first 2-3 Assgns → Solo
then team

5 Assgn
2 Lab test → 50-60%.
↑
8th March
12th April

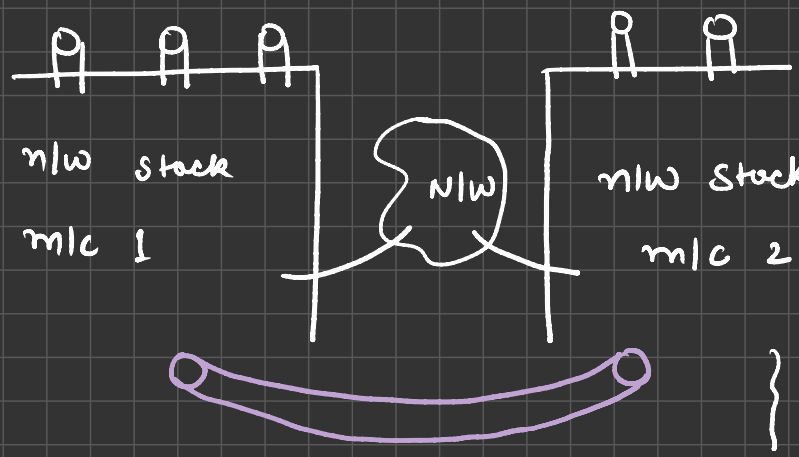
In a UDP concurrent server, all processes created are waiting on the same <IP address, port>. So a chunk sent by one client can be received by the process handling another client. Handling this properly makes UDP concurrent servers very complex (other than the fact that for this example, overhead will be too much), so you will hardly see any concurrent UDP server. It is a bad design..

#) Process - to - Process

< IP, Port >

↳ to identify the process (16 bit integer)

Socket



TCP
UDP

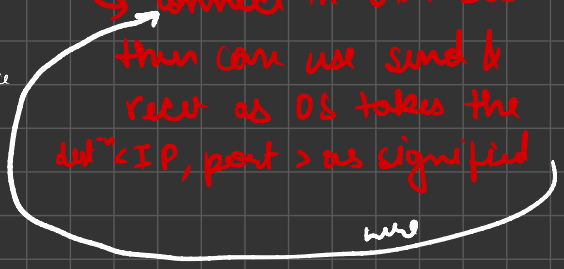
} TCP Sockets

Reliable, FIFO

gets the service → Client / Server → provides service

UDP → no guarantees about reliability

↳ connect in UDP sockets then can use send & recv as OS takes the dst <IP, port> as signified



Usable Socket always has a IP addr & Port. If not then OS does automatically.

sd = socket (..., ..., 0) ;

- sd → socket descriptor
- ... → family
- ... → UDP
- 0 → zero

blocking call → server waiting for msg. at recv from line until we receive smth. your code will stop

UDP → message centric Protocol

↳ one msg send one msg receive

UDP doesn't know max limit of server. → no specific order

- Packet sniffer

Read man pages for calls

IP addr. is for an interface.

(Eg - LAN & wifi will have diff. IP addr.)

"INTEROPERABILITY"

→ There is a specific network byte order.

htons(8181);
↓
host to network short as 16 bit.

need to pass len & addr too

will get those in return as well

#) TCP Socket :- → Connection oriented protocol

Client

"

connect (sockfd, (struct sockaddr *) & servaddr, sizeof(...))

↓ Blocking

↓ socket for that connection → so smaller function calls.

send (sockfd, buffer, 455, 0);

Server

sd= "

bind

listen (sd , 5); → tells the OS, if multiple clients try to connect then ★ no. of clients will be queued up & after that the clients will

clilen = sizeof (cliaddr);

```
while (1) {
newsd = accept (lsd, ....);
}
```

get an error msg.

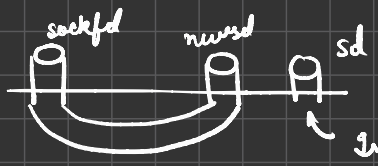
Blocking call

Breaking the connection → closing the socket

close(newsd);

returns a socket

→ sd1 & sd have the same IP & Port.

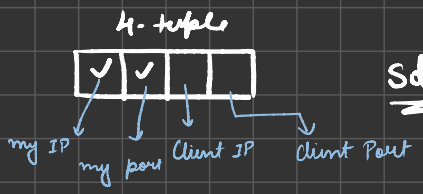


In TCP you cant use this directly.

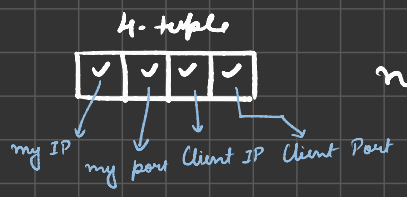
(sd) - Connection req. is a TCP packet.

(newsd) - Data is also a TCP packet.

Every TCP packet



Sd will have 2 fields empty.



new sd

Do longest match when a packet arrives.

byte oriented protocol -

→ How many bytes are sent in packet does not have a relation with the "send" calls.

→ Nothing gets lost.

- Let's say we send 100
- Max. receive = 1000
- Does not need to be that we get 100 in one go.

Web server → tcp concurrent server

→ 5 client then 6 threads

→ (5 new sd (s)
1 sd) one process (threads) for each client

#) TCP concurrent server — (there's TCP iterative server)

```
while (1) {

    newsd = accept ( ... );
    id = fork();

    if (id == 0) {
        close (sd);
        // Communicate with client
        close (newsd);
        exit(0);
    }
    else {
        close (newsd);
    }
}
```

→ use threads as processes are heavy

} no to hold on to unnecessary resources, prevents any unintended interference

} if not the resources will be consumed as it is copied for child.

- Servers should run non-stop.
- Make sure there's no memory leak.

read MAN pages.

TCP is a connection;

↳ receive but no data
↓
then block

Close one side then the other side would know.

#) send → blocking call → when send buffer is full.
↘ return when copied to buffer.

#) recv → blocking → As long as rec buffer is empty.

- In recv set MSG_WAITALL, will get entire chunk & then will go ahead. → • All specified in len are recieved.
- Check for <<EOF>> character. • connection close • error/signal
- MSG_DONTWAIT → recv is non-blocking
  - ↳ read data from buff
  - if no data then return −1.
    - / EAGAIN
    - \ EWOULD BLOCK

fcntl (sockfd, F_SETFL, O_NON BLOCK);

- ↳ non-blocking socket
  - ⇓ (accept, recv, recvfrom)
  - non-blocking
- ↳ ideal flag setup
  - ↓ int flag = fcntl (sockfd, F_GETFL, 0);
  - fcntl (sockfd, F_SETFL, O_NON BLOCK | flag);
    - ↓ doing or

Use RFC → for the assgn

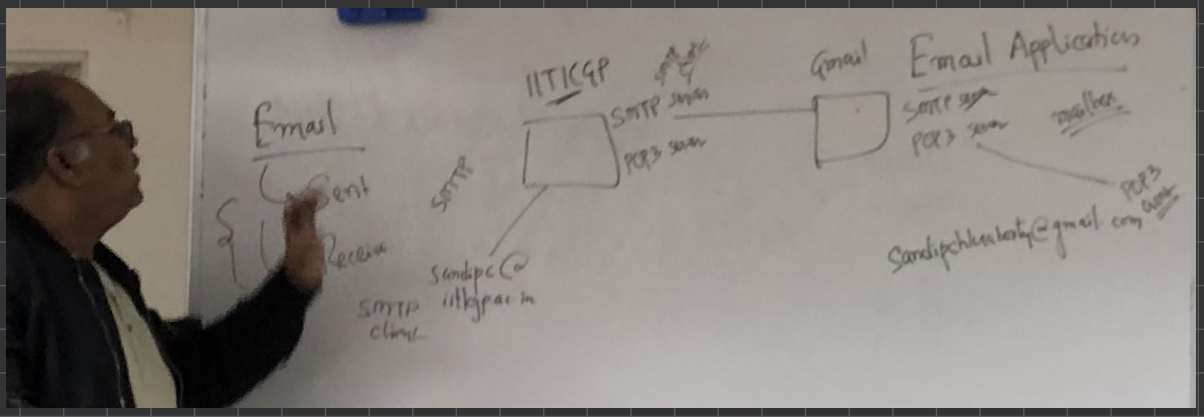
↓ RFC no.

MSG_PEEK → see msg w/o removing from queue.

#) Email –

↳ • Composing → client side

- Send email → Push type protocol
- Recieve email → Pull – do a login & get the mails
  - ↙ SMTP
  - → IMAP ↙ POP3



- need to have concurrent servers.
- RFC 8

Inter Sub → 2nd feb

text based email client

Q) How to wait for more than one thing??

select

```
fd_set fds;
while (1) {
    FD_ZERO (& fds);
```

↳ Set all bits to zero.

```
    FD_SET (sockfd1, & fds); // sets sockfd1 bit to 1
    FD_SET (sockfd2, & fds);
    nfds = max (sockfd1, sockfd2) + 1;
    select (nfds, & fds, 0, 0, 0);
```

1024 bits long

→ block until there is smth to read in sockfd1 or sockfd2.

Struct

→ readfds, writefds, exceptfds

```
if (FD_ISSET(sockfd1, &fds))
    {
        can now recieve without blocking.
    }
if (...        sockfd2    ...)
    {
    ...
    }
```

Send as a blocking call.

} not like this as then the next time it will be 12.

↳ wait for n+1 things
- n new sockfds
- OG sockfd

★ →
```
struct timeval t;
t.sec = 0
t.usec = 5000
```
Shows remaining time so will have to change later.

Poll - similar to select.

poll(nfds, struct pollfds * fds, timeout)

- nfds: no such limit as 1024
- pollfds: fd, events POLLIN, rev. (what did you get.)

Assgn 5

MTP → msg oriented, reliable, in order, flow controlled

TCP UDP
↳ use UDP to make MTP

OS

for every msg
↳ remember time of sending & ACK gotten or not
→ dont swarm the reciever

my_socket
my_bind
my_sendto
my_recvfrom
my_close

my_sendto, my_recvfrom → non-blocking

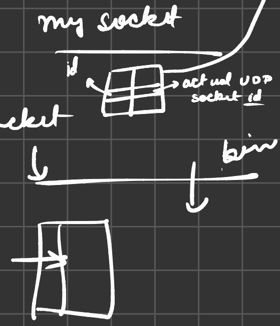
SM - shared memory

Two threads R & S

SM_ shared memory
↳ table is maintained by us → of sockfds
id

my-bind(id, _, _)

general sendto
↓
written to buff not that it is sent from the m/c



b/w multiple users, S & R thread so specific mutual exclusion.

if true that port & IP then taken

This goes here anyways

one send buffer, one recv buffer → per socket

put in this if it is from that Port & IP.

multiple sockets → use select on all fds

periodically check if there is a new thread or if there is a closed thread.

↳ timeout in select

sendto

go back -N

recvfrom

Selective repeat

↓

send ACK for longest in-order seq.

ACK → new window size

R & S → threads

static lib with implementation of the diff. calls.

thread → garbage collector

↳ PID of who made sockets then clean everything

setsockopt } → socket options

getsockopt

↳ snd buff size

rcv buff size

2 Sockets

↓

Reliable

↓

In order