Tutorial-2

① Why UDP is faster than TCP? Which function is responsible for sending SYN segment during TCP connection establishment phase? Illustrate the TCP-3 way & TCP 4-way handshake mechanism with suitable state transition diagram.

→ UDP is faster than TCP because UDP provides a connectionless service. There is no need to have a long-term relationship between a UDP client and server whereas TCP is a connection-oriented protocol which makes it slower compare to UDP.

→ The client issues an active open by calling Connect. This causes the client TCP to send a "Synchronize" (SYN) segment which tells the server the client's initial sequence number for the data that the client will send on the connection.

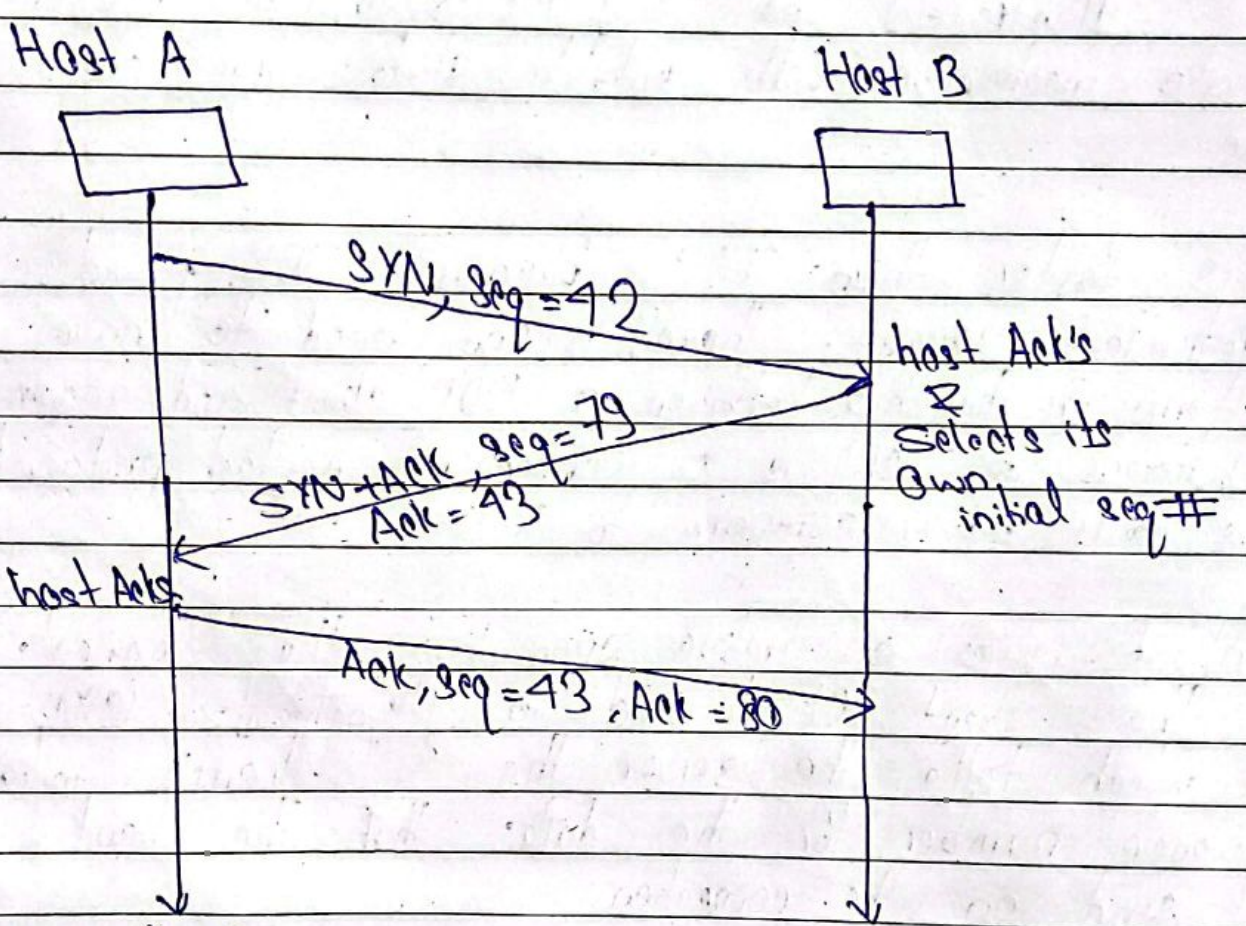
Three Way handshake

Steps:-

i) client host send TCP SYN segment to server
- specifies a random initial seq #
- no data.

ii) Server host receives SYN, replies with SYNACK segment.
- server allocates buffer.
- Specifies server initial seq #.

iii) client receives SYNACK replies with ACK segment, which may contain data.



three-way handshake

Seq #'s
- byte stream "number" of first byte in segments data

Acks:
- Seq # of next byte expected from other side
- Cumulative Ack.

TCP - 4 way handshake Mechanism

Closing a Connection

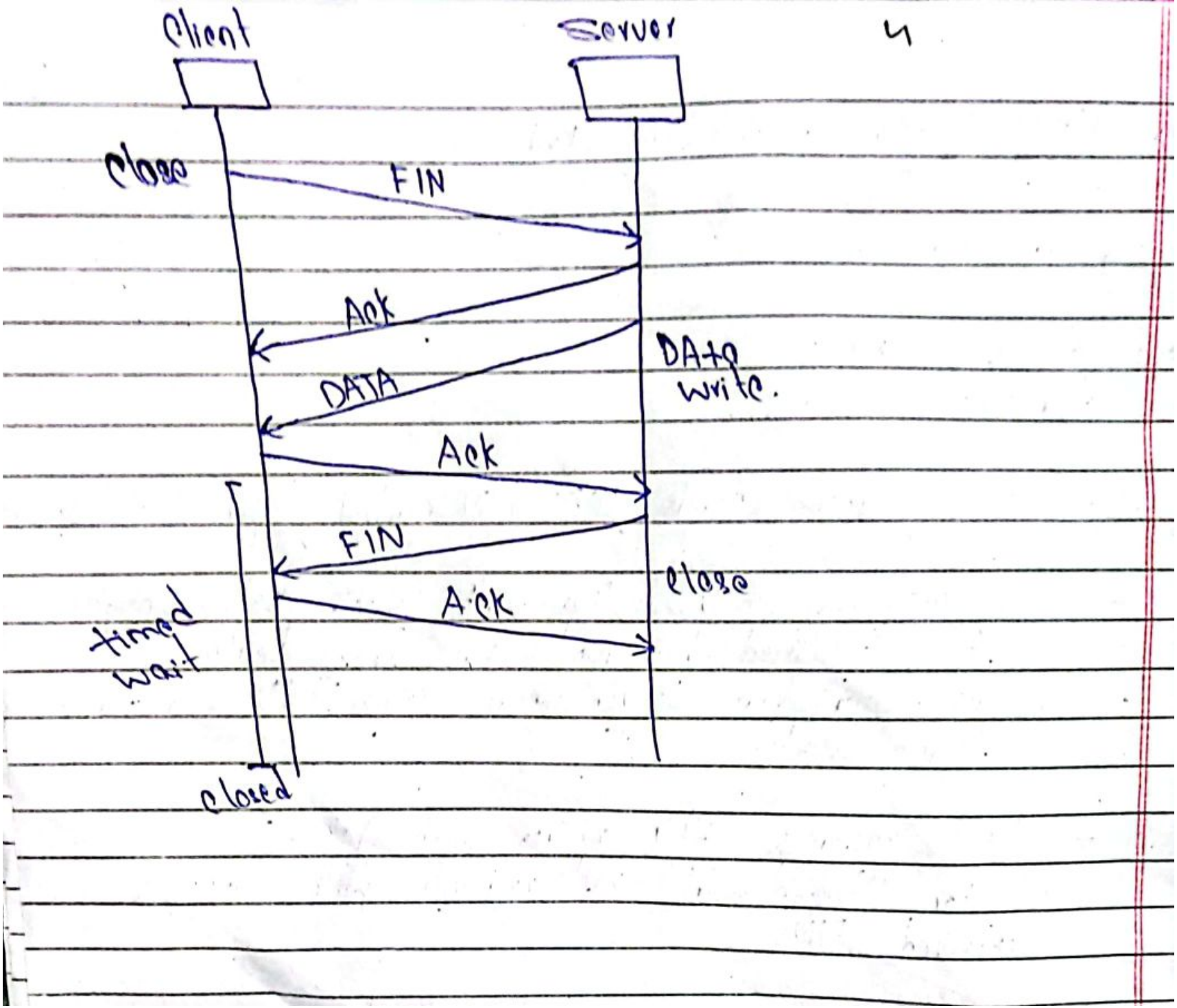
Client closes socket:
clientSocket.close();

Step 1 :- client end system sends TCP FIN control segment to server

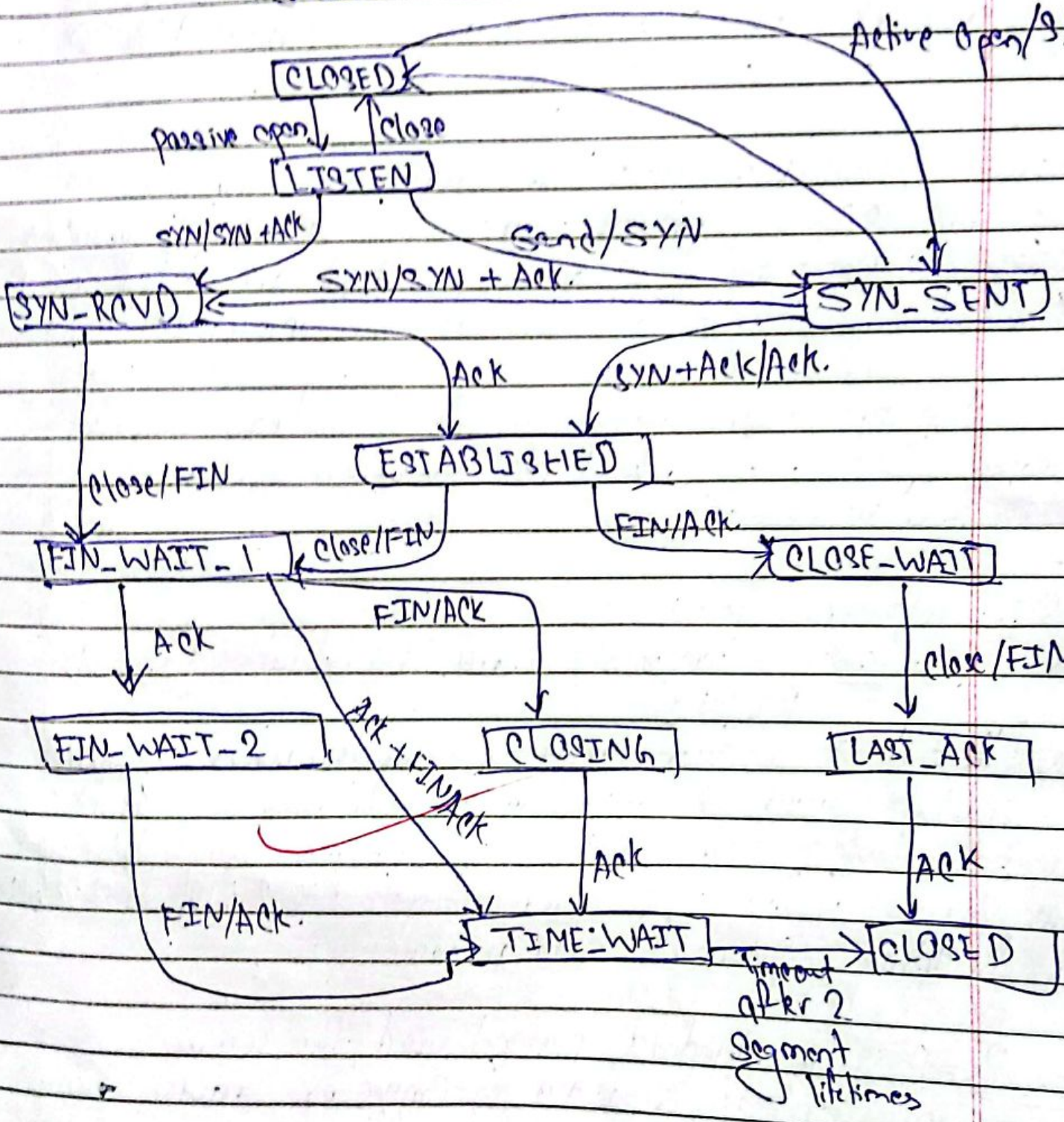
Step 2 :- Server receives FIN, replies with Ack. Server might send some buffered but not sent data before closing the connection. Server then sends FIN & moves to closing state.

Step 3 :- Client receives FIN, replies with Ack.
- enter "timed wait" - will respond with Ack to received FINs.

Step 4 :- Server receives Ack connection closed.


Client
Server
close
FIN
Ack
DATA
Data write.
DATA
Ack
FIN
close
Ack
timed wait
closed

TCP State-transition Diagram



) How network programming is different from Computer network? Explain different protocols (at least 3) in each layer of OSI reference model Compare TCP, UDP, SCTP.

⇒ Computer Networking aims to study and analyze the communication process among various computing devices or computer systems that are linked or networked together to exchange information and share resources.

Network programming is about writing computer programs that talk to eachother over a computer network.

⇒ OSI reference model.

i) physical layer ÷ protocols are Bluetooth, Rs-232, Rs-449

a) Bluetooth = it is a short-range wireless technology standard that is used for exchanging data between fixed & mobile device over short distance using UHF radio waves in the ISM bands from 2.402 GHz to 2.48 GHz, and building personal area network.

b) Rs-232 = it is a standard protocol used for serial communication, it is used for connecting computer and its peripheral devices to allow serial data exchange between them.

(c) RS-449 = is a data standard that specifies the functional & mechanical characteristics of the interface between data terminal equipment, typically a computer & data communications equipment typically a modem or terminal server.

ii) Data link layer ÷ Protocols are Frame Relay, ARP, SLIP

(a) Frame Relay = it is a protocol that defines how frames are routed through a fast-packet network based on the address field in the frame.

(b) ARP = it works between layer 2 and layer 3 of the OSI reference model which is used when a device wants to communicate with some other device on a local network.

(c) SLIP = it is a simple protocol that works with TCP/IP for communication over serial ports & routers.

iii) Network layer = protocols are IP, ARP, ICMP.

(a) IP = it is the network layer communications protocol in the internet protocol suite for relaying datagrams across network boundaries.

(b) ARP = it is a network protocol used to find out the hardware (MAC) address of the device from an IP address

ⓒ ICMP = the internet Control message protocol is a supporting protocol in the internet protocol suite. It is used by network devices. including routers to send error messages & operation information indication success or failure when communicating with another IP address.

iv) Transport layer = protocols are TCP, UDP, SCTP.

ⓐ TCP = it is designed to send packets across the internet and ensure the successful delivery of data and messages over networks.

ⓑ UDP = it allows data to be transferred very quickly but it can also cause packets to become lost in transmit and create opportunities for exploitation in the form of DDos attacks.

ⓒ SCTP = it provides multihoming support where one or both endpoints of a connection can consist of more than One IP address. This enables transparent failure between redundant network paths.

v) Session layer = protocols are RTP, PPTP, RPCP.

ⓐ RTP = it provide feedback on the quality of service (QOS) in media distribution periodically sending statistical information such as transmitted octet & packet counts or packet loss to the participants in the Streaming multimedia session. (Real-time transport Protocol).

(b) PPTP (point-to-point Tunneling protocol)

→ It provides a method for implementing virtual private networks. It provides security levels and remote access level comparable with typical VPN products.

(c) RPCP (Remote Procedure Call protocol)

→ it is used when a computer program causes a procedure (or a sub-routine) to execute in a different address space without programmer explicitly coding the details for the remote interaction.

vi) Presentation layer = Protocols are SSL, MIME, NDR.

(a) ~~SSL (Secure~~

(a) SSL (Secure Socket layer) = It is a protocol for web browser and servers that allows for the authentication, encryption and decryption of data sent over the internet

(b) MIME (Multipurpose Internet Mail Extension)

→ It is an internet standard that provides scalable capable of email for attaching of image, sound & text in a message.

(c) NDR (Network data Representation)

→ it provides or define various primitive data types, constructed data types & also several types of data Representations.

vii) Application layer = protocols are FTP, TFTP, Telnet

(a) FTP (file transfer protocol) = It promotes sharing of files via remote computers with reliable and efficient data transfer.

(b) TFTP (Trivial File Transfer Protocol)

→ it is the protocol of choice for transferring files between network devices & is simplified version of FTP.

(c) Telnet = it helps in terminal Emulation. It allows telnet clients to access the resources of the telnet server. It is used for managing files on the internet.

Comparing TCP, UDP & SCTP.

TCP	SCTP
(1) TCP doesn't supports multi-streaming	(1) SCTP supports multi streaming
(2) TCP data transfer is less secure	(2) SCTP has more secure data transfer.
(3) There is no partial data transfer in TCP.	(3) There is partial data transfer in SCTP.
(4) TCP doesn't supports multi-homing.	(4) Multi-homing is supported by SCTP.

TCP	SCTP
(5) TCP does not have unordered data delivery.	(5) There is unordered data delivery in SCTP.
(6) In TCP, the selective ACKs are optional.	(6) In SCTP, there are selective ACKs
(7) There is less reliable data transfer in TCP.	(7) There is more reliable data transfer in SCTP.

SCTP	UDP
(1) SCTP supports multistreaming	(1) UDP doesn't support multistreaming
(2) Data transfer is more reliable than SCTP.	(2) There is no reliable data transfer in UDP.
(3) Multihoming is supported by SCTP	(3) Multihoming is not supported by UDP
(4) SCTP protocol is connection-oriented.	(4) UDP protocol is not connection-oriented.
(5) There is partial data transfer in SCTP.	(5) There is no partial Data transfer in UDP.

③ What are the ways to pass the length of socket structure for different socket API's argument? Explain them in detail with function prototype & argument detail.

→ Most socket functions require a pointer to a socket address structure as an argument.

→ Generic socket address structure to hold socket information. It is passed in most of the socket function calls.

→ Unix socket programming provides IP version specific socket address structures.

Generic Socket address :-
- for address arguments to connect, bind & accept.

```
Struct sockaddr {
    unsigned short sa_family; /* protocol family */
    char sa_data[14]; /* address data */
};
```

Internet-specific socket address (IPv4 socket address structure) :-
- Must cast (sockaddr_in *) to (sockaddr *) for connect, bind, and accept.

```
Struct sockaddr_in {
    unsigned short sin_family; /* address family (always AF_INET) */
    unsigned short sin_port; /* port num in network byte order */
    Struct in_addr sin_addr; /* IP addr in network byte order */
    unsigned char sin_zero[8]; /* pad to size of (struct sockaddr) */
};
```

```
struct in_addr {
    unsigned long s_addr;   // this holds IPv4 address.
};
```

IPv6 Socket Address structure.

```
struct sockaddr_in6 {
    u_int16_t       sin6_family;    // AF_INET6.
    u_int16_t       sin6_port;      // this holds port number.
    u_int32_t       sin6_flowinfo;  // this holds IPv6 flow
                                    information.
    struct in6_addr sin6_addr;      // used to hold IPV6 address.
    u_int32_t       sin6_scope_id;  // scope id
};

struct in6_addr {
    unsigned char s6_addr[16];      // this holds IPv6
                                    address
};
```

→ When a socket address structure is passed to any socket function, it is always passed by reference. That is, a pointer to the structure is passed.

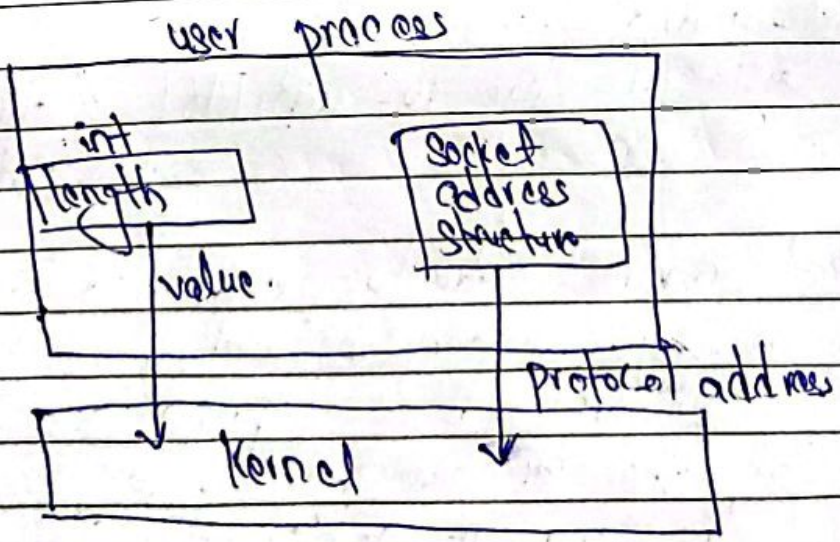
→ The length of the structure is also passed as an argument. But the way in which the length is passed depends on which direction the structure is being passed.

→ from process to kernel

bind(), connect(), and sendto() functions pass a socket address structure from process to the kernel. Arguments to those functions:
the pointer to the socket address structure
the integer size of the structure.

```
Struct sockaddr_in serv;
    /* fill in serv[] */
connect (sockfd, (SA *) &serv, sizeof (serv));
```

the datatype for the size of a socket address structure is actually socklen_t & not int, but the POSIX specification recommends that socklen_t be defined as uint32_t.



from kernel to process

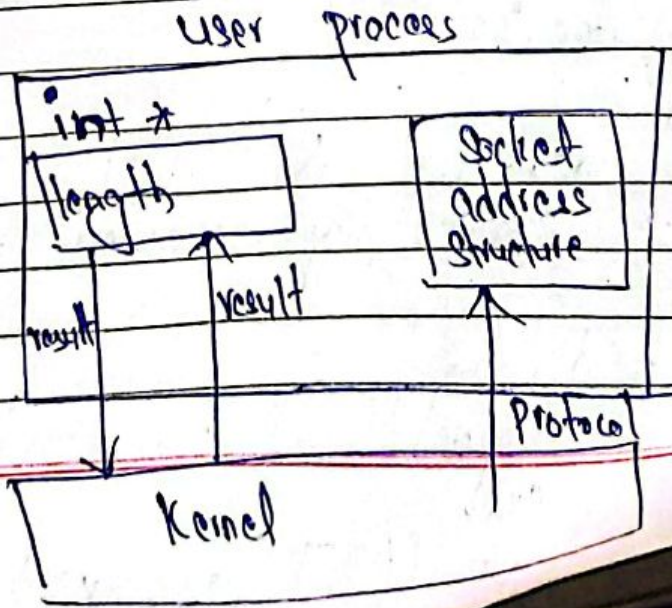
accept(), recvfrom(), getsockname(), and getpeername()

functions pass a socket address structure from the kernel to the process.

Arguments to these functions.
the pointer to the socket address structure.
the pointer to an integer containing the size of the structure.

```
struct sockaddr_un cli;  /* Unix domain */
socklen_t len;
len = sizeof(cli);       /* len is a value */
getpeername(unixfd, (SA *) &cli, &len);
                         /* len may have changed */
```

- Value-only = bind, connect, sendto (from process to kernel)
- Value-result = accept, recvfrom, getsockname, getpeername (from kernel to process, pass a pointer to an integer containing size)
  ↳ tells process how much information kernel actually stored.



4) Explain the statement: fork() is called once, and it returns twice. What is the significance of calling exec() after fork() system call? What will happen when close() is called in TCP socket during the implementation of concurrent server?

→ A call to fork() will create a completely separate sub-process which will be exactly the same as the parent.

→ the process that initiates the call to fork is called the parent process.

→ the new process created by fork is called child process.

→ fork() system call return an integer to both the parent & child processes.

→ -1 indicates an error with no child process created.

→ A value of zero indicates that the child process code is being executed.

→ positive integers indicates the child's process identifier (PID) and the code being executed is in the parent process

```
if(pid = fork() == 0)
    printf("I am child\n");
else
    printf("I am parent\n");
```

→ Significance of calling exec() after fork() system call is to terminate the currently running program and start executing a new one which is specified in the parameter of exec in the context of the existing process.

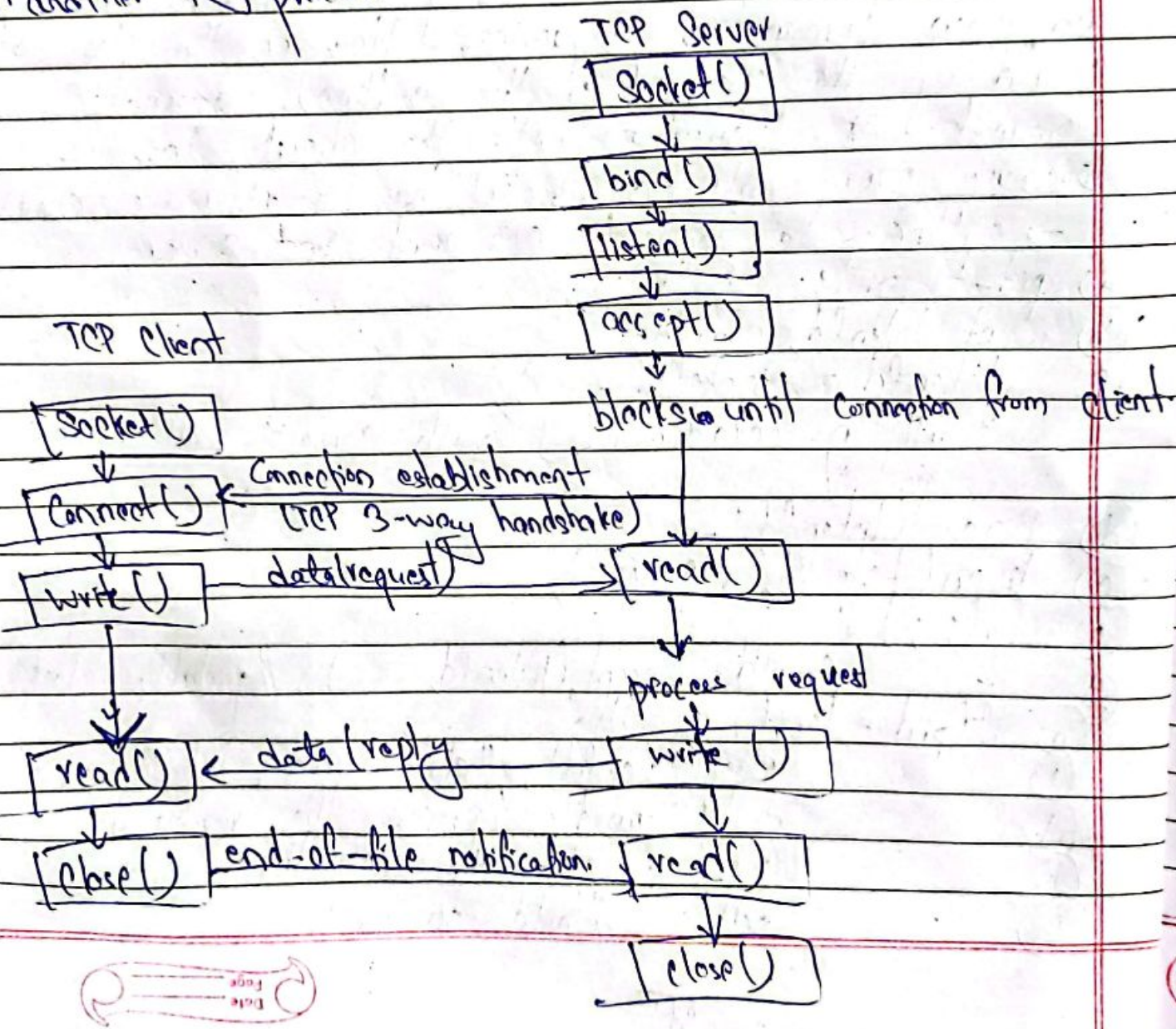
→ When close() is called on a TCP socket, it causes a FIN to be sent, followed by the normal TCP connection Termination sequence. However, the close of connfd by the parent (in the outline) doesnot terminate its connection with the client. This is because every file or socket has a reference count.

→ outline of typical concurrent server.

```
(a) pid_t pid;
(b) int listenfd, connfd;
(c) listenfd = Socket(---);
    /* fill in sockaddr_in{} with server's well-known
       port */
(d) Bind(listenfd, ---);
(e) listen(listenfd, LISTENQ);
(f) for(;;) {
(g)     connfd = Accept(listenfd, ---); // probably blocks
(h)     if((pid = Fork()) == 0) {
(i)         close(listenfd); // child closes listening socket
(j)         doit(connfd); // process the request
(k)         close(connfd); // done with this client
(l)         exit(0); // child terminates
        }
(m)     close(connfd);
```

(5) What is Socket? Explain different system call in specific order, required to create a TCP client and TCP server in the Unix System. [Hint: Explain each function with prototypes, return types and sample code].

→ Socket is an abstraction that is provided to an application programmer to send or receive data to another process.



i) Socket()

→ to perform network I/O, the first thing a process must do is call the socket function, specifying the type of communication protocol desired (TCP using IPV4, UDP using IPv6, Unix domain stream protocol, etc)

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

Returns ÷ non-negative descriptor if Ok, -1 on error.

family is one of
- AF_INET (IP v4), AF_INET6 (IPV6), AF_LOCAL (Local Unix), AF_ROUTE (access to routing tables), ~~AF_KEY (new~~ AF_KEY (new, for encryption)

type is one of
- Sock_STREAM (TCP), SOCK_DGRAM (UDP)
- SOCK_RAW (for special IP packets, PING, etc. must be root)
- Sock_SEQPACKET (sequenced packet socket)

protocol is one of
- IPPROTO_TCP
- IPPROTO_UDP
- IPPROTO_SCTP
- protocol is 0 (used for some raw socket options)

ii) Bind()

→ 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.

int bind(int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);

→ returns: 0 if OK, -1 on error

Sockfd is socket descriptor from socket()
myaddr is a pointer to address struct with:
- port number and IP address
- if port is 0, then host will pick ephemeral port
  - not usually for server (exception RPC port-map).
- IP address != INADDR_ANY (unless multiple nics)

addrlen is length of structure.
returns 0 if ok, -1 on error
- EADDRINUSE("Address already in use").

iii) LISTEN()

→ Connect function is used by a TCP client to establish a connection with TCP server.
Int listen(int sockfd, int backlog)
change socket state for TCP server.

- Sockfd is socket descriptor from Socket().
- backlog is maximum number of incomplete connections.
  - historically 5
  - rarely above 15 on a even moderate web server.

Sockets default to active (for a client)
- change to passive so OS will accept connection.

→ An incomplete connection queue, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of TCP 3-way handshake.
→ A complete connection queue which contains an entry for each client with whom the TCP three-way handshake has completed.
→ In term of the TCP state transition diagram, the call to the listen moves the socket from CLOSED state to LISTEN state.

(iv) CONNECT()

→ int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);

Connect to Server.
→ Sockfd is Socket descriptor from socket().
→ Servaddr is a pointer to a structure with:
  - port number and ip address
  - must be specified (unlike bind())
→ addrlen is length of structure.
→ returns socket descriptor if Ok, -1 on error.
→ the client does not have to call bind before calling connect, the kernel will choose both an ephemeral port & the source IP address if necessary.

→ ETIMEDOUT : host doesn't exist (connect time out)
→ ECONNREFUSED : no process is waiting for connections on the server host at the port specified.
→ EHOSTUNREACH : no route to host.

(V) ACCEPT()

→ int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);

Return next completed connection.
→ sockfd is socket descriptor from socket()
→ cliaddr and addrlen return protocol address from client.
→ returns brand new descriptor, created by the kernel. This new descriptor refers to the TCP connection with the client.
→ The listening socket is first argument (sockfd) to accept (the descriptor created by socket and used as the first argument to both bind and listen).

→ the Connected socket is the return value from accept the connected socket.
→ A given server normally creates only one listening socket, which then exists for the lifetime of the server.
→ the kernel creates one connected socket for each client connection that is
→ when a server is finished serving a given client, the connected socket is closed.

(vi) SENDING AND RECEIVING

→ int recv(int sockfd, void *buff, size_t mbytes, int flags)
int send(int sockfd, void *buff, size_t mbytes, int flags)

- Same as read() and write() but for flags.
- MSG_DONTWAIT (this send non-blocking)
- MSG_OOB (out of band data, 1 byte sent ahead)
- MSG_PEEK (look, but don't remove)
- MSG_WAITALL (don't give me less than max)
- MSG_DONTROUTE (bypass routing table).

(vii) CLOSE()

→ int close(int sockfd);
Close socket for use.
Sockfd is socket descriptor from socket()
closes socket for reading/writing.
- return (doesn't block)
- attempts to send any unsent data
- Socket option SO_LINGER.
  - block until data sent
  - or discard any remaining data.
- returns -1 if error.
- the default action of close with a TCP socket is to mark the socket as closed and return to the process immediately.
- The socket descriptor is no longer usable by the process. It cannot be used as an argument to read or write.

Sample Code

// daytime tcp server.

```
#include<stdio.h>
#include<netinet/in.h>
#include<sys/socket.h>
#include<arpa/inet.h>
#include<stdlib.h>
#include<unistd.h>
#include<errno.h>
#include<string.h>
#include<sys/types.h>
#include<time.h>

#define MAXLINE 1024
#define LISTENQ 10
typedef struct sockaddr SA;
int main(int argc, char **argv)
{
    int listenfd, connfd;
    struct sockaddr_in servaddr, cliaddr;
    char buff[MAXLINE];
    time_t ticks;
    int port;
    socklen_t len;
    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    port = atoi(argv[1]);
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
```

```
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(port);
bind(listenfd, (SA*) &servaddr, sizeof(servaddr));
printf("Server is waiting connection at port %d\t\n", port);
listen(listenfd, LISTENQ);
for(;;){
        cn = sizeof(cliaddr);
    connfd = accept(listenfd, (SA*) &cliaddr, &cn);
printf("connection from %s, port %d\n",
    inet_ntop(AF_INET, &cliaddr.sin_addr.s_addr,
        buff, sizeof(buff)), ntohs(cliaddr.sin_port));
ticks = time(NULL);
snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
write(connfd, buff, strlen(buff));
close(connfd);
}
}
```

// daytime tcp client

```
int main(int argc, char **argv)
{
    int sockfd, n;
    char recvline[MAXLINE+1];
    struct sockaddr_in servaddr;
    int port;
    if(argc!=3)
        printf("usage: a.out <IP address> <port no.>");
```

```
if((sockfd = socket(AF_INET, SOCK_STREAM, 0))<0)
    printf("Socket error");
port = atoi(argv[2]);
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(port);

if(inet_pton(AF_INET, argv[1], &servaddr.sin_addr)<=0)
    printf("inet_pton error for %s", argv[1]);

if(connect(sockfd, (SA*) &servaddr, sizeof(servaddr))<0)
    printf("Connect error");
while((n = read(sockfd, recvline, MAXLINE))>0 {
    recvline[n]=0
    printf("Server time");
    if(fputs(recvline, stdout) == EOF)
        printf("fputs error");
}
if(n<0)
    printf("read error");
    exit(0);
}
```

6) What will happen if you call bind() in tcp client program? Explain the situation with relevant example code where you can use send() & recv() in UDP socket & sendto() and recvfrom() in tcp socket.

⇒ if we call bind() in tcp client program then the client domain will be limited i.e. it will connect to specific port of the server only.

⇒ We can use send() and recv() in UDP socket by following :-

i) In Server Side

⇒ if socket descriptor is bind to particular Ip and port.
for eg :-

```
int sockfd, struct sockaddr_in address;
sockfd = socket(AF_INET, Sock_DGRAM, IPPROTO_UDP);
memset(& address, 0, size of (address));
address.sin.family = AF_INET.
address.sin_port = htons(PORT);
address.sin_addr.s_addr = htonl(INADDR_ANY).
bind(sockfd, (struct sockaddr *) & address, size of address);
send(new sockfd, buff, size of (buff)).
recv(new_sockfd, buff, sizeof (buff));
```

ii) In client side

⇒ if Connect() is used in UDP Socket we can use send() for sending and recv() for receiving.

for eg:-

(*) Sockfd = Socket(AF_INET, Sock_DGRAM, IPPROTO_UDP);
(*) memset(&address, 0, sizeof(address));
(*) address.sin_family = AF_INET;
(*) address.sin_port = htons(PORT);
(*) address.sin_addr.s_addr = inet_addr(IPADDRESS);
(*) Connect(Sockfd, (struct sockaddr*) &address, sizeof(address));
(*) send(Sockfd, buff, sizeof(buff));
(*) recv(Sockfd, buff, sizeof(buff));

→ We can use sendto() and recvfrom() in TCP Socket by:-

i) In Server Side
→ if we set NULL in argument of sendto and recvfrom it behaves like send and recv respectively.

for eg:-

(*) Sockfd = SOCKET(AF_INET, SOCK_STREAM, IPPROTO_TCP);
(*) memset(&address, 0, sizeof(address));
(*) address.sin_family = AF_INET;
(*) address.sin_port = htons(PORT);
(*) address.sin_addr.s_addr = htonl(INADDR_ANY);
(*) bind(Sockfd, (struct sockaddr*) &address, sizeof(address));
(*) Listen(Sockfd, BACKLOG);
(*) while(1){
(*) socklen_t addrlen = sizeof(client addr);
(*) newsockfd = accept(sockfd, (struct sockaddr*) &clientaddr, &addrlen);

ii) In Client Side

→ We can use sendto and recvfrom same as send and recv respectively, if we set NULL in the argument.

eg

(*) sockfd = SOCKET(AF_INET, SOCK_STREAM, IPPROTO_TCP);
(*) memset(&address, 0, sizeof(address));
(*) address.sin_family = AF_INET.
(*) address.sin_port = htons(PORT).
(*) address.sin_addr.s_addr = inet_addr(IPADDRESS));
(*) connect(sockfd, (struct sockaddr *) &address, sizeof(address));
(*) sendto(sockfd, buff, sizeof(buff), 0, NULL, NULL);
(*) recvfrom(sockfd, buff, sizeof(buff), 0, NULL, NULL).

(9) What is the purpose of bind() function? what will be the outcome if we do not specify IP address, port, both, or neither during bind() system call?

→ The bind function assigns a local protocol address to a socket.

i) Calling bind lets us specify the IP address, the port, both or neither. The following table summarizes the values to which we set sin_addr and sin_port, or sin6_addr and sin6_port depending on the desired result.

IP address	Port	Result
(*) wildcard	0	Kernel chooses IP address & port.
(*) wildcard	non-zero	Kernel chooses IP address, process specifies port.
(*) Local IP address	0	process specifies IP address, Kernel chooses port.
(*) Local IP address	non-zero	process specifies IP address and port.

ii) if we specify a port number of 0, the kernel chooses an ephemeral port when bind is called.

iii) if we specify a wildcard IP address, the kernel does not choose the local IP address until either the socket is connected (TCP) or a datagram is sent on the socket (UDP).

iv) With IPv4, the wildcard address is specified by the constant ~~INADDR~~ INADDR_ANY, whose value is normally 0. This tells the kernel to choose the IP address.

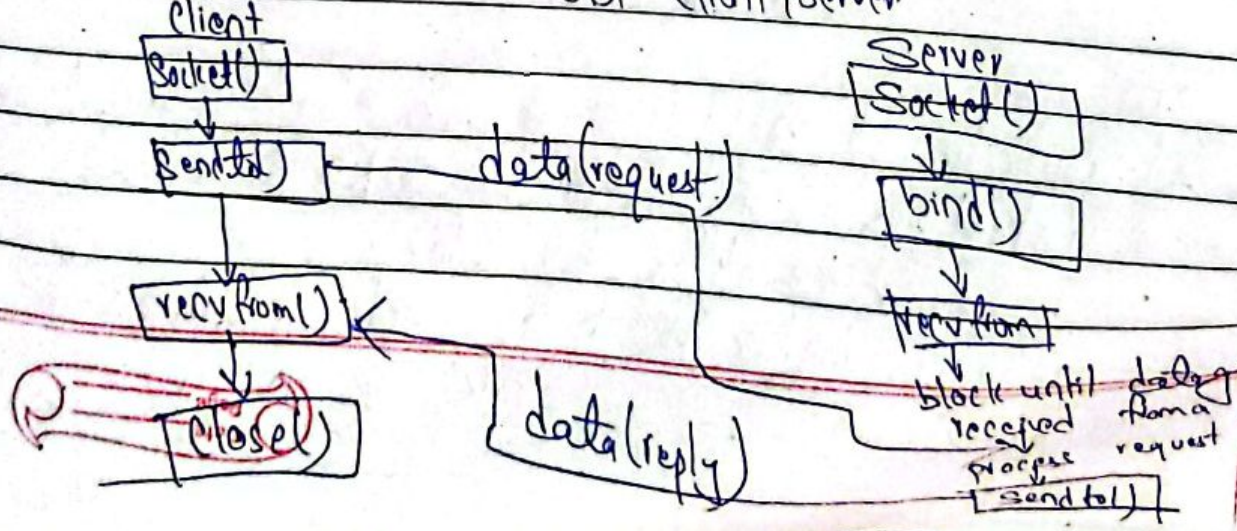
```
struct sockaddr_in servaddr.
servaddr.sin_addr.s_addr = htonl(INADDR_ANY); /* wild Ca
```

10) What is Completed Connection queue and incomplete connection queue()? Explain different system call in specific order required to create a UDP client & UDP server in the Unix system.

→ A completed connection queue, which contains an entry for each client with whom the TCP three way handshake has completed. This sockets are in the ESTABLISHED state.

→ An incomplete Connection queue, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three-way handshake. These sockets are in the SYN_RCVD state.

→ Socket Functions for UDP client/server.



i) Socket()

→ to perform network I/O, the first thing a process must do is call the socket function, specifying the type of communication protocol desired (TCP using IPv4, UDP using IPv6, etc).

- int Socket(int AF_INET, int SOCK_DGRAM, 0)
- Returns: non-negative descriptor if Ok; -1 on error.

ii) BIND()

→ The bind function assign a local protocol address to a socket with port number.

→ int bind (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);

→ Returns: 0 if Ok; -1 on error.

iii) RECVFROM() & SENDTO() Function.

→ #include<sys/socket.h>

Ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags, struct sockaddr *from, socklen_t *addrlen);

Ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags, const struct sockaddr *to, socklen_t addrlen);

//Both return: number of bytes read or written if Ok, -1 on error.

- the first three arguments, sockfd, buff and nbytes are identical to the first three arguments for read and descriptor, pointer to buffer to read into a write from and number of bytes to read or write.

- the to argument for sendto is a socket address structure containing the protocol address (eg: IP address and port number) of where the data is to be sent.

- the final argument to sendto is an integer value, while the final argument to recvfrom is a pointer to an integer value (a value-result argument).

(iii) ~~Close~~

(iv) close()

→ close socket for use

int close(int sockfd);

→ closes socket for reading/writing.

(11) Can we use Connect() system call while developing UDP socket? If yes, what will be the outcomes of using connect() in UDP socket? if no, how communicating processes know each other? Explain them in detail.

→ Yes, we can use connect() system call while developing UDP socket.

While calling connect for a UDP socket the kernel just checks for any immediate errors (eg:- an obviously unreachable destination), records the IP address and port number of the peer, and returns immediately to the calling process. Obviously there is no 3-way handshake with this capability, it can distinguish between

i) An unconnected UDP socket, the default when we create a UDP socket.

ii) A connected UDP socket the result of calling connect on a UDP socket.

→ With a connected UDP socket, three things change compared to the default unconnected UDP socket:

i) we can no longer specify the destination IP address and port for an output operation we don't use 'sendto' but 'write' or 'send' instead. Anything written to a connected UDP socket is automatically sent to the

protocol address. (eg:- IP address & port) specified by Connect

ii) We do not need to use recv from to learn the sender of a datagram, but read, recv or recvmsg instead. The only datagrams returned by the kernel for an i/p operation on a connected UDP socket are those arriving from the protocol address specified in Connect.

iii) Asynchronous errors are returned to the process for connected UDP sockets.

→ figure given summarizes point.

Types of Socket	Write or Send	Send to that doesnot specify a destination	Send to that specifies a destination
TCP socket	Ok	Ok	EISCONN
UDP socket Connected	Ok	Ok	EISCONN
UDP Socket Unconnected	Ok	Ok	EISCONN
	EDESTA DDRREQ	EDESTADDR REQ	Ok Ok.

table :- TCP & UDP sockets :- Can a destination protocol address be specified

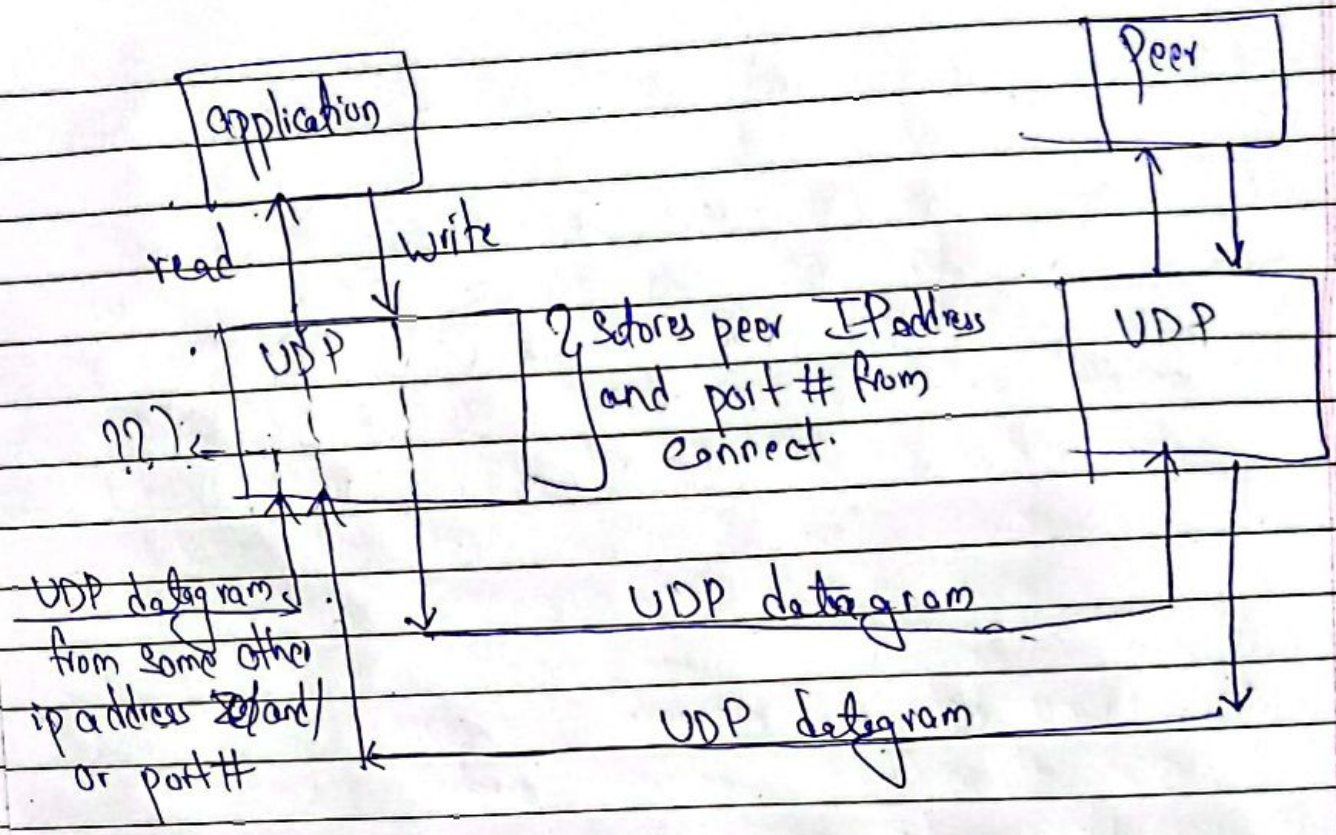


fig:- Connected UDP Socket

→ In Summary, UDP client or server can call 'connect' only if that process uses the UDP socket to communicate with exactly one peer. Normally, it is a UDP client that calls connect, but there application in which the UDP server communicates with a single client for a long duration (eg:- TFTP). in this case, both the client and server can call. connect.

12) The connect() function returns only when the connection is established or an error occurs, what are those errors, and when these errors occurred? Explain in detail.

→ int connect(int sockfd, const struct sockaddr * servaddr, socklen_t addrlen);

the connect() returns only when the connection is established or an error occurs. Those possible errors are :

i) if the client TCP receives no response to its SYN segment, ETIMEOUT is returned.

→ Connect moves from the CLOSED state (the state in which a socket begins when it is created by the socket function) to the SYN_SENT state and then, on success, the ESTABLISHED state.

ii) if the server's response to the client's SYN is a reset (RST), this indicates that no process is waiting for connections on the server, host at the port specified (the server process is probably not running). This is a hard error and the error ECONNREFUSED is returned to the client as soon as the RST is received.

iii) if the client's SYN elicits an ICMP "destination unreachable"

from some intermediate router, this is considered a soft error. The client kernel saves the message but keeps sending SYN with the same time between each SYN as in the first scenario. If no response is received after some fixed amount of time 175 seconds for 4.4 BSD the saved ICMP error is returned to the process as either EHOSTUNREACH or ENETUNREACH.

13) What is listening socket and connected socket? Explain with function prototype and sample code: when and why getsockname() and getpeername() required?

→ listening socket is the first argument (sockfd) to accept (the descriptor created by socket and used as the first argument to both bind & listen).

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

→ the connected socket is the return value from accept
the connected socket. accept() returns up to three values:
- an integer return code that is either a new socket descriptor or an error indication
- protocol address of the client process (through the cliaddr pointer).
- Size of this address (through the addrlen pointer).

#include <sys/socket.h>
int getsockname(int sockfd, struct sockaddr. * localaddr
socklen_t *addrlen).
int getpeername(int sockfd, struct sockaddr * peeraddr, socklen_t*
addrlen)

getsockname() and getpeername() is required due to following reasons.

- After connect successfully returns in a TCP client that doesn't call bind, getsockname() returns the local IP address and local port number assigned to the connection by the kernel.
- After calling bind with a port number of 0 (telling the kernel to choose the local port number), getsockname returns, the local port number that was assigned.
- getsockname can be called to obtain the address family of a socket.
- in a TCP server, that binds the wildcard IP address once a connection is established with a client, the server can call getsockname to obtain the local IP address assigned to the connection.
- When the server is execed by the process that calls accept, the only way the server can obtain the identity of the client is to call getpeername.

example :- obtaining the address family of a socket.
the ~~sockfd~~ sockfd_to_family function shown in below returns the address family of a socket.

lib/sockfd_to_family.c

```
#include "unp.h"
int sockfd_to_family(int sockfd)
{
    struct sockaddr_storage ss;
    socklen_t len;
    len = sizeof(ss);
    if (getsockname(sockfd, (SA *) &ss, &len) < 0)
        return(-1);
    return(ss.ss_family);
}
```

14) Explain the functions send(), sendto(), recv() and recvfrom(). What are the major difference bet wait() and waitpid()? Explain the possible option values that we can supply in waitpid() system call.

→ i) send()

⇒ send function is used to send data over stream sockets or connected datagram sockets. If we want to send data over UNCONNECTED datagram sockets, we must use sendto() function.

int send(int sockfd, void * buff, size_t nbytes, int flags);

ii) recv()

→ is used to receive data over stream sockets or CONNECTED datagram sockets. recv() receive data on a socket with descriptor socket & stores it in a buffer.

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

iii) sendto()

→ is use to send data over UNCONNECTED datagram sockets.

ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags, const struct sockaddr * to, socklen_t addrlen);

iv) recvfrom()
→ is use to receive data from Unconnected datagram sockets.

ssize_t recvfrom(int sockfd, void *buff, size_t n bytes, int flags, struct sockaddr * from, socklen_t * addrlen);

→ Either of wait & waitpid can be used to remove zombies. wait() and waitpid in its blocking form) temporarily suspends the execution of a parent process while a child process is running.
- once the child has finished, the waiting parent is restarted.

wait()	waitpid()
i) wait blocks the caller until a child process terminates	i) waitpid can be either blocking or non-blocking. * if option is 0, then it is blocking. * if option is WNOHANG, then it is non-blocking.
ii) if more than one child is running then wait() returns the first time one of the parents offspring exists	ii) waitpid is more flexible according to different values of pid.

→ the possible option values that we can supply in waitpid() system call are;

i) if pid == -1, it waits for any child process in this respect waitpid is equivalent to wait().

ii) if pid > 0, it waits for ~~any~~ the child whose process ID equals pid.

iii) if pid == 0, it waits for any child whose process group ID equals that of the calling process.

iv) if pid < -1 it waits for any child whose process group ID equals that absolute value of pid.

15) differentiate Unix domain Socket & internet domain Socket. Explain the mechanism of passing file descriptor in Unix System.

→ Unix domain Socket is used for same host, while internet domain socket is used for different host as well as on same host.

Mechanism of passing file descriptor in the Unix System :-

i) Create a Unix domain Socket either a stream socket or a datagram socket.
→ if the goal is to 'fork' a child and have the child open the descriptor & pass the descriptor back to the parent, the parent can call socket pair to create a stream pipe that can be used to exchange the descriptor.

ii) One process opens a descriptor by calling any of Unix functions that returns a descriptor. Any type of descriptor can be passed from one process to another.

iii) Sending process builds a "msghdr" structure containing the descriptor to be passed. The Sending process calls 'sendmsg' to send the descriptor across the Unix domain Socket.

- At this point, the descriptor is "in flight". Even if the

Sending process closes the descriptor after calling sendmsg but before the receiving process calls recvmsg. The descriptor remains open for the receiving process. Sending a descriptor increments the descriptor's reference count by one.

iv) The receiving process calls 'recvmsg' to receive the descriptor on the Unix domain socket. It is normal for the descriptor number in the receiving process to differ from the descriptor number in the sending process passing a descriptor involves creating a new descriptor in the receiving process that refers to the same file table entry within the kernel as the descriptor that was sent by the sending process.

16) What is Signal and why it is required? Explain POSIX signal handling mechanism. Also Explain two signal types.

→ Signal is a notification to a process that an event has occurred. Signals are sometimes called software interrupts. Signals usually occur synchronously.

→ Signal is required to notify a process that an event has occurred.

POSIX Signal handling mechanism :-

- Every signal has a disposition, which is also called the action associated with the signal we set the disposition of a signal by calling the 'sigaction' function. We have three choices for disposition.

i) We can provide a function that is called whenever a specific signal occurs. This function is called a signal handler and the action is called catching a signal. The two signals SIGKILL & SIGSTOP cannot be caught.

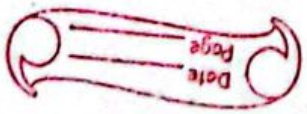
ii) We can ignore a signal by setting its disposition to SIG_IGN. The two signals SIGKILL and SIGSTOP cannot be ignored.

iii) We can set the default disposition for a signal by setting its disposition to SIG_DFL.

There are few signals whose default disposition is to be ignored; SIGCHLD and SIGURG

→ few signal types are:-

Name	Default Action	Description
i) SIGINT	terminate process	interrupt program.
ii) SIGQUIT	Create Core Image	quit program.
iii) SIGILL	Create Core image	illegal instruction.
iv) SIGKILL	terminate process	Kill program.
v) SIGBUS	Create Core Image	bus error.
vi) SIGTERM	terminate process	Software termination signal
vii) SIGURG	discard signal	urgent Condition present on Socket.
viii) SIGSTOP	Stop process	stop (Cannot be Caught or ignored).
ix) SIGCONT	discard signal	Continues after stop.
x) SIGCHLD	discard signal	child status has changed.
xi) SIGIO	discard Signal	I/O is possible on a descriptor.
xii) SIGINFO	discard signal	Status request from keyboard.



Q-18

18) What is I/O multiplexing? Explain the use of select function in the context of I/O multiplexing in detail. (Hint: Do not forget possibilities of wait options).

→ I/O multiplexing is the capability to tell the kernel that we want to be notified if one or more I/O conditions are ready like input is ready to be read, or descriptor is capable of taking more output.

Select function

→ Allows the process to instruct the kernel to wait for any one of multiple events to occur and to wake up the process only when one or more of these events occurs or when a specified amount of time has passed. What descriptors we are interested in (readable, writeable or exception condition) and how long to wait?

```
#include <sys/select.h>
#include <sys/time.h>
int select(int maxfdp1, fd_set *readset, fd_set
           *writeset, fd_set *exceptset,
    const struct timeval *timeout);
/* returns: +ve count of ready descriptors, 0 on
   time out, -1 on error */
struct timeval {
        long tv_sec; /* seconds */
        long tv_usec; /* microseconds */
```

. the final argument timeout tells, the kernel how long to wait for one of the specified file descriptors to become ready. A ~~timeout~~ timeval structure specifies the number of second and microseconds.

→ possibilities for select function :

i) wait forever
→ return only when descriptor(s) is ready (specify timeout argument as NULL).

ii) wait upto a fixed amount of time.
→ Return when one of the specified descriptor is ready for I/O, but donot wait beyond the number of seconds and microseconds specified in the timeval structure pointed to by the timeout argument.

iii) Do not wait at all.
→ return immediately after checking the descriptors (called polling) (Specify timeout argument as pointing to a timeval structure where the time value is 0).

19) What is the basic difference between select() and poll() and pselect() function

→ Basic difference bet select() and poll() are:-

i) Select() call helps to create three bitmasks to mark which sockets and file descriptors we want to watch for reading, writing and errors and then the Os marks which ones in fact have had some kind of activity. poll() helps to create a list of descriptors ID's and the Os marks each of them with the kind of event that occurred.

ii) Select() is rather ~~chunky~~ outdated & inefficient, poll() approach works much better because we keep re-using the same data structure.

→ Poll()

→ int poll (struct pollfd *fdarray, unsigned long nfds, int timeout);

returns: count of ready descriptors, 0 on timeout, -1 on error.

* Each element of fd array is a ~~polle~~ pollfd structure that specifies the condition to be tested for a given descriptor.

* "poll fd" contains file descriptor fd, events of interest on fd, and events that occured on fd.

* to switch off a descriptor set the fd member of the pollfd structure to a negative value.

timeout value	Description.
INFTIM	Wait forever.
0	return immediately dont block.
>0	wait specified number of millisrcnd.

iii) Pselect()

```
#include<sys/select.h>
#include<signal.h>
#include<time.h>
int pselect(int manfdp1, fd_set *readset, fd_set *
    writeset, fd_set *except set, const struct
    timespec *timeout, const sigset_t *
    sigmask);
```

- Returns: Count of ready descriptors 0 on time out, 1 on error.
- pselect contains two changes from the normal select function.
- pselect uses the timespec structure, another POSIX invention, instead of 'timeval' structure.

```
Struct timespec{
    time_t tv_sec; /*seconds*/
    long tv_nsec /*nano seconds*/}
```

iv) -> the difference in these 2 structures is with the second member. The tv_nsec member of the newer structure specifies nanoseconds, whereas the tv_usec member of the older structure specifies microseconds.

(v) pselect adds a sixth argument, a pointer to a signal mask. This allows the program to disable the delivery of certain signals, test some global variables that are set by the handlers for these now disabled signals & then call pselect telling it to reset the signal mask.

17) Why I/O & I/O multiplexing is needed in Socket programming? Explain various I/O models in Unix System?

→ I/O is needed in socket programming because it allows bi-direction communication between client & server.

→ I/O multiplexing is needed in socket programming to tell the kernel that we want to be notified if one or more I/O conditions are ready (i.e. inputs is ready to be read or the descriptor is capable of taking more output). this capability is provided by the select and poll functions.

→ When a client is handling multiple descriptor I/O multiplexing is used.