TRANSPORT LAYER

* Applications of Transport Layer:-

- Process to Process delivery: provide logical communication b/w app processes running on different hosts.
- Segmentation and Reassembly: TP on sender's side breaks down the messages into smaller units called segments. These segments are passed to the network layer. On the reciever's side, transport layer reassembles the segments back into original application message [Multiplexing and Demultiplexing].
- Flow and Error control (Checksum).

* Protocols used in Transport Layer:

TCP (Transmission Control Protocol)	UDP (User Datagram Protocol)
- connection-oriented (acknowledgement)	- connectionless (no acknowledgement)
- flow control	- no flow control
- congestion control	- no congestion control
- slower	- faster
- Guarantees order & data arrival	- no guarantee
- suitable for applications requiring accurate data transfer	- suitable for applications requiring speed of data transfer
- ex: sending emails, web browsing	- ex: video streaming, online gaming
- Robust error checking & correction mechanism	- basic error checking & correction mechanism

* Multiplexing and Demultiplexing:

- occurs at sender's side of communication.
- gathers data from multiple applications (sockets).
- TCP adds a transport header to ensure correct delivery.
- ensures data from multiple applications is prepared for transmission.

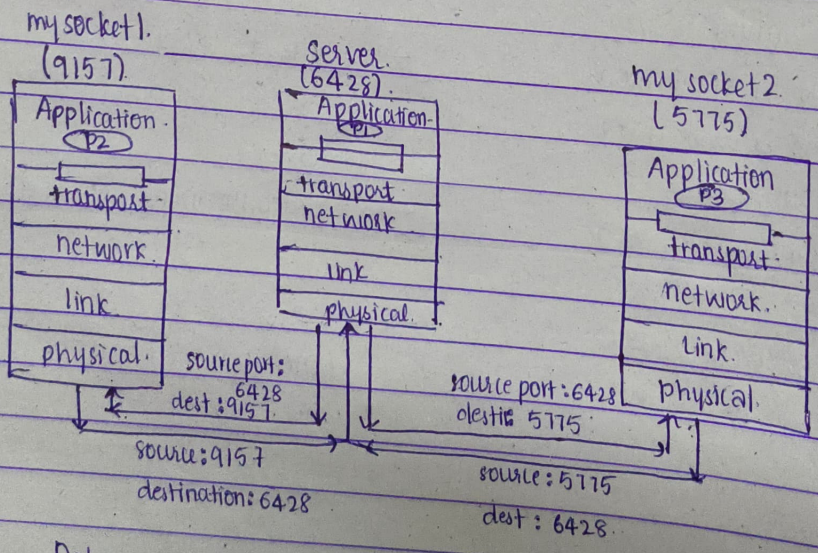
- occurs at recievers side of communication.
- transport layer examines header of each received packet (segment).
- ** uses source IP, dest IP. IP Address source port number, dest port number and port numbers to direct segment
- ensures data reaches the correct application at the reciever.

- Connectionless demultiplexing:

When host recieves UDP segment it enquires the port numbers (source and destination) then directs to the socket with the respective port number.

↳ datagrams with same destination port numbers will be directed to same sockets irrespective of their source port number / IP addresses being different.

Example:



Datagrams are transferred respective of their source and destination port numbers.

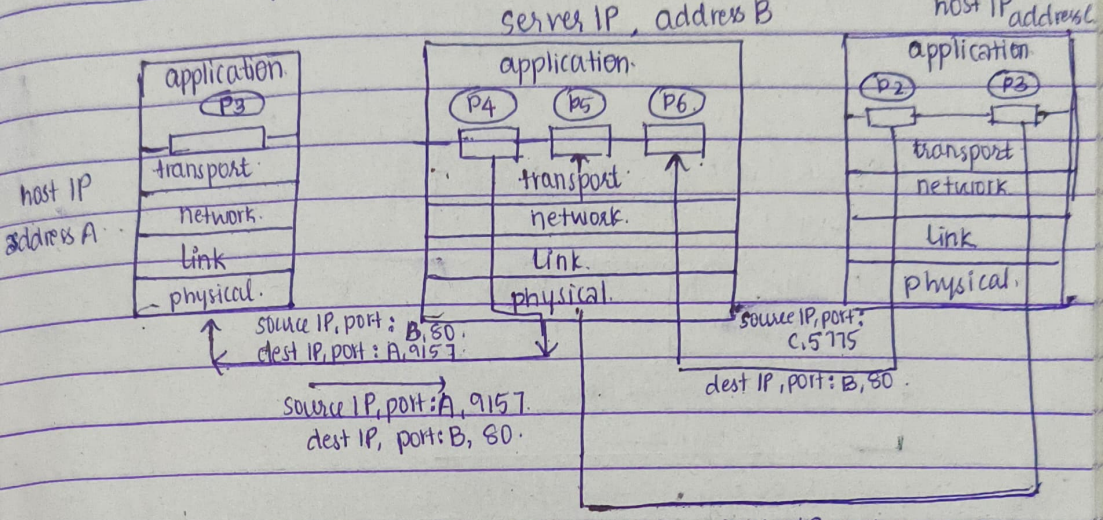
- Connection oriented demux :

TCP socket identified by 4 tuples:

i source IP address
ii destination IP address.
iii source port number.
iv destination port number.

receiver uses all 4 values to direct segment to appropriate socket.

- server host may support many simultaneous TCP sockets.



three segments, all destined to IP address: B, dest port: 80 are demultiplexed to different sockets.

User Datagram Protocol (UDP):

- UDP is a lightweight and simple transport layer protocol in the IP suite.
- It is connectionless (doesn't establish persistent connection b/w the sender and receiver before sending data).
- "no frills" or "bare bones": UDP doesn't provide many of the features that TCP does, such as error checking, retransmission and flow control.
- "best effort" means that the network makes an attempt to deliver packets from sender to receiver but doesn't guarantee delivery.
- It doesn't require a connection setup b/w the sender and receiver.
- Each UDP segment is handled independently: there's no sequence or order enforced.
- UDP applications: > streaming multimedia → video/audio streaming
  > DNS
  > SNMP → network monitoring & management.
- It is simple, which makes it faster but reliable.

* UDP: Segment header.

source port #	dest port #
length	checksum
application data (payload)	

32 bits

bytes of UDP segment including header.

In hexadecimal format:

0045 DF00 0058 FE20.

i) what is source port no? → 0045

ii) what is dest port no? → DF00

iii) total length of user datagram? → 88 bytes (0058 is converted to decimal)

iv) length of data? → 80 bytes.

source port # (0045)	desti port # (DF00)
length (0058)	checksum (FE20)
application data (payload)	

Decimal equivalent: 88 bytes → total length of datagram.

header size of UDP = 8 bytes (64 bits)

∴ Datalength = 88 − 8 = 80 bytes.

Checksum: used for error detection in transmitted segment.

ex- flipped bits, loss of packets

Sender's side	Receiver's side
– takes entire UDP segment (header + data) as a 16 bit integer	– computes checksum of received segment and compares with value stored in checksum field.
– checksum is calculated by adding up all these 16 bits using 1's complement arithmetic.	If same: no errors detected.
– Sender places calculated checksum into dedicated checksum field in UDP header.	If not same: Yes error detected.
	* not a guarantee ~~to~~ against all types of transmission errors.

example: add two 16 bit integers

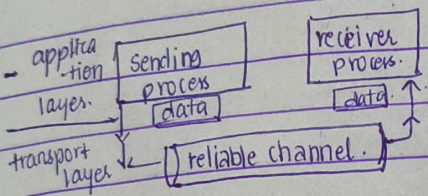
```
  1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
  1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
(1)1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
                                + 1
  1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
```

add to result

1's comp → 0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1.
(checksum).

Principles of Reliable Data Transfer:

- Reliable Data Transfer is crucial across multiple layers of the network stack (application, transport & datalink layers).

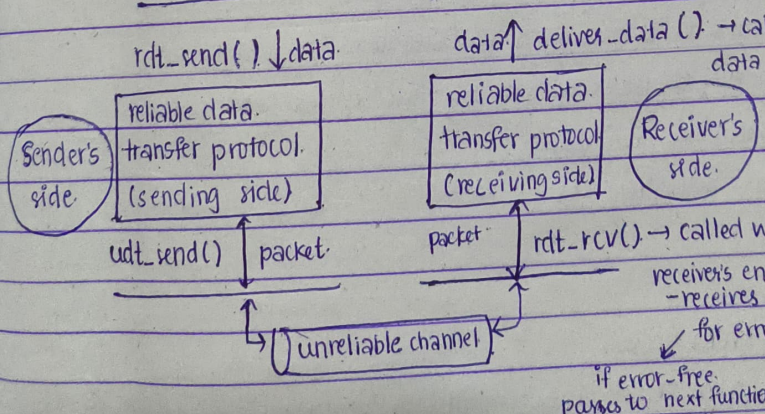


→ ~~transmitt~~ data is transferred over a reliable channel, which is assumed to be error-free and guarantees delivery.

- unreliable channel can experience packet loss, corruption (or) reordering

↓

here RDT protocol, helps in implementing data transfer over this unreliable channel.. → employs various techniques to ensure data integrity and delivery.

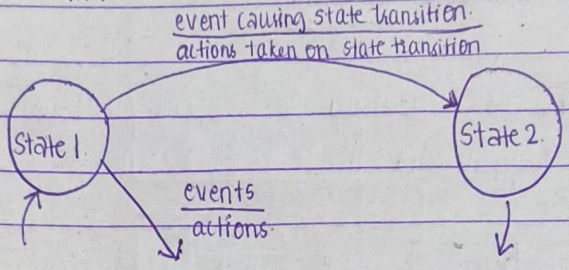


rdt_send() → function is called by upper layers (application) to initiate transmission
- encapsulates the data to be sent into a packet by adding headers.

udt_send() → called by RDT layer to transmit packet over unreliable channel.
- takes packet as input and transmits to receiver.

Reliable Data Transfer (RDT):

- incrementally develop sender, receiver sides of rdt.
  ↓
  protocol will be built step by step.
- Unidirectional Data Transfer: data will flow only in one direction i.e from sender to receiver or vice versa.
  but acknowledgements or control information will flow in both directions.
- FSM (Finite State Machines): used to model the behaviour of the sender and receiver.



States: represent different conditions / phases the system can be in.
(e.g: idle, data ready, waiting for acknowledgement)

Transitions: show how system moves from one state to another.

Events: Triggers that cause a state transition.

Actions: operations performed when a state transition occurs.

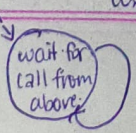
* **Rdt 1.0 [Reliable Transfer over a Reliable Channel]**

→ underlying channel is assumed to have:
- > no bit errors - data is transmitted without any corruption.
- > no loss of packets - all packets sent by the sender are guaranteed to reach the receiver.

- separate FSMs fo sender, receiver:

Sender

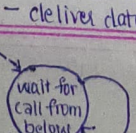
State: wait for call from above.
Event: data arrives from appln layer.
Action: - data encapsulated to packet using ~~rdt~~ make_pkt(data)
- send data to receiver using udt_send(packet).



rdt_send(data)
packet = make_pkt(data)
udt_send(packet)

Receiver.

State: wait for call from below.
Event: packet is received from underlying channel
Action: - extract data from packet using extract(packet, data)
- deliver data to Appln layer using deliver_data(data).



rdt_rcv(packet)
extract(packet, data)
deliver_data(data).

* RDT 2.0 Channel with Bit Errors (no Loss).

- underlying channel may flip bits in packet. (bits in packet can be flipped during transmission).
- to detect the errors, it introduces checksum mechanism.
- It employs two mechanisms for error recovery:
  - > Acknowledgements (ACKs) - When receiver successfully receives and processes a packet, it sends an ACK message back to sender. this ACK message serves as confirmation that packet is received correctly.
  - > Negative Acks (NAKs) - if error detected, receiver sends NAK message to sender, which means packet was corrupted and needs to be retransmitted.
- What is different from RDT 1.0? (Add-ons)
  - > error detection) - use of checksums
  - > Receiver feedback - introduction of ACKs and NAKs.

retransmission: is not recieved properly, then send it again

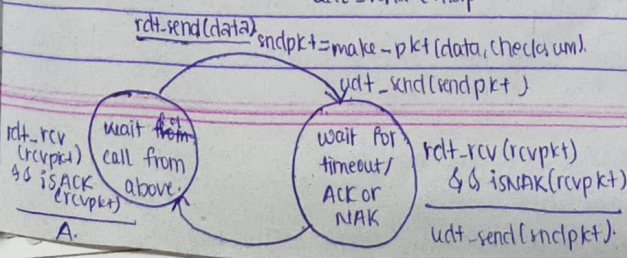
FSM specification.

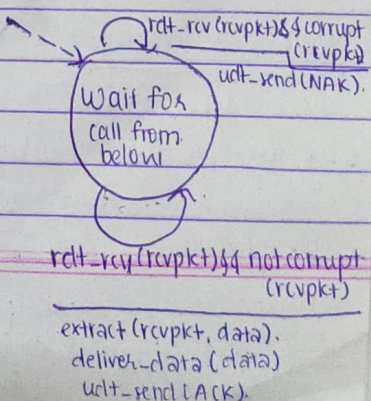
Sender

1. State = "Wait for call from above".
   Event - data arrives from Appl^n layer.
   Action - encapsulate data into packet with checksum using make_pkt(data, checksum).
   - send to receiver [udt_send(sndpkt)]
   - set a timer.
2. State = "Wait for ACK or NAK / timeout".
   Event - Timeout / ACK or NAK received.
   Action:
   > if timeout : retransmit using udt_send(sndpkt).
   > if ACK : go back to "Wait for call from above" state.
   > if NAK: retransmit packet using udt_send(sndpkt).

Receiver.

State = "wait for call from below".
Event - packet received.
Action - check checksum using corrupt(rcvpkt)
- if corrupted: send NAK udt_send(NAK)
- if not corrupted:
  - extract data using extract(rcvpkt, data)
  - deliver data to appl^n layer. deliver_data(data).
  - send an ACK to sender udt_send(ACK)





RDT 2.0 has a fatal flaw!

What happens when ACK or NAK message gets corrupted?

- this may lead to incorrect actions.
- unnecessary retransmission might happen (due to corrupted NAK).
- missing acknowledgements, sender might not receive confirmation for a successfully delivered packet.

To address this issue, RDT 2.0 introduces the concept of "sequence Numbers". Each packet is assigned a unique sequence number. The receiver can discard if already received (duplicate packets).

Stop-and-Wait Protocol: Where sender ~~receives~~ sends one packet and waits for acknowledgement before sending next.

[Go through slide 37 & 38 for diagram].

RDT 2.1 Protocol:

Sender	Receiver
- Seq # added to pkt. packet is assigned with a seq. number to distinguish b/w packets and handle duplicates.	- must check if received packet is duplicate. because sender might retransmit due to timeouts or corrupted acknowledgements.
- two seq #'s are sufficient as they need only to keep track of current sequence number and expected acknowledgments.	- receiver's state keeps track of sequence numbers.
	* receiver cannot ~~know~~ directly confirm if sender has received ACK/NAK messages, which can lead to errors

RDT 2.2 Protocol:

uses only ACK messages.

If receiver gets a packet correctly, sends an ACK with same sequence number. If sender doesn't receive an ACK within a certain time, it retransmits the packet.

- receiver must explicitly include sequence number of packet being acknowledged~~ment~~ in the ACK message.

[Go through diagram from slide 41]

RDT 3.0

- Can loose or corrupt packets.
- Checksums, seq #, ACKs, retransmissions will be of help, but not enough.
- mechanisms:
  - > sender waits for a "reasonable" amount of time for ACK from receiver.
  - > If no ACK is received within the timeout period, sender ✱ retransmits the packet.
  - > If packet or ACK is just delayed (not lost), retransmission will be a duplicate. But sequence numbers handle it.
  - > receiver must specify sequence number of packet being acknowledged.

Sender: (refer diagram in slide 43)

RDT 3.0 uses alternating sequence numbers (0 and 1) to handle retransmissions and avoid confusions with duplicate ~~items~~ packets.

States: i Wait for call from above
ii wait for ACK0 - sender has a sent a packet with seq num 0 and is waiting for ack with seq num 1.
iii wait for ACK1 - same as above but with seq num 1.

Transitions: i. Receive data from above - creates packet with the data, calculates checksum sends the packet.
ii. Receive a corrupt packet / ACK with wrong sq number - if sender receives ACK with wrong sequence number, ignores and remain in current state.
iii Receive a timeout - if timer expires before ACK is received, sender retransmits the packet and restarts timer.
iv Receive a correct ACK - if sender receives a correct ACK, it stops the timer and transitions to other wait state.

To check how RDT 3.0 works, one go through slides 44 and 45.

Calculations:

i) Transmission time: transmission delay for sending a packet.

$$D_{trans} = \frac{L \rightarrow \text{packet size (bits)}}{R \rightarrow \text{link speed (bits per sec)}}$$

ii) Sender Utilisation (U) - fraction of time sender is actively sending data.

$$U_{sender} = \frac{L/R}{RTT + L/R}$$

iii) Throughput - actual data transfer rate.

$$\text{Throughput} = U_{sender} \times \frac{\text{Packet size}}{\text{packet sending interval}}$$

(Once go through calculations in slide 46 and 47).

* Pipelined Protocols:

- Allow multiple packets to be in transit simultaneously. These packets are referred as "in-flight" packets.
- to handle multiple in-flight packets, range of sequence numbers must be increased. to ensure order.
- Both sender and receiver need to have buffers to store multiple packets. Sender's buffer holds packet waiting to be sent. while receiver's " " " that have to be processed yet.

This helps in improving throughput and reduces latency.

$$\text{~~Throughput~~ } U_{sender} = \frac{nL/R}{RTT + L/R}$$

Pipelined Protocols

GBN - back - N

Sender can have N unacknowledged packets in pipeline. can send N packets without waiting for ACKs.

~~if first apa~~

Selective Repeat

same.

- sends individual ACKs for each packet received correctly. So sender identifies specific packets which are received and which need to be retransmitted.

GBN: if sender doesn't receive an acknowledgement for a packet within a certain time period, it assumes that packet with subsequent packets has been lost or corrupted. so sender retransmits all unacknowledged packets starting from one that was lost.

→ GO-Back-N: sender ~

- Sender maintains a window of upto N unacknowledged packets. It can send N packets without waiting for acknowledgements.
- Each packet is assigned a unique sequence number.
- Receiver sends cumulative acknowledgements
  ↓ acknowledges the highest numbered packet received, implicitly acknowledging all previously received packets.
- Sender sets a timer for older unacknowledged packet. If timer expires before an acknowledgement then it retransmits all unacknowledged packets ~~start~~ from the oldest one.

FSM:

Sender	Receiver.
State: Wait.	State: Wait for a packet.
Transitions:	
i. Receive data - checks ~~for~~ if there's room in the window, if it is, there it creates a packet and sends the packet then starts a timer.	Transitions: i Receive a packet - - If received packet is not corrupt. and has expected sequence number. receiver extracts data and send ack message.
ii. Timeout - if timer expires, sender retransmits all unacknowledged packets and restarts timer.	ii - If received packet is corrupt, it is discarded, receiver re-sends ack for highest-numbered packet received correctly.
iii Receive a corrupt packet. - If sender receives a corrupt packet, it simply discards it.	
iv. Received a correct ack - updates to next seq number. timer is stopped or restarted. after checking base == next seq num.	

[Example in ppt].

→ Selective Repeat n.

- Transmits only lost or corrupted packets, rather than all unacknowledged packets.

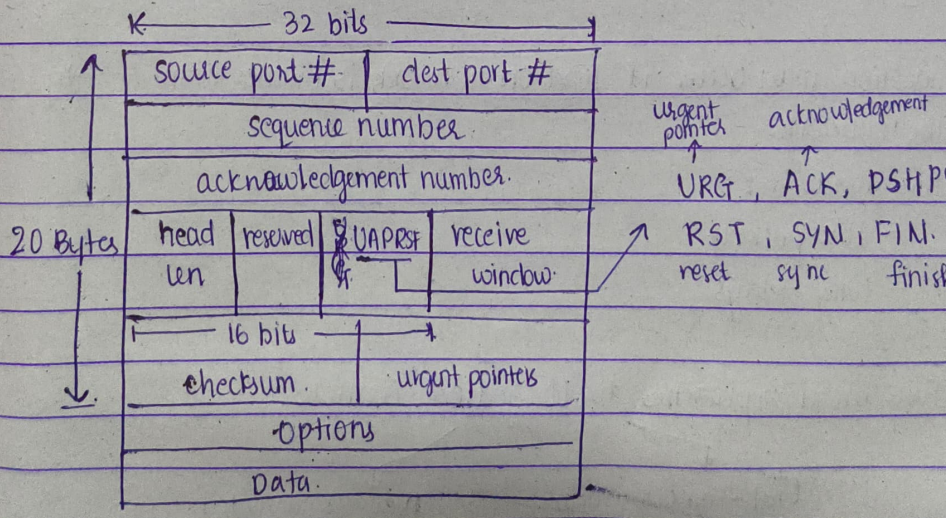
- Receiver sends individual acknowledgements for each correctly received packet. Allows sender to identify which packets have been received and which are to be retransmitted.
- Receiver buffers out-of order until they can be delivered in the correct sequence.
- Sender maintains a window of unacknowledged packets.
- FSM:

Sender	Receiver.
State: wait for data from application layer.	State: wait for packets to be received.
transition:	transition:
i. Receive data - If has space in window (data → packet). assigns a seq num and sends.	i. If received packet correctly, delivers to appl^n layer and sends an acknowledgement.
ii ~~wait~~ timeout - if time for packet expires, retransmits only that packet.	ii If out of order, buffered until missing packet arrives.
iii receive ack - Sender receives an acknowledgement, marks ~~the~~ packet as ack~~nowledged~~owledged and updates window base if necessary.	iii If duplicate packet is received, it simply discards it.

Transmission Control Protocol (TCP)

- Reliable, connection-oriented protocol.
- Ensures reliable, in-order delivery of data b/w two applications.

- TCP establishes a direct connection b/w two end points, a sender a receiver.
- Data can flow in both directions simultaneously (through pipelines).
- Ensure data is delivered in same order it was sent.
- ensures no data is lost or corrupted.
- Treats data as a continuous stream of bytes.
- Establishes a connection before data transmission.
  (Three way Handshake). both exchange control messages.
- # Sender maintains a window size to control the number of segments it can send.
- Flow control:
  > sender does not overwhelms the receiver.
  > receiver sends window size updates to the sender, indicating how much data it can accept.
  > prevents congestion.



- Each byte in a TCP is assigned a unique sequence number.
- it will be used to track the order of bytes.
- when a receiver receives a segment, it sends ACK to the sender.
- ACK contains a sequence num of next segment receiver expects to receive.

- TCP doesn't specify how a receiver should handle out-of-order segments. that is left to the implementer.

EstimatedRTT = $(1-\alpha)$ * EstimatedRTT + α * SampleRTT

current estimate. smoothing factor.

typical value of $\alpha = 0.125$

DevRTT = $(1-\beta)$ * DevRTT + β * |SampleRTT - EstimatedRTT|

typical $\beta = 0.25$.

Timeout interval = EstimatedRTT + 4 * DevRTT

~~Refer~~

Congestion control:

→ Congestion may occur if load on network is greater than capacity of the network.

→ This leads to:
- lost packets.
- long delays.

→ Two broad approaches to Congestion control:

End-end congestion control.	Network assisted congestion control.
- no feedback given ~~by~~ to sender.	- routers provide feedback.
- finds out watching signs of:	
> packet loss.	
> delays.	

Open Loop Congestion Control.	Closed Loop Congestion Control.
- policies are applied to prevent congestion before it happens.	- try to alleviate congestion after it happens.
- congestion control is handled by either source or destination.	-
- handled by retransmission, window size managing, or acknowledging.	- reduces size of transmission window and retransmitting.

* Congestion control in TCP is based on both open loop and closed-loop mechanisms.

AIMD: Additive Increase Multiplicative Decrease.

TCP increases congestion window size slowly ~~to probe~~ & until loss detected.

↑s. cwnd by 1 MSS every RTT

maximum segment size

If packet loss occurs, TCP reduces the window size drastically.

Process involves:

i. Slow start - Starts with a slow rate of transmission, ↑s rate exponentially to reach a threshold.

ii Congestion avoidance - Linear growth until congestion is detected.

iii Multiplicative decrease - reduces window size and threshold when packet loss occurs