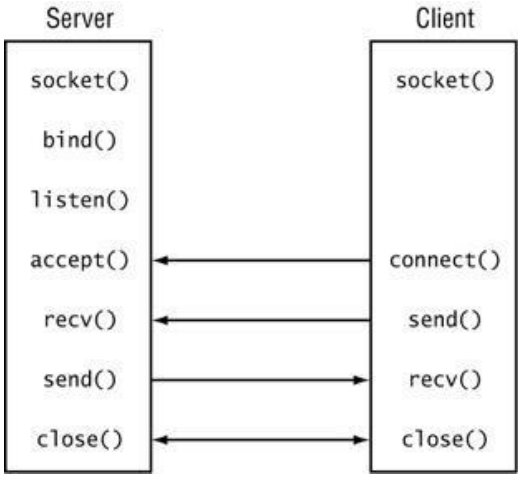
Socket Programming using TCP

- **Connectionless vs Connection oriented communication**

Connectionless	Connection Oriented
No need to setup the connection before starting communication.	Connection needs to be setup before the start of communication.
Packets may get corrupted.	Packets do not corrupt.
Packets may get lost.	Packets loss does not occur.
Packets may arrive out of order.	Packets are passed to the application in an order.
UDP is a connectionless transport protocol.	TCP is a connection oriented transport protocol.
Commonly used for real time loss tolerant applications such as VoIP.	Commonly used for reliable transportation delay tolerant applications.

- **Socket Programming using TCP**
  TCP communication model is shown below. Just like UDP, each block of this model is a system call. Unlike UDP, for TCP based communication, we first have to initiate a connection with the server. Once the server accepts the connection request, only then we can start communication with the server.



  - **socket()**
    Same as explained in Lab 2.
  - **bind()**
    Same as explained in Lab 2.
  - **int listen(int sock, int backlog);**
    - **sock**
      sock is the socket which was created in the earlier step on which connection requests will be listened.

- **backlog**
  Backlog specifies the length of the queue, which will be holding pending connections.
- **return value**
  This call returns -1 in case of any error.
- **Usage**
  listen(sock, 10);

- **int rv = connect(int sock, struct sockaddr *server_addr, int addrlen);**
  - **sock**
    The socket created at client end to communicate with the server.
  - **server_addr**
    Contains the address information of the server.
  - **addrlen**
    It is simply the length of the serv_addr.
  - **rv**
    it is the return value of the connect method. It returns -1 in case of error while connecting with the server.
  - **Usage**
    connect(sock, struct sockaddr*)& server_addr, sizeof (server_addr));

- **int accept(int sock, struct sockaddr *client_addr, socklen_t *addrlen);**
  - **sock**
    The socket which is being used for listening.
  - **client_addr**
    The address of the remote peer will be received in this structure.
  - **addrlen**
    The length of the address will be received in this variable.
  - **return value**
    Returns a new socket, on which the communication will take place with the connecting party.
  - **Usage**
    comsock = accept(lsock, (struct sockaddr *)&client_addr, &addr_size);

- **int send(int comsock, const void *sbuff, int len, int flags);**
  - **comsock**
    The socket which was returned by the accept call when the client's connect request was accepted.
  - **sbuff**
    It is the buffer which contains data to be sent to the remote peer.
  - **len**
    The length of the data buffer.

- **flags**
  Set flags to 0.
- **Usage**
  bytes_sent = send(comsock, sbuff, len, 0);

- **int recv(int comsock, void *rbuff, int len, int flags);**
  - **comsock**
    The socket which was returned by the accept call when the client's connect request was accepted.
  - **rbuff**
    it is the buffer which will receive the incoming data.
  - **len**
    The length of the data buffer.
  - **flags**
    Sets flags to 0.
  - **Usage**
    Bytes_received = recv (comsock, rbuff, len, 0);

- **close()**
  Same as explained in Lab 2.

Lab Activity

Task

In this lab, you have to write a simple TCP half duplex chat server and client. Half duplex is a mode of communication in which only one of the two parties can send data at any given instant of time.

How to do it?

Check Beej's Guide to Network Programming, which can be found at the following link: http://cse.iitkgp.ac.in/~agupta/compsyslab/Socket-Tutorial.pdf. You can use client and server templates given in this guide. For the given task, you will have to use **Send** and **Recv** calls in a loop so that half duplex communication can take place. You will have to arrange the sequence of **Send** and **Recv** in such a way that when the client sends data, the server will be waiting to receive that data, and when the server is sending data, the client is waiting to receive that data.

The chat server will be running on port 12345. A client will send the connect request to the server using **connect()** call. The server, upon receiving the request, will send the message, **welcome,** to the client and then client will send a **Hello** reply packet. In this way, simplex communication will start.

Server Side Display	**Client Side Display**
	Server: Welcome
Client: Hello	Client: Hello
Server: How are you?	Server: How are you?
Client: What are you up to?	Client: What are you up to?

For any queries,
Contact ifrah.saeed@uet.edu.pk