Module 7
Transport Layer

Dr. Natarajan Meghanathan
Professor of Computer Science
Jackson State University, Jackson, MS 39232
E-mail: natarajan.meghanathan@jsums.edu

Module 7 Topics

- 7.1 UDP vs. TCP
- 7.2 UDP Header
- 7.3 TCP Header and Connection Establishment
- 7.4 TCP Flow Control and Congestion Control

Need for End-to-End Transport Protocols

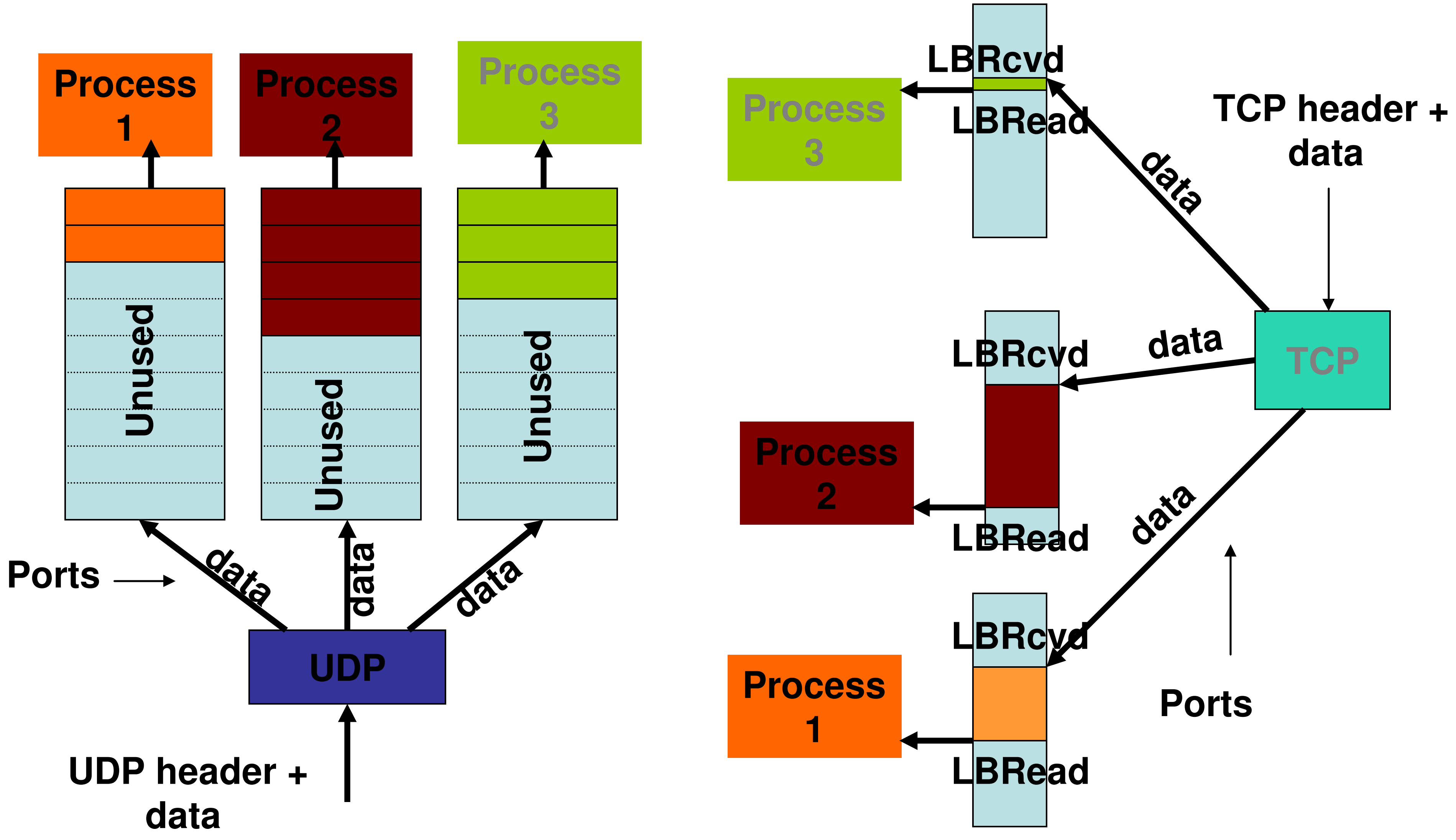
- Though IP can transfer datagrams from a source computer to a destination computer across one or more networks, it cannot distinguish between packets of different application programs running on the two computers.
- In computers where multiple application programs can run concurrently, how to identify the actual end points, the two application programs, which want to communicate by exchanging packets over the internet?
- Transport layer protocols operate above the network layer protocols and allow individual application programs to be identified as the end-points of communication.
- The TCP/IP protocol suite provides two transport protocols: User Datagram Protocol (UDP) and Transmission Control Protocol (TCP).

Ports

- Ports are used for a process running in one host to identify a process running in the destination host.
- Why not process ids for ports? Ports can be assigned the process ids only when the whole internet is a “closed” distributed system in which a single OS runs all the hosts and assigns each process a unique id.
- This is not possible in an internet where the participating computers may be run with different OS. For a given application process (say time server), the id of the process assigned in one system may not match with another.
- With ports, we want to provide an internet-wide unique abstraction for the application processes. For example, the time server process is referred using port number 13 irrespective of the computer and the OS in which the process is run.
- A port is merely an abstraction. It may be implemented as a Buffer (storing bytes) by TCP or as a message queue by UDP.
- Port numbers below 1024 are designated as well-known ports and are assigned to a fixed application program. For example, port number 21 for FTP, 22 for SSH, 23 for telnet, 24 for SMTP, 53 for DNS, 80 for HTTP, etc.
- For user-defined application programs, we need to use define port numbers greater than or equal to 1024.

Ports

A port is merely an abstraction. It may be implemented as a Buffer (storing bytes) by TCP or as a message queue by UDP.



7.1 TCP vs. UDP

Differences between UDP and TCP

- UDP is connectionless; TCP is connection-oriented
  - Connectionless: the source and destination processes do not communicate to know each other before starting to exchange data packets
  - Connection-oriented: the source and destination processes communicate to learn about the resources available at each side and set up initial values for the parameters for reliable, in-order communication.

- TCP – session-based and full-duplex; UDP – unidirectional
  - TCP connections are typically run as part of a session between a source and destination machine. A TCP connection can permit packets to be sent in both the directions simultaneously.
  - Each process/machine can communicate to any other process/machine whenever it wants to. So, there is no such concept of simultaneous communication or session.

Differences between UDP and TCP

- UDP is message-based and TCP is byte-stream based
  - UDP just packages whatever the higher-layer application wants to send as a segment and sends down to the IP layer.
    - Message boundaries are preserved. The receiving application sees reads as messages from the lower transport layer.
  - TCP: The data received from the higher-layer application is buffered at the transport layer (at the byte-level) and the bytes are packaged into segments, depending on the MTU of the underlying network.
    - Message boundaries are not preserved. Receiving application may not read the same number of bytes in one read operation that were sent as one segment.

Differences between UDP and TCP

- UDP is best-effort service based and TCP provides reliable, in-order delivery.
  - UDP does not bother about keeping track of whether the message sent from one end host (source) has reached the other end host (destination).
    - UDP runs on the top of IP that also provides only best-effort service.
    - If reliability and in-order delivery are needed, the higher-layer application has to take care of that.
  - The source-side TCP buffers the segments sent until it receives an ACK from the destination. Segments are retransmitted, if not acknowledged. The destination-side TCP buffers the segments received out-of-order and delivers only the bytes in-order to the higher-layer application.

Differences between UDP and TCP

- <u>UDP is preferred for real-time applications; TCP is preferred for delay-tolerant applications.</u>
  - Real-time applications (like video streaming) are delay-sensitive and they need the packets to be delivered within a certain time; the loss of one or fewer packets may be OK and could be handled with redundant info present in adjacent packets.
  - TCP is preferred for delay-tolerant applications for which every byte needs to be received in the same order they were sent from the application at the source side.
- <u>UDP is used for short-duration communication; TCP is preferred for lengthy and critical communications where reliability is important.</u>
  - For short communication (like DHCP) that involves only one or few message exchanges, it would be too much of an overhead to go through a connection-establishment process before sending any actual data packets.
  - For lengthy and critical communications (like file download, e-transfer), it would be just a one-time delay to go through a connection establishment process for reliable communication.

Differences between UDP and TCP

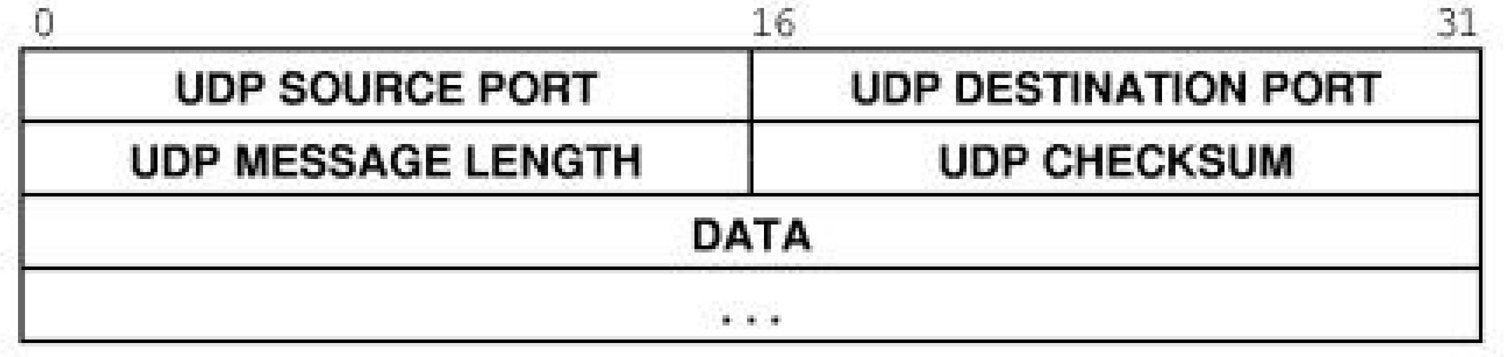
- <u>UDP is used for unicast, multicast and broadcast; TCP for unicast only</u>
  - The semantics of TCP is such that it cannot be used for multicast and broadcast communications.
    - Difficult to make sure that every message sent from the source has reached all the intended destinations.
  - Multicast and broadcast communication are typically done using UDP as the transport layer protocol.

- <u>UDP: Datagram fragmentation is possible in the source network itself; TCP – no datagram fragmentation possible.</u>
  - Since the higher-layer application decides the message size, if the underlying network cannot handle the message, the IP protocol would have to fragment the data before sending.
  - The Internet layer protocol (IP) at the destination has to keep track of the fragments and reassemble them. As IP provides only best effort service, there is no guarantee that every message sent is received. Hence, ***UDP messages are typically small*** so that fragmentation is not needed

7.2 User Datagram Protocol (UDP)

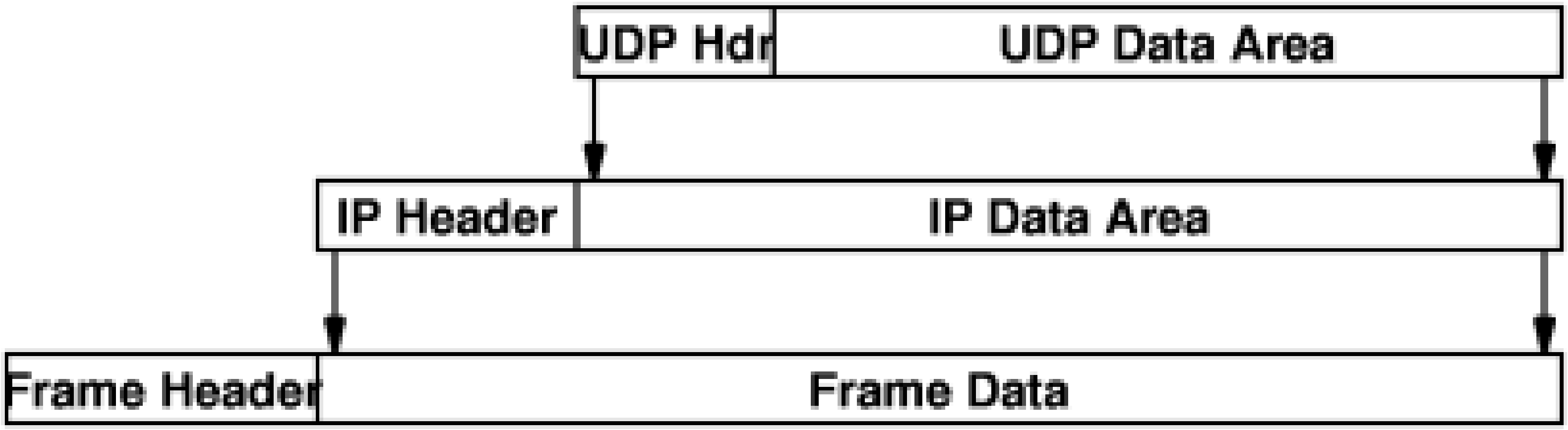
UDP Datagram Format

- UDP SOURCE PORT and UDP DESTINATION PORT contain respectively the port numbers of the sending and receiving processes/ applications.
- UDP message length specifies the total size of the UDP DATA in bytes.



- UDP computes a checksum of the following fields: UDP SOURCE PORT, UDP DESTINATION PORT, UDP MESSAGE LENGTH, UDP DATA and IP SOURCE ADDRESS, IP DESTINATION ADDRESS and IP H.LEN fields (the last three fields are called the pseudo header fields – used to make sure the communication is between the appropriate source and destination machines).

UDP Encapsulation



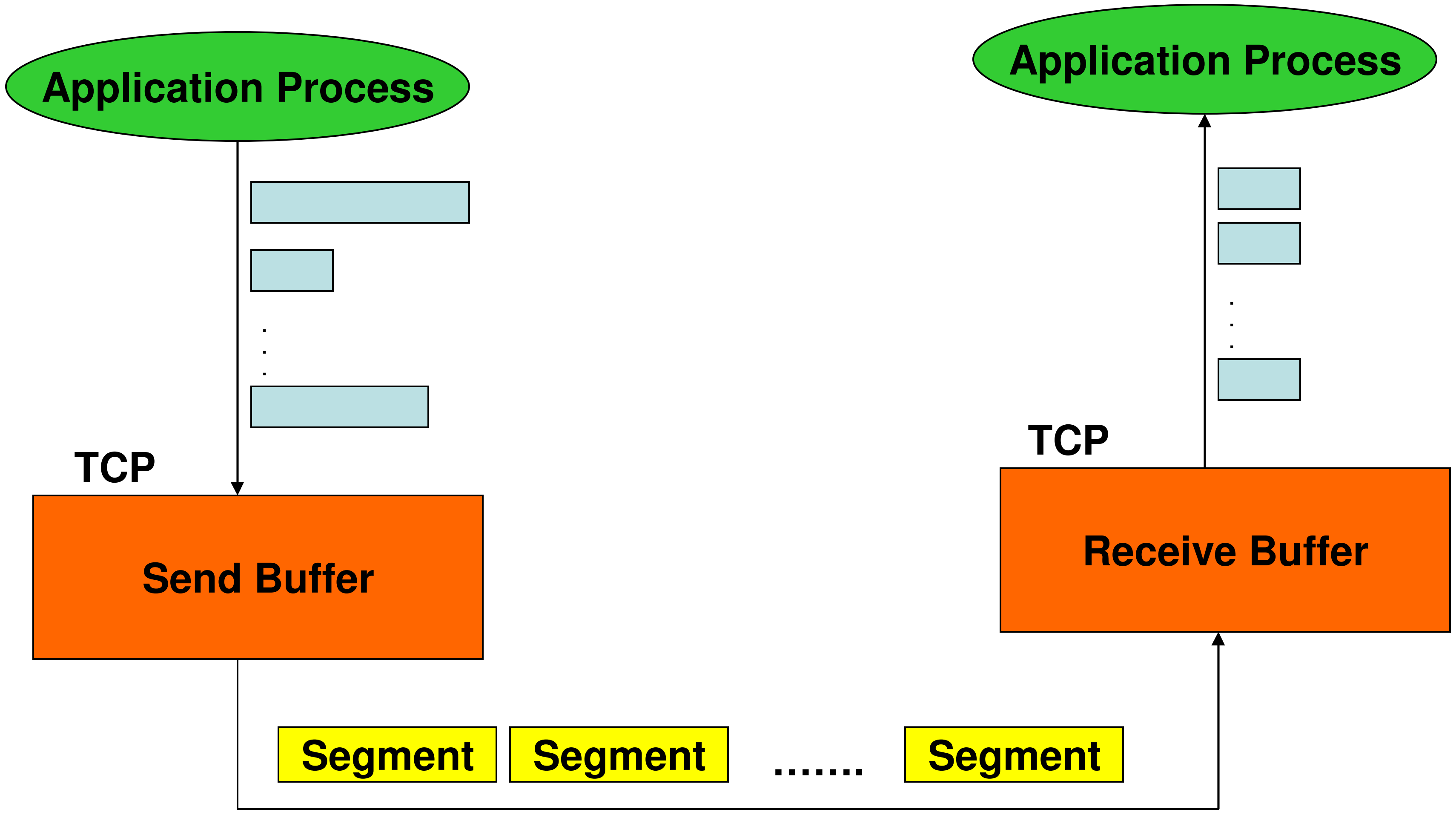
7.3 Transmission Control Protocol (TCP)

TCP Header, Connection Establishment

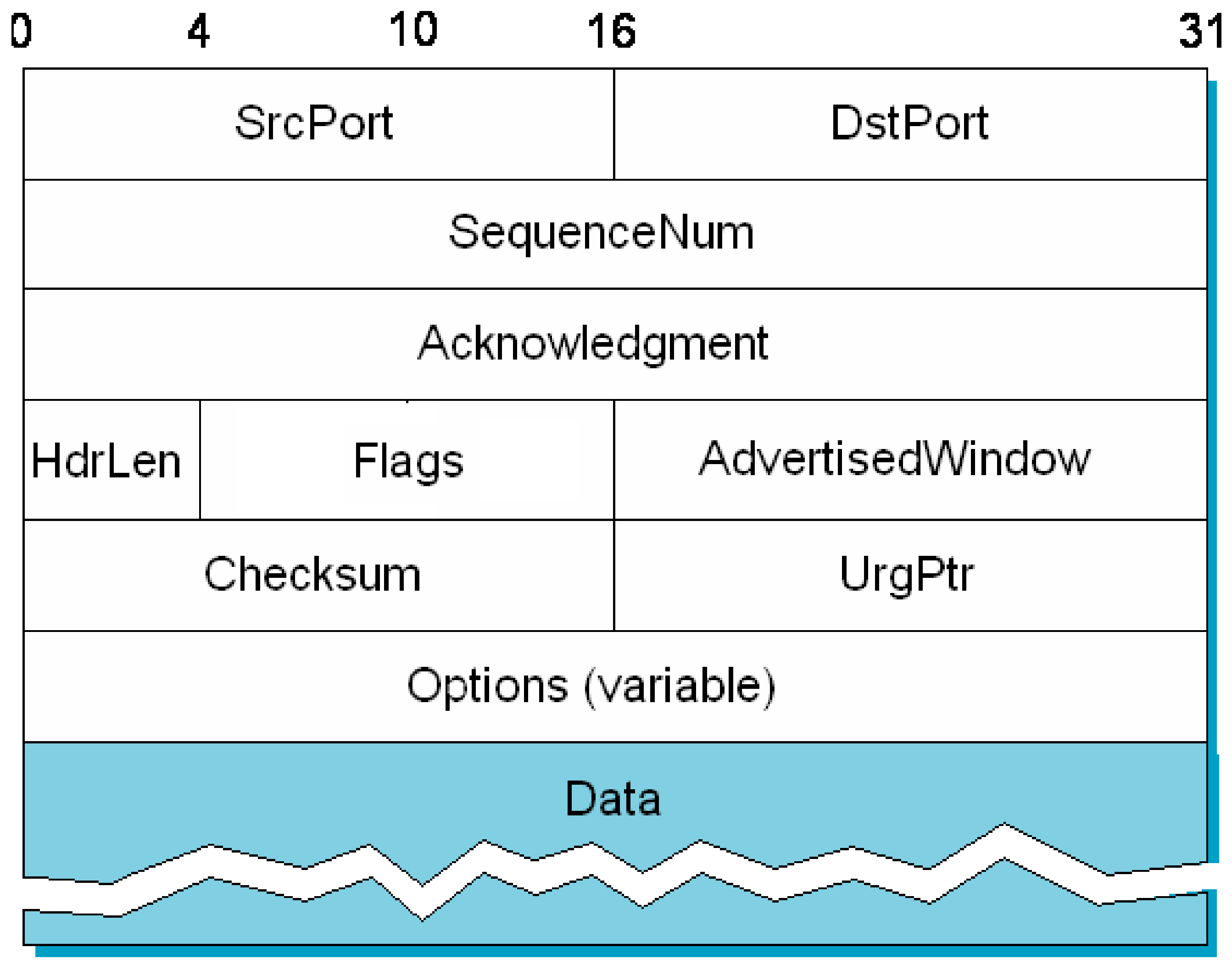
TCP: Byte Stream Management

- TCP is a byte-oriented protocol: the sending process writes bytes into a TCP connection and the receiving process reads bytes out of the connection.
- Though TCP offers “byte-stream” service to application processes, TCP does not transmit data over the internet in the form of bytes.
- A single TCP connection supports byte streams flowing in both directions.
- TCP on the source host buffers the bytes written by the sending process until the bytes can be filled in to form a reasonably sized message (called TCP segment) and then sends the segment to its peer TCP running at the destination host.
- The TCP at the destination host, on receiving the TCP segment, empties the contents of the segment into a receive buffer, which is read from (extracted) by the receiving process at its leisure.
- The receiving process does not read data in the same size of pieces that were inserted into the connection by the sending process. The fundamental unit of data that is common to both the sending and receiving host processes is byte and hence TCP is called a byte-stream oriented protocol.

TCP: Byte Stream Management



TCP Header Format



TCP Header Format

- Since TCP is a byte-oriented protocol, each byte of data has a sequence number; the sequenceNum field contains the sequence number for the first byte of data carried in a segment.
- The Acknowledgement and AdvertisedWindow (used to indicate the buffer space available in bytes) fields are filled in the ACK packet sent to acknowledge the receipt of a data packet. These fields are involved in the sliding window algorithm.
- The checksum is computed over the TCP header, TCP data, pseudo header-the source and destination addresses and length fields from the IP header.
- The HdrLen field indicates the length of the TCP header in 32-bit words.

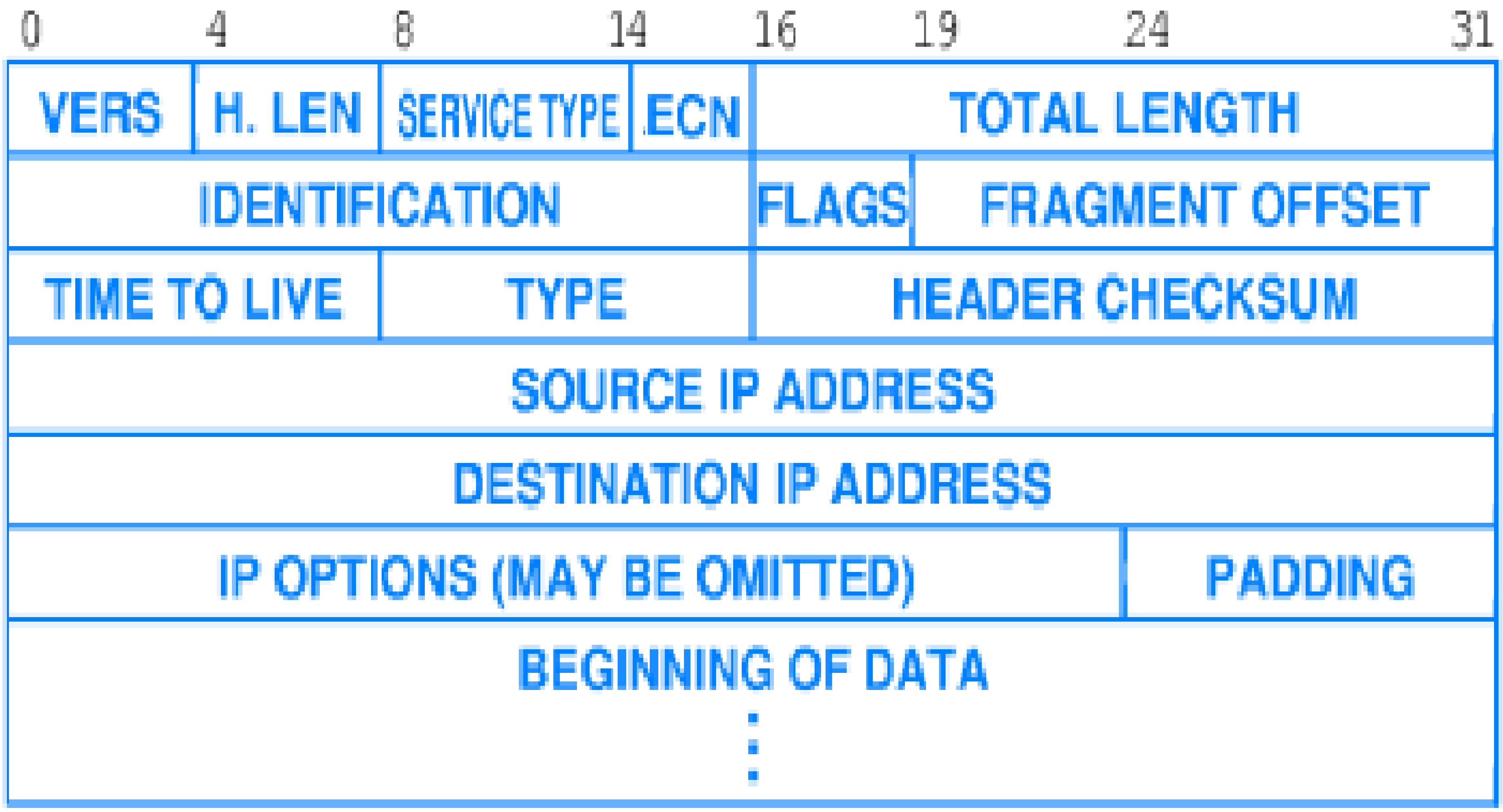
TCP Options

- The format of the options is similar to the one in the IP header.
  - 8-bit Options Type; 8-bit Options Length and (variable length) Options Data
- Possible Options
  - Window scaling factor: To indicate Advertised Window sizes that are larger than 2^{16}-1 bytes, the advertising end host can indicate a value ≤ in the Advertised Window and include a corresponding scaling factor in the Options field.
    - For example, to indicate an Advertised Window of size 80,000 bytes, the advertising host can advertise 20,000 in the Advertise Window and set the Data portion of the Window scaling factor options field to 4.
  - Maximum Segment Size (MSS): To indicate the MTU of the underlying network to the opposite end.
    - MSS = MTU – [Max. IP header Size + Max. TCP header Size]
  - Timestamp: Used for protection against wrapped around sequence numbers.
    - For each value of the timestamp field, there can be 2^{32} different sequence numbers for the bytes.

TCP Flags

- Flags – 12 bits; the first four bits are not used.
- The next two bits (ECE and CWR flags) are used for Explicit Congestion Notification-related purposes.
  - The end hosts sets the ECE flag in the ACK packets of the 3-way handshake to indicate their support for ECN at the transport layer.
- The last six bits/flags fields are SYN, FIN, RESET, PUSH, URG and ACK.
  - The SYN flag is used to establish a TCP connection.
  - The FIN flag is used to teardown a connection.
  - The RESET flag is used by the receiver to abort a connection.
  - The PUSH flag is set by the sender in order to indicate the receiver that the segment was sent as a result of invoking the push operation.
  - The URG flag signifies that the segment contains urgent data. The UrgPtr field indicates where the non-urgent data contained in the current segment begins. The urgent data is contained in the front portion of the segment data body.
  - The ACK flag is set when the receiver of the segment should pay attention to the Acknowledgement field.

IP Header Format (v4)



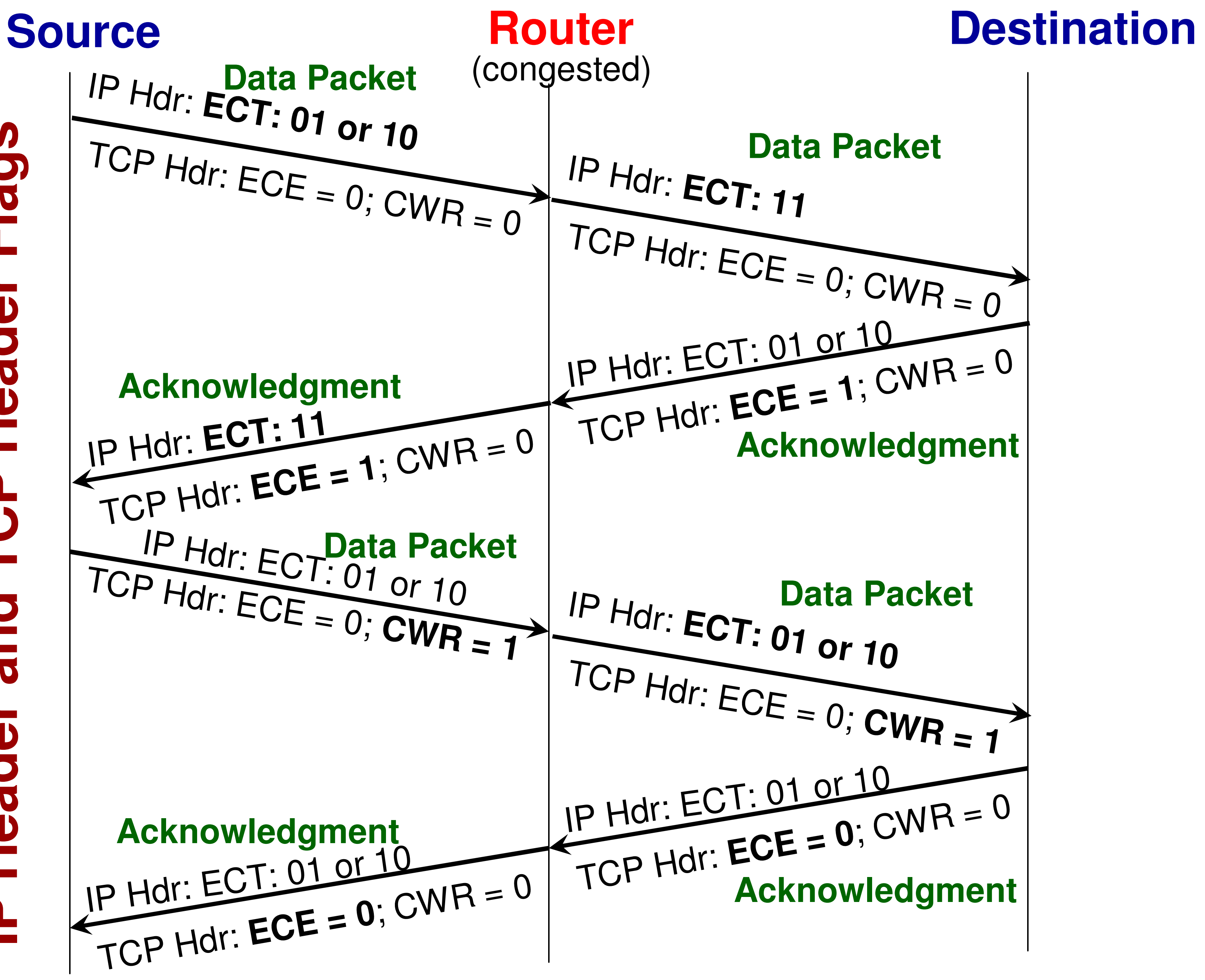
IP Header Format

- ECN bits (2 bits) for Explicit Congestion Notification
  - 2 bit-combinations
    - 0 0 (Non-ECT – EC not supported at transport layer)
    - 0 1 or 1 0 (ECT–EC supported at the transport layer)
    - 1 1 (CE: Congestion Experienced)
  - If the end hosts can support ECN, the source sets either 0 1 or 1 0 in the IP header of the datagrams sent.
  - A router experiencing congestion, (instead of dropping the packet right away) will overwrite the ECT bits with the CE bits, letting the destination know that the datagram was forwarded in spite of the impending congestion.
  - The destination has to now echo this EC notification in the ACK packet sent to the source (through the ECE flag in the TCP header)

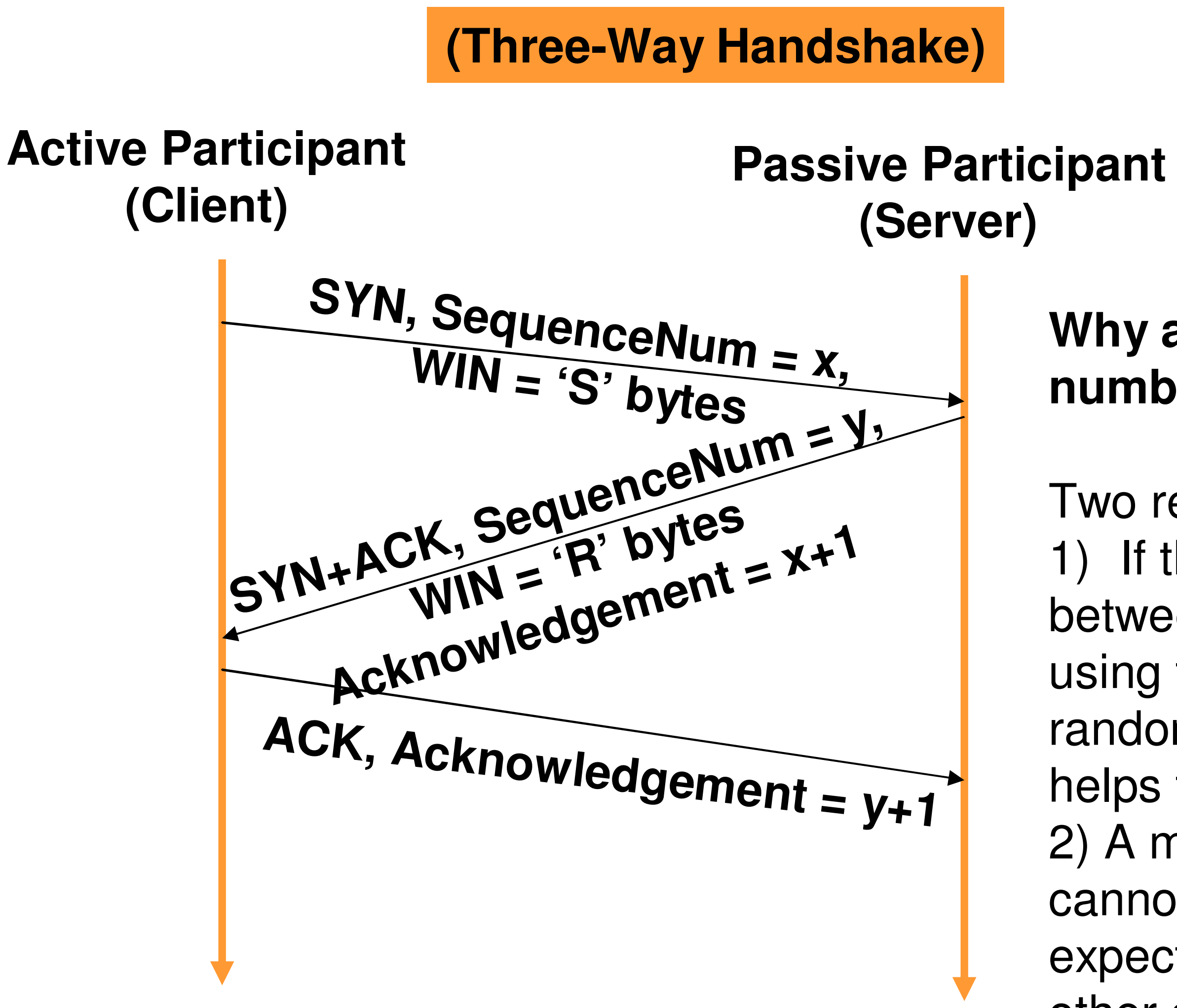
Explicit Congestion Notification

- The idea is that if a router senses an impending congestion in its queue (mechanisms are available to make this prediction), it can notify the end hosts to slow down rather than dropping their packets right away.
- The router notifies the destination end host through the ECT-flags in the IP header.
- The destination notifies the source by setting the ECE (EC Echo) flag in the TCP header for the ACK packets until it sees a data packet with the CWR set.
- When the source slows down to send the subsequent segments, it sets the CWR (Congestion Window Reduced) flag in the TCP header to indicate that it has slowed down.
- The CWR flag is an indication to the destination not to set the ECE flag for awhile
  - If the router continues to set the ECT flags in the IP header in spite of the source setting the CWR flag, the destination again sets the ECE flag in the TCP ACK, triggering the source to further slow down.
- The intermediate routers stop setting the ECT flags in the IP header after they see the probability of an impending congestion is below a threshold.

Congestion Notification using IP Header and TCP Header Flags


Source
Router
(congested)
Destination
Data Packet
IP Hdr: **ECT: 01 or 10**
TCP Hdr: ECE = 0; CWR = 0
Data Packet
IP Hdr: **ECT: 11**
TCP Hdr: ECE = 0; CWR = 0
IP Hdr: ECT: 01 or 10
TCP Hdr: **ECE = 1**; CWR = 0
Acknowledgment
Acknowledgment
IP Hdr: **ECT: 11**
TCP Hdr: **ECE = 1**; CWR = 0
Data Packet
IP Hdr: ECT: 01 or 10
TCP Hdr: ECE = 0; **CWR = 1**
Data Packet
IP Hdr: **ECT: 01 or 10**
TCP Hdr: ECE = 0; **CWR = 1**
IP Hdr: ECT: 01 or 10
TCP Hdr: **ECE = 0**; CWR = 0
Acknowledgment
Acknowledgment
IP Hdr: ECT: 01 or 10
TCP Hdr: **ECE = 0**; CWR = 0

TCP Connection Establishment



(Three-Way Handshake)
Active Participant (Client)
Passive Participant (Server)
SYN, SequenceNum = x, WIN = 'S' bytes
SYN+ACK, SequenceNum = y, WIN = 'R' bytes Acknowledgement = x+1
ACK, Acknowledgement = y+1

Why a random starting sequence number from each side?

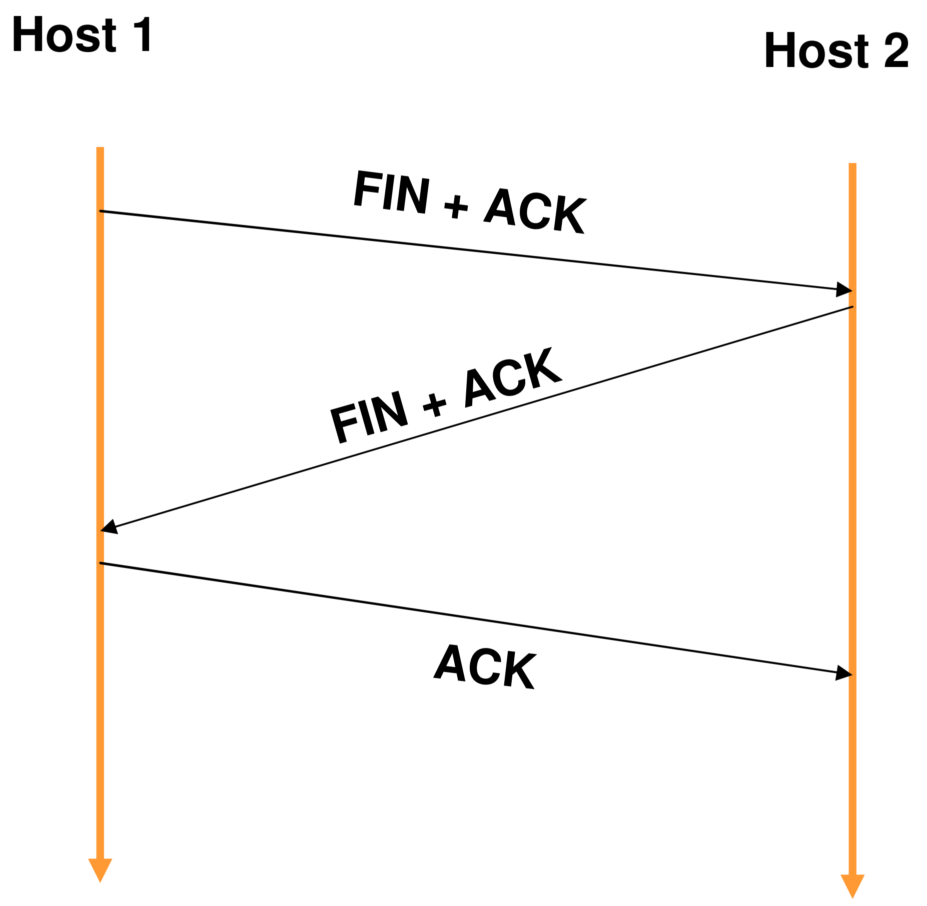
Two reasons:

1) If there are multiple sessions between the source-destination using the same port numbers, a random starting sequence number helps to distinguish packets

2) A man-in-the-middle attacker cannot easily guess the next expected sequence number at the other end and hijack a TCP session

TCP Connection Termination

(Three-Way Handshake)



Segment Triggering Techniques

- Maximum Segment Size (MSS) – the maximum size of the data that can be transmitted by the TCP protocol at the sending host.
- MSS = (MTU of the underlying network to which the sending host is attached) – (Size of the IP header + Size of the TCP header)

When to send a segment from the sending host to a receiving host for a given pair of application processes?

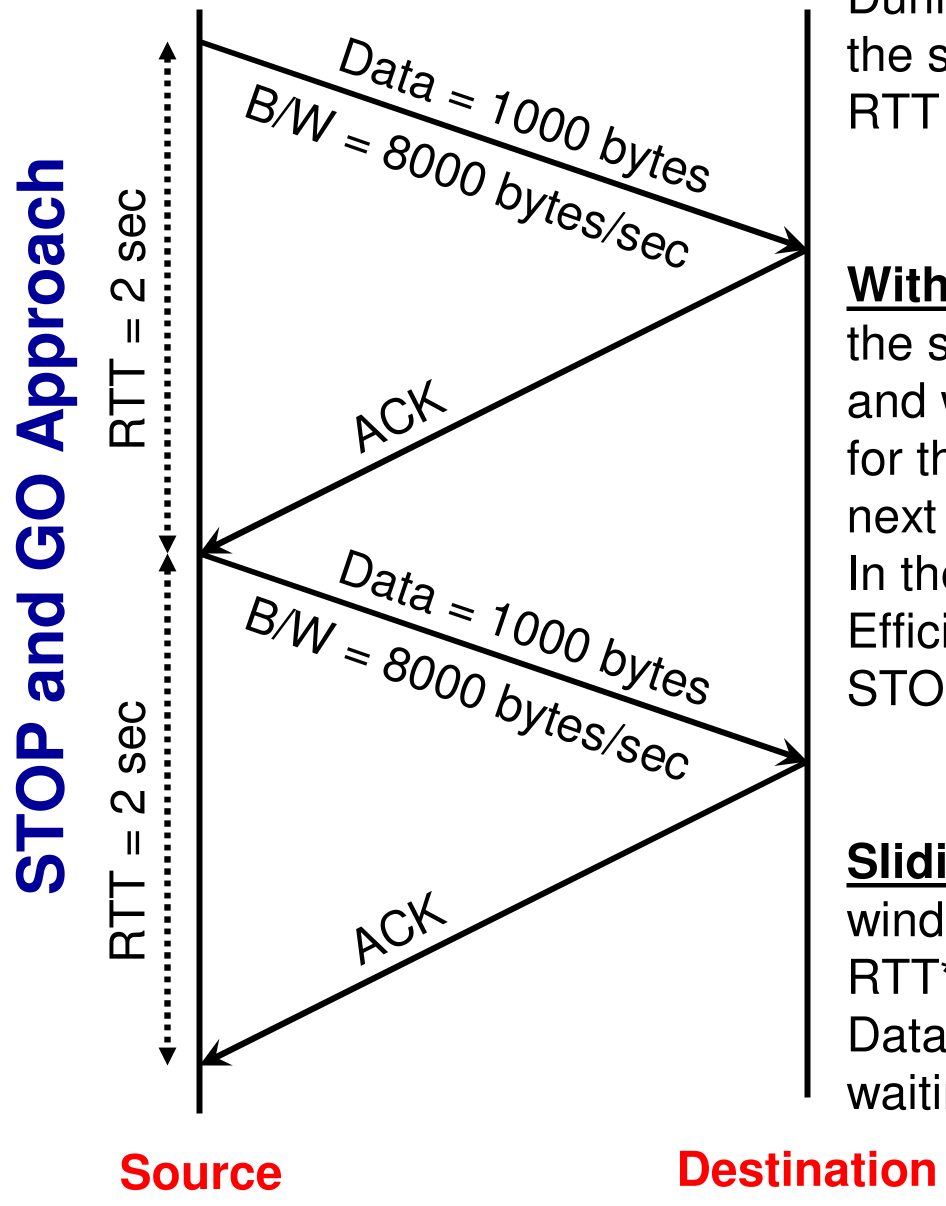
- When bytes totaling up to MSS have accumulated at the send buffer for the process.
- Periodically using a timer to trigger after a timeout.
- When the sending process wants to indicate that it wants to send whatever has accumulated in the buffer so far and wants the receiver to process them right away, then it invokes a PUSH operation. Whatever the amount of non-sent data (of course size <= MSS) that has accumulated at the Send buffer is used to form a segment and transmitted to the receiving process.

7.4 TCP Flow Control and Congestion Control

Flow Control

- Flow Control is the mechanism of adjusting the sending rate according to the resources available at the destination.
- During the TCP connection establishment process, the source and destination learn about the resources (i.e., the buffer space) that each side can allocate for the connection and then periodically update the available buffer space through the ‘Advertised Window Size’ field in the TCP header of the Acknowledgment and data packets.
- The Sliding Window algorithm is used to dynamically adjust the number of outstanding packets (packets that have been sent but not yet acknowledged).
- **Classic TCP:** Acknowledgments are sent only for the bytes that have arrived in-order so far. The application at the receiver side can read only the bytes received in-order so far.
- The bytes received out-of-order are simply buffered at the receiver side. When the missing bytes come, a cumulative ACK indicating the sequence number of the last byte received in-order is sent.

Motivation for Sliding Window



During the RTT of 2 seconds,
the source could have sent
RTT * B/W = 2 sec * 8000 bytes/sec
= 16,000 bytes of data

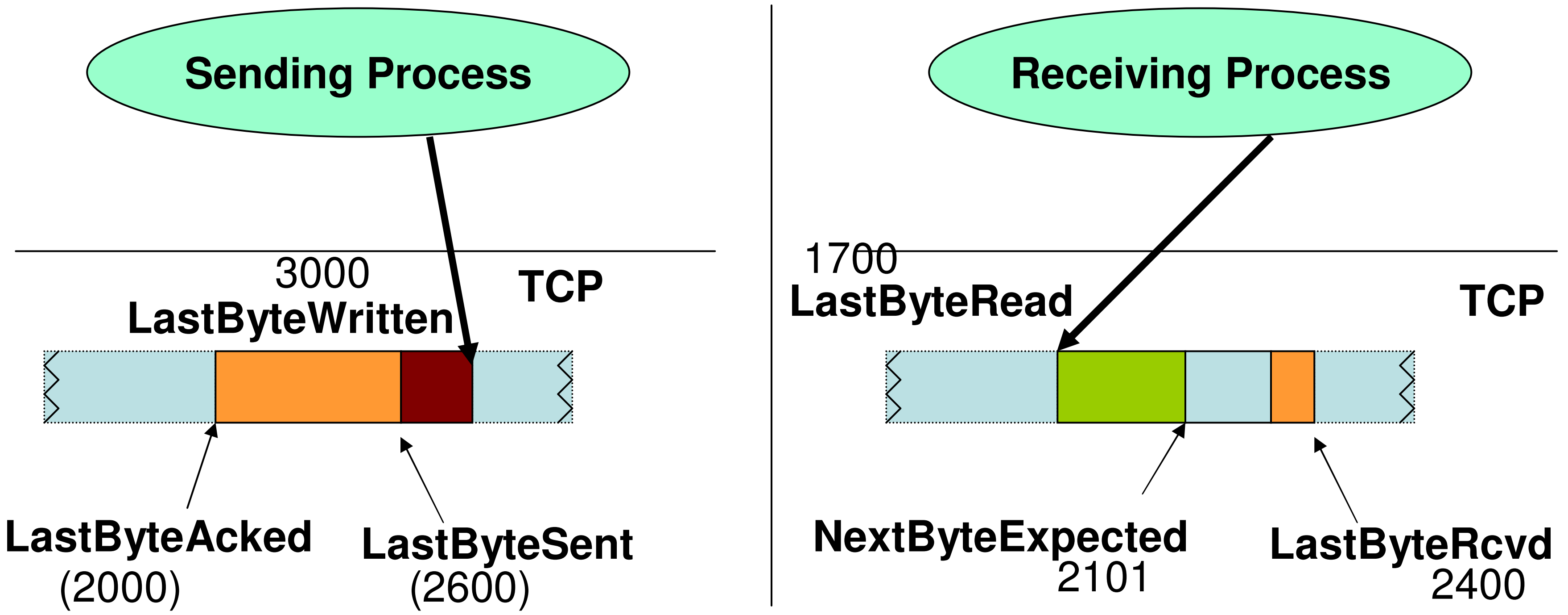
With a STOP and GO approach,
the source just sends one data packet and waits for an acknowledgment (ACK) for that packet before sending the next one.
In the example shown here, the Efficiency of channel utilization for the STOP and GO approach is
1000 /16000 = 1/16 < 10%

Sliding Window Approach: If the advertised window allows, we could send RTT*Bandwidth amount of Data (i.e., keep the pipe full) before waiting for an ACK

Motivation for Sliding Window

- Example: Assume that we use the stop and go approach (send only one data packet and wait for an ACK before sending the next data packet)
- Let the bandwidth of the underlying network be 8000 bytes/sec and the RTT (round trip time from source to destination networks) be 2 seconds.
- If the data packet size is 1000 bytes, then we have basically sent only 1000 bytes/sec over a period of 2 seconds if we use the Stop and go approach. Whereas, we could have sent a total of 16,000 bytes over a period of 2 seconds. The % efficiency of link utilization is only 1/16th.
- <u>If the Advertised Window can allow</u>, we should try to "keep the pipe full" by sending the RTT*Bandwidth amount of data (a window of data packets) before we expect the first acknowledgment.
- The data packets (bytes) that have been sent and not yet acknowledged are called outstanding packets (bytes).

Flow Control: Example 1



Assume Receiver Buffer Size = 1,500 bytes

Advertised Window = Receiver Buffer Size – (Last Byte Rcvd – Last Byte Read)

Advertised Window = 1,500 – (2400 – 1700) = 800 bytes

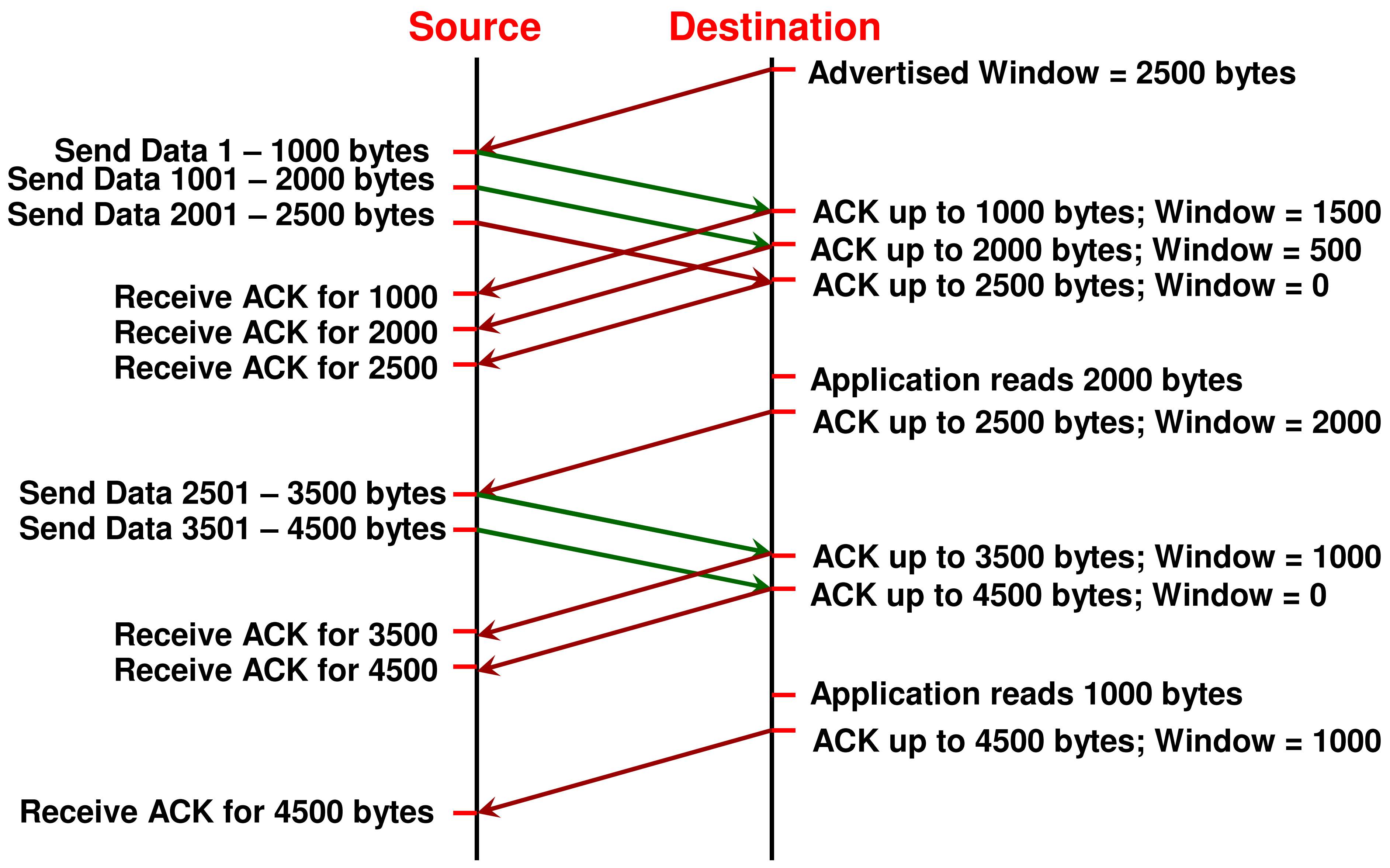
Outstanding bytes = Last Byte Sent – Last Byte Acked = 2600 – 2000 = 600 bytes

Effective Window = Advertised Window – Outstanding bytes

= 800 – 600 = 200 bytes.

Hence, the sender could only send 200 more bytes.

Flow Control: Example 2


Source
Destination
Advertised Window = 2500 bytes
Send Data 1 – 1000 bytes
Send Data 1001 – 2000 bytes
Send Data 2001 – 2500 bytes
ACK up to 1000 bytes; Window = 1500
ACK up to 2000 bytes; Window = 500
ACK up to 2500 bytes; Window = 0
Receive ACK for 1000
Receive ACK for 2000
Receive ACK for 2500
Application reads 2000 bytes
ACK up to 2500 bytes; Window = 2000
Send Data 2501 – 3500 bytes
Send Data 3501 – 4500 bytes
ACK up to 3500 bytes; Window = 1000
ACK up to 4500 bytes; Window = 0
Receive ACK for 3500
Receive ACK for 4500
Application reads 1000 bytes
ACK up to 4500 bytes; Window = 1000
Receive ACK for 4500 bytes

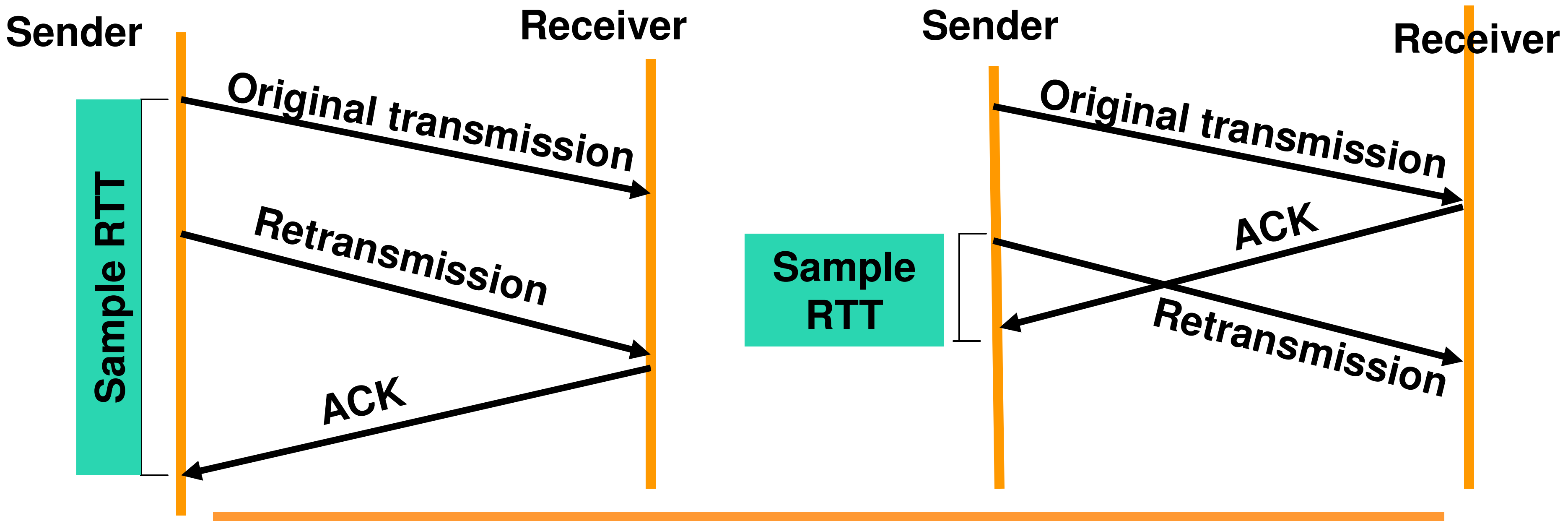
Congestion Control

- Congestion Control is the mechanism of adjusting the sending rate according to the resources (i.e., bandwidth and router queue size) available in the intermediate networks.

- Congestion Control is heavily dependent on the ‘Timeout’ value set at the source in order to decide about retransmitting a data packet that has not been acknowledged yet.
- As the Round-trip-time (RTT) between a source and destination across the Internet dynamically changes, estimating a proper RTT is key to setting the appropriate Timeout value to avoid unnecessary retransmissions and at the same time effectively utilize the channel bandwidth.

- The effective window (i.e., the amount of data the sender can send to the receiver satisfying the conditions of flow control and congestion control) is MIN(CongestionWindow, AdvertisedWindow) – (LastByteSent – LastByteAcked).

Associating the Acknowledgements with Retransmission



Which Sample RTT to be used to calculate the Estimated RTT?
Solution: Karn/ Partridge Algorithm

1. **Use the simple retransmission algorithm, but measure the Sample RTT only for messages that were not retransmitted.**
2. **For every timeout, set the next timeout twice the value of the last timeout, a binary exponential backoff approach useful to handle congestion.**

A Simple Retransmission Algorithm

- The round-trip time (RTT) for each connection should be estimated by measuring the time it takes to receive a response.
- Each time TCP sends a message it starts a timer, measures the time at which the acknowledgement arrives; the difference between these two times is called the Sample RTT.
- For the first message, the Estimated RTT is the same as the Sample RTT. For other messages, Estimated RTT is the weighted average between the previous estimate and Sample RTT.

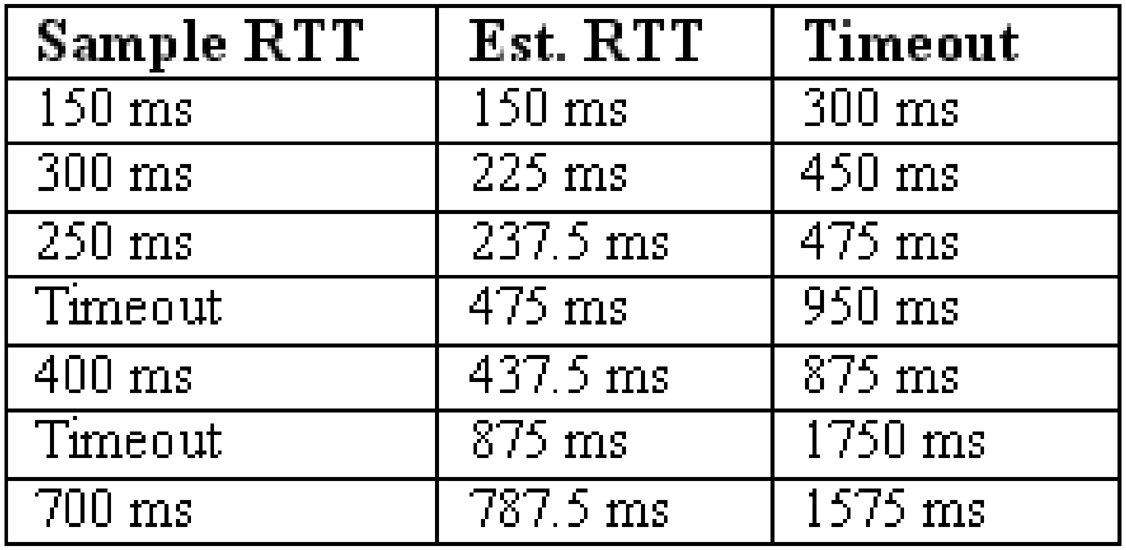
Estimated RTT = α * Estimated RTT + (1-α) * Sample RTT

Timeout = 2 * Estimated RTT

- A smaller value for α could track changes in the Sample RTT and is heavily influenced during temporary fluctuations. A larger value for α makes the retransmission algorithm not quick enough to adapt to real changes.

Sample Question: Retransmission Algorithm Example

- The following are the sample round-trip times (Sample RTTs) for the acknowledgments or timeouts for a sequence of packet transmissions at the sender side: 150 ms, 300 ms, 250 ms, timeout, 400 ms, timeout and 700 ms. Compute the estimated timeout value at the end of each acknowledgment received or timeout incurred. Use Karl's simple retransmission algorithm (α =0.5).
  - For the first packet, Est. RTT = Sample RTT
  - For subsequent packets, Est. RTT = 0.5 * Sample RTT + 0.5 * Est. RTT



Sample RTT	Est. RTT	Timeout
150 ms	150 ms	300 ms
300 ms	225 ms	450 ms
250 ms	237.5 ms	475 ms
Timeout	475 ms	950 ms
400 ms	437.5 ms	875 ms
Timeout	875 ms	1750 ms
700 ms	787.5 ms	1575 ms

ERTT = 0.5*300 + 0.5*150

ERTT = 0.5*250 + 0.5*225

ERTT = 2 * ERTT

ERTT = 0.5*400 + 0.5*475

ERTT = 2 * ERTT

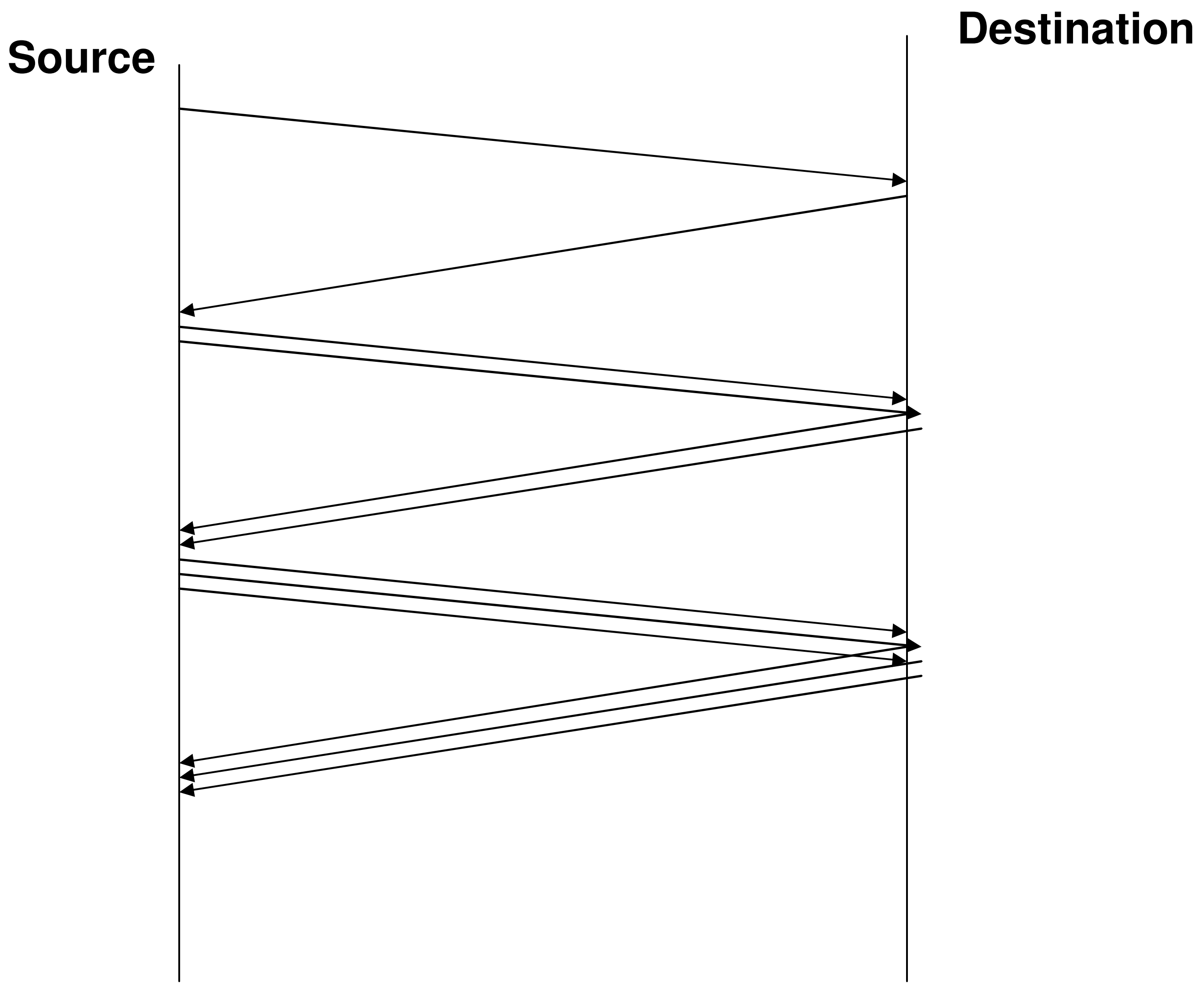
ERTT = 0.5*700 + 0.5*875

Additive Increase / Multiplicative Decrease (AIMD)

- Idea of congestion control: As packet losses are rarely to occur due to hardware errors/ transmission errors, a packet loss is considered by the sender as a sign of congestion in the network and hence it begins to slow down.

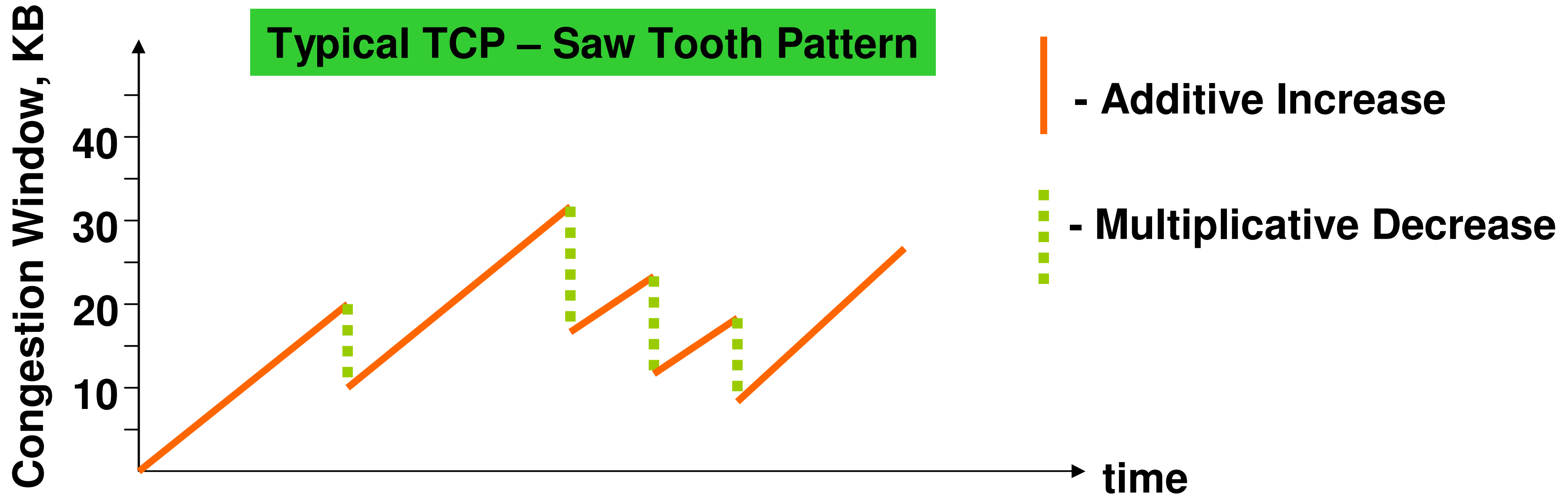
- Additive Increase:
- Initially, the sender does not know the congestion window. So, it starts very conservatively sending only one segment per RTT (i.e., congestion window = 1 segment).
- If an ACK is received within the timeout period, the sender sends two segments for the next RTT (i.e., congestion window = 2 segments).
- If the sender receives ACKs for both the segments with in their timeout period, it sends three segments for the next RTT and waits for three ACKs within their timeout period. (i.e., congestion window = 3 segments)
- The above procedure is continued until the congestion window size equals the advertised window or the congestion window size has to be dropped due to packet loss.

Example: Additive Increase



Additive Increase / Multiplicative Decrease (AIMD)

- Multiplicative Decrease:
- For each packet loss, the sender decreases its congestion window by one half of its current value.
- The congestion window size is not allowed to fall below one segment.



Slow Start

- The additive increase mechanism is too slow to ramp up a connection especially when starting from scratch.
- Slow start uses a congestion threshold (<= Advertised window) such that the congestion window is exponentially increased until reaching the congestion threshold and after that we increase the congestion window incrementally similar to that in AIMD.

- Initially, the congestion window is equal to 1 segment.
- When one segment is transmitted and an ACK received, the sender doubles the congestion window (congestion window 2 segments) for the next RTT.
- If the ACKs for both the segments arrive, then the sender doubles the congestion window (i.e., 4 segments) for the next RTT.
- The above procedure is repeated until the congestion window reaches the advertised window or there is a packet loss.

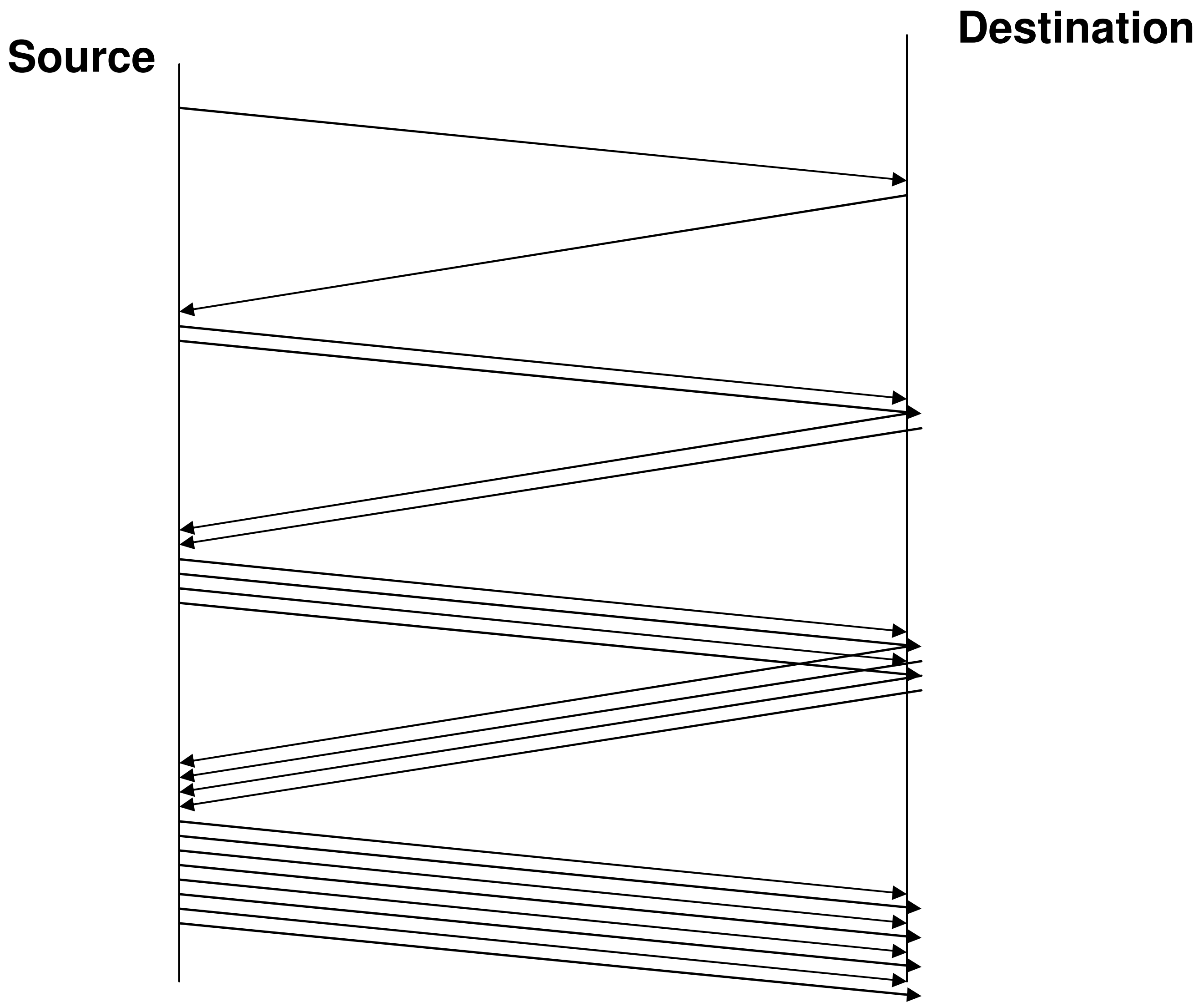
Slow Start

- If there is a packet loss, the congestion threshold is set to half of the current value of the congestion window, and the congestion window is set to 1. The congestion window is then again ramped up using the previously described exponential increase approach.
- When the congestion window reaches the congestion threshold, we employ additive increase rather than exponential increase.

Slow Start

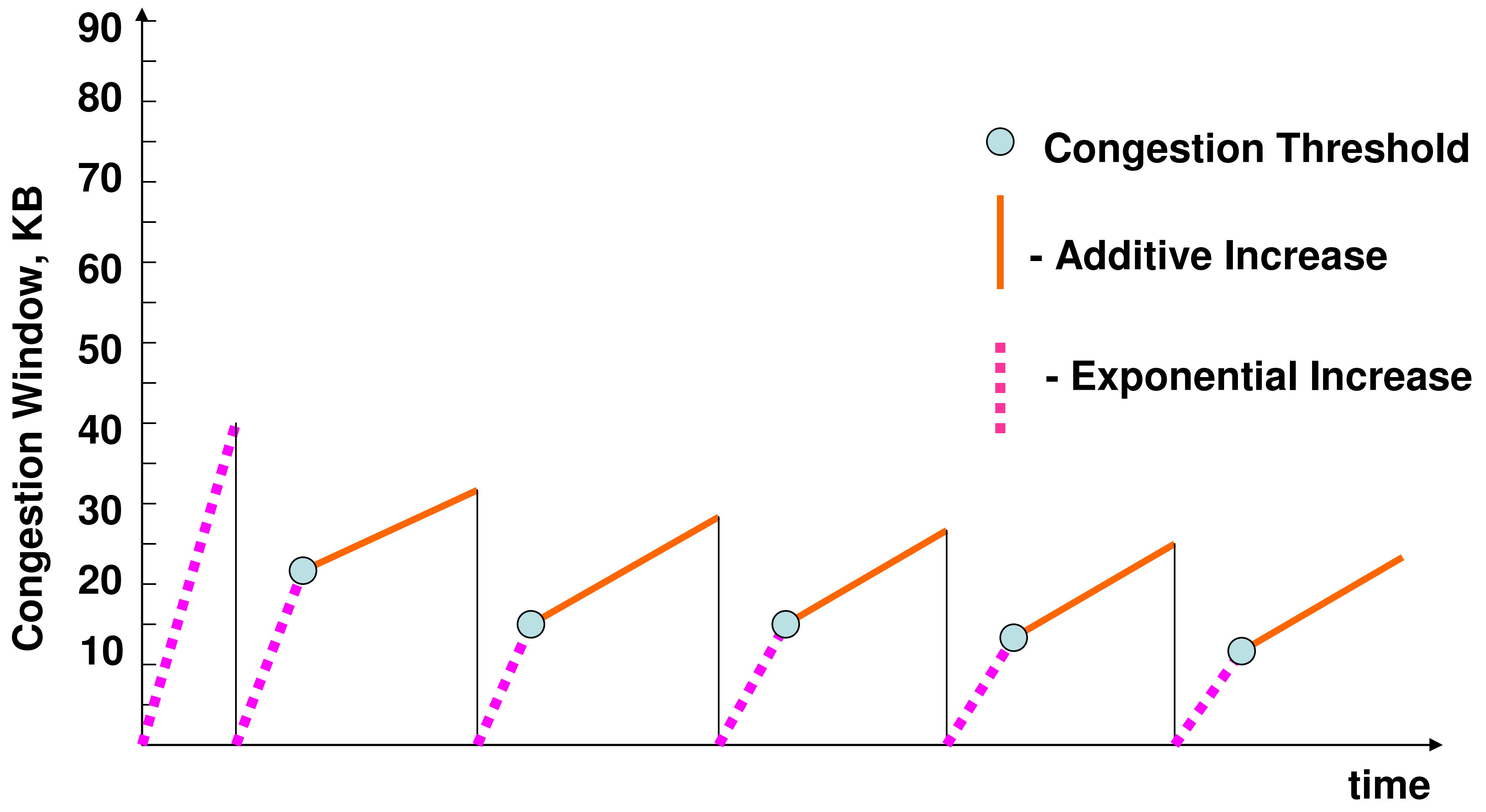
- The whole idea of ramping up exponentially until the congestion threshold is that in the previous round, we knew that until the congestion window was less than or equal the congestion threshold, there was no loss of packets.
- When the congestion window was twice the congestion threshold, we incurred a packet loss. So the actual capacity of the network that would avoid a packet loss is somewhere between the congestion threshold and the congestion window.
- So, in the current round, we proceed incrementally after the congestion threshold aiming to reduce packet loss and get a stable congestion window.

Slow Start



Exponential increase until Congestion Window reaches Congestion Threshold or Advertised Window

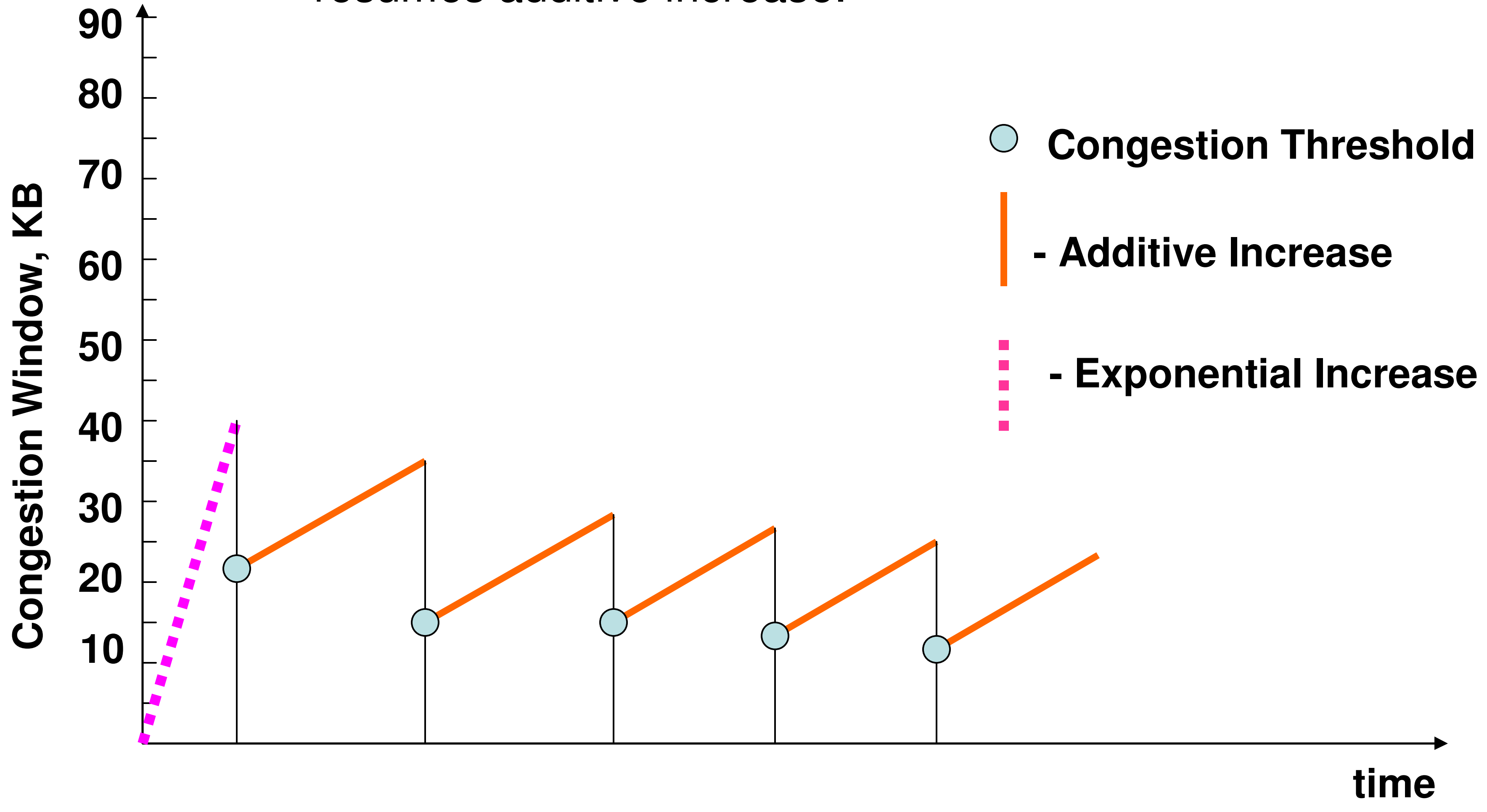
Slow Start



90
80
70
60
50
40
30
20
10
Congestion Window, KB
time
Congestion Threshold
- Additive Increase
- Exponential Increase

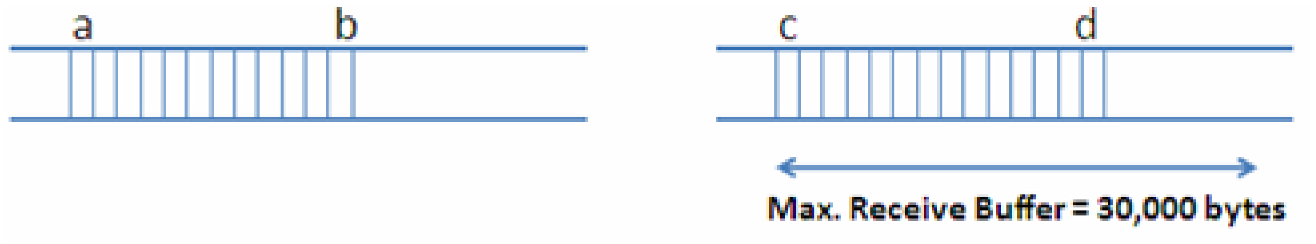
Fast Recovery

- With Fast Recovery, the source avoids the slow start and instead simply cuts the congestion window by half and resumes additive increase.



Sample Question: Flow Control and Congestion Control

- Consider the status of a TCP connection at the source and destination as shown in the Figure and Table below. Let the Congestion Window size be 15,000 bytes.



Source Process Destination Process

Notation	Description	Byte Sequence Number
a	Last Byte Acknowledged	20,000
b	Last Byte Sent	30,000
c	Last Byte Read	15,000
d	Last Byte Received	20,000

Sample Question: Flow Control and Congestion Control (continued…)

- **What would be the Effective Window Size (the amount of data that can be sent) by the source considering:**
- **(a) Only Congestion Control**

  Effective Window Size = Congestion Window Size – (Last Byte Sent – Last Byte Acknowledged)

  = 15,000 – (30,000 – 20,000) = 15,000 – 10,000 = 5,000 bytes

- **(b) Only Flow Control**

  Advertised Window = Max. Receiver Buffer – (Last Byte Received – Last Byte Read)

  = (30,000) – (20,000 – 15,000) = 25,000 bytes

  Effective Window Size = Advertised Window Size – (Last Byte Sent – Last Byte Acknowledged)

  = 25,000 – (30,000 – 20,000) = 15,000 bytes

- **(c) Both Flow Control and Congestion Control**

  Effective Window Size = Min (Congestion Window Size, Advertised Window Size) – (Last Byte Sent – Last Byte Acknowledged)

  = Min (15000, 25000) – (30000 – 20000)

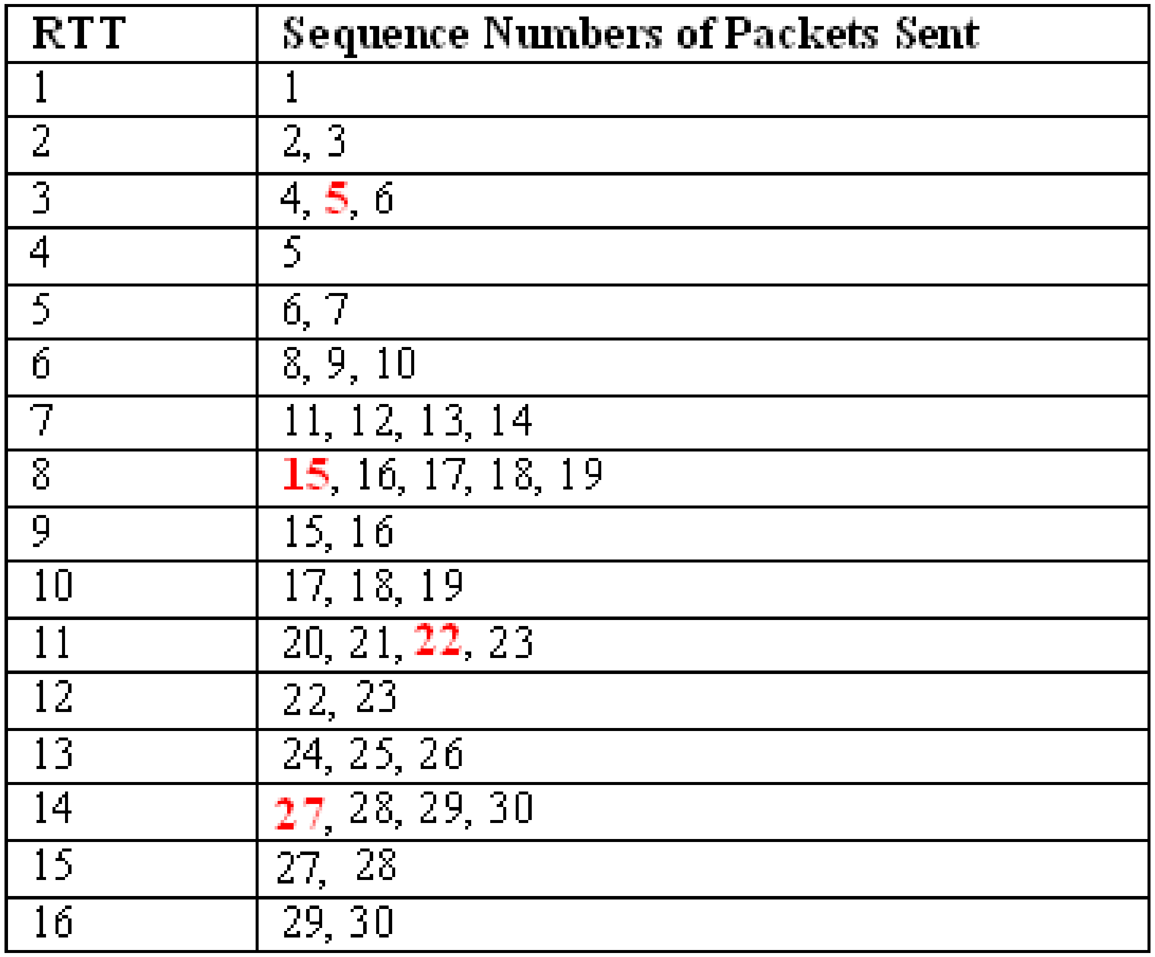
  = 5000 bytes

Sample Question 1: Congestion Control

- Consider each of the three congestion control algorithms that work in units of packets and that start each connection with a congestion window equal to one packet. Assume an ACK is sent for each packet received in-order, and when a packet is lost, ACKs are not sent for the lost packet and the subsequent packets that were transmitted. The lost packet and the subsequent packets have to be retransmitted by the sender. Whenever there is a packet loss and the sender times out in a RTT, the congestion window size in the next RTT has to be reduced to half of its size in the current RTT.

- For simplicity, assume a perfect timeout mechanism that detects a lost packet exactly 1 RTT after it is transmitted. Also, assume the congestion window is always less than or equal to the advertised window, so flow control need not be considered.
- Consider the loss of packets with sequence numbers **5, 15, 22 and 27** in their first transmission attempt. Assume these packets are delivered successfully in their first retransmission attempt.
- Fill the following table to indicate the RTTs and the sequence numbers of the packets sent. The sequence numbers of the packets sent range from **1 to 30**.
- Compute the effective throughput achieved by this connection to send packets with sequence numbers **1 to 30**, each packet holds 1KB of data and that the RTT = 100ms.

Sample Question: AIMD



RTT	Sequence Numbers of Packets Sent
1	1
2	2, 3
3	4, 5, 6
4	5
5	6, 7
6	8, 9, 10
7	11, 12, 13, 14
8	15, 16, 17, 18, 19
9	15, 16
10	17, 18, 19
11	20, 21, 22, 23
12	22, 23
13	24, 25, 26
14	27, 28, 29, 30
15	27, 28
16	29, 30

5, 15, 22, 27
- Lost packets

It takes 16 RTTs to send 30 packets.
The throughput = (30 packets * 1 KB/packet) / (16 RTTs * 100 ms /RTT)
= 153600 bits/sec

Sample Question: Slow Start

RTT	Sequence Numbers of Packets Sent
1	1
2	2, 3
3	4, 5, 6, 7
4	5 Cong. Threshold = 2
5	6, 7
6	8, 9, 10
7	11, 12, 13, 14
8	15, 16, 17, 18, 19 Cong. Threshold = 2
9	15
10	16, 17
11	18, 19, 20
12	21, 22, 23, 24 Cong. Threshold = 2
13	22
14	23, 24
15	25, 26, 27 Cong. Threshold = 1
16	27
17	28, 29
18	30

5, 15, 22, 27
- **Lost packets**

It takes 18 RTTs to send 30 packets.
The throughput = (30 packets * 1 KB/packet) / (18 RTTs * 100 ms /RTT)
= 136533 bits/sec

Sample Question: Fast Recovery

RTT	Sequence Numbers of Packets Sent	
1	1	
2	2, 3	
3	4, 5, 6, 7	Cong. Threshold = 2
4	5, 6	
5	7, 8, 9	
6	10, 11, 12, 13	
7	14, 15, 16, 17, 18	Cong. Threshold = 2
8	**15, 16**	
9	**17, 18**, 19	
10	20, 21, 22, 23	Cong. Threshold = 2
11	**22, 23**	
12	24, 25, 26	
13	27, 28, 29, 30	Cong. Threshold = 2
14	**27, 28**	
15	**29, 30**	

5, 15, 22, 27
- **Lost packets**

It takes 15 RTTs to send 30 packets.
The throughput = (30 packets * 1 KB/packet) / (15 RTTs * 100 ms /RTT)
= 163840 bits/sec

Sample Q2: Congestion Control

- Consider each of the three congestion control algorithms that work in units of packets and that start each connection with a congestion window equal to one packet. Assume an ACK is sent for each packet received in-order, and when a packet is lost, ACKs are not sent for the lost packet and the subsequent packets that were transmitted. The lost packet and the subsequent packets have to be retransmitted by the sender. Whenever there is a packet loss and the sender times out in a RTT, the congestion window size in the next RTT has to be reduced to half of its size in the current RTT.

- For simplicity, assume a perfect timeout mechanism that detects a lost packet exactly 1 RTT after it is transmitted. Also, assume the congestion window is always less than or equal to the advertised window, so flow control need not be considered.
- Consider the loss of packets with sequence numbers **10, 25, 34 and 45** in their first transmission attempt. Assume these packets are delivered successfully in their first retransmission attempt.
- Fill the following table to indicate the RTTs and the sequence numbers of the packets sent. The sequence numbers of the packets sent range from **1 to 50**.
- Compute the effective throughput achieved by this connection to send packets with sequence numbers **1 to 50**, each packet holds 1KB of data and that the RTT = 100ms.

Sample Question 2: AIMD

RTT	Sequence #
1	1
2	2, 3
3	4, 5, 6
4	7, 8, 9, 10
5	10, 11
6	12, 13, 14
7	15 , 16, 17, 18
8	19, 20, 21, 22, 23
9	24, 25, 26, 27, 28, 29
10	25, 26, 27
11	28, 29, 30, 31
12	32, 33, 34, 35, 36
13	34, 35
14	36, 37, 38
15	39, 40, 41, 42
16	43, 44, 45, 46, 47
17	45, 46
18	47, 48, 49
19	50

10, 25, 34, 45
Lost packets

Throughput = 50 packets * 1024 bytes * 8 bits/byte
-- = 50*1024*8/ (19*0.1sec) = 215, 578 bits/sec
100 ms/RTT * 19 RTTs

Sample Question 2: Slow Start

RTT	Sequence #
1	1
2	2, 3
3	4, 5, 6, 7
4	8, 9, 10, 11, 12, 13, 14, 15 (congestion window = 8; congestion threshold = 4)
5	10
6	11, 12
7	13, 14, 15, 16
8	17, 18, 19, 20, 21
9	22, 23, 24, 25, 26, 27 (congestion window = 6; congestion threshold = 3)
10	25
11	26, 27
12	28, 29, 30
13	31, 32, 33, 34 (congestion window = 4; congestion threshold = 2)
14	34
15	35, 36
16	37, 38, 39
17	40, 41, 42, 43
18	44, 45, 46, 47, 48 (congestion window = 5; congestion threshold = 2)
19	45
20	46, 47
21	48, 49, 50

10, 25, 34, 45
Lost packets

$$\text{Throughput} = \frac{50 \text{ packets} * 1024 \text{ bytes} * 8 \text{ bits/byte}}{100 \text{ ms/RTT} * 21 \text{ RTTs}} = 50*1024*8/\ (21*0.1\text{sec}) = 195,\ 047 \text{ bits/sec}$$

Sample Question 2: Fast Recovery

10, 25, 34, 45
Lost packets

RTT	Sequence #
1	1
2	2, 3
3	4, 5, 6, 7
4	8, 9, 10, 11, 12, 13, 14, 15 (congestion window = 8; congestion threshold = 4)
5	10, 11, 12, 13
6	14, 15, 16, 17, 18
7	19, 20, 21, 22, 23, 24
8	25, 26, 27, 28, 29, 30, 31 (congestion window = 7; congestion threshold = 3)
9	25, 26, 27
10	28, 29, 30, 31
11	32, 33, 34, 35, 36 (congestion window = 5; congestion threshold = 2)
12	34, 35
13	36, 37, 38
14	39, 40, 41, 42
15	43, 44, 45, 46, 47 (congestion window = 5; congestion threshold = 2)
16	45, 46
17	47, 48, 49
18	50

$$\text{Throughput} = \frac{50 \text{ packets} * 1024 \text{ bytes} * 8 \text{ bits/byte}}{100 \text{ ms/RTT} * 18 \text{ RTTs}} = 50*1024*8/\ (18*0.1\text{sec}) = 227,\ 555 \text{ bits/sec}$$