2️⃣

Unit 2 - Transmission Control and IP

▼ TCP : Principles & Reliable Data Transfer

▼ Packet Loss

- Packet loss occurs because of corrupt packet discarded at the receiver
- Packet was discarded at a router due to lack of buffer space
- Packets experience long queuing delays in a router

▼ Reliable Data Transfer

- In a real network, packets can be corrupted, lost or delayed
- UDP doesn’t provide reliability but TCP does by using RDT (reliable data transfer)
- The idea of RDT is the sender and receiver must cooperate using acknowledgements (ACKs), negative acknowledgements (NAKs), timers and retransmissions
- Reliable Data Transfer between hosts A and B is achieved when both agree to monitor the packets exchanged and notify when packet loss is detected. This is achieved by some handshaking between A and B before the packets are exchanged
- We start with an ideal case and incrementally add complexity to transport layer protocol
  So, transport layer protocol will be referred to as *RDT Protocol*

▼ Stages of Development of RDT Protocol

▼ Stop and wait RDT Protocols

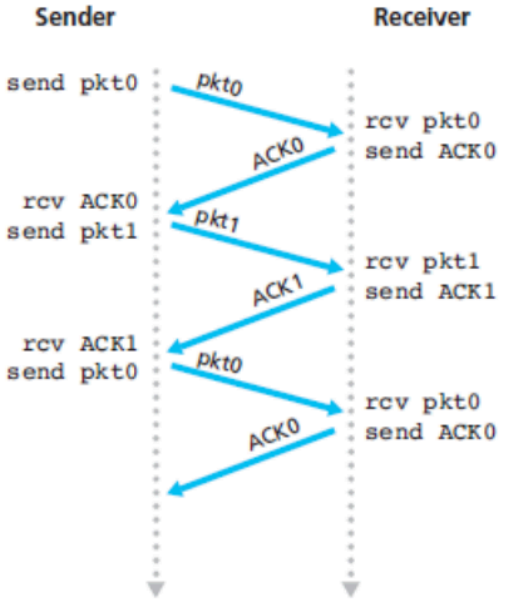
- This is the simplest baseline idea

- Host A sends one packet at a time, waits for ACK from host B to transmit neck packet
- This ensures order and delivery

▼ 4 incremental versions of stop and wait RDT protocols

▼ RDT 1.0 - Ideal world

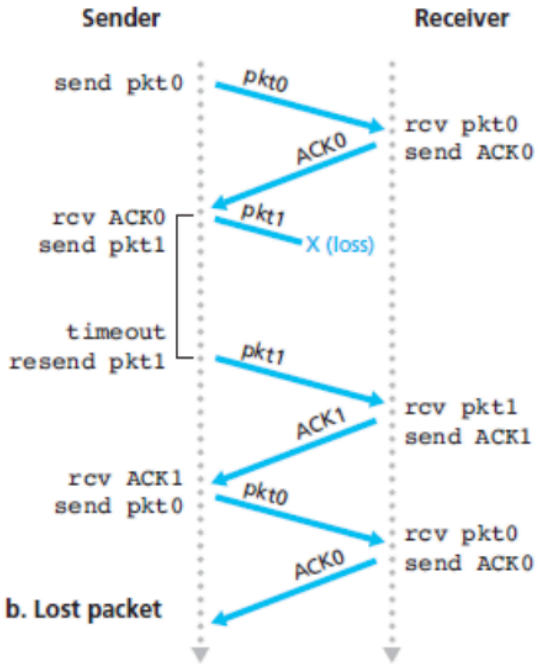
- No errors, no loss
- Sender just sends → Receiver gets it. Done



a. Operation with no loss

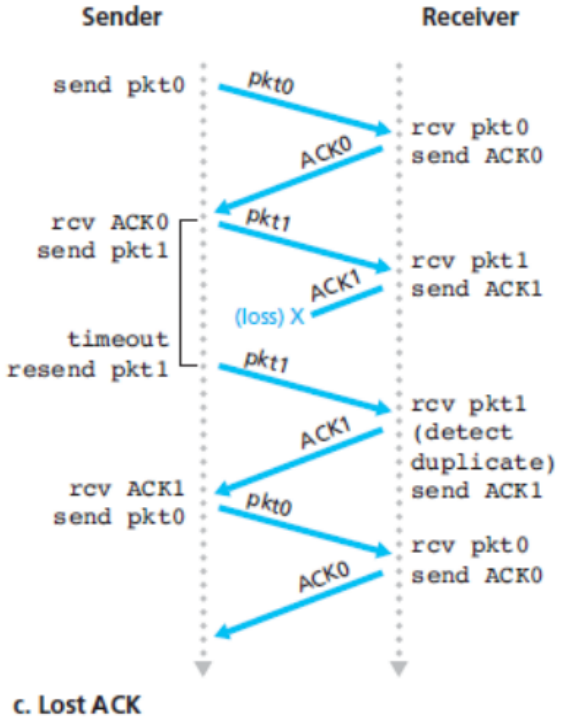
▼ RDT 2.0 - Errors possible

- Add *checksum* to detect errors
- Receiver replies with ACK (all good) or NAK (corrupt)
- Sender resends if it gets NAK



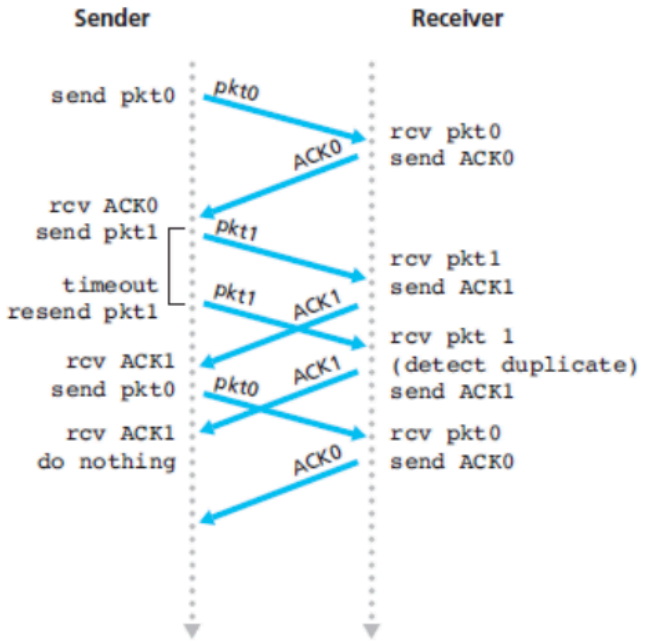
▼ RDT 2.2 - Handling corrupted ACK/NAK

- Now if ACK/NAK only get corrupted, then we add a sequence number (0 or 1 alternately) to each packet
- Receiver just sends ACK with sequence number
- Sender resends if wrong or missing

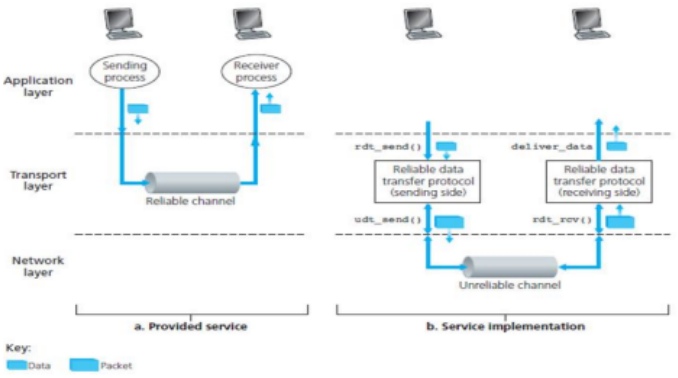


▼ RDT 3.0 - Losses and delays happen

- If a packet or ACK just disappears, then sender will wait forever
- To avoid this, sender sets a timer when sending a packet
- If timer expires without valid ACK → retransmit
  Receiver ignores duplicate packets using sequence number

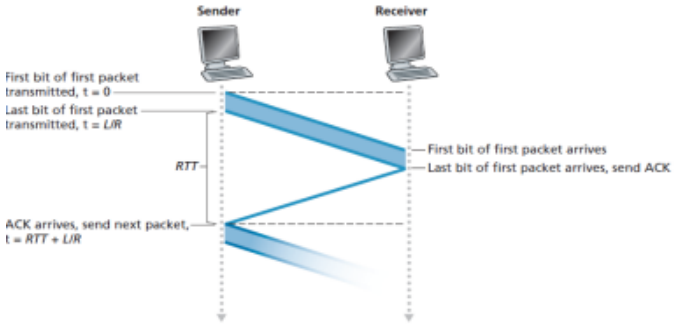


d. Premature timeout

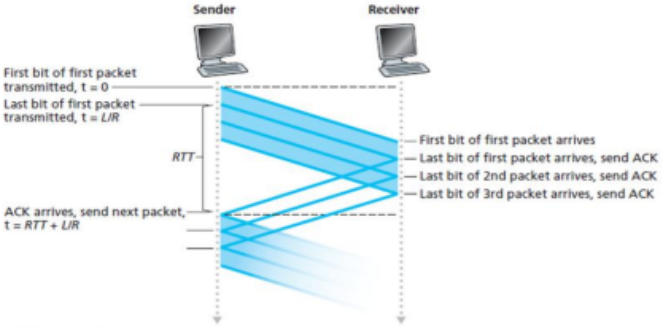


▼ Pipelining RDT Protocols

- In Stop-and-Wait, sender sends only 1 packet a time. So if link has high bandwidth x delay, utilization is very poor



- In pipelining, Host A sends multiple packets to Host B at a time



- Host A waits for acknowledgements from Host B with in a fixed time interval known as *timeout*
- Upon learning successful delivery of packets in previous round, next batch of packets are transmitted
- Compared to stop and wait RDT protocols, these pipelining RDT protocols provide better link utilization

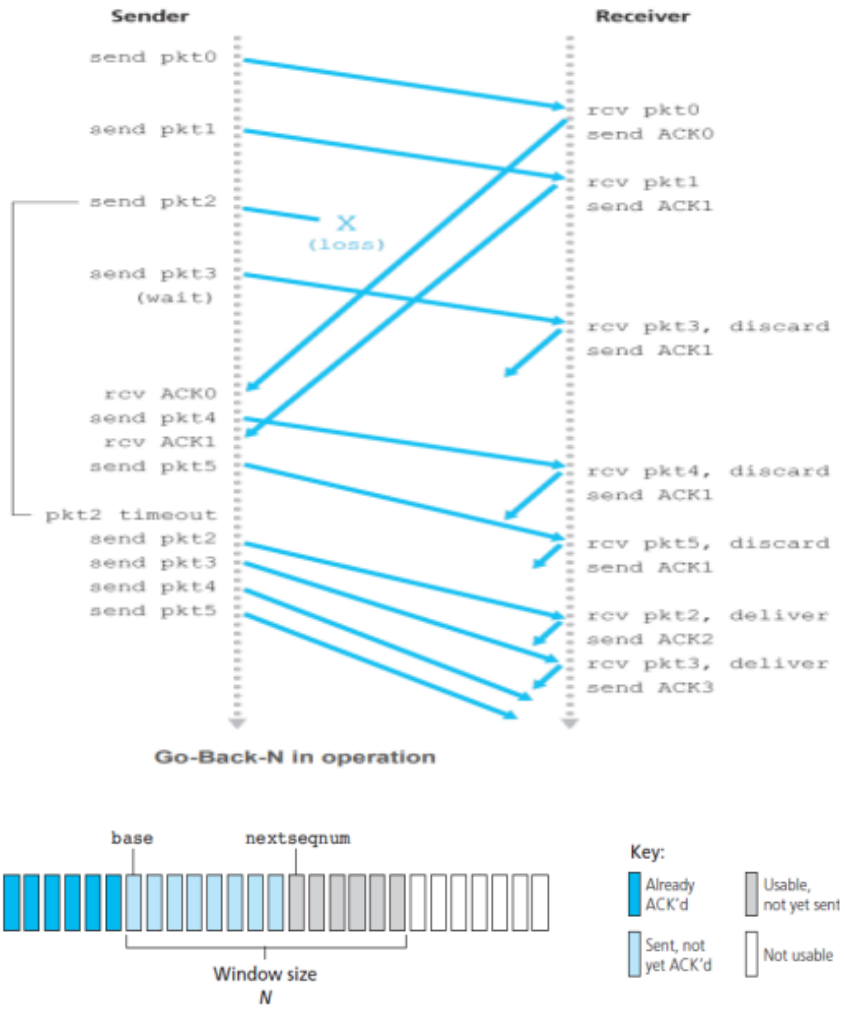
▼ 2 versions of pipelining RDT protocols

▼ GBN

- Stands for Go-Back N Protocol
- Sender :
  - Maintains a window of N packets (sends N unacknowledged packets)
  - If an ACK is not received for the earliest packet, sender retransmits that packet and all following packets in the window
- Receiver :
  - Only keeps track of the next expected packet

- If packet is missing or corrupted, receiver discard it and all following packets
- Sends cumulative ACK (last in-order packet received)

- It is simple, cumulative ACKs reduce feedback overhead
- But it wastes bandwidth, a single loss causes retransmission of many packets



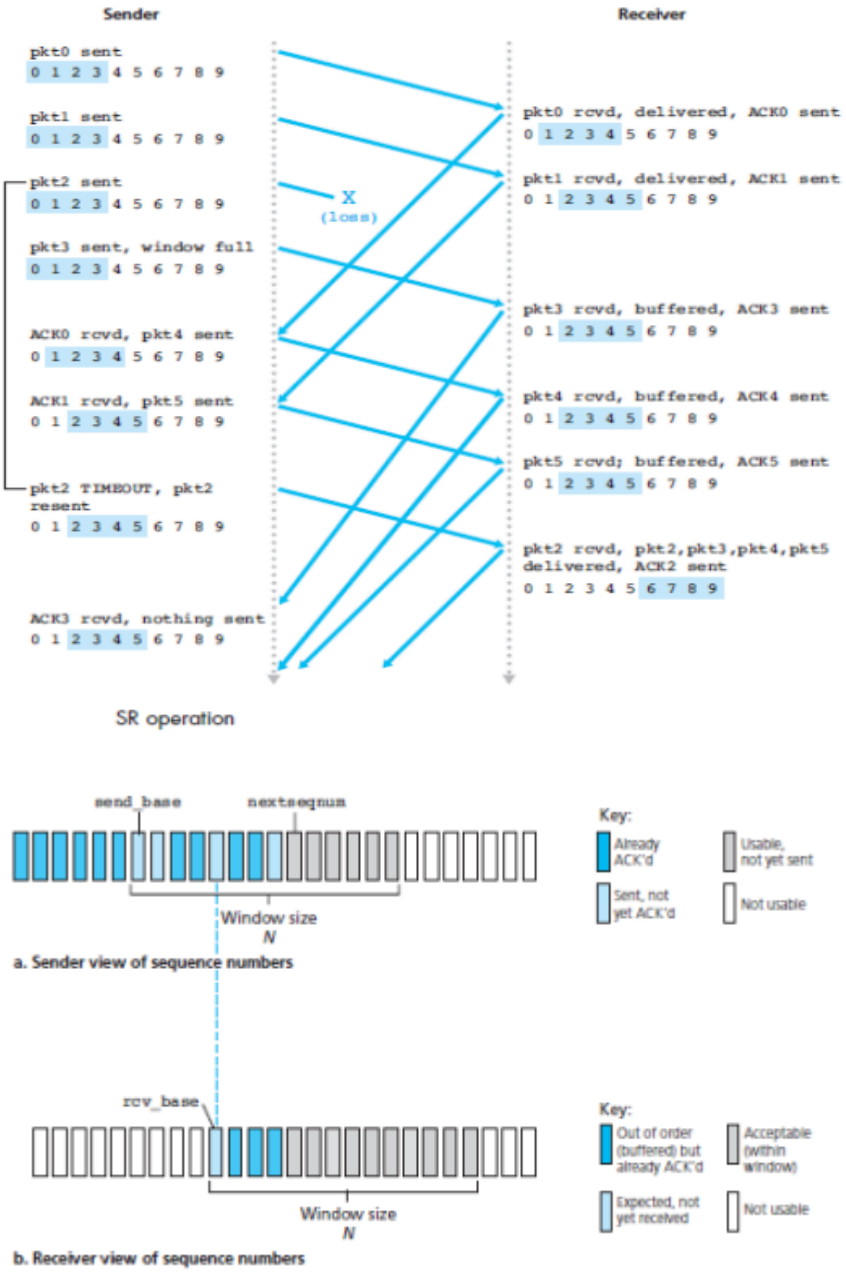
Sender's view of sequence numbers in GBN

▼ SR

- Stands for Selective Release Protocol
- Sender :
  - Maintains a window of N packets

- But only retransmits the packets that were lost/corrupted

- Receiver :
  - Buffers correctly received out-of-order packets until missing one arrives
  - Sends individual ACKs for each packet



Sender's and Receiver's view of sequence numbers

▼ Comparison between GBN and SR

Go-Back N	Selective Repeat
GBN takes cumulative acknowledgements	SR doesn't take cumulative acknowledgements
If some ACKs are missing, GBN retransmits all packets under the sliding window	If some ACKs are missing, SR retransmits selectively
A receiver using GBN doesn't store out-of-order packets	A receiver using SR stores out-of-order packets
Sender and Receiver sliding windows under GBN are in sync	Sender and Receiver sliding windows under SR can be out of sync
Retransmit unacknowledged packets upon timeout	Retransmit unacknowledged packets upon timeout

▼ Limitations of GBN and SR

- Timer values are fixed arbitrarily
- Transmission rate (sliding window length) is fixed
- They can't exploit low network congestion nor prevent causing network congestion
- Message segmentation is performed arbitrarily
- Buffer overflow may occur at receiver
- Two-way data transmission can't be implemented
- Only one pair of processes between sender-receiver can communicate at any time
- To overcome all these limitations we use Transmission Control Protocol

▼ Transmission control Protocol (TCP)

- It is a hybrid of the above pipelining RDT protocols but with sophistication of its own
- TCP is robust compared to pipelining RDT protocols

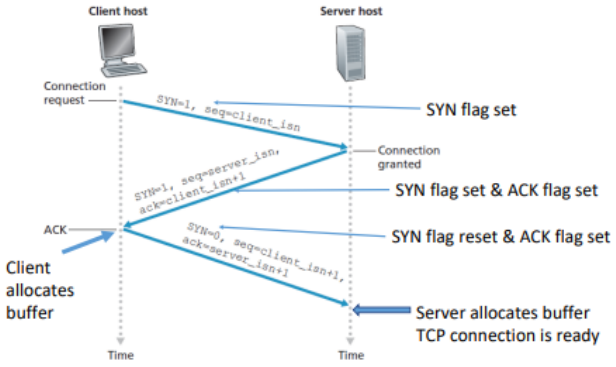
- TCP makes host A adaptive to network congestion and packet overflow problems that may occur at host B

▼ MSS

- MSS stands for Maximum Segment Size.
- It defines the largest size of the packet (or) application-layer data in the segment that can be transmitted as a single entity in a network connection
- The size of the MTU (Maximum Transition Unit) dictates the amount of data that can be transmitted in bytes over a network
- MSS is set to ensure that a TCP segment + TCP/IP header length will fit into single link-layer frame
  MSS ≠ Max amount of TCP segment + header
  MSS = Max amount of application layer data in the segment

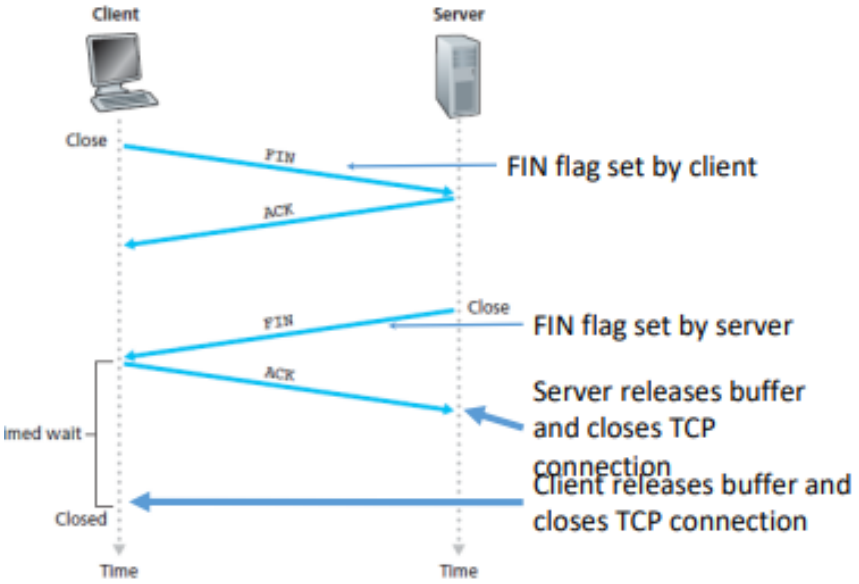
▼ TCP Connection Establishment (3-way Handshake)

- Client → Server : Sends a SYN (synchronize) segment
  - `SYN = 1`
  - client_ISN = Randomly chosen sequence number
  - `ACK = 0`
- Server → Client : Replies with SYN+ACK
  - `SYN = 1`
  - `ACK = 1`
  - server_ISN = Random sequence number
  - Acknowledgment number = client_ISN + 1
- Client → Server : Sends final ACK
  - `SYN = 0`
  - `ACK = 1`
  - Acknowledgment number = server_ISN + 1



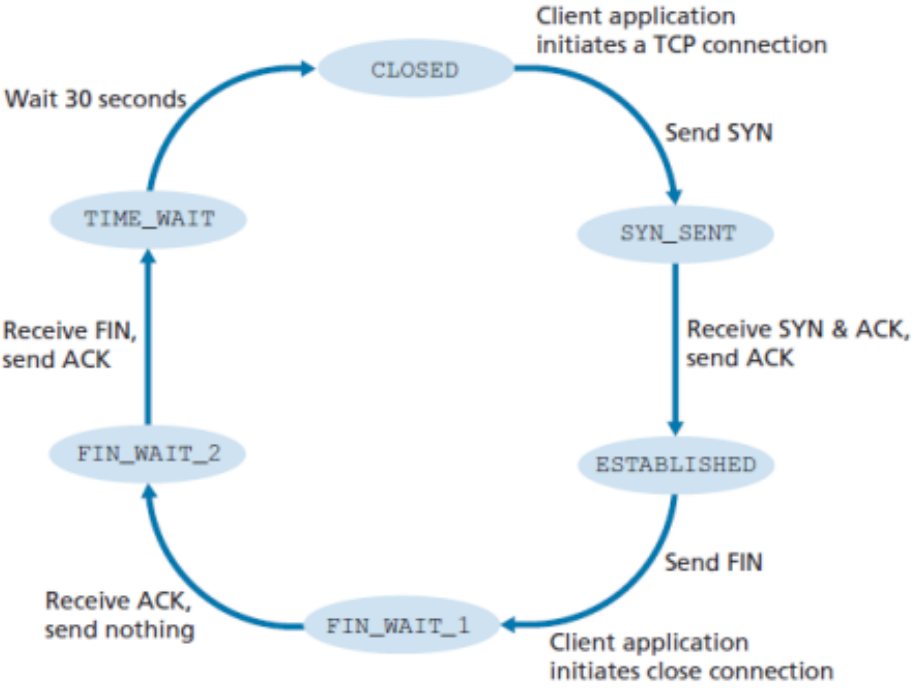
▼ TCP Connection Termination (4-way Handshake)

- Client sets `FIN = 1` (FIN : finish), which means client wants to close
- Server acknowledges then sets `FIN = 1` later
- Both sides release buffers and close the connection

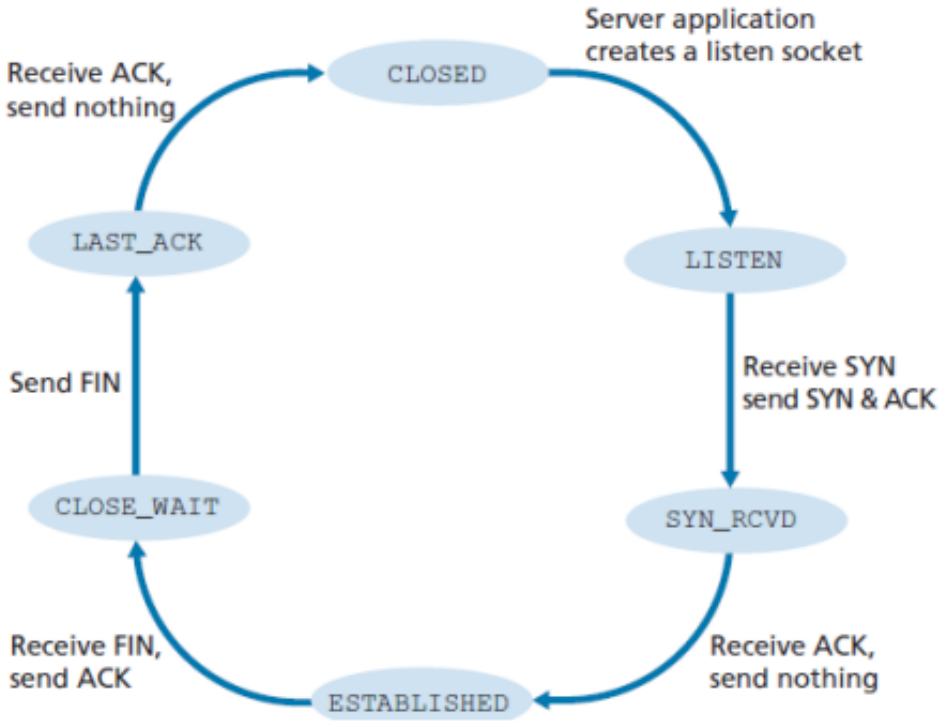


▼ States seen by client and server

- Client



- Server

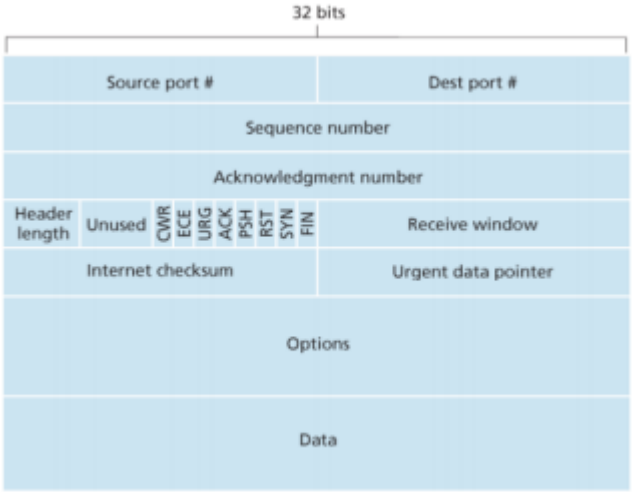


▼ TCP Segment Format

- Source Port (16 bits) & Destination port (16 bits)
  - Identify which application process is talking (like HTTP = 80, DNS = 53)

    - Together with IP address, they form socket
- Sequence number (32 bits)
    - Number of the first byte of data in this segment
    - Ex : if seq = 1000 and 500 bytes of data, the next seq = 1500
- Acknowledgment number (32 bits)
    - Next byte receiver expects to get
    - Example if ack = 1500 → all bytes up to 1499 have been received
- Data Offset/ Header Length (4 bits)
    - Length of TCP header
    - Min = 5 (20 bytes) ; Max = 15 (60 bytes)
    - in 20 bytes, UDP = 8 bytes and DNS = 12 bytes
- Reserved (4 bits)
    - Currently unused, set to 0
- Flags (6 bits, 1 bit per flag)
    - CWR : Congestion Window Reduced Flag → Congestion Notifications
    - ECE : Explicit Congestion Echo → Congestion Notifications
    - URG : Urgent Pointer Field valid
    - ACK : Acknowledgement Field valid
    - PSH : Push data immediately (skip buffering)
    - RST : Reset connection
    - SYN : Synchronize Sequence numbers
    - FIN : Sender finished sending data
- Receive window (16 bits)
    - Flow control : max number of bytes receiver is ready to accept

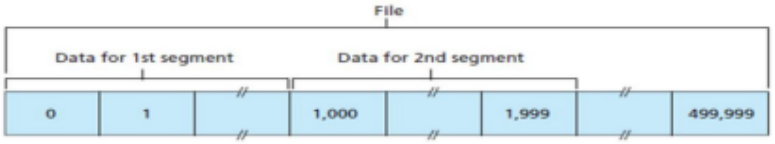
    - Tells sender how much it can send before needing ACK
- Internet Checksum(16 bits)
    - Error detection (includes TCP header + data + parts of IP header)
- Urgent Data Pointer (16 bits)
    - Used with URG flag which shows where urgent data ends. Rarely used nowadays
- Options (variable length, multiple of 4 bytes)
    - Common ones : MSS, Window Scaling Timestamps
- Data (variable length)
    - Application payload (ex : HTTP request, email content)



▼ TCP Sequence numbers and ACKs

▼ Sequence Numbers

- Every byte of data in TCP has sequence number
- A message maybe divided based on the MSS into TCP segments
- The 1st byte in each segment is assigned a unique sequence number before being transmitted
- For the 1st segment a random 32 bit number is assigned
- The received segments are reordered using these numbers



▼ ACKs

- Used by receiver to inform the sender of missing segments
- Receiver acknowledges each received segment
- Receiver stores out-of-order segments
- Replies with acknowledgement of the last correctly received in-sequence segment
- Cumulative acknowledgments are accepted by sender in case out-of-order acknowledgments are received
- The acknowledgement for client-to-server data is carried in a segment carrying server-to-client data. This acknowledgment is said to be piggybacked on server-to-client data segment

▼ Example

- Suppose that Host A has received all bytes numbered 0 through 535 from B and suppose that it is about to send a segment to Host B. Host A is waiting for byte 536 and all the subsequent bytes in Host B's data stream. So Host A puts 536 in the acknowledgment number field of the segment it sends to B.
- Suppose that Host A has received one segment from Host B containing bytes 0 through 535 and another segment containing bytes 900 through 1,000. For some reason Host A has not yet received bytes 536 through 899. In this example, Host A is still waiting for byte 536 (and beyond) in order to re-create B's data stream. Thus, A's next segment to B will contain 536 in the acknowledgment number field. Because TCP only acknowledges bytes up to the first missing

byte in the stream, TCP is said to provide cumulative acknowledgments.

- ▼ Round Trip Time Estimation and Timeout
  - ▼ What and Why RTT Estimation
    - RTT is the time taken to get an acknowledgement since the segment was passed to the network layer
    - TCP retransmits if an ACK doesn't come back in time but the time it should wait ?
      TCP dynamically estimates the Retransmission Timeout (RTO) using measured RTTs
    - ▼ We have 4 different values we can calculate
      - ▼ Sample RTT
        - RTT is measured for one of the TCP segments at a time
        - Instantaneous value of RTT is denoted as *Sample RTT*
        - $Sample\ RTT = Time\ ACK\ received - Time\ segment\ sent$
      - ▼ Estimated RTT
        - Time averaged statistics are generated for the RTTs
        - The mean value is Estimated RTT
        - $Estimated\ RTT = (1 - \alpha) * Estimated\ RTT + \alpha * Sample\ RTT$
        - Usually $\alpha = 0.125$ which is an exponential weighted moving average (EWMA) which smooths out fluctuations
      - ▼ Dev RTT
        - The standard deviation is Dev RTT
        - It measures how much RTT is changing (jitter)

- $Dev\ RTT = (1 - \beta) * Dev\ RTT + \beta * |Sample\ RTT - Estimated\ RTT$
- Typically $\beta = 0.25$

▼ Timeout Interval

- Final RTO (timeout)
- $Timeout\ Interval = Estimated\ RTT + 4 * Dev\ RTT$
- The extra 4*Dev RTT gives a safety margin in case of spikes in delay

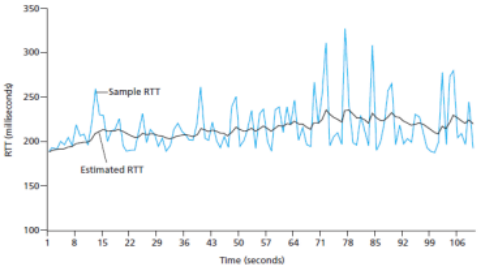


Figure: RTT versus transmission rounds

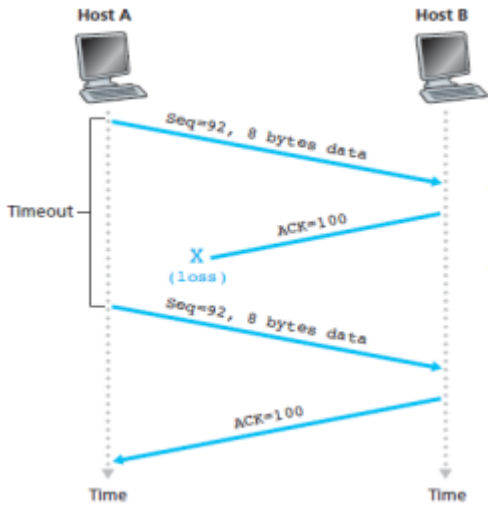
▼ Final Procedure for updating Timeout interval

- Initialize the Timeout interval to 1 second
- Select a fresh segment and measure its Sample RTT
- If all ACKs are received before timer expired, then Timeout interval is updated for next round using the previous equations
- If timeout occurs, Timeout interval is doubled
- Packets without ACKs in previous round are retransmitted and timer is set to the Timeout interval
- This steps are repeated as TCP segments with payload are sent

▼ Reliable Data Transfer

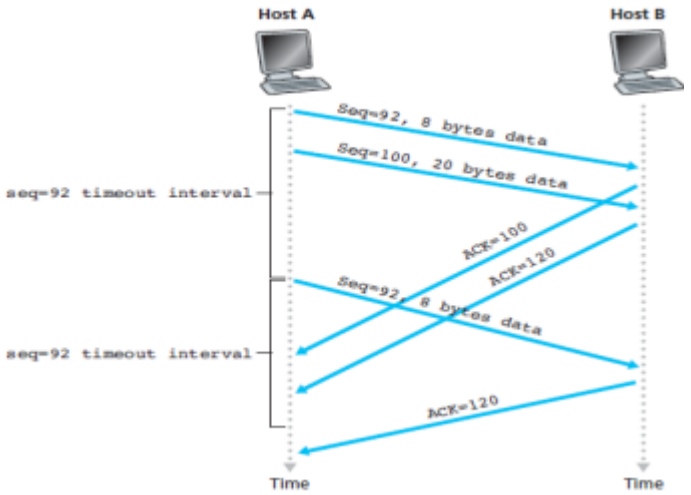
▼ Case 1 : ACK Lost + Timeout

- Suppose sender sends a segment
- Receiver gets it and sends ACK but ACK is lost in the network
- The sender’s timer expires (Time out)
- Sender retransmits the segment even though receiver already got it
- TCP also doubles the timeout value



▼ Case 2 : ACK arrives after Timeout

- After sender sends the segment, receiver sends the ACK but it reaches after Timeout. ACK is delayed but not lost
- After timeout sender retransmits the old segment anyways and timer is doubled
- Receiver ignores duplicate because it already moved forward
- TCP can’t know if ACK is lost or delayed, so it stays on the safe side by retransmitting



▼ Case 3 : Cumulative Acknowledgment

- For example, ACK for seq=92 is lost
- But later ACK for seq=100 arrives before timeout
- Because TCP uses cumulative ACKs, sender knows that everything up to 100 is received and no retransmission is required

▼ Case 4 : Duplicate ACKs (Out-of-order segments)

- Segments can arrive out of order
- Receiver uses a conservative approach when acknowledging
- If packet seq=100 is missing, but seq = 101, 102 arrive, then it cannot ACK 101 or 102 yet

▼ Questions

▼ Consider using a stop and wait protocol between two hosts who are connected by a direct link of rate 1 Mbps. Let the one way propagation delay be 99.5 ms. For a packet of size 1000 bits, determine the link utilization (%) and the throughput of the sender (bits/s). Explain your result with appropriate formulae

- $Link\ Utilization = \frac{Transmission\ delay}{Transmission\ delay + RTT} = \frac{\frac{1000}{10^6}}{\frac{1000}{10^6} + 2*99.5*10^{-3}}$ = 0.005 or 0.5%
- $Throughput = \frac{packet\ size}{transmission\ delay + RTT} = \frac{1000}{\frac{1000}{10^6} + 2*99.5*10^{-3}}$ = 5000 bps

▼ Suppose a pipelining protocol is used instead of a stop and wait protocol in the question 1A. Calculate the number of packets to achieve 100% link utilization. What will be the number of bits required to represent the sequence numbers in this case?

- Since 1 packet achieves 0.5% Utilization
  Then to achieve 100% Utilization, 100% * 1/0.5% = 200 packets
- The number of bits required to represent the sequence numbers is
  $\log_2(200) = 7.6439 \Rightarrow 8bits$

▼ Consider a Link bandwidth (R) = 1 Mbps, One-way propagation delay d_{prop} = 5 ms, Packet size (L) = 1000 bytes = 8000 bits, Now find the Line utilization for both Stop-and-Wait RDT as well as Pipelining RDT of window size 4

- Transmission Delay $T_{trans} = \frac{L}{R}$ = $\frac{8000}{10^6} = 0.008s = 8ms$
- $RTT = 2 * d_{prop} = 2 * 5ms = 10ms$
- For Stop-and-Wait RDT,
  $U = \frac{T_{trans}}{RTT+T_{trans}} = \frac{8}{30+8} = 0.21$ ⇒ 21% Utilization
- For Pipelining RDT,
  $U = min(1, \frac{N*T_{trans}}{RTT+T_{trans}}) = min(1, \frac{4*8}{30+8}) = \frac{32}{38} = 0.84$ ⇒ 84% Utilization
- From this we can say that, Stop-and-wait is wasteful if RTT >> Transmission delay
  and Pipelining increases utilization drastically

▼ Assume five samples of RTT were obtained as 106ms, 120ms, 140ms, 90ms and 115ms. Assume the initial values of EstimatedRTT and DevRTT as 100ms and 5ms respectively. Calculate the Timeout interval for each transmission round using the corresponding values of SampleRTT given above. Consider the case of 'No packet loss in any round'.

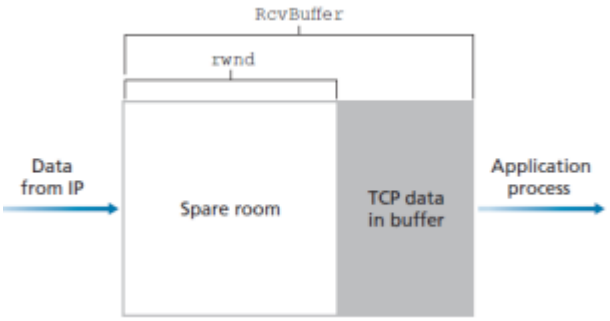
- $Dev\ RTT = (1 - \beta) * Dev\ RTT + \beta * |Sample\ RTT - Estimated\ RTT$

- $Estimated\ RTT = (1 - \alpha) * Estimated\ RTT + \alpha * Sample\ RTT$
- $Timeout\ Interval = Estimated\ RTT + 4 * Dev\ RTT$
- After first sample RTT 106 ms:
  DevRTT = (0.75∗5) + 0.25∗|106-100| = 3.75 + 1.5 = 5.25 ms
  EstimatedRTT = (0.875∗100) + (0.125∗106) = 100.75 ms
  Timeout interval = 100.75 + 4∗5.25 = 121.75 ms
- After second sample RTT 120 ms:
  DevRTT = (0.75∗5.25) + 0.25∗|120-100.75| = 8.75 ms
  EstimatedRTT = (0.875∗100.75) + (0.125∗120) = 103.16 ms
  Timeout interval = 103.16 + 4∗8.75 = 138.16 ms
- After third sample RTT 140 ms:
  DevRTT = (0.75∗8.75) + 0.25∗|140-103.16| = 15.77 ms
  EstimatedRTT = (0.875∗103.16) + (0.125∗140) = 107.76 ms
  Timeout interval = 107.76 + 4∗15.77 = 170.84 ms
- After fourth sample RTT 90 ms:
  DevRTT = (0.75∗15.77) + 0.25∗|90-107.76| = 16.27 ms
  EstimatedRTT = (0.875∗107.76) + (0.125∗90) = 105.54 ms
  Timeout interval = 105.54 + 4∗16.27 = 170.62 ms
- After fifth sample RTT 115 ms:
  DevRTT = (0.75∗16.27) + 0.25∗|115-105.54| = 14.57 ms
  EstimatedRTT = (0.875∗105.54) + (0.125∗115) = 106.72 ms
  Timeout interval = 106.72 + 4∗14.57 = 165 ms

▼ TCP : Flow & Congestion Control

▼ Flow Control

- Every host allots a buffer to store the incoming signals
- The segments are accessed by application layer
- When application reads the data slower than rate at which segments were received, then there is packet overflow
- So to overcome this, the sender can be notified using the *receive buffer field* in the TCP header
- Because TCP is full-duplex, sender at each side of the connection maintains a distinct receive window



- Each receiver has a *receive buffer*
- Receiver advertises a window size (called rwnd, receive window) to the sender which is the amount of free space left in the buffer
- So to prevent overflow at receiver buffer of B, the condition to ensure is
  Last Byte Rcvd - Last Byte Read $\leq$ RcvBuffer
- The receive window (rwnd) is given by
  rwnd = RcvBuffer - [Last Byte Rcvd - Last Byte Read]

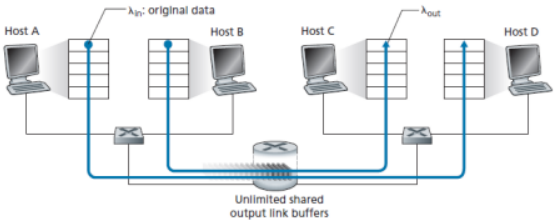
▼ Congestion Control

▼ Principles of Congestion Control

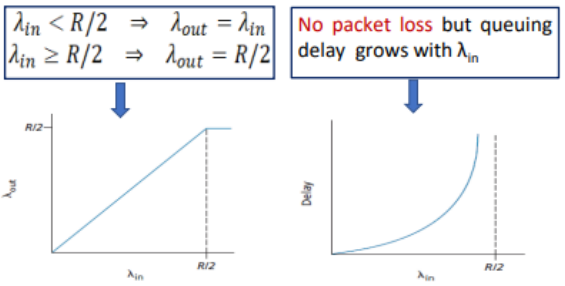
- Congestion refers to a condition where the network experiences flow rates exceeding transmission rates
- In such situations, routers could experience queue saturations leading to some packets being dropped (packet loss) and wasted work
- Luckily, TCP implements congestion control algorithm to minimize the impact of network congestion on the application performance

▼ Lets assume 3 scenarios

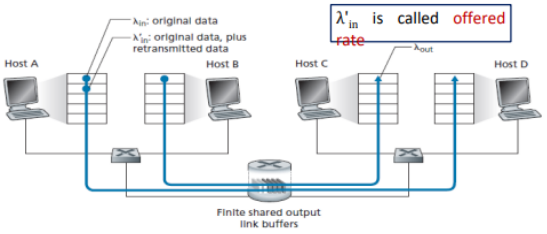
- Infinite Buffer Router
  - In this case, 2 hosts send traffic at rate λ each and link capacity = R
  - Router split bandwidth equally (Each get R/2 throughput)



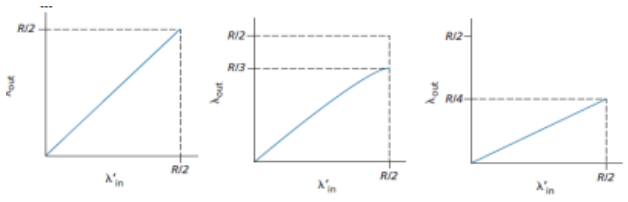
- No packet loss because buffer is infinite
- But this leads to indefinite queueing delay as λ approaches R/2 (Unrealistic)



- Finite Buffer Router
  - Line rate is R but now buffer size is limited

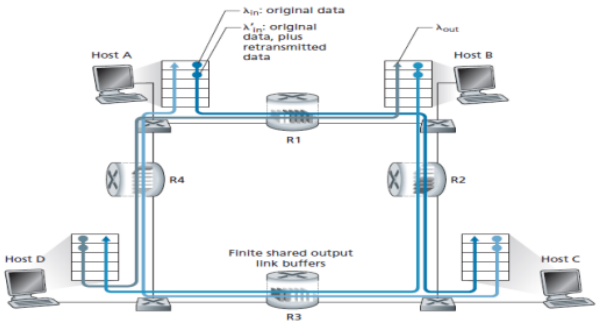


  - If $\lambda'_{in} < \frac{R}{2}$ → no packet loss, throughput = λ'_{in}
  - If $\lambda'_{in} \geq \frac{R}{2}$ → packets dropped and throughput = R/3
  - Retransmission occurs after every loss cause $\lambda_{in} = \frac{\lambda'_{in}}{2}$ → Wasted bandwidth and inefficiency



- Multi-hop Flows

- Four Flows: A→C B→D C→A D→B through multiple routers
- If router R2 drops packets from flow A→C due to congestion caused by B→D, then work done by upstream R1 is wasted
- Even through A→C wasn't the cause of congestion, it suffers losses and may get zero throughput if continuously blocked



- Without congestion control, TCP/IP networks would collapse under heavy load
- Congestion control ensures high utilization, Fairness among flowing competing flows, and avoidance of collapse

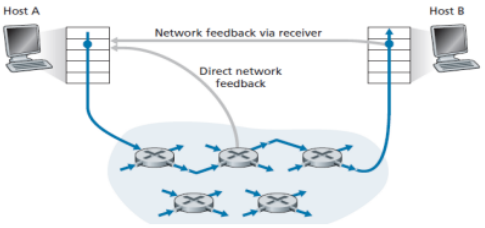
▼ Approaches to congestion control

▼ End-to-end congestion control

- Congestion inferred without explicit help from the network
- Sender reduces rate upon detection congestion (Timeout events, Triple duplicate ACKs)
- Used by TCP
- Advantage is that it doesn't need to modify routers

▼ Network-Assisted Congestion Control

- Routers provide explicit feedback about congestion
- 2 methods
  - Direct: Router sends special *congestion notification* packet to sender



    - Indirect: Router sends a choke packet to receiver, which notifies sender
  - Used in order networks (ATM, IBM, SNA, DECnet)
  - Not used widely in Internet's TCP/IP model

▼ TCP Congestion window

- Transmission rate depends on congestion window and RTT
- Congestion window in TCP is adaptive
  (cwnd inversely proportional to network congestion)
- cwnd (congestion window/sliding window) is a sender-side variable controlling no of unacknowledged bytes "in flight"
- Transmission rate = cwnd/RTT
- TCP segment length is called MSS (Maximum segment size) MSS is negotiated by sender and receiver using SYN and SYNACK segments via options field in TCP header
- Packer length (L) transmitted = MSS + IP Header + Link Layer Header

▼ Classic TCP Congestion Control

- TCP's congestion control is designed to maximize throughput while avoiding network collapse
- TCP detects congestions in 2 events
  - Timeout event : ACK not received withing the retransmission timeout (RTO) which indicated heavy congestion
  - Triple duplicate ACKs : 3 repeated ACKs for the same packet → indicates one packet is lost, but others are still flowing (lighter congestion)
- TCP Modifies cwnd dynamically :

- If no packet loss → increase cwnd (probe for available bandwidth)
- If packet loss → decrease cwnd (relieve congestion)
- This probing strategy is conservative (avoids overloading)

▼ Congestion control algorithm has 3 phases (described in RFC5681)

▼ Slow start phase (Present in TCP Reno and TCP Tahoe)

- Initial cwnd = 1 MSS
- On each ACK received :
  - cwnd increases by 1 MSS
  - Effectively doubles every RTT
- Transition to next phase when :
  - cwnd $\geq$ ssthresh (slow start threshold)
  - Or when packet loss occurs
- On timeout :
  - ssthresh = 0.5*cwnd
  - cwnd reset to 1 MSS (restart slow start)

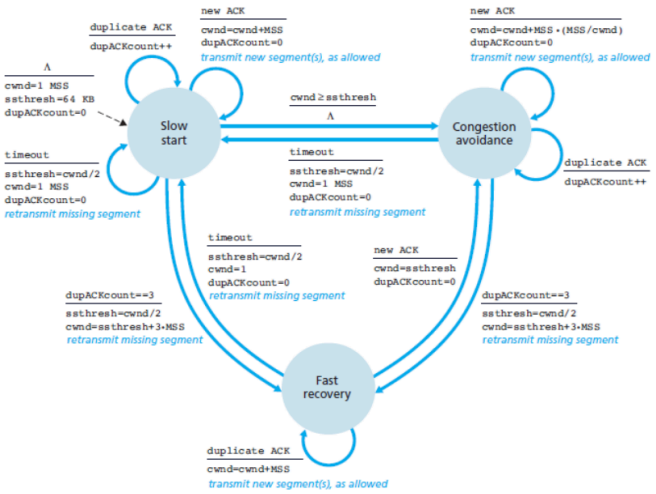
▼ Congestion avoidance (Present in TCP Reno and TCP Tahoe)

- Starts when cwnd $\geq$ ssthresh
- Growth is linear : cwnd increases by ~1 MSS per RTT
- This is the additive increase part of AIMD
- On packet loss :
  - ssthresh = 0.5*cwnd
  - cwnd reduced accordingly
  - Timeout → reset to slow start
  - Triple duplicate ACK → fast recovery (if Reno)

▼ Fast recovery (Present only in TCP Reno)

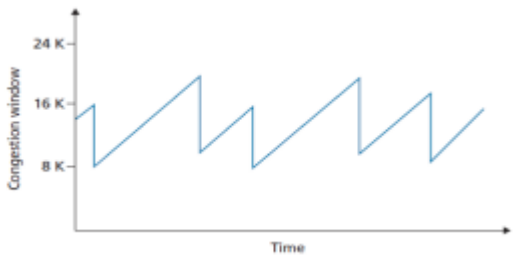
- Triggered by triple duplicate ACKs

- Retransmit lost segment immediately
- ssthresh = 0.5*cwnd
- cwnd = ssthresh + 3 MSS (accounts for the 3 ACKs)
- Continue with congestion avoidance
- Avoids going all the way back to cwnd = 1 (unlike Tahoe which has no fast recovery, always restart slow start after loss)



▼ AIMD Behavior

- Ignoring slow start, TCP follows additive increase, multiplicative decrease (AIMD)
- Additive increase → cwnd increases linearly with time (during congestion avoidance)
- Multiplicative decrease → cwnd cut in half upon loss
- This balances fairness (all flows get share) and efficiency (links are kept busy)



▼ Visual Behavior

- Slow start : Exponential growth
- Congestion avoidance : Linear growth
- Loss event :
  Tahoe → drop to 1 MSS (reset)
  Reno → drop to half, fast recovery, then linear growth

▼ Questions

▼ Host A and B are directly connected with a 100 Mbps link. There is one TCP connection between the two hosts, and Host A is sending to Host B an enormous file over this connection. Host A can send its application data into its TCP socket at a rate as high as 120 Mbps but Host B can read out of its TCP receive buffer at a maximum rate of 50 Mbps. Describe the effect of TCP flow control.

- Link capacity = 100 Mbps, so Host A's sending rate can be at most 100 Mbps
- Host A sends data into receive buffer faster than B can remove data from the buffer
- The buffer fills up around 40 Mbps and once its full, It signals Host A to stop sending data by setting RcvWindow = 0
- Host A then stops sending until it receives a TCP segment with RcvWindow > 0
- On average, the long-term rate at which Host A sends data to Host B as part of this connection is no more than 60 Mbps

▼ Consider sending a large file from a host to another over a TCP connection that has no loss. Suppose TCP uses AIMD for its congestion control without slow start. Assuming cwnd increases by 1 MSS every time a batch of ACKs is
received and assuming approximately constant round-trip times, how long does it take for cwnd increase from 6 MSS to 12 MSS (assuming no loss events)?
• How long did it take to transmit from the 6th segment to 12th segment?

- TCP connection, no loss, no slow start

- cwnd increases by 1 MSS per RTT
- starts at cwnd = 6 MSS
- cwnd growth (takes 6 RTTs to grow from 6 to 12)

RTT	cwnd size
0	6 MSS
1	7 MSS
2	8 MSS
3	9 MSS
4	10 MSS
5	11 MSS
6	12 MSS

- Segments transmitted between cwnd = 6 and cwnd = 12.
  Segment 6 sent in RTT 0 (cwnd = 6)
  Segment 12 sent in RTT 1 (cwnd = 7)
  So it takes 1 RTT to transmit from segment 6 to 12

RTT	cwnd size	Segments sent
0	6 MSS	1 → 6
1	7 MSS	7 → 13
2	8 MSS	14 → 21
3	9 MSS	22 → 30
4	10 MSS	31 → 40
5	11 MSS	41 → 51
6	12 MSS	52 → 63

-

▼ IP Layer

▼ Network Layer Functions

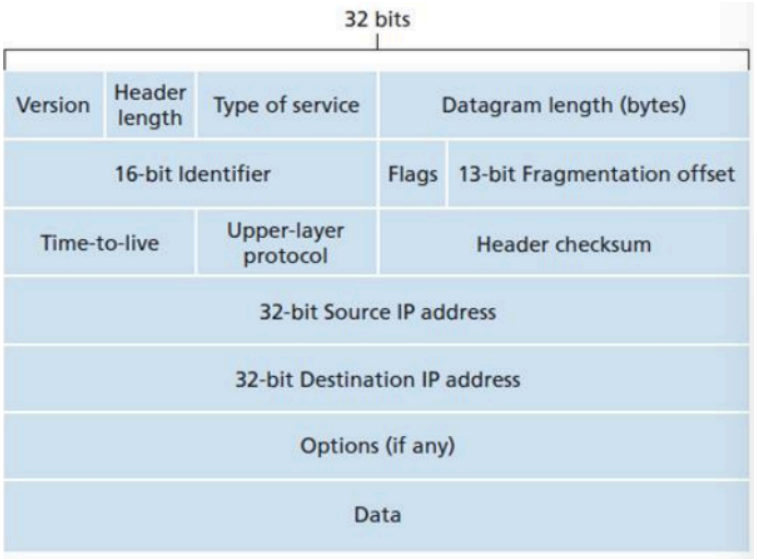
- Routers are oblivious (ignorant) of the transport layer services which support the application
  Therefore, transport layer in source host passes the segments to its network layer

- Network layer prepares datagrams by inserting IP Header which are sufficient for the routers to guide the datagrams until they reach the destination host
- Functions at host level :
  - Obtaining IP address for host
  - Converts segments to datagrams with the specified IP version
  - Error detection of IP address
- Functions at network level :
  - Assigning address to hosts
  - Forwarding datagrams
  - Switching operation withing a router
  - Routing the internet

▼ IPv4 Datagram Format

- Specified in RFC 791
- Version : 4 bit number specifying IP version 4
  Routers identify the datagram type using this field
- Header length : 4 bit field to give the size of the IP header. It is expressed as 32 bit words. When options field is (rarely) present the length exceeds 20 bytes
- Types of service : 8-bit field for providing differentiated services in networks
- Datagram length : 16-bit field for providing the length of IP header + Payload
- Identifier, Flags, Fragmentation offset : IPv4 routers fragment large datagrams based on the MTU supported by the links.
  When fragmentation is performed, these fields help in reassembled the original datagram at the destination host. Identifier is a 16-bit field, flags occupy 3 bits and fragmentation offset is 13-bit field
- TTL (Time to Live) : 8-bit field which indicates the number of hops the packet can travel without being discarded. A router decrements this field by 1 upon processing

- Protocol : 8-bit field to describe the upper layer protocol for the datagram
- Header checksum : 16-bit field for error detection. Only the header fields are used in the calculation. Whenever a datagram is forwarded or fragmented, this field is updated
- Source and Destination IP addresses : 32-bit fields each. Routers use the destination IP field for delivering the datagrams to the destination host
- Options : This field is rarely used and has been removed in IPv6
- Data : Variable length field which holds the upper layer information encapsulated into the datagram. The max size of data is 65535 bytes, however, this size rarely occurs. Datagram size in ethernet range is 46-1500 bytes only



▼ Addressing Datagram Networks

- Host/router connects with a communication link via an interface (Ethernet interface or Wireless interface)
- Each interface of a host/router is assigned an IP address which is a 32-bit number expressed in dotted decimal format
- Example
  192.168.1.1 → 11000001.10101000.00000001.00000001

▼ Subnet

- Subnet or IP network represents a network in which IP addresses have common portion starting with the MSB. The common portion is called prefix.
- Routers lie on the periphery of a subnet. Subnet can consist of switches, hubs, cables or wireless interfaces

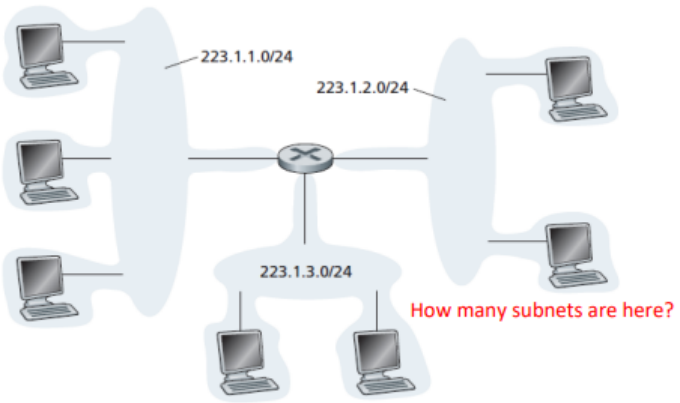
▼ Network addressing

- Network address is of the form `a.b.c.d/x`
  a, b, c, d are 8 bit decimal numbers and x is prefix
- So for example, 192.168.1.1/24 means the first 24 bits are common among the IP addresses in the subnet. Remaining 32-x bits are variable (all 0s to all 1s of length 32-x)
- This format is called *Classless Interdomain Routing* (CIDR) defined in RFC4632

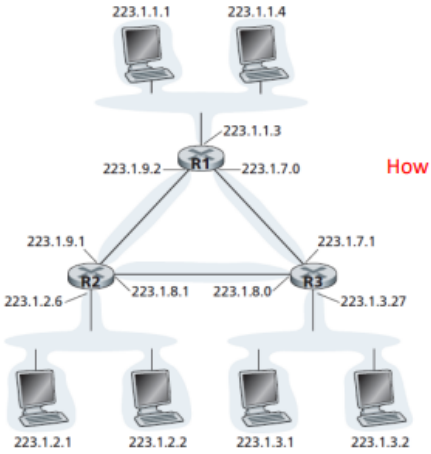
▼ Subnet Mask

- It is an alternate way of representing the prefix in 32-bit dotted decimal format
- The subnet mask is expressed as x ones and 32-x zeros
- For example, /24 is expressed as 255.255.255.0
  /26 is expressed as 255.255.255.192

▼ Questions



- Ans - 3 Subnets



- Ans - 3 Subnets

▼ 223.1.3.0/24 Subnet?

- 24 prefix bits, 8 host bits → 2^8 = 256 total addresses
- Range : `223.1.3.0` to `223.1.3.255`
- `.0` → network address
- `.255` → broadcast
- Usable : `.1` - `.254` (254 hosts)

▼ 223.1.3.64/8 Subnet?

- 28 prefix bits → 4 host bits → 2^4 = 16 total addresses
- Range : `233.1.3.64` to `233.1.3.79`
- `.64` : network
- `.79` : broadcast
- Usable : `.65` to `.78` (14 hosts)
- Subnet mask = `255.255.255.240`

▼ Subnetting a /24 into 4 subnets

- Original : `223.1.3.0/24` (256 addresses)
- Divide into 4 equal subnets → each with 64 addresses
- New prefix = 24 + 2 = `/26`
- Subnets :
  - `223.1.3.0/26` (.0 to .63)

        - `223.1.3.64/26` (.64 to .127)
        - `223.1.3.128/26` (.128 to .191)
        - `223.1.3.192/26` (.192 to .255)

▼ CIDR vs Classful Addressing

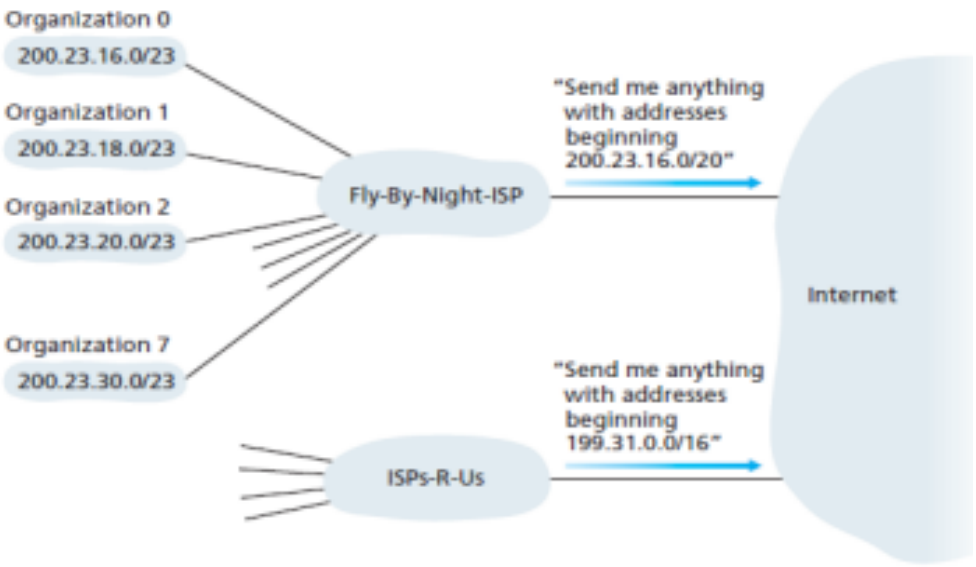
- In the old system :
  Class A : /8 (16M Hosts)
  Class B : /16 (65K Hosts)
  Class C : /24 (254 Hosts)
- Wasted many IPs → replaced by CIDR (RFC 4632)
- CIRD allows flexible allocation (prefix can be any length)
- Hierarchical distribution : ICANN → Registers → ISPs → Users

▼ Route Aggregation

- Multiple subnets with common prefix can be grouped into 1 entry
- Example : Subnets `223.1.3.64/28, 223.1.3.80/28, 223.1.3.96/28, 223.1.3.112/28`
  All share `/26` → summarized as `223.1.3.64/26`
- Reduces size of routing tables

▼ Hierarchical Addressing

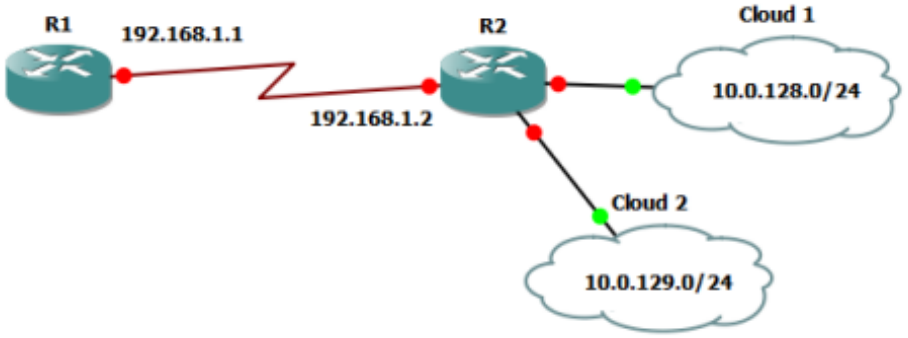
- IP allocation is hierarchical :
  ICANN/IANA → Regional Registries → ISPs → Organizations → Users



ISP's block	200.23.16.0/20	11001000 00010111 00010000 00000000
Organization 0	200.23.16.0/23	11001000 00010111 00010000 00000000
Organization 1	200.23.18.0/23	11001000 00010111 00010010 00000000
Organization 2	200.23.20.0/23	11001000 00010111 00010100 00000000
. . .	. . .	. . .
Organization 7	200.23.30.0/23	11001000 00010111 00011110 00000000

- Helps in efficient route aggregation and reduces global routing table entries

▼ Write one network address which can be used by router R2 to advertise both 10.0.128.0/24 and 10.0.129.0/24



- 10.0.128.0/24 = 00001010.00000000.10000000.00000000
  10.0.129.0/24 = 00001010.00000000.10000001.00000000

- They only differ in 9th bit of 3rd octet
  first 23 bits are common
- Common prefix = /23
  Aggregated network address = 10.0.128.0/23
  which covers 10.0.128.0 to 10.0.129.255

▼ Questions

▼ 1

P8. Consider a datagram network using 32-bit host addresses. Suppose a router has four links, numbered 0 through 3, and packets are to be forwarded to the link interfaces as follows:

Destination Address Range	Link Interface
11100000 00000000 00000000 00000000 through 11100000 00111111 11111111 11111111	0
11100000 01000000 00000000 00000000 through 11100000 01000000 11111111 11111111	1
11100000 01000001 00000000 00000000 through 11100001 01111111 11111111 11111111	2
otherwise	3

a. Provide a forwarding table that has five entries, uses longest prefix match ing, and forwards packets to the correct link interfaces.
b. Describe how your forwarding table determines the appropriate link inter face for datagrams with destination addresses:

11001000 10010001 01010001 01010101
11100001 01000000 11000011 00111100
11100001 10000000 00010001 01110111

184

- **Solution:**

a)

Prefix Match	Link Interface
11100000 00	0
11100000 01000000	1
1110000	2
11100001 1	3
otherwise	3

b) Prefix match for first address is 5th entry: link interface 3
Prefix match for second address is 3nd entry: link interface 2
Prefix match for third address is 4th entry: link interface 3

▼ 2

- Consider a router that interconnects three subnets: Subnet 1, Subnet 2, and Subnet 3. Suppose all of the interfaces in each of these three subnets are required to have the prefix 223.1.17/24. Also suppose that Subnet 1 is required to support at least 60 interfaces, Subnet 2 is to support at least 90 interfaces, and Subnet 3 is to support at least 12 interfaces. Provide three network addresses (of the form a.b.c.d/x) that satisfy these constraints.

To interconnect three subnets (Subnet 1, Subnet 2, and Subnet 3) with the prefix `223.1.17/24`, and given the requirements for supporting at least 60, 90, and 12 interfaces respectively, we need to perform subnetting.

1. Prefix Analysis and Requirements:

- **Subnet 1** requires at least 60 addresses.
- **Subnet 2** requires at least 90 addresses.
- **Subnet 3** requires at least 12 addresses.
- The original block `223.1.17.0/24` gives 256 addresses. We need to divide this into smaller blocks.

2. Determining Subnet Sizes:

- For Subnet 2 (largest), we need at least 90 addresses. The smallest power of 2 greater than or equal to 90 is 128, so we will use a `/25` subnet, which provides 128 addresses.
- For Subnet 1, we need at least 60 addresses. The smallest power of 2 greater than or equal to 60 is 64, so we will use a `/26` subnet, which provides 64 addresses.
- For Subnet 3, we need at least 12 addresses. The smallest power of 2 greater than or equal to 12 is 16, so we will use a `/28` subnet, which provides 16 addresses.

3. Assigning Subnets:

Now we can allocate addresses from the `223.1.17.0/24` block:

- **Subnet 2** (90 interfaces):
  - Network address: `223.1.17.0/25`
  - Address range: `223.1.17.0` to `223.1.17.127`
- **Subnet 1** (60 interfaces):
  - Network address: `223.1.17.128/26`
  - Address range: `223.1.17.128` to `223.1.17.191`
- **Subnet 3** (12 interfaces):
  - Network address: `223.1.17.192/28`
  - Address range: `223.1.17.192` to `223.1.17.207`

▼ 3

• Consider the topology shown in Figure 4.20. Denote the three subnets with hosts (starting clockwise at 12:00) as Networks A, B, and C. Denote the subnets without hosts as Networks D, E, and F.

a. Assign network addresses to each of these six subnets, with the following constraints: All addresses must be allocated from 214.97.254/23; Subnet A should have enough addresses to support 250 interfaces; Subnet B should have enough addresses to support 120 interfaces; and Subnet C should have enough addresses to support 120 interfaces. Of course, subnets D, E and F should each be able to support two interfaces. For each subnet, the assignment should take the form a.b.c.d/x or a.b.c.d/x – e.f.g.h/y.

b. Using your answer to part (a), provide the forwarding tables (using long est prefix matching) for each of the three routers.

From 214.97.254/23, possible assignments are

a) Subnet A: 214.97.255/24 (256 addresses)
Subnet B: 214.97.254.0/25 - 214.97.254.0/29 (128-8 = 120 addresses)
Subnet C: 214.97.254.128/25 (128 addresses)

Subnet D: 214.97.254.0/31 (2 addresses)
Subnet E: 214.97.254.2/31 (2 addresses)
Subnet F: 214.97.254.4/30 (4 addresses)

b) To simplify the solution, assume that no datagrams have router interfaces as ultimate destinations. Also, label D, E, F for the upper-right, bottom, and upper-left interior subnets, respectively.

Router 1

Longest Prefix Match	Outgoing Interface
11010110 01100001 11111111	Subnet A
11010110 01100001 11111110 0000000	Subnet D
11010110 01100001 11111110 000001	Subnet F

Router 2

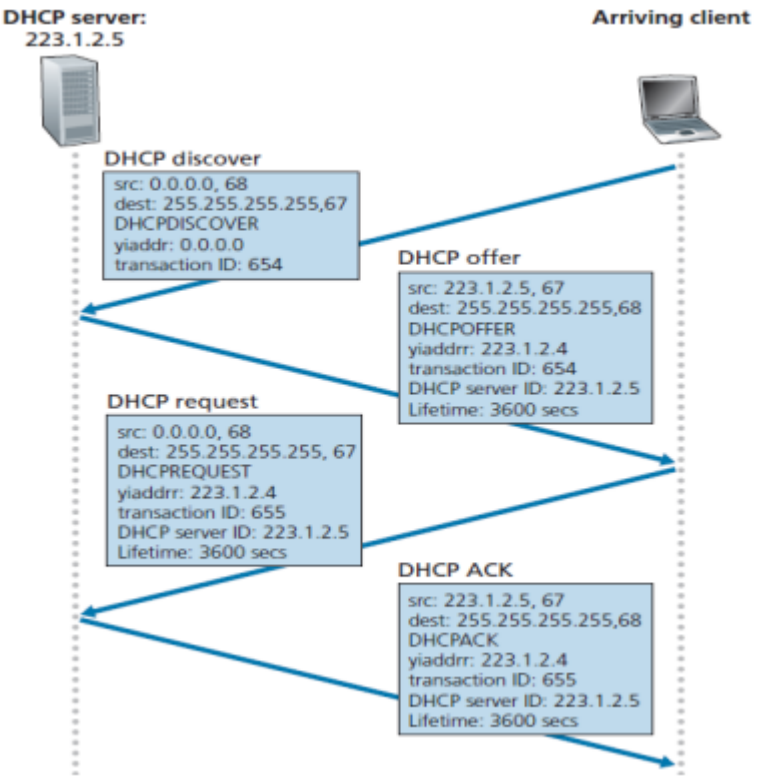
Longest Prefix Match	Outgoing Interface
11010110 01100001 11111111 0000000	Subnet D
11010110 01100001 11111110 0	Subnet B
11010110 01100001 11111110 0000001	Subnet E

Router 3

Longest Prefix Match	Outgoing Interface
11010110 01100001 11111111 000001	Subnet F
11010110 01100001 11111110 0000001	Subnet E
11010110 01100001 11111110 1	Subnet C

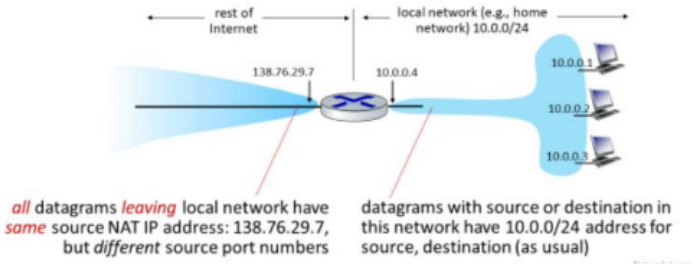
▼ IPv4 DHCP (Dynamic Host Configuration Protocol)

- Once an organization obtains a block of addresses, it can either manually assign IP addresses to various interfaces or alternatively use the DHCP to automate IP assignment
- DHCP : Dynamic Host Configuration Protocol
  - client server model for obtaining IP address
  - Server listens for UDP messages on port 67
- Process
  i. Hosts sends a DHCP discovery message
  ii. DHCP servers replies with DHCP offer messages
  iii. Host replies with DHCP request
  iv. DHCP server acknowledges with assigned IP address



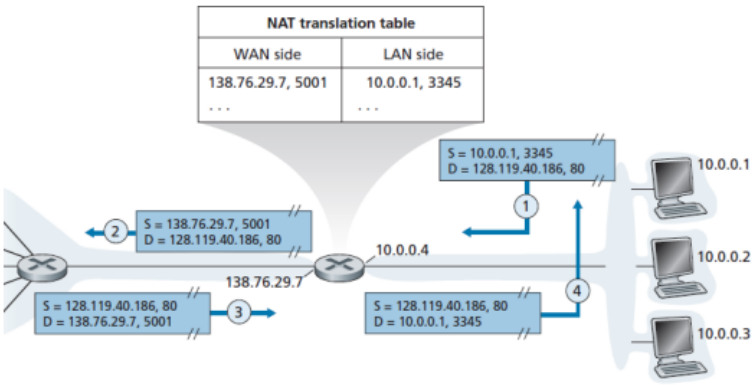
▼ Network Address Translation

- NAT or Network address translation solves shortage of public IP addresses
- All devices in a local network share just one IPv4 address as far as outside world is concerned



- IP addresses allotted by an ISP to a consumer could be insufficient. in such cases, private IP networks are created and public IPs that are allotted by ISPs are shared
- Private IP address can be assigned manually or using DHCP
- Private hosts can access public internet using NAT
- NAT maps the private IP addresses to public IP addresses

- NAT table maintains such mappings



- Criticism faced :
  - Uses port numbers for host distinction
  - Routers should perform only level 3 functions
  - Breaks end-to-end principle
  - Slows down IPv6 adoption

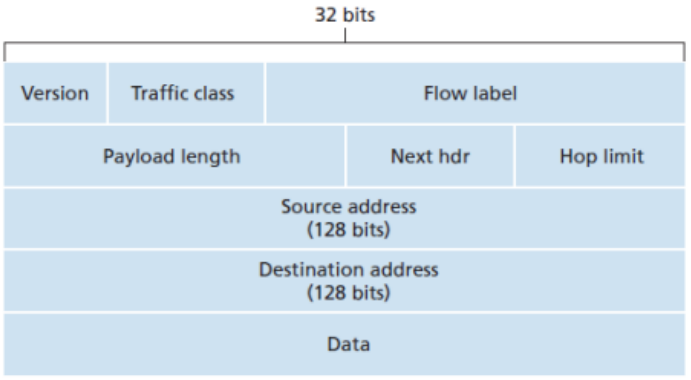
▼ IPv6 Datagram Format

- Proposed by IETF
  (initiated in 1998 but implemented in 6th June 2012)
- Provides $3.4 * 10^{38}$ IP addresses of 128 bits each
- Currently implemented by new ISPs and carries
- About 22% of IP traffic is because of IPv6 in 2018
  About 40% of IP traffic is because of IPv6 in 2025
- Fixed 40 byte header compared to 20 byte (variable length) in IPv4. Simpler
- It is more robust to IP spoofing and attacks
- Supports unicast, multicast and anycast

▼ Datagram Format

- Version : 4-bit field identifies the IP version number
- Traffic class: 8-bit traffic class field, like the TOS field in IPv4. Used to give priority to certain datagrams within a flow, or it can be used to give priority to datagrams from certain applications.

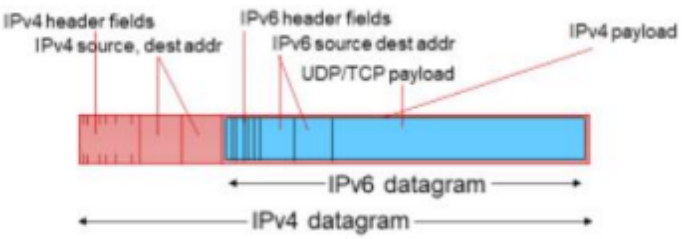
- Flow label: 20-bit field is used to identify a flow of datagrams.
- Payload length: 16-bit value is treated as an unsigned integer giving the number of bytes in the IPv6 datagram following the fixed length, 40-byte datagram header.

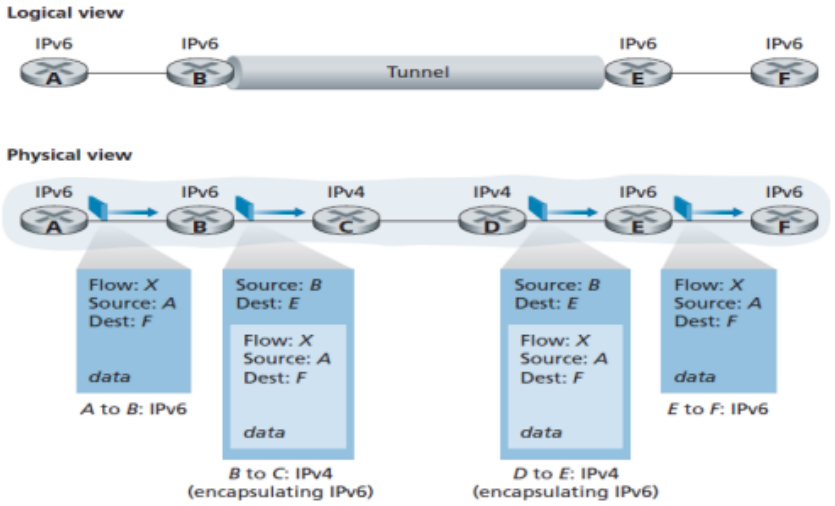


- Compared to IPv4.... IPv6 doesn't have checksum, fragmentation/reassembly/options

▼ IPv6 Tunneling

- Not all routers can be upgraded simultaneously
  - no "flag days"
  - how can network operate when IPv4 and IPv6 routers are mixed
- We use a concept called tunneling (RFC 4213) which encapsulated IPv6 packets inside IPv4 during migration
- Helps transition as not all devices/networks support IPv6 natively




Logical view
IPv6
IPv6
IPv6
IPv6
A
B
Tunnel
E
F
Physical view
IPv6
IPv6
IPv4
IPv4
IPv6
IPv6
A
B
C
D
E
F
Flow: X
Source: A
Dest: F
data
A to B: IPv6
Source: B
Dest: E
Flow: X
Source: A
Dest: F
data
B to C: IPv4
(encapsulating IPv6)
Source: B
Dest: E
Flow: X
Source: A
Dest: F
data
D to E: IPv4
(encapsulating IPv6)
Flow: X
Source: A
Dest: F
data
E to F: IPv6