23/01/23

⊛ Socket programming :-

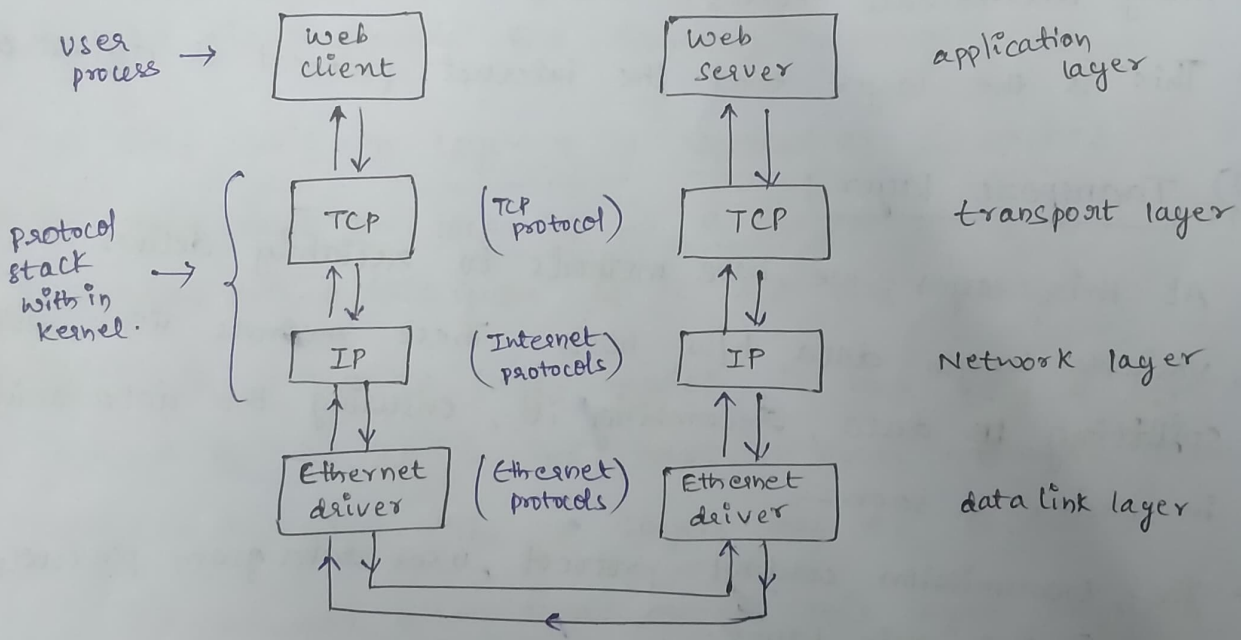
⇒ Most network applications can be divided into two pieces. one is server & second one is client

⇒ we can have only one server but can have multiple clients

⇒ server should run 1st & it should wait for the clients request. once it receives the request it processes the data & sends the response back.

⇒ Most of the servers based on TCP protocol.

⇒ Client to server communication over ethernet is TCP.



⇒ RJ45 port (female port) to connect ethernet cable

⇒ OSI layer Model (open system interconnection)

⇒ we have total 7 layers.

① physical layer :- This is the level of physical communication in the real world. At this level, we have specifications for things such as the voltage levels on an Ethernet cable

and each pin description on a connecter and the radio frequency of wifi and the light flashes over an optical fibre.

② Data link layer :- This is builds on physical layer. It deals with protocols for directly communicating between 2 nodes. It defines how a direct message b/w nodes starts & ends (framing, error detection, correction & flow control)

③ Network layer :- This Network level/layer provides the methods to transmit the data sequences called packets b/w nodes in different networks. It provides methods to route packets from one node to another by transferring through many intermediate nodes.

⇒ This is the layer that the internet protocol is defined on.

④ Transport layer :-

⇒ At this layer, we have methods to reliably deliver variable length data b/w hosts. These methods deals with splitting up data, recombining it, ensuring the data arrives in order & soon..

⇒ The transmission control protocol, user data gram protocols are used in this layer.

⑤ session layer :-

⇒ This layer builds on transport layer by adding methods to establish checkpoint, suspend, resume & transport terminate dialouges.

(6) Presentation layer :-

⇒ This is the lowest layer at which data structure & Presentation for an application are defined. Here, we can see concern such as data encoding, serialization & encryption handling

(7) Application layer :-

⇒ The Applications that the user interfaces with exist here, like, web browser & email clients.

⇒ These applications makes use of the services provided by the 6 lower layers.

⇒ The data unit on a layer 2 is called frame that means, layer 2 is responsible for framing messages.

⇒ The data unit on layer 3 is referred as a packet

⇒ The data unit on layer 4 is a segment; If it is a TPC connection (or) a data gram if it is a UDP message.

⇒ If it is a TCP IP layer model, it has only 4 layers instead of 7. It is the most common network communication model used today. The 4 layers are as follows :-

① Network access layer :- On this layer, physical connections & data framing happens.

② Internet layer :- In this layer, an ip address is defined, this layer deals with concerns of addressing packets & routing them over multiple inter connection networks.

③ host to host layer :- This layer provides 2 protocols i.e., TCP & UDP. These protocols address concerns such as data order, data segmentation, network congestion, error correction

④ process (or) Application layer :- Here, In this layer STTP, SMTP, FTP are implemented.

⊛ Data Encapsulation :-

⇒ A web browser needs only to implement the protocols dealing specifically with websites - STTP, HTML, CSS & so on.. It doesn't need to bother with implementing TCP/IP & the packet encoded on ethernet or wifi

⇒ when communicating over a network, data must be processed down through the layers at the sender & up again through the layers at the receiver

⇒ A web page contains a few paragraphs of text, this text must be encoded in an HTML structure.

⇒ AS the text cannot be transmitted directly. it must be transmitted as a part of HTTP response.

⇒ The web server does this by applying the appropriate HTTP response header to the HTML

⇒ The HTTP is transmitted as a part of TCP session This is not done by the web server. This is taken care by operating systems tcp/IP stack.

⇒ The TCP packet is routed by an IP address.

⇒ This is transmitted over the wire in an ethernet packet or another protocol.

⇒ The lower level concerns are handelled automatically when we use the socket API's for network programming.

(*) Internet protocols :-

It comes in two (2) versions. They are :-

① IP V4

② IP V6

① IPV4 uses 32 bit address, it can't be more than 32 bit value.

② IPV6 uses 128 bit address, it was designed to replace IPV4. Every operating system supports both IPV4 & IPV6 that is called a dual stack configuration.

(*) Address :-

⇒ All internet protocols traffic routes to an address

⇒ IPV4 addresses are 32 bit long. They are commonly divided into 4 8 bit sections. Each section is displayed as a decimal number between 0 & 255 & separated by a period (.) [dot].

⇒ A special address, called loop back address is reserved at 127.0.0.1 (self address).

⇒ IPV4 reserves some address ranges for private use. If we are using a router, it has an IP address. These are reserved as follows.

① 10.0.0.0 to 10.255.255.255.

② 172.16.0.0 to 172.31.255.255.

③ 192.168.0.0 to 192.168.255.255.

⇒ Class less inter domain routing address are as follows :-

① 10.0.0.0 to 10.0.0.8.

② 172.16.0.0 to 172.16.0.12.

③ 192.168.0.0 to 192.168.0.16.

⇒ IPV6 addresses are 128 bit long. They are written as 8 groups of 4 hexa decimal characters separated by colons. We must use lower cases to represent IPV6 addresses.

⊛ Link local addresses :-

⇒ These are usable only on the local links.

⇒ routers never forward packets from these addresses.

⇒ They are useful for a system to access auto configuration functions before having an assigned IP address. These addresses ranges from 169.254.0.0 to 169.254.0.16.

⊛ Domain names :- (....com)

⇒ The internet protocols can only route packets to an IP address, not a name

⇒ If we try to connect to a website, the system must resolve the domain name into an IP address. This is done by connecting a domain name to system server.

⇒ You connect to a domain name server by knowing in advance its IP address.

⇒ The IP address for a domain name server is usually assigned by ISP

(*) Internet routing :-

⇒ If all networks contained only a maximum of 2 devices. then there is no need for routing.

(*) Local n/w & address translation

⇒ When a packet originates from a device on an IPV4 local network it must undergo network address translation before being routed on the internet.

⇒ The devices on same the same LAN can directly address one another by their local address. However, any traffic communicated to the internet must undergo address translation by the router. The router does this by modifying the source IP address from the original private LAN IP address to its public Internet IP address

(*) Multicast, Broadcast, Anycast :-

⇒ When a packet is routed from one sender to one receiver it uses unicast addressing. This is the simplest & most common type of addressing.

⇒ Broadcast addressing allows a single sender to address a packet to all recipients simultaneously. It is typically used to deliver a packet to Every receiver on an entire subnet.

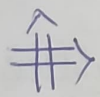
⇒ If a broadcast is one to all communication then multicast is one to many communication.

⇒ Multicast involves some group management, and a message is addressed & delivered to the members of a group

⇒ Any cast addressed packets are used to deliver a message to one receipent when you don't care who that receipent is.

⇒ IPV4 and lower network levels supports local broadcast addressing.

⇒ IPV6 mandates multicasting support while providing additional features over IPV4 multicasting.



⊛ Port numbers :-

⇒ An IP address gets a packet routed to a specific system but a port number is used to route the packet to a specific application on that system.

⇒ When your computer receives a TCP segment (or) UDP datagram your operating system looks at the destination port number in that packet. That port number is used to look up which application should handle it.

⇒ port numbers are stored as unsigned 16 bit integers. There are fixed port numbers are assigned by IANA (Internet assigned number authority).

⇒ 20, 21 ⇒ For FTP (File transport protocol)

⇒ Port 22 ⇒ For secure shell (ss)

⇒ port no. 23 ⇒ Telenet

⇒ port no. 25 ⇒ Simply mail transfer protocol (SMTP)

⇒ port no. 53 ⇒ Domain name system (DNS) (internally uses UDP)

⇒ port no. 80 ⇒ Hyper text transfer protocol (HTTP)

⇒ port no. 110 ⇒ post office protocol (POP3)

⇒ port no. 143 ⇒ Internet Message Access protocol (IMAP)

⇒ port no. 194 ⇒ Internet Relay chat (IRC)

⇒ port no. 443 ⇒ HTTP over TLS or SSL

⇒ port no. 993 ⇒ IMAP over TLS/SSL

⇒ port no. 995 ⇒ POP3 over TLS/SSL

⊛ Clients & servers :-

⇒ In this model, a server listens for the connections, the server listening on client knowing the address & port no that the establishes the connection by sending the first packet.

⊛ Socket :-

⇒ A socket is one end point of a communication link b/w systems.

⇒ It is an abstraction in which your application can send & receive the data over the network.

$ sudo eth0 → ethernet port 0
" " 1
" " 2
etc.

wifi router → WLAN 0
" 1
" 2

⇒ An open socket is uniquely defined by 5 tupples consisting of the following :-

1. local IP address
2. local port no
3. Remote IP address
4. Remote port no
5. protocol TCP / UDP

⇒ $ ifconfig → for linux To see IP address.

$ ipconfig → for windows

⇒ ifconfig command shows the IP address of each adopter on or system or computer

⇒ sockets come in 2 basic types :-

1. connection oriented
2. connection less.

⇒ The two protocols that are used today are transmission control Protocol & user data gram protocol.

⇒ TCP is a connection oriented protocol. (transmission control protocol)

⇒ UDP is a connection less protocol. (user datagram protocol)

⇒ Each application will have fixed port numbers

⇒ Each system will have IP address.

⇒ $ ping (IP address) command is used to verify the connections established or not.

$ ping IP address

⊛ TCP (Transmission control protocol) :- (connection oriented)

⇒ TCP is a connection oriented protocol that provides a reliable, full duplex byte stream to its users.

⇒ TCP sockets are an examples of stream sockets.
(continuous) packet transmission

⇒ TCP takes care of details such as aknowledgments, time outs, retransmissions etc.

⇒ Most internet application programs use TCP. This TCP can either use IPV4 (or) IPV6.

⊛ UDP (user ~~defined~~ datagram protocol) :-

⇒ UDP is a connection less protocol.

⇒ UDP sockets are an example of datagram sockets.

⇒ There is no guarantee that, UDP datagrams reaches to their end points or destination. It can also use IPV4 (or) IPV6 address.

⇒ ~~Structure used is struct~~

⇒ An IPV4 (32 bit) socket address structure is

#include <netinet/in.h>

```
struct sockaddr_in
{
unsigned int 8 bit → uint8_t   sin_len;  (length of the structure)
    sa_family_t   sin_family;  (Internet family)   IPV4 → AF_INET
                                                   IPV6 → AF_INET6
    in_port_t     sin_port;    (port no)
    struct in_addr  sin_addr;  → struct in_addr { in_addr_t s_addr; };
                                  (32 bit IP address)
                                  ↳ IPV4 address
    char sin_zero[8];
                ↳ unused.
};
```

(*) For IPV6 (128 bit address)

#include <netinet/in.h>

⇒ struct sockaddr_in6
{
    uint8_t sin6_len;
    sa_family_t sin6_family;
    in_port_t sin6_port;
    uint32_t sin6_flowinfo; → flow information undefined
    struct in6_addr sin6_addr; → struct in6_addr { uint8_t s6_addr[16]; }; (128 bit address of IPV6)
    uint32_t sin6_scope_id; → set of interfaces for a scope
};

⇒ If the communication is on the same Host machine, we can use AF_UNIX (or) AF_LOCAL in sin6_family.

(*) Find out the header files in which these macros are defined & corresponding MACROS. #include<sys/socket.h>

AF_local, AF_UNIX, AF_INET, AF_INET6, PF_INET, PF_INET6.

(*) Differences b/w TCP & UDP.

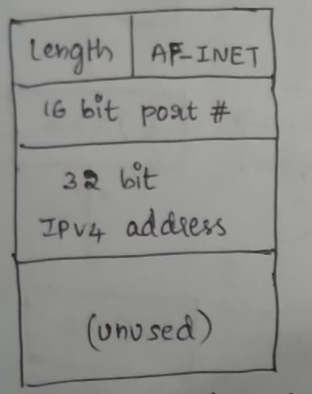
⇒ If we want to Establish TCP protocol. we use SOCK_STREAM

UDP " → SOCK_DGRAM

Find out the macro definitions.

IPV4

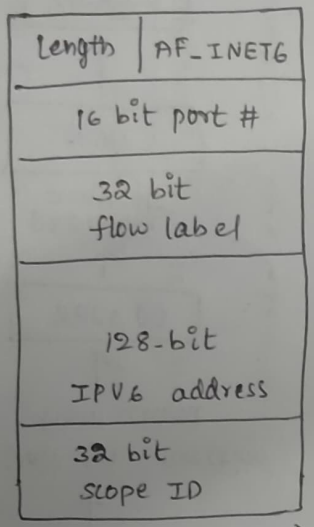
sockaddr_in{ }



Fixed length (16 bytes)

IPV6

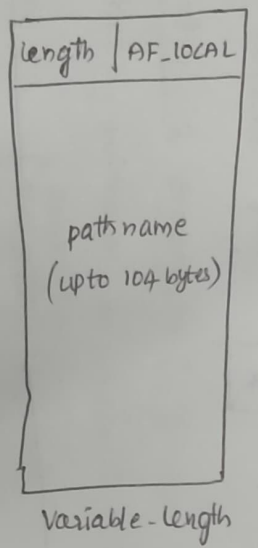
sockaddr_in6{ }



Fixed length (28 bytes)

Unix

sockaddr_un{ }



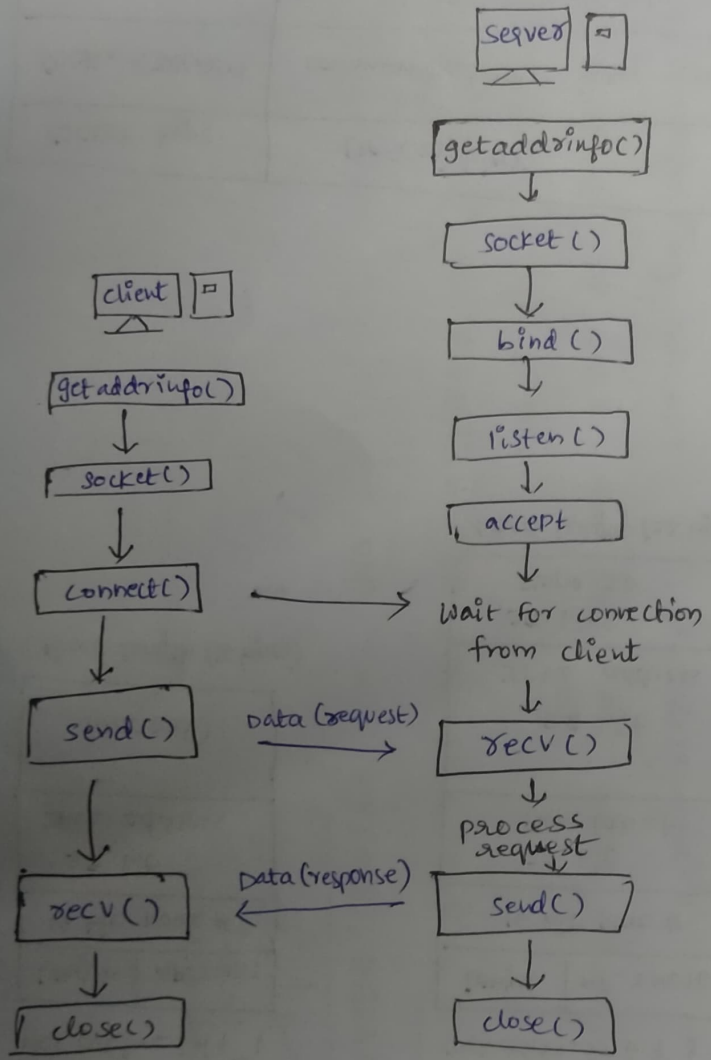
Variable-length

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

#define PF_INET   2// IP protocol family
#define PF_INET6  10 // IP version 6.

#define AF_INET   PF_INET
#define AF_INET   PF_INET6
```

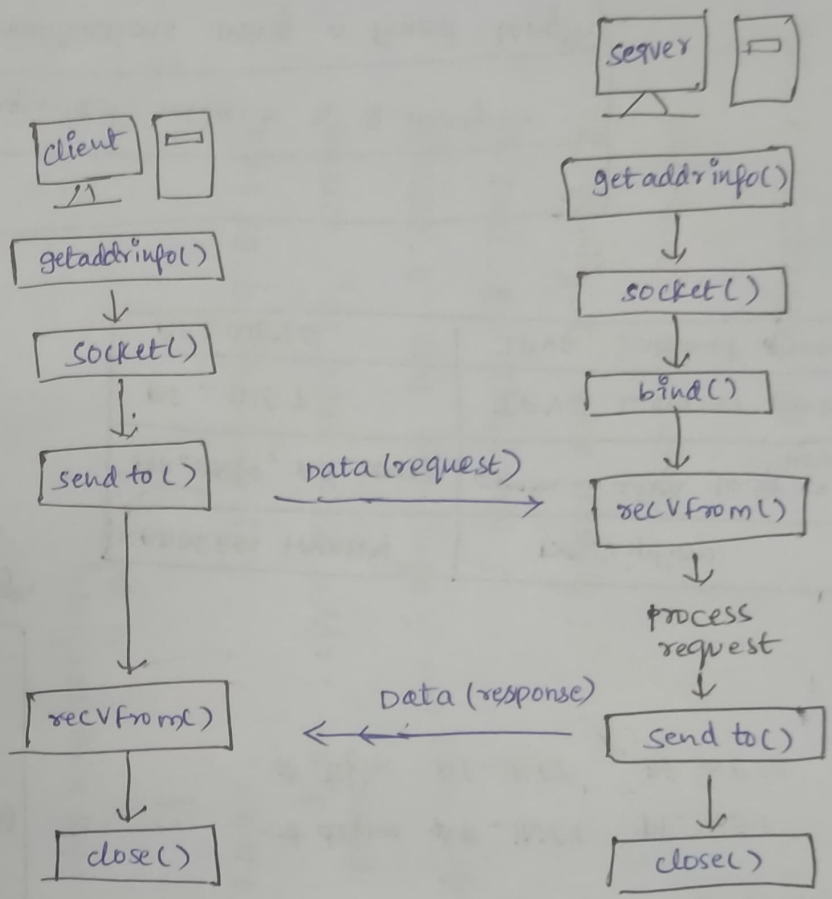
ADDRESS FAMILY	Description
AF_UNIX, AF_LOCAL	communication local to same host
AF_INET	IPV4 Internet protocols
AF_INET6	IPV6 Internet protocols.

Socket type	Description
SOCK_STREAM	communications are connection based, sequenced, reliable & 2-way
SOCK_DGRAM	connectionless, unreliable message-type communications using a fixed length

Flow of a TCP client & server



Flow of a UDP client & server



⊛ socket functions :-

⇒ socket() creates and initializes a new socket.

⇒ bind() associates a socket with a particular local IP address & port no.

⇒ listen() is used on the server to cause a TCP socket to listen for new connections.

⇒ connect() is used on the client to set the remote address & port. In the case of TCP, it also establishes a connection.

⇒ accept() is used on the server to create a new socket for an incoming TCP connection.

⇒ send() & recv() are used to send & receive data with a socket.

⇒ sendto & recvfrom() are used to send & receive data from sockets without a bound remote address.

⇒ close() (Berkeley sockets) and closesocket() (winsock sockets) are used to close a socket. In the case of TCP, this also terminates the connection.

⇒ shutdown() is used to close one side of a TCP connection. It is useful to ensure an orderly connection teardown.

⇒ select() is used to wait for an event on one or more sockets.

⇒ getnameinfo() and getaddrinfo() provide a protocol-independent manner of working with hostnames and addresses.

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

⇒ The socket call creates a network socket & returns a descriptor on success, on failure it will return -1.

⇒ Protocol family: AF_INET → IPV4 (or) AF_INET6 → IPV6.

⇒ Type: SOCK_STREAM → TCP

SOCK_DGRAM → UDP

⇒ protocol: IPPROTO_TCP / IPPROTO_UDP.

⇒ int close (int socket);

⇒ close () tells the underlying protocol stack to initiate any actions required to shut down communications & deallocate any resources associated with the socket, close () returns 0 on success and -1 on failure.

⇒ int bind (int socket, struct sockaddr *local address, unsigned int address length);

⇒ socket: integer, socket descriptor

⇒ Local address: struct sockaddr, the (IP) address & port of the machine.

⇒ For TCP/IP server, internet address is usually set to INADDR_ANY, i.e., chooses any incoming interface.

⇒ Address length: the size (in bytes) of the addrport structure

⇒ status :- on success it will return 0.
on failure it will return -1.

⇒ int connect (int socket, struct sockaddr *foreign addr...,
unsigned int address length);

⇒ Foreign address will always be a pointer to a socket addr-in containing the internet address & port of the server, address length specifies the length of the address structure & is invariably given as size of (struct sockaddr-in). When connect() returns successfully, the socket is connected & communication can proceed with calls to send() & recv().

⇒ int listen (int socket, int queue limit);

⇒ listen() causes internal state changes to the given socket, so that incoming TCP connection requests will be handled & then queued for acceptance by the program. The queue limit parameter specifies an upper bound on the number of incoming connections that can be waiting at any time.

⇒ It will return 0, on success
on failure → -1.

⇒ int accept (int socket, struct sockaddr *client address,
unsigned int *address length);

⇒ accept() dequeues the next connection on the queue for socket. If the queue is empty, accept() blocks until a connection request arrives. When successful, accept() fills in the sockaddr structure, pointed to by client address, with the address of the client at the other end of the connection, address length specifies the maximum size of the client address

address structure & contains the no. of bytes used for the address upon return. If successful, accept() returns a descriptor for a new socket that is connected to the client. The socket sent as the first parameter to accept() is unchanged (not connected to the client) & continues to listen for new connection requests. On failure, accept() return -1. The server communicates with the client using send() & recv();

⇒ int send(int socket, const void *msg, unsigned int msgLength, int flags);

⇒ int recv(int socket, void *rcvBuffer, unsigned int bufferLength, int flags);

⇒ send() & recv() have very similar arguments, socket is the descriptor for the connected socket through which data is to be sent or received, For send(), msg points to the message to send, and msglength is the length (bytes) of the message. The default behaviour for send() is to block until all the data is sent.

⇒ Ssize_t sendto(int socket, const void *msg, size_t msglength, int flags, const struct sockaddr *destAddr, sockLen_t addrLen);

⇒ msg, msgLength, flags, count : same with send()

⇒ destAddr : struct sockaddr, Address of the destination.

⇒ AddrLen : size of (foreign Addr)

⇒ ssize_t recvfrom (int socket, void *msg, size_t msgLength,
int flags, struct sockaddr * srcAddr, socklen_t *addrLen);

⇒ msg, msgLength, flags, count : same with recv()
⇒ srcAddr : struct sockaddr, address of the client.
⇒ addrLen : sizeof (clientAddr)

⇒ HOST Byte-ordering : the byte ordering used by a host (bit or little)

⇒ Network Byte-ordering : the byte ordering used by the N/w (always big Endian)

⇒ u_long htonl (u_long x);
⇒ u_short htons (u_short x);
⇒ u_long ntohl (u_long x);
⇒ u_short ntohs (u_short x);

⇒ On big-endian machines, these routines do nothing
⇒ On little-endian machines, they reverse the byte order.

⇒ int inet_aton (const char *cp, struct in_addr *inp);

⇒ converts the internet host address cp from the IPV4 numbers and dots notation into binary form (in network byte order).
⇒ stores it in the structure that inp points to.
⇒ it returns non zero if the address is valid, and 0 if not.
⇒ char *inet_ntoa (struct in_addr in);
⇒ converts the internet host address in, given in n/w byte order, to a string in IPV4 dotted decimal notation.

```
typedef uint32_t in_addr_t;
struct in_addr
{
    in_addr_t s_addr;
};
```

24/01/23

⇒ server to client communication using (TCP/IP)

Server

```
⇒ main()
{
    int sockfd, newsockfd, client_size, ret;
    char buf[256];
    struct sockaddr_in serv, client;
    sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    // memset(&serv, 0, sizeof(struct sockaddr_in));
    bzero(&serv, sizeof(struct sockaddr_in));
    serv.sin_family = AF_INET;
    serv.sin_port = htons(5000);
    serv.sin_addr.s_addr = INADDR_ANY;
    bind(sockfd, (struct sockaddr *)&serv, sizeof(serv));
    listen(sockfd, 5);
    client_size = sizeof(client);
    newsockfd = accept(sockfd, (struct sockaddr *)&client, &client_size);
```

- socket: fd
- TCP
- bzero: it will clear with 0.
- port number.
- htons: host to n/w (converts little endian to big endian)
- INADDR_ANY: to accept any one of client (or read client IP to accept particular client)
- bind: to make it generic
- accept: fd; client socket fd; blocking call for clients

```
    ret = read(newsockfd, buf, 256);
    write(1, buf, ret);
    ret = write(newsockfd, "Hello", strlen("Hello"));
    close(newsockfd);
}
```

- ret: no. of characters read
- write(1, ...): prints on terminal
- close(newsockfd); → closes complete server

client

```
⇒ main()
{
    int sockfd, ret;
    struct sockaddr_in serv;
    char buf[256] = "Hi";
    sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    bzero(&serv, sizeof(serv));
    serv.sin_addr.s_addr = inet_addr("127.0.0.1");
    serv.sin_port = htons(5000);
    serv.sin_family = AF_INET;
    connect(sockfd, (struct sockaddr *)&serv, sizeof(serv));
    ret = send(sockfd, "Hi", strlen("Hi"), 0);
    ret = recv(sockfd, buf, 256, 0);
    write(1,
    close(n fd);
}
```

- inet_addr: converts string into IP address
- unblocks: 1 client is connected
- sends hi to server
- receive hello from server to client.

Project - 2 (TCP)

⇒ write a client to server program to perform below bank operations like.

login
Register
deposit
Withdraw
bank balance.

Rules

① Maintain database for each user (user details & transaction history (maintain time))

② Below menu should be displayed.

① Login
② Register
③ exit.

① Login → verify login details in the database at server side.

if successfully verified → perform the operation requested by user.

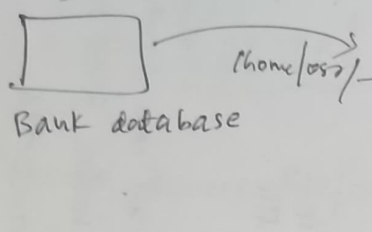
① deposit
② withdraw
③ Balance
④ Transaction History
⑤ profile info (display) → (Name, balance, DOB, registered date, Nominee name, account no (~~login id~~ + Aadhar no / mobile no.))
⑥ logout.

② Register → If user info doesn't exist in the server data-base, ask for below info.

⇒ user name
(⇒) ⇒ user id ⇒ Gender ⇒ Account no
⇒ pass word (> 8 characters) & 16
⇒ mobile no (exact 10 digits)
⇒ address (location) H-NO, Area, location, Dist., state, pincode.
⇒ registered date.
⇒ DOB ⇒ Balance
⇒ Nominee ~~⇒ name, ph no.~~ (name, ph no, Aadhar no)
⇒ Aadhar no (12 digits)

Once registration is successful → relogin to perform the operations.

Server



→ 1st file → profile info (register details)

→ 2nd file → transaction history.

↳ amount debited / credited, balance

read

1. login → { user id, password } if exist → 1. file info

2. register → { details }

filed in structures

↳ compares

OK → result 0

-1

#⇒ Server

Database (folder) specific path

(If user not registered with bank then ⇓)

↳ User details

↳ each user. folder (with user id as name)

↳ 2 files

1. profile
2. Transaction History

Registration / profile structure

(signed int) Balance
(character array) user name
(unsigned int) user id
(char array) password
(unsigned long int) Account no
(unsigned long int) Aadhar no
(unsigned int) Mobile no
(char) → Gender, Date of Birth → Another structure ⇒ { date, month, year }

registered time & date → (time to print, time / use predefined structure & calls to read & store time

Name, Mobile no., Aadhar no. ← Nominee details

Address → H.No, area, location, Dist., state, Pincode

(time stamps) Addition (or) subtraction.

Server

① Read info passed by client

② create a folder in database with user id

③ create 2 files
→ profile & → transaction history

④ copy profile info in profile file

done successfully

Client

1st menu
① → register
② → login
③ → exit

Client

Balance = 0

updates in
① profile balance
② update in to transaction history file.

User name
User id, gender
password
Mobile no
Aadhar no
Account no
DOB
Nominee details → { Name, mobile no, Aadhar no }
Address → { HNO, area, location, Dist, state, pin code }

Registered time & date — Use predefined structure & calls

client

2nd menu

separate structure
① Deposit
② withdraw
③ Balance
④ transaction history
⑤ profile info → Display
⑥ Logout

Name
Balance
DOB
Account no
Nominee name
registered date & time.

separate structure at server & client

server

updates profile file & close the file.

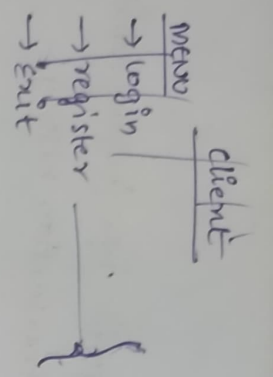
Server

① If database is not opened then open it.

② search a folder with userid
if exist file in structure

Ack = on success
-1 on failure
(or) msg

② relogin (or) Login
→ { user id, password }



Tip :-

⇒ create 1 header file & keep all the structures in it & just include the header file to use the desired structures.