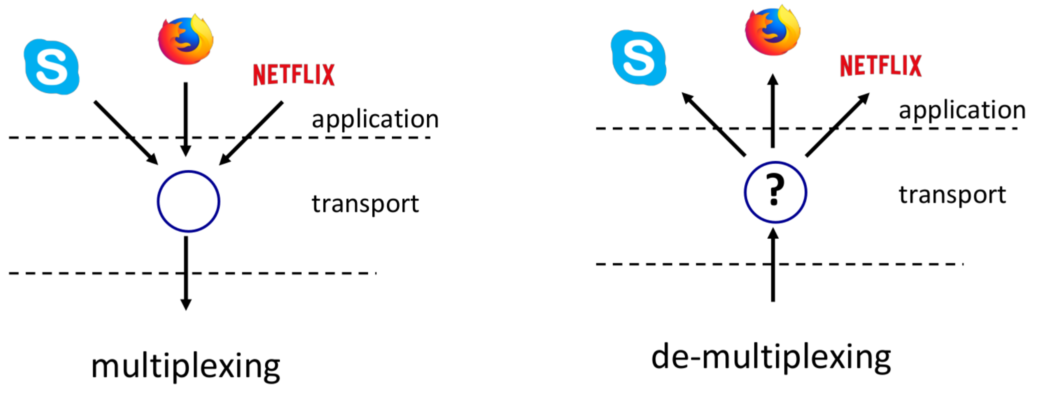
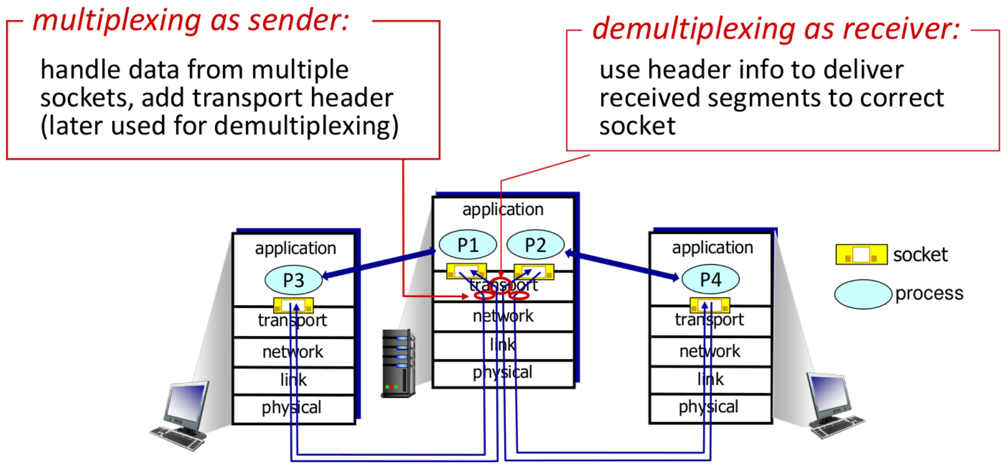
Unit 3

▼ Introduction & Responsibilities

- Responsible for End to End data delivery (port to port)
- Data unit at Transport layer is known as " **Segment** ". Data is sent from Application layer as a stream of bytes and that data is segmented in transport layer at sender side.
- We know that network layer and data link layer is also responsible for delivery of message from sender to receiver but Transport layer specifically manage, which process in sender agent is sending the message and which process in receiver agent is going to receive the message
    - example: p1 process from A PC is sending msg to p3 process of B PC, so transport layer will navigate the msg to specific port(process or socket).
- Transport layer commonly uses either of 2 Protocols TCP (Transmission control protocol) & UDP (User Datagram Protocol)
- When we're using TCP (connection oriented transmission) at transport layer protocol, it will provide
    - Reliability (ensuring that every packet reaches the destination).
    - Correct Ordering (the sequence of packets sent will be the same at receiver side).
    - Flow control (uses algorithm like Stop & wait, Go back n, Selective Repeat).
    - Congestion Control.
    - Error control functionality in Transport layer uses Checksum method to ensure data correctness.

    ▼ **Multiplexing & DeMultiplexing**



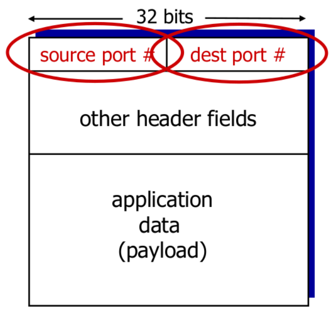


multiplexing as sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)
demultiplexing as receiver:
use header info to deliver received segments to correct socket
application
P3
transport
network
link
physical
application
P1
P2
transport
network
link
physical
application
P4
transport
network
link
physical
socket
process

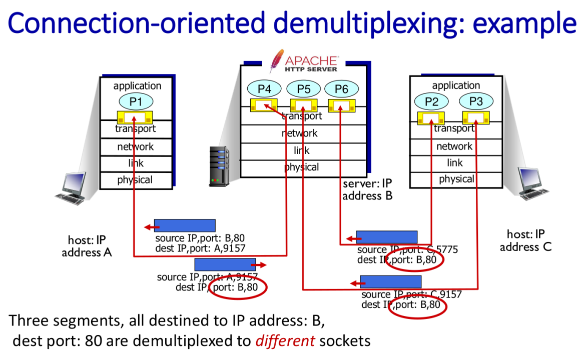
▼ **How De-Multiplexing Works?**

- IP/Network layer at receiver side will send Datagram/Packet to Transport layer, This datagram/packet is now segment at transport layer, Segment can be TCP segment and UDP segment, These segments have Source & Destination IP address & port number.
- Host uses these fields to navigate segment to appropriate socket.
- UDP: demultiplexing using destination port number (only)

- TCP: demultiplexing using 4-tuple: source and destination IP addresses, and port numbers



TCP/UDP segment format


Connection-oriented demultiplexing: example

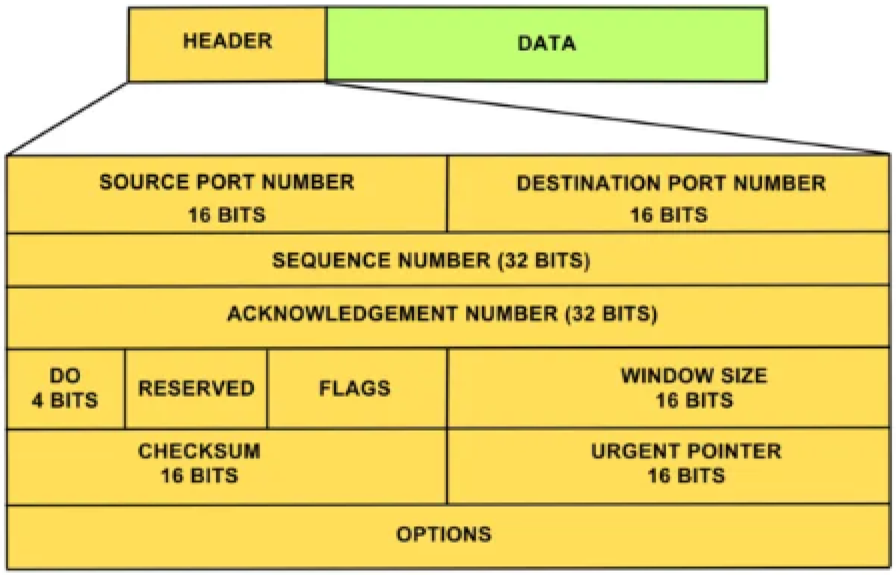
Three segments, all destined to IP address: B, dest port: 80 are demultiplexed to *different* sockets

▼ TCP

- Uses 3 Way handshaking ⇒
  1. Request for connection establishment to server.
  2. Response from server for connection acknowledgement.
  3. Request for Content / object.
- TCP is **full duplex** ⇒ Means at a time both server and client can communicate with each other.
- PiggyBanking ⇒ we can send data too with acknowledgement (for segment transmission) from server side.
- Error control
- Flow control (using stop & wait, go back n, selective repeat etc)
- Congestion control
- Services Not Supported by TCP:
  - delay guarantees
  - bandwidth guarantees

▼ **TCP Header**

- Size of TCP Header is around 20 to 60 Bytes.


HEADER
DATA
SOURCE PORT NUMBER 16 BITS
DESTINATION PORT NUMBER 16 BITS
SEQUENCE NUMBER (32 BITS)
ACKNOWLEDGEMENT NUMBER (32 BITS)
DO 4 BITS
RESERVED
FLAGS
WINDOW SIZE 16 BITS
CHECKSUM 16 BITS
URGENT POINTER 16 BITS
OPTIONS

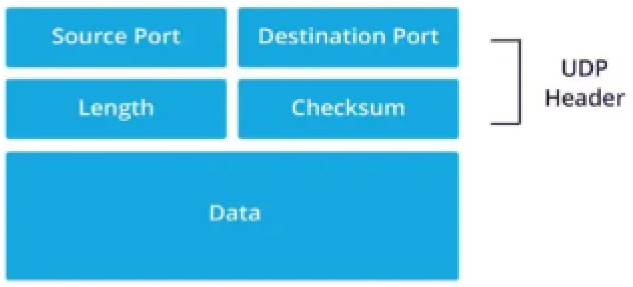
- In Transport layer segment there are more than one bytes and each byte is assigned a sequence number starting from a randomly choosen number, let's assume i th byte is sent and if the transmission was error free then receiver will send acknowledgement containing the i+1 number indicating that now i'm requesting for next byte.

▼ UDP

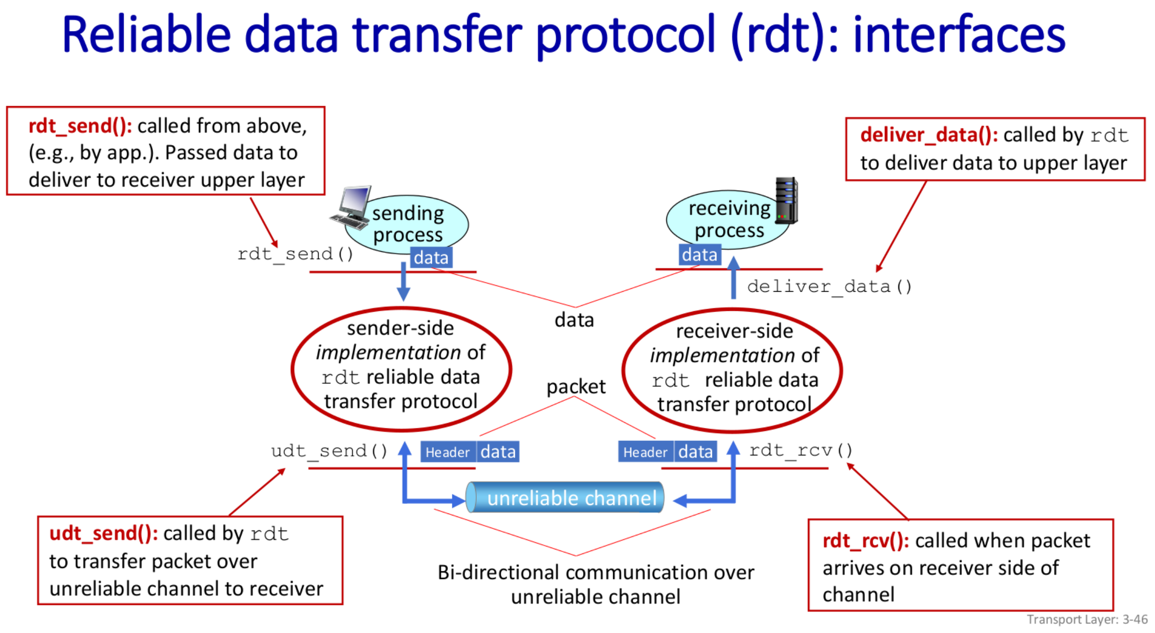
- Connectionless, Unreliable service for Data transmission.
- No handshaking between client & server.
- Each UDP segment handled independently of others.
- No ordering of packets.
- Benefits:
    - simple
    - fast, because no connection so no Extra RTTs.
    - Smaller header size
    - no congestion control
        - UDP can blast away as fast as desired!

- can function in the face of congestion

▼ UDP Header



▼ Reliable data transfer

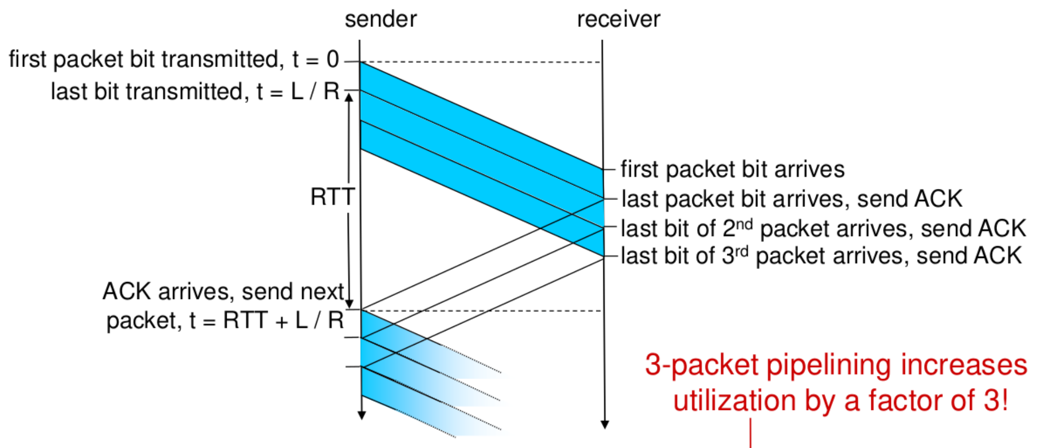

Reliable data transfer protocol (rdt): interfaces
rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer
deliver_data(): called by rdt to deliver data to upper layer
sending process
receiving process
data
rdt_send()
deliver_data()
data
sender-side implementation of rdt reliable data transfer protocol
receiver-side implementation of rdt reliable data transfer protocol
packet
udt_send()
Header data
rdt_rcv()
unreliable channel
udt_send(): called by rdt to transfer packet over unreliable channel to receiver
Bi-directional communication over unreliable channel
rdt_rcv(): called when packet arrives on receiver side of channel
Transport Layer: 3-46

- rdt 1.0 : Reliable transfer over reliable channel
- rdt 2.0 : channel with bit errors, checksum is used to find bit errors.
  - for bit error correction ACK, NACK (negative) concepts are used
  - but what if ACK, NACK itself is corrupted
- rdt 2.1 : if ACK, NACK not received or is corrupted, then server retrasmits packet with the same sequence id, so receiver discards the possible duplicates

- rdt 2.2 : a NACK free protocol, only send ACK with next sequence number, if the packet has not successfully reached receiver then receiver will again send ACK for same sequence number so that sender sends the same packet again just like NACK, but NACK is replaced by duplicate ACK.
- rdt 3.0 : channel with errors, loss
    - checksum, sequence #s, ACKs, retransmissions will be of help, but not quite enough
    - sender sends packet and then waits for ACK, if receiver does not get packet it will not send ACK, or may be ACK packet might be lost in the network so, after a certain time period sender will automatically send packet again to receiver, assuming that previously sent packet has not reached it's destination.

▼ Pipelining

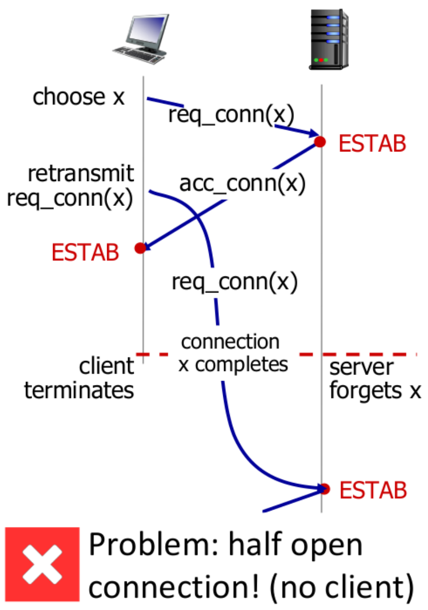
Pipelining: increased utilization


sender
receiver
first packet bit transmitted, t = 0
last bit transmitted, t = L / R
RTT
ACK arrives, send next packet, t = RTT + L / R
first packet bit arrives
last packet bit arrives, send ACK
last bit of 2nd packet arrives, send ACK
last bit of 3rd packet arrives, send ACK
3-packet pipelining increases utilization by a factor of 3!

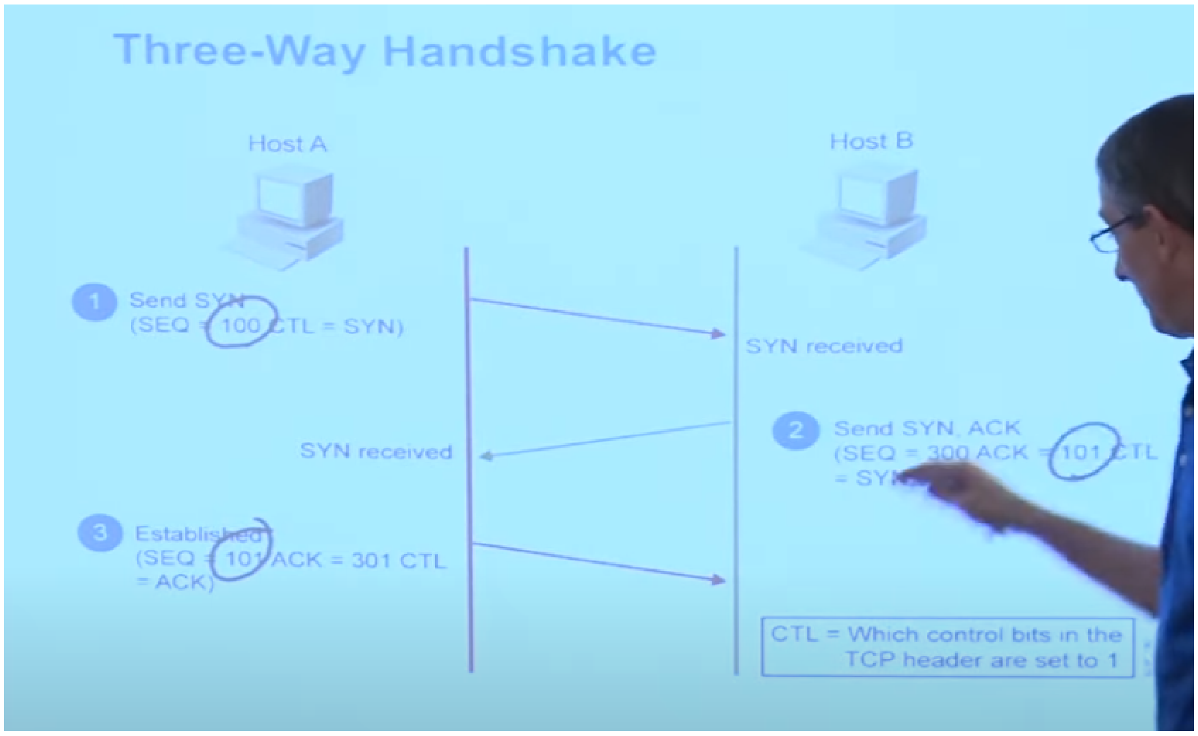
$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

▼ two way handshaking

2-way handshake scenarios

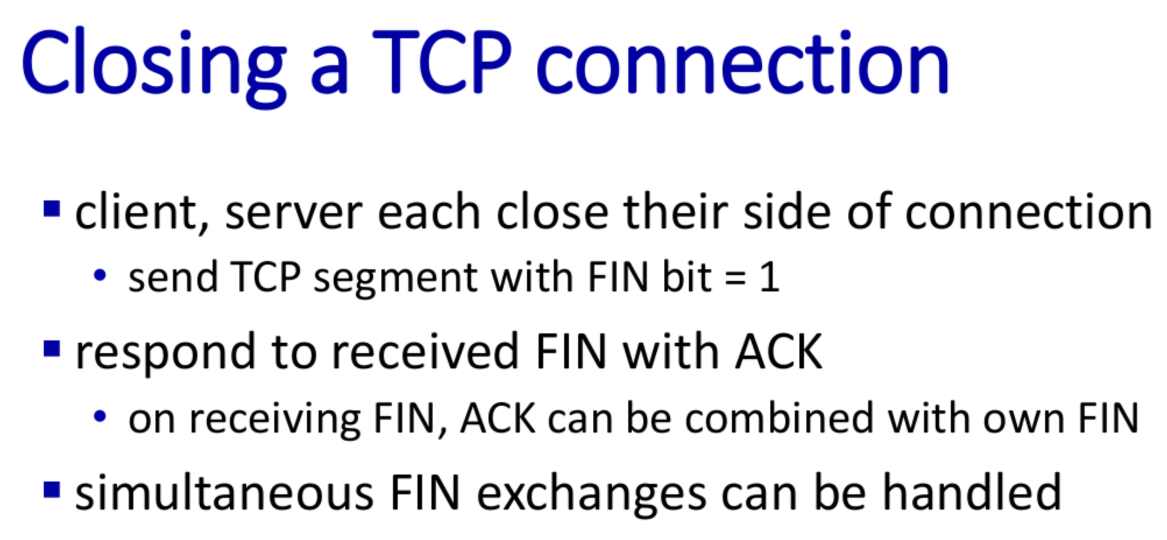


▼ 3-way handshaking



Three-Way Handshake
Host A
Host B
1 Send SYN (SEQ = 100 CTL = SYN)
SYN received
2 Send SYN, ACK (SEQ = 300 ACK = 101 CTL = SYN)
SYN received
3 Established (SEQ = 101 ACK = 301 CTL = ACK)
CTL = Which control bits in the TCP header are set to 1

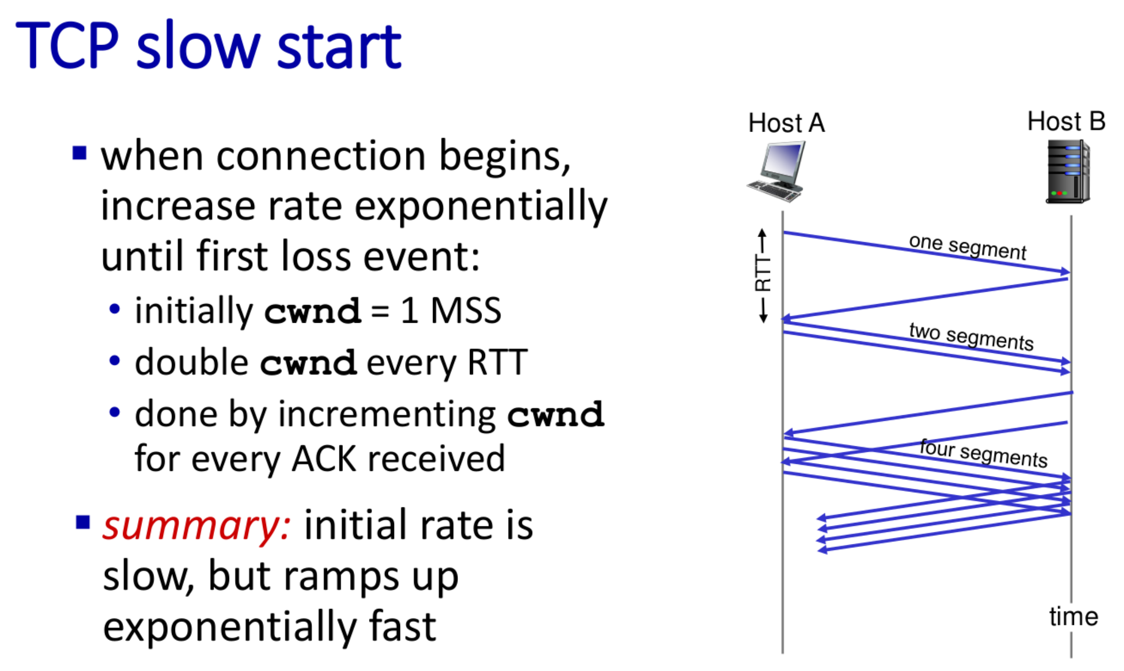
▼ TCP closing connection



Closing a TCP connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

▼ Slow Start & congestion control



TCP slow start

- when connection begins, increase rate exponentially until first loss event:
  - initially **cwnd** = 1 MSS
  - double **cwnd** every RTT
  - done by incrementing **cwnd** for every ACK received
- *summary:* initial rate is slow, but ramps up exponentially fast

Not mentioned topic

Go back n

Selective Repeat