**Aim:** Socket Programming in C Language on Linux.

a) TCP Client, TCP Server

b) UDP Client, UDP Server

**Theory:**

**Socket Programming with TCP**

**Processes Communicating Across a Network**

An network application involves two processes in two different hosts communicating with each other over a network. The two processes communicate with each other by sending and receiving messages through their *sockets*. A process's socket can be thought of as the process's door: a process sends messages into, and receives message from, the network through its socket. When a process wants to send a message to another process on another host, it shoves the message out its door.

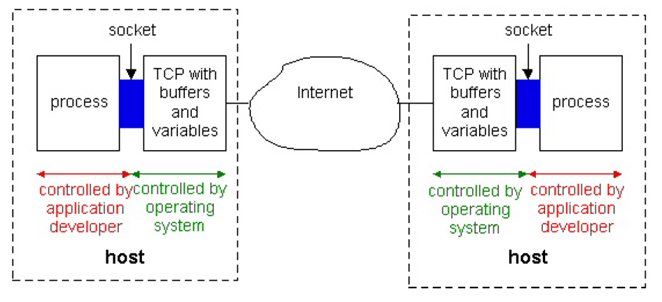


Fig 1: Application processes, sockets, and the underlying transport protocol

The Fig 1 illustrates socket communication between two processes that communicate over the Internet. As shown in this figure, a **socket** is the interface between the application layer and the transport layer within a host. It is also referred to as the **API (Application Programmers interface)** between the application and the network, since the socket is the programming interface with which networked applications are built in the Internet.

The arrows in the following Fig 2 show the position of a socket, and the communication layer that the socket provides.

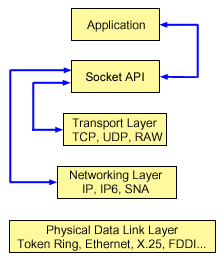


Fig 2: The position of a socket, and the communication layer that the socket provides.

The application developer has control of everything on the application-layer side of the socket but has little control of the transport-layer side of the socket. The only control that the application developer has on the transport-layer side is

1. The choice of transport protocol and

2. The ability to fix a few transport-layer parameters such as maximum buffer and maximum segment sizes.

**Point-to-Point Communication**

A socket represents a single connection between exactly two pieces of software. More than two pieces of software can communicate in client/server or distributed systems (for example, many Web browsers can simultaneously communicate with a single Web server) but multiple sockets are required to do this. Socket-based software usually runs on two separate computers on the network, but sockets can also be used to communicate locally (interprocess) on a single computer.

**Types of Sockets**

Socket interfaces can be divided into three categories.

1. Stream Socket
2. Datagram Socket
3. Raw Socket

Essentially, a "stream" requires that the two communicating parties first establish a socket connection, after which any data passed through that connection will be guaranteed to arrive in the same order in which it was sent.

Datagram sockets offer "connection-less" semantics. With datagram’s, connections are implicit rather than explicit as with streams. Either party simply sends datagram’s as needed and waits for the other to respond; messages can be lost in transmission or received out of order, but it is the application's responsibility and not the socket's to deal with these problems.

The third type of socket -- the so-called raw socket -- bypasses the library's built-in support for standard protocols like TCP and UDP. Raw sockets are used for custom low-level protocol development.

**Addresses and Ports**

Many parts of the Internet work with naming services, so that the users and socket programmers can work with computers by name. Stream and datagram sockets also use IP [port numbers](http://compnetworking.about.com/library/glossary/bldef-port.htm) to distinguish multiple applications from each other. The socket APIs are relatively small and simple. Many of the functions are similar to those used in file input/output routines such as read(), write(), and close().

## How do TCP sockets work?

In a connection-oriented client-to-server model, the socket on the server process waits for requests from a client. To do this, the server first establishes (binds) an address that clients can use to find the server. When the address is established, the server waits for clients to request a service. The client-to-server data exchange takes place when a client connects to the server through a socket. The server performs the client's request and sends the reply back to the client.

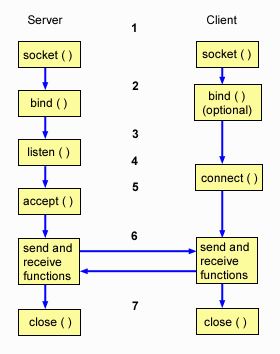


Fig 3: TCP Socket events

Typical flow of events for a connection-oriented socket:

1. The [**socket()**](http://publib.boulder.ibm.com/infocenter/iseries/v5r3/topic/apis/socket.htm) function creates an endpoint for communications and returns a socket descriptor that represents the endpoint.
2. The [**listen()**](http://publib.boulder.ibm.com/infocenter/iseries/v5r3/topic/apis/listen.htm) function indicates a willingness to accept client connection requests. When a listen() is issued for a socket, that socket cannot actively initiate connection requests. The listen() API is issued after a socket is allocated with a socket() function and the [**bind()**](http://publib.boulder.ibm.com/infocenter/iseries/v5r3/topic/apis/bind.htm) function binds a name to the socket. A listen() function must be issued before an accept() function is issued.
3. The client application uses a connect() function on a stream socket to establish a connection to the server.
4. The server application uses the [**accept()**](http://publib.boulder.ibm.com/infocenter/iseries/v5r3/topic/apis/accept.htm) function to accept a client connection request. The server must issue the bind() and listen() functions successfully before it can issue an accept().
5. Clients and servers have many data transfer functions from which to choose, such as [**send ()**](http://publib.boulder.ibm.com/infocenter/iseries/v5r3/topic/apis/send.htm), [**recv()**](http://publib.boulder.ibm.com/infocenter/iseries/v5r3/topic/apis/recv.htm), read(), write(), and others.
6. When a server or client wants to cease operations, it issues a [**close()**](http://publib.boulder.ibm.com/infocenter/iseries/v5r3/topic/apis/close.htm) function to release any system resources acquired by the socket.

**Socket Programming with TCP**

When two processes communicate over TCP, from the perspective of the processes it is as if there is a pipe between the two processes. This pipe remains in place until one of the two processes closes it. The sending process does not have to attach a destination address to the bytes because the pipe is logically connected to the destination.

## How do UDP sockets work?

The following figure shows the typical flow of events (and the sequence of issued functions) for a connection-less socket session. An explanation of each event follows the Fig 4

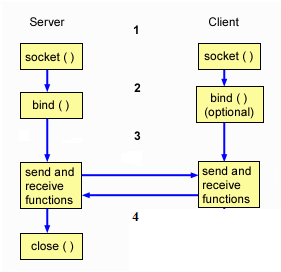


Fig 4: UDP Socket events

Typical flow of events for a connection-less socket:

1. The [**socket()**](http://publib.boulder.ibm.com/infocenter/iseries/v5r3/topic/apis/socket.htm) function creates an endpoint for communications and returns a socket descriptor that represents the endpoint.
2. Clients and servers have many data transfer functions from which to choose, such as [**send()**](http://publib.boulder.ibm.com/infocenter/iseries/v5r3/topic/apis/send.htm), [**recv()**](http://publib.boulder.ibm.com/infocenter/iseries/v5r3/topic/apis/recv.htm), read(), write(), and others.
3. When a server or client wants to cease operations, it issues a [**close()**](http://publib.boulder.ibm.com/infocenter/iseries/v5r3/topic/apis/close.htm) function to release any system resources acquired by the socket.

**Conclusion**

Done Socket Programming in C Language on Linux of TCP Client, TCP Server & UDP Client, UDP Server.