24/8/24

CH.3 TRANSPORT LAYER

TRANSPORT LAYER SERVICES AND PROTOCOLS

→ Whatever is being sent by sender is only to be interpreted by the reciever and not by the intermediate devices (no interception)
↳ & transport medium

→ i.e Communication is end to end b/w sender's transport layer to reciever's transport layer.

→ provides logical comm. between app processes running on diff hosts.

→ logical end to end transport means even though there are ~~end~~ intermediate devices it is like the ends are directly connected.
Also, at each layer some new data could be added and that data can only be understood by that layer of the reciever machine.

→ Intermediate devices do not have a transport layer hence, transport protocols run only in end systems

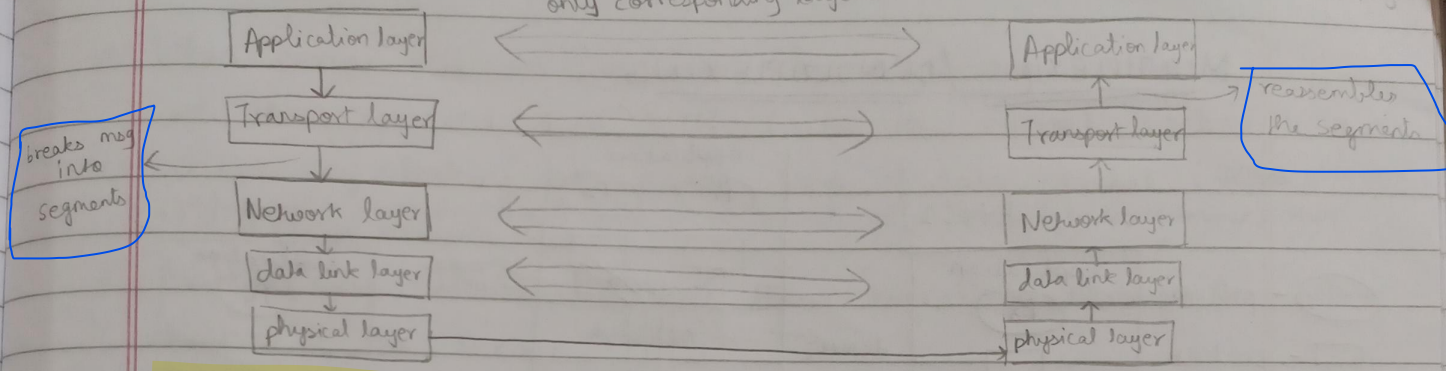
- send side: breaks app messages into segments, passes to network layer (of the same machine) (finally transfer is done b/w physical layers of both machines)

- reciever side: ~~xxxx~~ transport layer reassembles segments into messages and passes it to application layer of its machine.

SENDER

RECIEVER

only corresponding layers can understand data added at that layer



TRANSPORT VS NETWORK LAYER

→ Network layer : logical comm b/w hosts

→ transport layer : logical comm b/w processes.

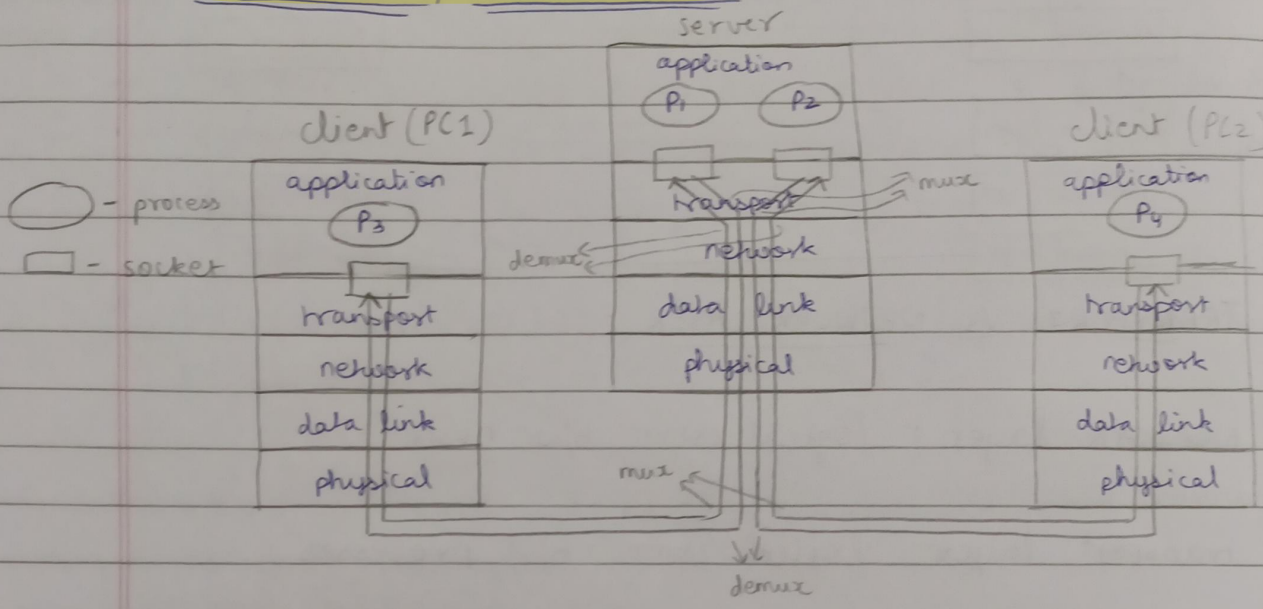
INTERNET TRANSPORT LAYER PROTOCOLS

→ reliable, in order delivery (TCP)
- congestion control
- flow control
- connection setup (3 way handshaking)
  ↳ SYN, SYN + ACK, ACK

→ unreliable, unordered (no packet numbering) delivery (UDP)
- no-frills extension of "best-effort" IP (hope for the best and see what happens)
- DNS request & reply happens with UDP
- video streaming uses UDP as less overhead (TCP has overhead)

→ services not available for both TCP & UDP:
- delay guarantees (cannot guarantee delivery time)
- bandwidth guarantees (allotted bandwidth uniformly shared, no priority)

MULTIPLEXING / DEMULTIPLEXING



→ P1 talking to P3

→ P2 talking to P4

→ Sending data → using Write API. Socket's file handler is used by write API

→ recieving data → using read API "

→ connection always initiated by client ~~send~~ using write API

→ All of them have a single Network card.
here in server same network card is being used for both processes.

→ When server acts as sender multiplexing is done.
data handled from multiple sockets, add transport header (later used for demultiplexing i.e to decide which client is the reciever of this data they use IP address to demultiplex in the physical layer)

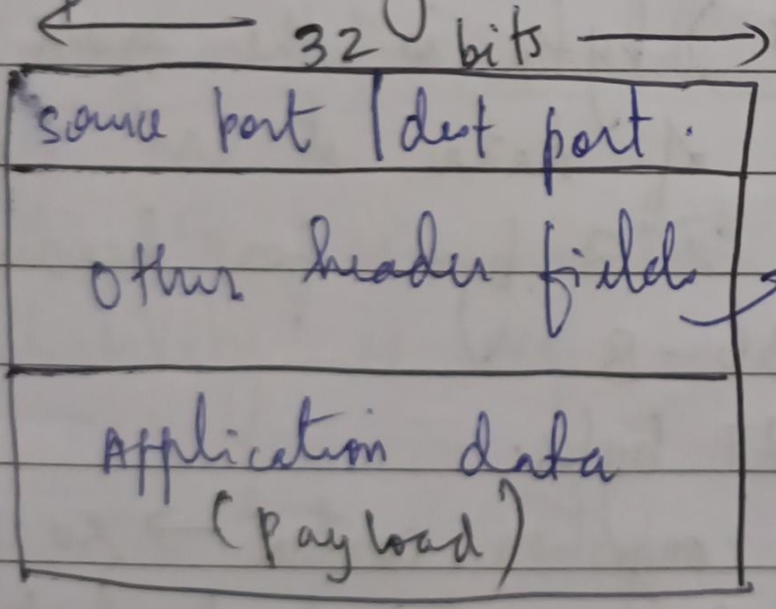
→ When server acts as reciever demultiplexing is done.
i.e the recieved data is sent to the correct socket in the server by looking at header info of data packet. (when data recieved from multiple clients first it is multiplexed at physical layer (combined) then demuxed at transport layer)

→ Multiplexing allows data to flow parallely through the other layers of the server.
(in this eg: mux & demux happening at server but it could happen at client also if multiple processes)

demultiplexing at receiver

either port No or IP address

uses header information to deliver the received info to the correct socket in application layer



each port No is 16 bit

TCP/UDP segment format

UDP Protocol

→ 'no frills', 'bare bones' internet transport protocol (i.e very simple)

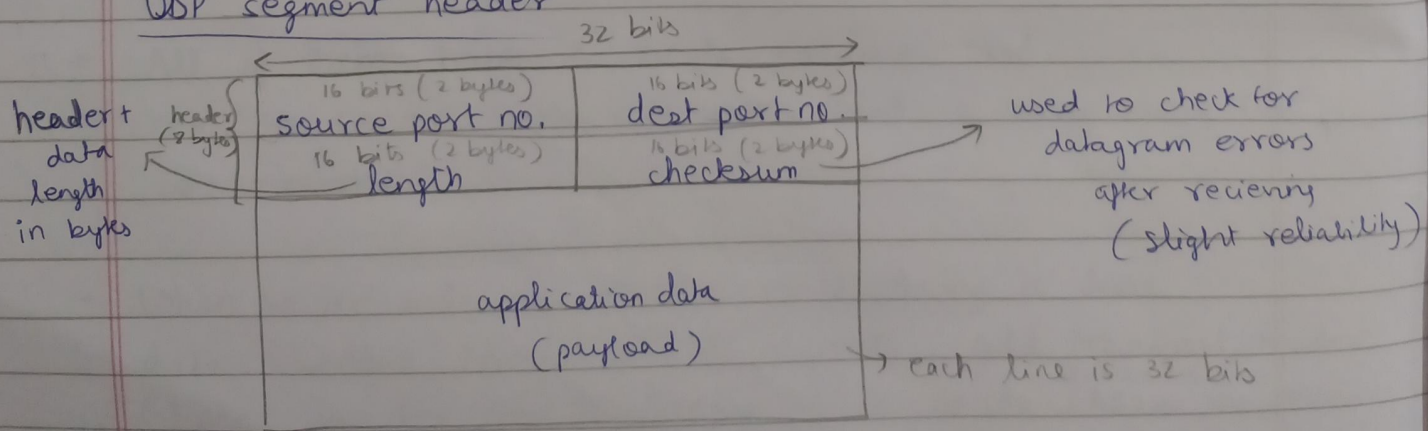
→ "best effort" service, UDP segments may be:
- lost
- delivered out-of order

→ Connectionless:
- no handshaking b/w UDP sender, reciever
- each UDP segment handled independantly of others (i.e all may take diff path unlike TCP)

→ UDP use:
- streaming multimedia
- DNS
- SNMP

→ reliable transfer over UDP can happen if:
- reliability added at application layer (manually add it to your program)
- application-specific error recovery.

UDP segment header



→ max amount of data bytes?

Max size of UDP data packet (max length) = 65535

↳ length field can have 16 bits so max size = $2^{16} - 1$
= 65535

data bytes = total length − header
= 65535 − 8
= 65527

→ UDP has no congestion control: it can blast away as fast as desired.

→ its simple (no connection maintained)

→ Check Sum

→ entire header & data is divided into 16 bit segments. (initially all 16 bits of checksum are 0)

→ then add all the groups. if there is a carry add it to the LSB again (wrap around)

→ finally 1's complement the sum, that is checksum.

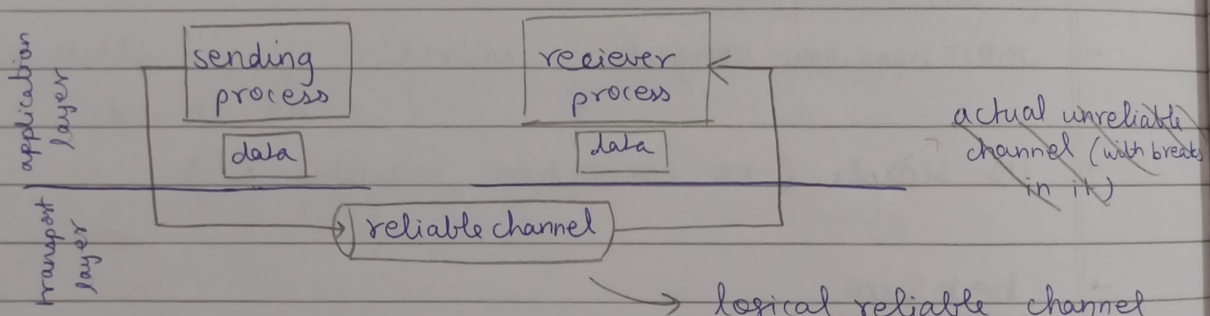
source port no.
+ dest port no.
+ length
+ checksum (initially 0)
+ data group 1 (16 bits)
+ data group 2 (16 bits)
⋮
sum

this sum is 1's complemented to get checksum.

→ At the reciever end again checksum is calculated. if this matches the sent checksum field then data delivered correctly.

Principles of Reliable Data Transfer

→ UDP is unreliable
→ communication channels are also unreliable.
→ Basically how to develop a reliable protocol on an unreliable network? using TCP.

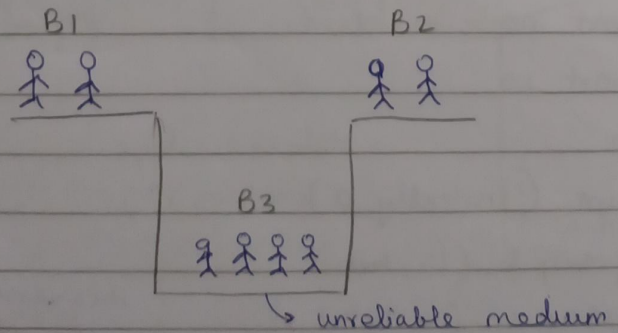


(a) provided service

2 army problem

→ two batallions of our army B1 & B2 &
1 batallion of enemy B3 (unreliable medium)

→ B1 & B2 have no communication, how to to attack?



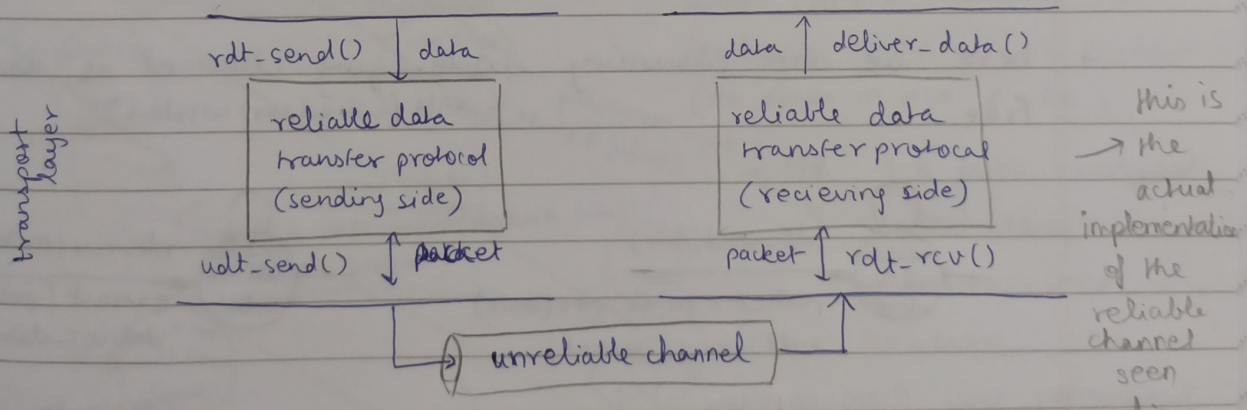
→ a messenger from B1 sent to B2 with time of attack
→ B1 doesnt know if messenger reached
→ B2 sends an acknowledgement messenger.
→ B2 doesnt know if he reached
→ B1 send acknowledgement of the acknowledgement.

: goes on

→ Theoretically if you want 100% efficiency this continues infinitely.

→ TCP will not do this, it will ~~make a limited no of~~ never send an acknowledgement for an acknowledgement.

→ ~~if no IP else OR~~ TCP will send 5 data packets (ping) to reach an IP address, if that IP is not reachable and all 5 ~~data packets are~~ attempts fail then it says destination unreachable (not 100% reliable but efficient)



(b) implementation of service

→ unreliable channel is converted to logical reliable using sockets.

→ packet is prepared using the data by the transfer

protocol (has extra header info like source, dest. etc.)

→ This data is extracted from the packet at reciever end.

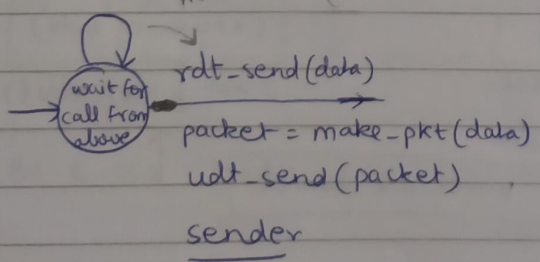
→ reliable channel means no bit errors, no loss of packets.
↳ these errors might occur in unreliable channel.
(these errors are masked by the module i.e reliable channel that we create) ↓ protocol technique.

→ rdt - reliable data transfer
→ udt - unreliable data transfer

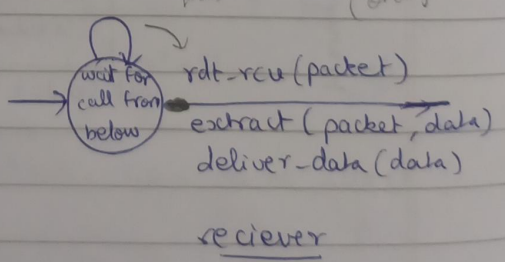
Reliable data transfer (ideal) (rdt 1.0)

→ ~~incrementally develop~~ separate finite state machines (FSMs) for sender, reciever:
- sender sends data into underlying channel.
- reciever reads data from underlying channel.

→ here we are assuming underlying channel is 100% error free ↳ (wired (or) wireless)



sender



reciever

→ Practically this is not possible as underlying channel will not be 100% error free.

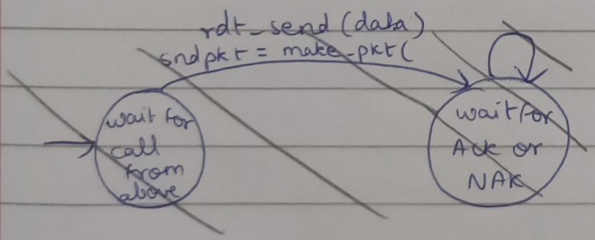
Rdt 2.0 (practical)

→ underlying channel may flip bits in packet.
- checksum to detect bit errors.

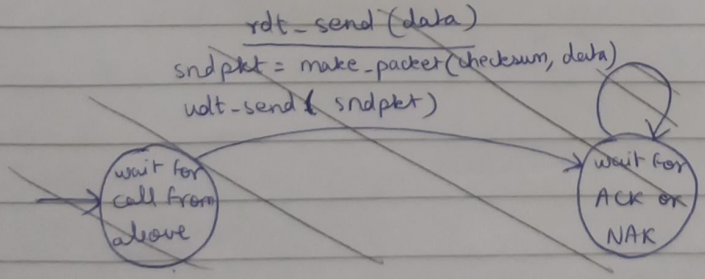
→ how to recover from errors?
- acknowledgements (ACKs) : reciever explicitly tells sender that packet recieved OK.
- negative acknowledgements (NAKs) : reciever explicitly tells sender that packet had errors. (i.e checksum did not match)
- Sender retransmits packet on receipts of NAKs.

→ New mechanisms in rdt 2.0 (beyond rdt 1.0)
- error detection
- feedback : control msgs (ACK, NAK) from reciever to sender

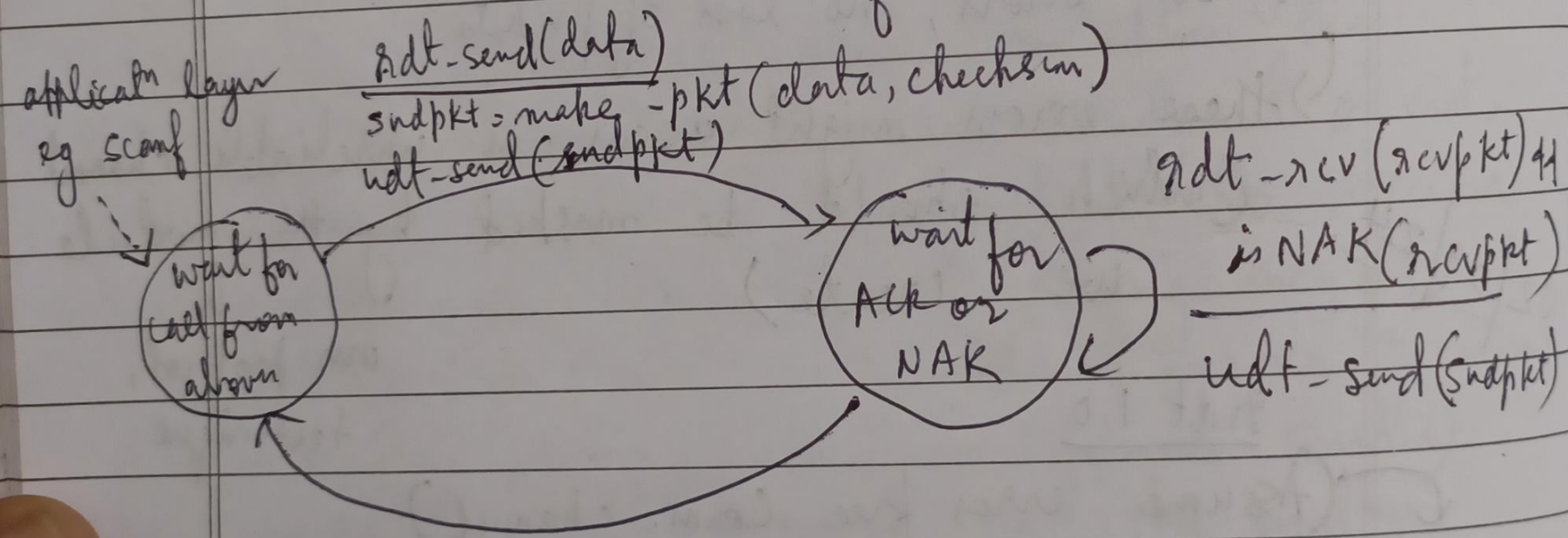
→ does not deal with packet loss. (i.e what if pkt never reached)



→ also rdt 2.0 assumes that feedback mechanism is perfect (no error in ACK & NAK)



→ if ACK/NAK corrupted
- sender doesn't know what happened at reciever
- can't just retransmit bcoz reciever could end up having duplicate packets.



application layer
eg scanf

rdt_rcv(rcvpkt) && is ACK(rcvpkt)

(If +ve Ack then continue with transmission of next data)

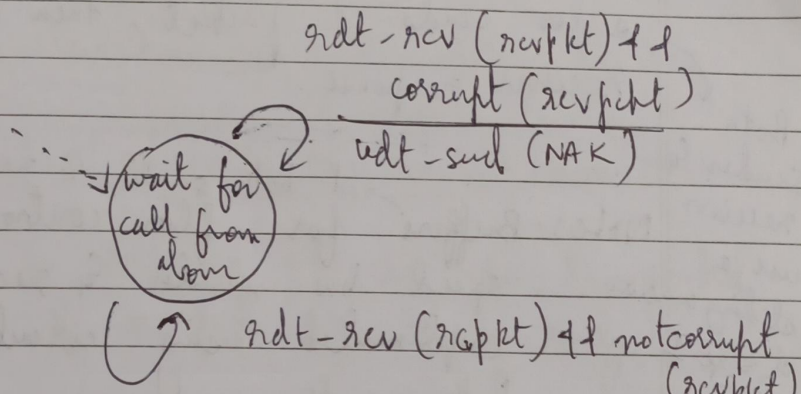
SENDER END

Here, we assume that ACK, NAK are error free i.e feedback mechanism is 100% efficient.

papergrid

Date: / /

RECEIVER END



(dealt in rdt 2.1)

→ If ACK/NAK are corrupted
- then sender doesn't know what happened at receiver
- can't just retransmit (could lead to duplicate)

→ solve duplication by giving each data packet a different number (or ID) even if data is same.
  - ↳ sender retransmits current pkt if ACK/NAK corrupted (not received)
  - ↳ sender adds sequence number to each pkt
  - ↳ receiver discards duplicate pkt. (If it had received pkt but ACK corrupt)

→ STOP and WAIT protocol

sender sends 1 packet, then waits for receiver response.

Both sender & receiver use a sliding window of size 1

Notes: • Buffers at both sender & receiver for flow control (matching the speed bw sender & receiver) is implemented using circular queue at both ends.

• This includes sliding window.

Sliding window in circular format

→ Sender & reciever both create a buffer for flow control. (i.e speed matching)

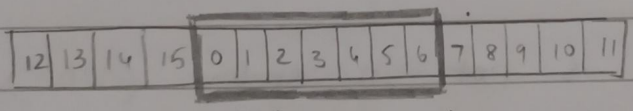
→ This is implemented using a circular queue (allows reusing of memory as cannot have ∞ memory) at both ends.

→ outstanding packet - packet has been sent but no response yet.

→ Sliding window in linear format is just flattened out circular queue.

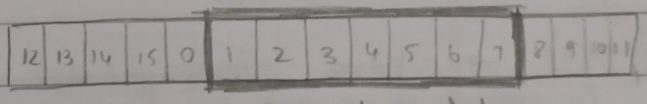
→ window is max no. of outstanding packets you're okay with having

→ once a packet is acknowledget the window slides by 1 position



↳ window of size 7

7 packets have been sent
window is full
waiting for ACK



↳ window slides

packet 0 has been acknowledged
window slides, packet 7 ready to send.

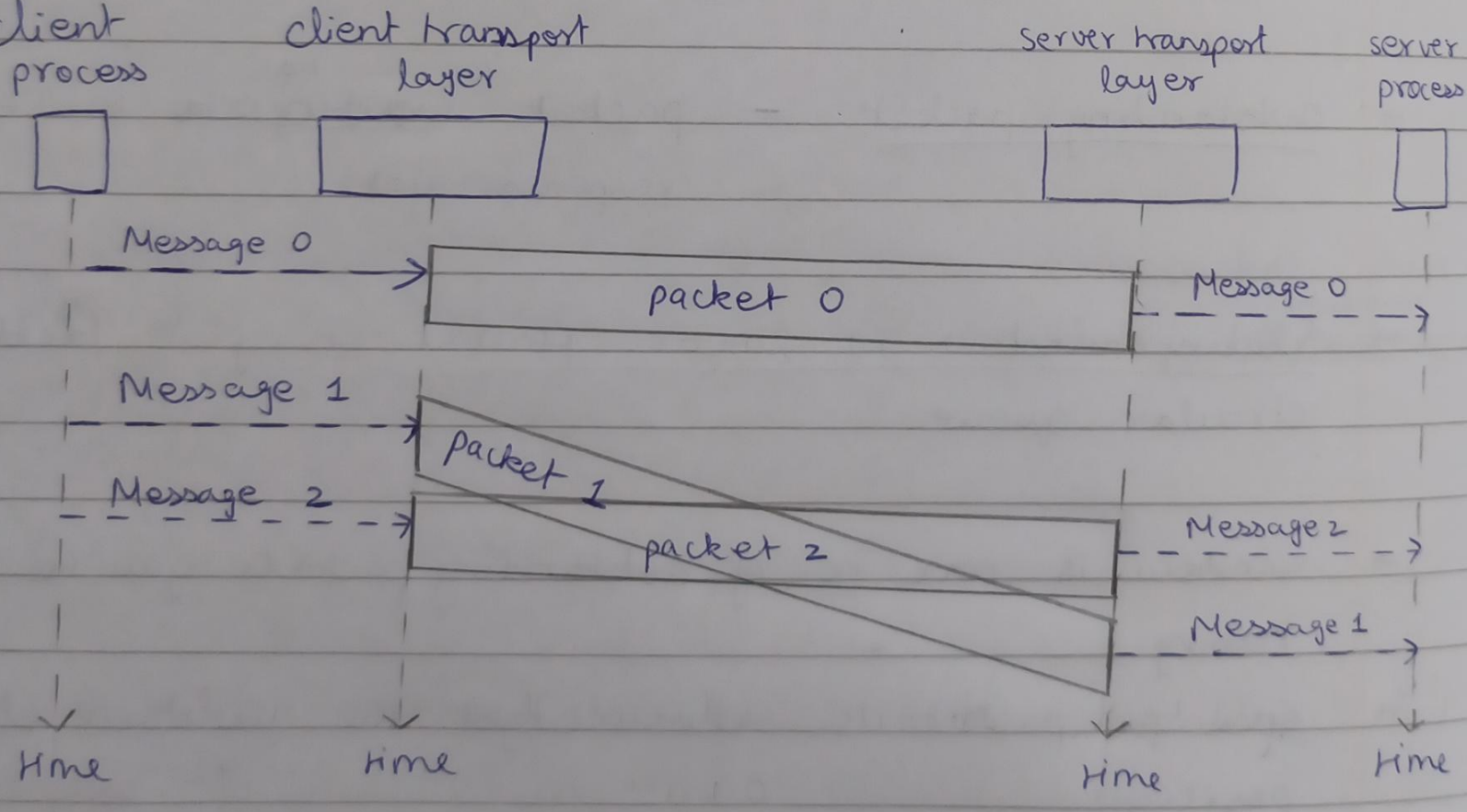
→ any packets outside the window are ready to be sent in application layer but not being considered to send at the moment.

→ here after packet 0 is acknowledged it is removed from buffer and new packet can be filled there.

→

Time sequence diagram

eg: connectionless service (i.e packets take diff path each time)



→ here message delivery is not in order. (UDP)
Since all packets may take diff path, some packets reach before others.

kari line time time

stay in transport layer.

UDP → Not in order due to dynamic routes

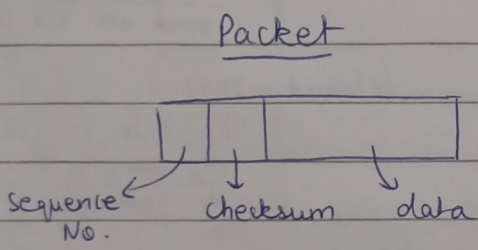
- In TCP even if Packet 2 reaches before 1, P2 wait for P1 to reach (∵ N/W is unreliable)
- There is connection - open request from client & then connection establishment, before packets are delivered.
- Then there connection close request from client

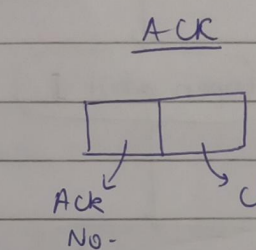
→ Simple protocol (rdt 1.0)

→ Stop & Wait Protocol

→ Sliding window of size 1
→ send 1 packet wait for response.
Until response recieved no other packets are sent.
→ Timer is also used for flow control & error control
(retransmit if no response recieved in the set time)

→ response is always next packet to send i.e reciever says send next packet.
→ if there was an error in packet, the reciever will give the same packet number for next packet.





→ because ACK is also a data packet which could get corrupted. if Ack No. has error and gets changed then duplicates could be sent (wrong packet)

→ Once a packet is recieved & response (Ack) is sent, that packet is removed from the buffer as it was safely delivered.
→ when packet is sent timer starts. It will stop when ACK

recieved.

→ But if timer runs out and no ACK recieved it is assumed (that something went wrong eg: packet was lost & never recieved by reciever). Then this packet is retransmitted.

→ even if packet recieved properly but ACK got lost then we have to retransmit the packet.
But at reciever end it was expecting next packet so it discards this duplicate packet and sends ACK asking for next packet. → (it already had the prev packet) identified by sequence no.

→ Every data packet has a TTL - time to live. Within this lifetime it should reach the destination else it will be deleted from the network.

→ since sliding window size = 1
seq no can be 0 & 1 only.
these can be repeated as after packet 0 recieved & acknowledged it is removed from buffer so a new packet can be named packet 0.

Q) Assume that, in a Stop & wait

GO-BACK-N Protocol (GBN)

→ To improve transmission efficiency (max utilisation of resources), multiple packets must be in transmission while the sender waits for acknowledgement (not letting it sit idle)

→

to support this we need:

→ seq no.

→ ack no.

→ send window - how many packets sender can push at once (send) without waiting for Ack

→ recieve window - how many packets reciever can hold

→ timers

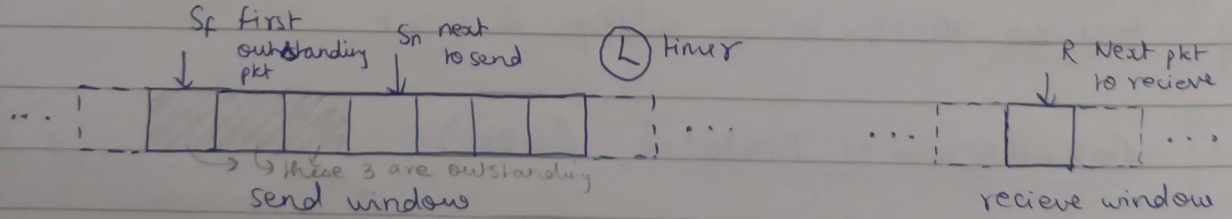
→ resending packets

- send window should be acc. to what recieve window can manage.

- Both these sizes are announced when communication established.

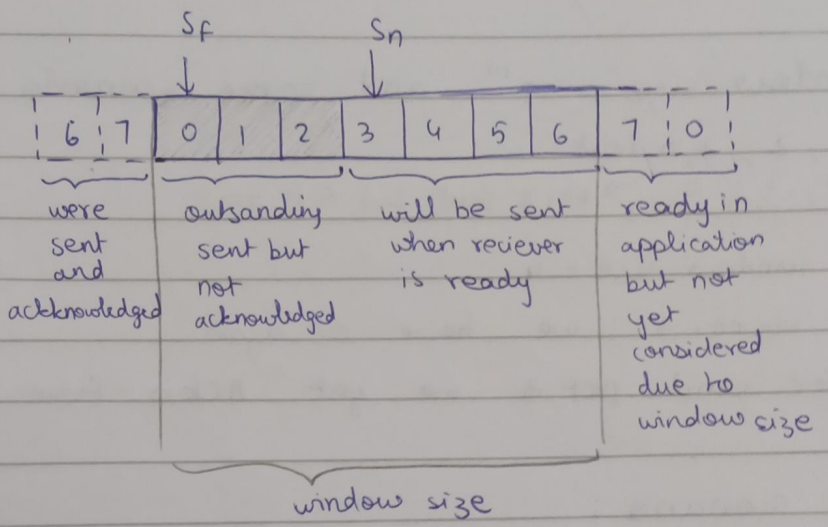
- Timer is for the send window i.e when first data packet sent timer starts and timer times out after all packets in window sent + some extra time to wait for Acks. Timer stopped in the middle if a packet is recieved and Ack arrives, then window slides by 1 position & timer starts again.

- if 1st packet does not get recieved (i.e lost), entire window retransmitted. (hence called go back n protocol i.e all n resent packets)

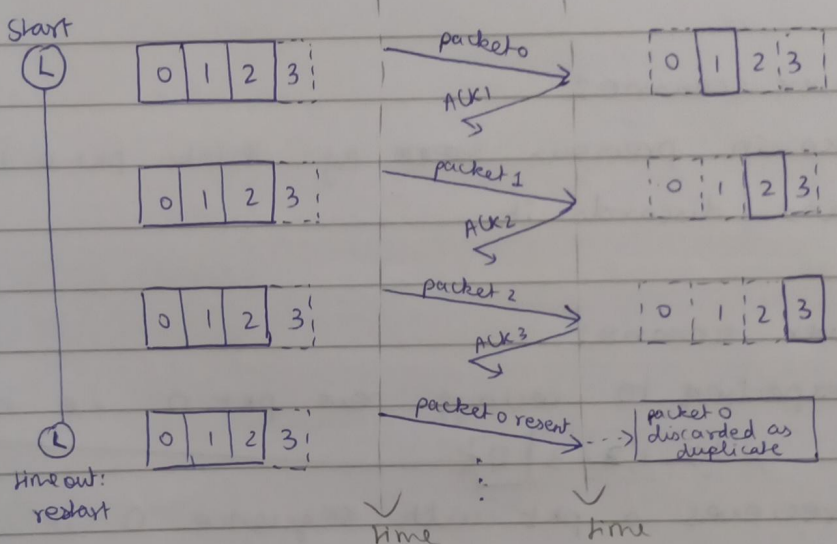


→ Sf packet will be kept until it is acknowledged. and then window slides by 1 position.

→



→ here sequence no. field is 3 bits. hence 0 to 7 (000 to 111) so after 7 you have 0 again. (recycled)



here, send window size $< 2^m$

where send window size = 3

and m = no. of bits in sequence no.

$2^m = 2^2 = 4$

Note: reciever window is 1 to ensure in order recieval of packets.

→ here we never recieved ACK 1 before timer timed out, so packet 0 resent, similarly all other packets are resent

→ if window size = 2^m and same scenario as earlier

ACK 1, 2, 3, 0 lost

pkt 0 pkt 1 pkt 2 → pkt 3 sent but ACK lost

since window size = 4

after seq no. 3 we have 0 again

so after send pkt 3 we get ACK 0 from reciever

→ Sender scenario:

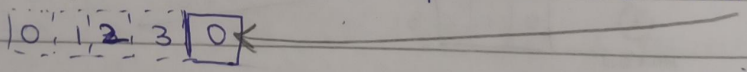
it has not recieved ACK for anything so it resends everything starting with pkt 0.

→ (ideal) reciever scenario:

reciever like in previous ~~scenario~~ eg: thinks pkt 0 is a duplicate and discards it.

→ (actual) reciever scenario:

reciever is expecting to recieve new pkt 0 i.e next seq no.



when it recieves a pkt with sequence 0 it thinks it is the new pkt 0 not the old one and thus does not realise its a duplicate and erroneously accepts it

---

→ if a packet is ~~received~~ lost (not recieved), the reciever will keep sending ACK for it eg: pkt 2 not recieved, reciever keeps sending ACK 2.

if sender sends subsequent packets 3, 4, 5, 6, reciever will ~~dis~~ recieve them but sliding window will be stuck at 2 as it was not recieved and it will keep sending ACK 2.

if pkt 2 is retransmitted before timer runs out reciever will directly jump to ACK 7 because it has already recieved pkt 3, 4, 5, 6.

by the time Ack 2 arrives

but if timer runs out, sender will think reciever has not recieved anything (it will also think pkt 1 was not recieved as Ack 2 is a confirmation that pkt 1 recieved now send pkt 2)
So everything from pkt 1 till the end of the window will be resent.

→ Say a pkt 3 was recieved but its Ack 4 was lost
reciever thinks Ack 4 was sent properly so when it recieves pkt 4 from sender it sends Ack 5
At sender side : even though it did not recieve Ack 4, when it recieves Ack 5 it understands that both pkt 3 and pkt 4 were recieved because if there was a problem then Ack 5 would not have come.
hence sliding window of sender moves 2 positions (removes pkt 3 & 4 from buffer) and starts timer again.

SELECTIVE REPEAT ALGORITHM

→ ~~inefficient of~~ prev protocol had a drawback of retransmitting even those packets which have already been recieved.

→ This protocol makes sure only the data packet which is lost is resent

→ eg: 0-2 sent
3 lost
4-7 sent
in go back n 3-7 will be resent
but here instead of sending ACK 3 reciever sends
SRJ 3 this indicates send only pkt 3
↳ selective reject