | smtp, mail | UNIX mail |
| 37 | time, timeserver | Time server |
| 42 | nameserver | Name resolution (DNS) |
| 70 | gopher | Text/menu information |
| 79 | finger | Current users |
| 80 | www, http | Web server |

- **Ports 0 – 1023**, are reserved and servers or clients that you create will not be able to bind to these ports unless you have root privilege.
- **Ports 1024 – 65535** are available for use by your programs, but beware other network applications maybe running and using these port numbers as well so do not make

assumptions about the availability of specific port numbers. Make sure you read Stevens for more details about the available range of port numbers!

### 1.6 Overview of Client-Server Communication with Connectionless/Connection oriented Protocol

| Connectionless Protocol | Connection Oriented Protocol |
|---|---|
|  Server<br>Client<br>socket()<br>bind()<br>socket()<br>bind()<br>recvfrom()<br>sendto()<br>sendto()<br>recvfrom()<br>close()<br>close() |  Server<br>Client<br>socket()<br>bind()<br>socket()<br>listen()<br>connect()<br>accept()<br>send()<br>recv()<br>recv()<br>send()<br>close()<br>close() |
| ● There are two end points called server and client. Therse two end points are going to open a socket and they are going to bind the IP address and whatever the port they are going to listen to the particular socket.<br>● Binding is a process to give the identity of a socket, that is to associate a IP address to a port number.<br>● After doing this, both the hosts are going to free to communicate with each other through reveivefrom() and sendto() functions. | |

# 2 Communication through UDP - The un-reliable communication

## 2.1 Outline of a UDP Server and Client

- **Step-1: Creating a socket or get a file descripter.**

  (Applicable for both client and server)

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

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

**int sockfd = socket(domain, type, protocol);**

Where,

**sockid:** socket descriptor, an integer (like a file-handle)

**domain:** integer, communication domain, e.g.,

**AF_INET**, IPv4 protocols, Internet addresses (typically used)

**AF_INET6,** IPv6 protocol, Imnternet address

**PF_UNIX**, Local communication, File addresses

**type:** communication type

**SOCK_STREAM** - reliable, 2-way, connection-based service

**SOCK_DGRAM** - unreliable, connectionless, messages of maximum length

**protocol:** specifies protocol

PPROTO_TCP or IPPROTO_UDP

usually set to 0 (i.e., use default protocol)

**upon failure returns -1**

**Program-1: Create sockets for a client and server separately. Check whether sockets are created or not, display appropriate message.**

| Server | Client |
|---|---|
| #include <sys/types.h><br>#include <sys/socket.h><br>int main()<br>{<br>  int ssid;<br>  ssid = socket(AF_INET, SOCK_DGRAM, 0);<br>  if(ssid==-1)<br>  {<br>     printf(“\n No socket for server application is not created successfully.”);<br>     exit(0);<br>  }<br>  printf(“\n A Socket for server application is created sussessflly.”);<br>  return 0;<br>} | #include <sys/types.h><br>#include <sys/socket.h><br>int main()<br>{<br>  int csid;<br>  csid = socket(AF_INET, SOCK_DGRAM, 0);<br>  if(csid==-1)<br>  {<br>     printf(“\n No socket for server application is not created successfully.”);<br>     exit(0);<br>  }<br>  printf(“\n A Socket for server application is created sussessflly.”);<br>  return 0;<br>} |

- **Step-2: Used to associate a socket with a port and IP of the local machine, called binding :** After creation of the socket, bind function binds the socket to the address and port number specified.
  (Applicable for both client and server)

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

```
int bind (int socket_file_descriptor, const struct sockaddr
*LocalAddress, socklen_t AddressLength);
```

Where,

bind (Berkeley Internet Name Domain)

- **Argument-1: socket_file_descriptor** is the socket file descriptor or socket-id created previously by socket() function.
- **Argument-2: LocalAddress** is a pointer to struct sockaddr that contains information about IP address & Port number.
- **Argument-3: AddressLength** is the size of the struct sockaddr.
- The **return value** of bind() is **0 for success** and **–1 for failure.**

<table>
<tr><th>Internal structure definition of sockaddr</th><th>Internal structure definition of sockaddr_in</th></tr>
<tr><td><b>sockaddr</b> is an structure defined as follows: This structure holds socket address information for many types of sockets:</td><td><b>sockaddr_in</b> is an parallel structure created by programmers to deal with struct sockaddr. This structure makes it easy to reference elements of the socket address.</td></tr>
<tr><td>struct sockaddr {<br>unsigned short sa_family;<br>char sa_data[14];<br>};</td><td>struct sockaddr_in {<br>short int sin_family;<br>unsigned short int sin_port;<br>struct in_addr sin_addr;<br>unsigned char sin_zero[8];<br>};</td></tr>
<tr><td>Where,<br><b>sa_family</b> is the structure member used to hold the socket's address family.<br><b>sa_data[]</b> is the structure member used to hold port number, IP address etc that will be associated with the particlar socket name/id.</td><td>Where,<br><b>sin_family</b> is the structure member that holds the socket's address family<br><b>sin_port</b> is the structure member that holds the port number to be associated with the particlar socket name/id.<br><b>sin_addr.s_addr</b> is the structure member used to hold IP address to be associated with the particlar socket name/id.<br>Now,<br>// Internet address<br>struct in_addr {<br>unsigned long s_addr;<br>};</td></tr>
<tr><td colspan="2">Now if we declare a variable of struct sockaddr_in as clientaddr; then clientaddr.sin_port holds 2 byte port address, address.sin_addr.s_addr holds 4-byte ip address (in Network Byte Order). sin_family corresponds to sa_family in a struct sockaddr and should be set to "AF_INET". clientaddr.sin_family=AF_INET</td></tr>
</table>

- **As port number require 2 bytes and address are unsigned, so ports can be numbers from 0 to $2^{16}$-1=65535.**
- **All ports below 1024 are reserved.**
- **You can use ports above 1024 upto 65535, provided they are not already use.**
- **Some common examples of TCP and UDP with their default ports:**

| SL. NO. | APPLICATION PROTOCOL | CONNECTION PROTOCOL | PORT NUMBER |
|---|---|---|---|
| 1 | DNS LOOKUP | UDP | 53 |
| 2 | FTP | TCP | 21 |
| 3 | HTTP | TCP | 80 |
| 4 | POP3 | TCP | 110 |
| 5 | Telnet | tcp | 23 |

**Program-2: Create sockets for a client and server separately. Bind the sockets with each machines appropriate port and IP address. Assume client and server machines are the same machine. Check whether sockets are created or not, binding done properly or not. display appropriate message.**

**Server**

```
#include <sys/types.h>
#include <sys/socket.h>
int main()
{
  int ssid, status_ss;
  struct sockaddr_in ssaddr;
  /*Server Socket Creation*/
  ssid = socket(AF_INET, SOCK_DGRAM, 0);
  if(ssid==-1)
  {
     printf("\n No socket for server application is
not created successfully.");
     exit(0);
  }
  printf("\n A Socket for server application is
created sussessflly.");

  /*Code of Binding on same machine*/
  ssaddr.sin_family=AF_INET;
  ssaddr.sin_port=htons(1025);
  ssaddr.sin_addr.s_addr=htonl(INADDR_ANY);
  /*or
ssaddr.sin_addr.s_addr=inet_addr("127.0.0.10");
*/
  /* zero the rest of the struct*/
  memset(&(ssaddr.sin_zero), '\0', 8);
  status_ss=bind(ssid, (struct sockaddr *)
&ssaddr, sizeof(ssaddr);
  if (status_ss!=-1)
     printf("\nBinding Success");
```

**Client**

```
#include <sys/types.h>
#include <sys/socket.h>
int main()
{
  int csid, status_cs;
  struct sockaddr_in csaddr;
  /*Server Socket Creation*/
  csid = socket(AF_INET, SOCK_DGRAM, 0);
  if(csid==-1)
  {
     printf("\n No socket for server application is
not created successfully.");
     exit(0);
  }
  printf("\n A Socket for server application is
created sussessflly.");

  /*Code of Binding on same machine*/
  csaddr.sin_family=AF_INET;
  csaddr.sin_port=htons(1026);
  csaddr.sin_addr.s_addr=htonl(INADDR_ANY);
  /*or
csaddr.sin_addr.s_addr=inet_addr("127.0.0.10");
*/
  /* zero the rest of the struct*/
  memset(&(csaddr.sin_zero), '\0', 8);
  status_ss=bind(ssid, (struct sockaddr *)
&csaddr, sizeof(csaddr);
  if (status_cs!=-1)
     printf("\nBinding Success");
```

| | |
|---|---|
| else<br>printf("\nBinding Failure");<br>return 0;<br>} | else<br>printf("\nBinding Failure");<br>return 0;<br>} |

- **Step-3: sendto() and recvfrom() - DGRAM Style**
  (Applicable for both client and server)

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

Where,

- **Argument-1: sockfd** is the socket file descriptor or socket-id created previously by socket() function. This is the socket through which data is to be sent.
- **Argument-2: msg** is a pointer to the data you want to send
- **Argument-3: len** is the length of the message.
- **Argument-4: flags** is
- **Argument-5: to** is a pointer to a struct sockaddr, that it contain the destination's socket address (i.e.IP and port).
- **Argument-6: tolen** is sizeof (struct sockaddr) or sizeof (to)

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

Where,

- **Argument-1: sockfd** is the socket file descriptor or socket-id created previously by socket() function. This is the socket through which data is received/read.
- **Argument-2: buf** is a pointer to the data you want to read from
- **Argument-3: len** is the max.length of the buffer.
- **Argument-4: flags** is set to 0.
- **Argument-5: from** is a pointer to a local struct sockaddr, that will be filled with IP and port number of originating machine.
- **Argument-6: frmlen** is sizeof (struct sockaddr) or sizeof (from)
- **recvfrom() returns the number of bytes received**, or -1 on error (with errno set accordingly.)

**Program-3: Create sockets for a client and server separately. Bind the sockets with each machines appropriate port and IP address. Assume client and server machines are the same machine. Check whether sockets are created or not, binding done properly or not. display appropriate message. Now send a message entered through keyboard from the client to server. Check whether the message sent by client is received by server or not. Also display number of bytes need to send and actually sent by client and number of bytes received by server.**

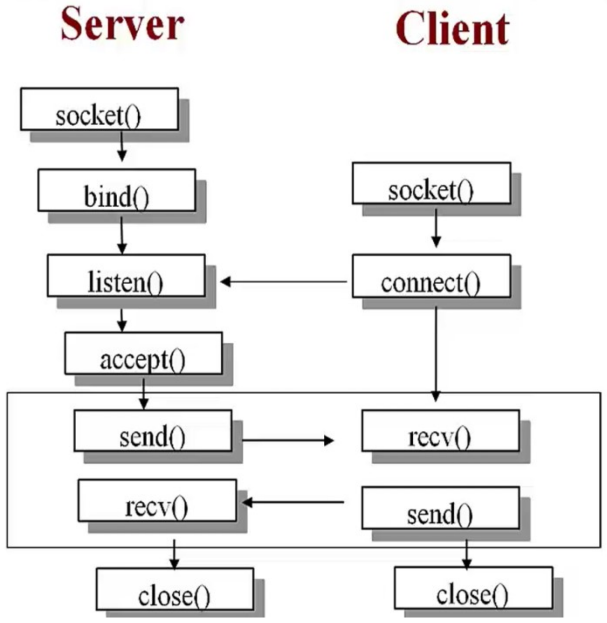
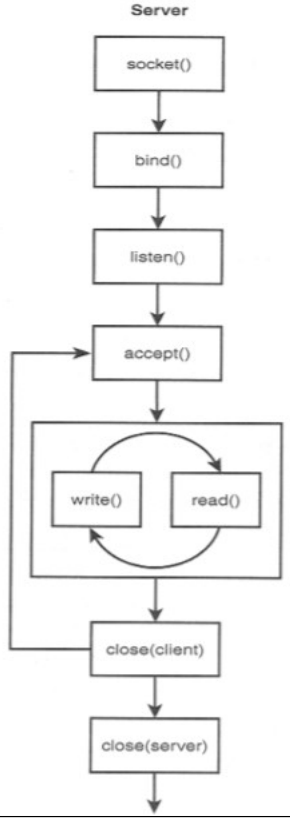
| **Server** | **Client** |
|---|---|
| #include <sys/types.h><br>#include <sys/socket.h><br>int main()<br>{ | #include <sys/types.h><br>#include <sys/socket.h><br>int main()<br>{ |

```
 int ssid, status_ss;
 struct sockaddr_in ssaddr, fromclientaddr;
 char servbuf[40];
 /*Server Socket Creation*/
 ssid = socket(AF_INET, SOCK_DGRAM, 0);
 if(ssid==-1)
 {
    printf(“\n No socket for server application is
not created successfully.”);
    exit(0);
 }
 printf(“\n A Socket for server application is
created sussessflly.”);

 /*Code of Binding on same machine*/
 ssaddr.sin_family=AF_INET;
 ssaddr.sin_port=htons(1025);
 ssaddr.sin_addr.s_addr=htonl(INADDR_ANY);
 /*or
ssaddr.sin_addr.s_addr=inet_addr(“127.0.0.10”);
*/
 /* zero the rest of the struct*/
 memset(&(ssaddr.sin_zero), ’\0’, 8);
 status_ss=bind(ssid, (struct sockaddr *)
&ssaddr, sizeof(ssaddr);
 if (status_ss!=-1)
    printf(“\nBinding Success”);
 else
    printf(“\nBinding Failure”);


 /*Waiting to receive message from client*/
 smsgbyte=recvfrom(ssid, serverbuf, 40, 0,
fromclientaddr, int *fromlen);

 return 0;
}
```

```
 int csid, status_cs, cmsgbyte;
 struct sockaddr_in csaddr, toserveraddr;
 char clientbuf[40];
 /*Server Socket Creation*/
 csid = socket(AF_INET, SOCK_DGRAM, 0);
 if(csid==-1)
 {
    printf(“\n No socket for server application is
not created successfully.”);
    exit(0);
 }
 printf(“\n A Socket for server application is
created sussessflly.”);

 /*Code of Binding on same machine*/
 csaddr.sin_family=AF_INET;
 csaddr.sin_port=htons(1026);
 csaddr.sin_addr.s_addr=htonl(INADDR_ANY);
 /*or
csaddr.sin_addr.s_addr=inet_addr(“127.0.0.10”);
*/
 /* zero the rest of the struct*/
 memset(&(csaddr.sin_zero), ’\0’, 8);
 status_ss=bind(ssid, (struct sockaddr *)
&csaddr, sizeof(csaddr);
 if (status_cs!=-1)
    printf(“\nBinding Success”);
 else
    printf(“\nBinding Failure”);

 /*Sever address is stored in toserveraddr*/
 toserervaddr.sin_family=AF_INET;
 toserveraddr.sin_port=htons(1025);
toserveraddr.sin_addr.s_addr=htonl(INADDR_A
NY);
/*or
toserveraddr.sin_addr.s_addr=inet_addr(“127.0.0
.10”); */
/* zero the rest of the struct*/
memset(&(toserveraddr.sin_zero), ’\0’, 8);

 /*User input meaasge to send*/
 prinf(“\nEnter a message:”);
 gets(clientbuf);

 /*send the message to server*/
 cmsgbyte=sendto(csid, str, sizeof(str), 0, (struct
sockaddr *) &toserveraddr,
sizeof(toserveraddr));
    printf(“\n Client want to send message
byte:%d”, sizeof(str));
```

| | |
|---|---|
| | printf("\n Client actually sent message byte:%d", cmsgbyte);<br><br>return 0;<br>} |

## 3 Communication through TCP - The un-reliable communication

| Communication through TCP | Outline of a TCP Server |
|---|---|
|  Server<br>Client<br>socket()<br>bind()<br>listen()<br>accept()<br>send()<br>recv()<br>close()<br>socket()<br>connect()<br>recv()<br>send()<br>close() |  Server<br>socket()<br>bind()<br>listen()<br>accept()<br>write()<br>read()<br>close(client)<br>close(server) |
| | |

### 3.1 Outline of a TCP Server

- **Step-1: Creating a socket - socket() function**
  (Applicable for client & server)
  **Same as UDP Step-1**

- **Step-2: Creating a Socket Address (Binding an address and port number) - bind() functiomn :**
  (Required for server, optional for client)

- **Step-3: Listen for incoming connections - listen() function**
  (Required for server only)

**Syntax**

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

Where,

- **sockfd** is the is the socket file descriptor returned by a call to socket() function.
- **backlog** is the number of active participants that can "wait" for a connection. It is used to determining how many connections, the server will connect with. Typical values for backlog are 5 – 10.
- The return value of **listen() is 0 for success** and **–1 for failure**.

- **Step-3: Connecting to a Server - connect() function :**Connect performs the three-way handshake with the server and returns when the connection is established or an error occurs. Once the connection is established you can begin reading and writing to the socket.

  (Required for client only)

  **Syntax**

  ```
  int connect(int sockfd, const struct sockaddr *ServerAddress, int
                          AddressLength)
  ```

  Where,

  - **sockfd** is the is the socket file descriptor, as returned by the socket() call
  - **ServerAddress** is a pointer to a structure struct sockaddr that is it contains the address of destination server's socket address (Port and IP address)
  - **AddressLength** can be set to sizeof(struct sockaddr) that is size of the server address
  - **return -1** on error.

- **Step-4: Sending a connection - connect() function & Accepting a connection - accept() function**

  (connect() is required for client only, accept() is required for server only)

  **Syntax**

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

  Where,

  - **sockfd** is the is the socket file descriptor returned by a call to socket() function. sockfd is the listen()ing socket descriptor.
  - **ClientAddress:** After a connection with a client is established, the address of the client must be made available to your server, otherwise how could you communicate back with the client? ClientAddress is usually be a pointer to a local struct sockaddr_in, where the information about the incoming connection (host address & port number) will be stored automatically.
  - **addrlen** is a local integer variable that should be set to sizeof(struct sockaddr_in) before its address is passed to accept().
  - **accept() returns a new file descriptor**, that is used to receive and send mesages from/to the client.
  - **accept()** returns **-1 and sets errno** if an error occurs
  - **accept()** is **blocking:** waits for connection before returning. It **dequeues** the **next connection** on the queue for socket (sockfd)

- **Step-5: Read and Writing to the socket (Sending messages and receiving messages) - send() and recv() function repectively.**

  (Required for both client & server)

  **Syntax for send() function**

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

Where,

- **Sockfd** is the socket descriptor **that need to send data to** (it's the one returned by socket() in client side or server side)
- **msg** is a pointer to the data to be sent by user.
- **len** is the **length of the data** sent in bytes.
- **flags**: set to 0.
- **send() returns** the number of bytes actually sent out–this might be less than the number of bytes you told it to send. It'll fire off as much of the data as it can, and trust you to send the rest later.
- If the **value returned by send() doesn't match the value in len**, it's up to you to send the rest of the string. The good news is this: if the packet is small (less than 1K or so) it will probably manage to send the whole thing all in one go.
- **-1 is returned on error**, and errno is set to the error number.

**<u>Syntax for recv() function</u>**

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

Where,

- **Sockfd** is the socket descriptor **that need to receive data from**(whether it's the one returned by socket() in client side or the one that is returned by accept() in server side)
- **buf** is the buffer to read the information into.In otherwords, it is a pointer to buffer where message sent by send() function is stored.
- **len** is the **maximum length of the buffer.**
- **flags** can again be set to 0.
- **recv()** returns t**he number of bytes actually received into the buffer,** or **-1 on error** and errno is set to the error number.
- **recv() can return 0**. This can mean only one thing: the remote side has closed the connection on you! A return value of 0 is recv()'s way of letting you know this has occurred.

- **Step-6: Shutting down sockets- close() function**

  (Required for both client & server)

  **<u>Syntax for close() function</u>**

```
int close(int filedescriptor);
```

Where,

- It will prevent any more reads and writes to the socket.
- returns **0 on success**, and **-1 on error.**

**<u>Syntax for shutdown() function</u>**

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

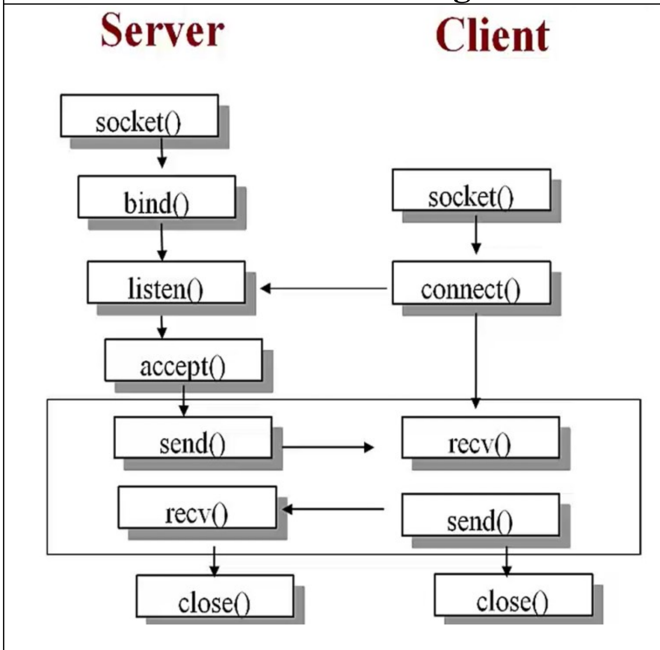
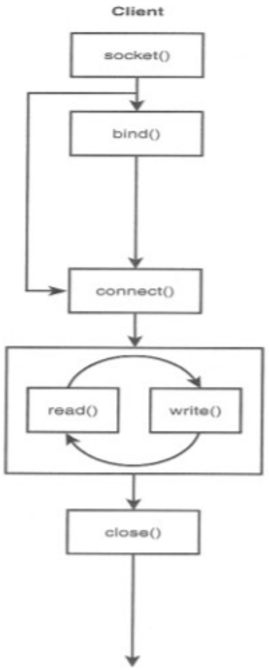
Where,

- It allows you **to cut off communication in a certain direction, or both ways** (just like close() does.)
- **sockfd** is the socket file descriptor you want to shutdown.
- **how** is one of the following:

  0 – Further receives are disallowed
  1 – Further sends are disallowed
  2 – Further sends and receives are disallowed (like close())
- **shutdown()** returns **0 on success**, and **-1 on error.**

- It's important to note that **shutdown() doesn't actually close the file descriptor**–it just changes its usability. To free a socket descriptor, you need to use close().

# 4 Communication through TCP - Outline of a TCP Client

| Communication through TCP | TCP Client |
|---|---|
|  Server<br>Client<br>socket()<br>bind()<br>listen()<br>accept()<br>send()<br>recv()<br>close()<br>socket()<br>connect()<br>recv()<br>send()<br>close() |  Client<br>socket()<br>bind()<br>connect()<br>read()<br>write()<br>close() |
| | |

## 4.1 Outline of a TCP Client

- **Example-4:** Write a program to implement a ***chat server and client*** in C using ***TCP sockets*** where both of them will exchange messages with each other continuously. If any one of them will receive the "quit" message from the other end then both of them will close the connection.
  a) Assume both the client and server are running with in the same host.
  b) Do necessary changes, so that the same programs will run with different host.

```
/*chat_server.c :    A stream socket server*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>
```

```
#define MYPORT 3490 // the port users will be connecting to
#define BACKLOG 10 // how many pending connections queue will hold
void sigchld_handler(int s)
{
while(wait(NULL) > 0);
}

/*main() function of server.c A stream socket server*/
int main(void)
{
    int sockfd, new_fd; // listen on sock_fd, new connection on new_fd
    struct sockaddr_in my_addr; // my address information
    struct sockaddr_in their_addr; // connector's address information
    int sin_size;
    struct sigaction sa;
    int yes=1;
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("Socket Creation");
        exit(1);
    }
    /*
    if (setsockopt(sockfd,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == -1)
    {
         perror("setsockopt");
         exit(1);
    } */
    my_addr.sin_family = AF_INET; // host byte order
    my_addr.sin_port = htons (MYPORT); // short, conerting to network byte order
     my_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
     bzero(&(my_addr.sin_zero), 8); // zero the rest of the struct
     if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr))== -1)
     {
         perror("Socket Bind");
         exit(1);
     }

     /*Listing a connection from client*/
     if (listen(sockfd, BACKLOG) == -1)
     {
        perror("Socket Listen");
        exit(1);
     }
     /*
     sa.sa_handler = sigchld_handler; // reap all dead processes
     sigemptyset(&sa.sa_mask);
     sa.sa_flags = SA_RESTART;
     if (sigaction(SIGCHLD, &sa, NULL) == -1)
     {
         perror("sigaction");
         exit(1);
     } */

    while(1) { // main accept() loop
    sin_size = sizeof(struct sockaddr_in);
    if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size)) == -1)
    {
       perror("accept");
       continue;
    }
    printf("server: got connection from %s\n", inet_ntoa(their_addr.sin_addr));
    if (!fork())    // this is the child process
    {
       close(sockfd); // child doesn’t need the listener
       if (send(new_fd, "Hello, world!\n", 14, 0) == -1)
             perror("Socket Send");
       close(new_fd);
       exit(0);
    }
    close(new_fd); // parent doesn’t need this
}
return 0;
}
```

```
/*chat_client.c :    A stream socket client*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#define PORT 3490 // the port client will be connecting to
#define MAXDATASIZE 100 // max number of bytes we can get at once

int main(int argc, char *argv[])
{
    int sockfd, numbytes;
    char buf[MAXDATASIZE];
    struct hostent *he;
    struct sockaddr_in their_addr; // connector’s address information
    if (argc != 2)
    {
       fprintf(stderr,"\nUsage: Client Hostname");
       exit(1);
    }
    if ((he=gethostbyname(argv[1])) == NULL) // get the host info
    {
       perror("\nSocket-gethostbyname");
```

```
            exit(1);
        }

        if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        {
            perror("socket");
            exit(1);
        }
        their_addr.sin_family = AF_INET; // host byte order
        their_addr.sin_port = htons(PORT); // short, network byte order
        their_addr.sin_addr = *((struct in_addr *)he->h_addr);
        bzero(&(their_addr.sin_zero), 8); // zero the rest of the struct
        if (connect(sockfd, (struct sockaddr *)&their_addr, sizeof(struct sockaddr)) == -1)
        {
            perror("\nSocket connect");
            exit(1);
        }
        if ((numbytes=recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1)
        {
            perror("\nSocket recv");
            exit(1);
        }
        buf[numbytes] = '\0';
        printf("\nReceived: %s",buf);
        close(sockfd);
        return 0;
    }
```

## 5 Commonly used network functions

| Sl. No. | Function Name | Header File | Prototype and Use |
|---|---|---|---|
| 1 | **perror**<br>(print a system error message) | #include <stdio.h> | **Prototype**<br>**void perror(const char *s);**<br>**Description**<br>● s is a null byte ('\0') terminated string, canot be NULL<br>● First, the argument string s is printed, followed by a colon and a blank. Then an error message corresponding to the current value of errno and a new-line.<br>● To be of most use, the argument string should include the name of the function that incurred the error.<br>● In the C Programming Language, the perror function write an error message to the stderr stream in the following format:<br>**string: error-message**<br>**Use**<br>**perror("Socket bind");** |
| 2 | **memset** | #include | **Prototype** |

| | | | |
|---|---|---|---|
| | (used to fill a block of memory with a particular value) | <string.h> | **void *memset(void *str, int c, size_t n)**<br>**<u>Description</u>**<br>● This function copies the character c (an unsigned char) to the first n characters of the string pointed to, by the argument str.<br>● This function returns a pointer to the memory area str.<br>● **str** − This is a pointer to the block of memory to fill.<br>● **c** − This is the value to be set. The value is passed as an int, but the function fills the block of memory using the unsigned char conversion of this value.<br>● **n** − This is the number of bytes to be set to the value.<br>**<u>Use</u>**<br>char str[]=”kiit”;<br>memset(str, '#', 2);<br>printf(“%s”, str); => o/p: ##it |
| **3** | **bzero**<br>(used to set all the socket structures with null values) | | **<u>Prototype</u>**<br>**void bzero(void *s, int nbyte);**<br>**<u>Description</u>**<br>● places nbyte null bytes in the string s<br>● This function does not return anything.<br>● **s** − It specifies the string which has to be filled with null bytes. This will be a point to socket structure variable.<br>● **nbyte** − It specifies the number of bytes to be filled with null values. This will be the size of the socket structure.<br>**<u>Use</u>**<br>char str[]=”kiit”;<br>bzero(str,4);<br>printf(“%s”, str); => o/p: |