Chapter 3: Transport Layer

Transmission Control Protocol(TCP)

Instructor: HOU, Fen

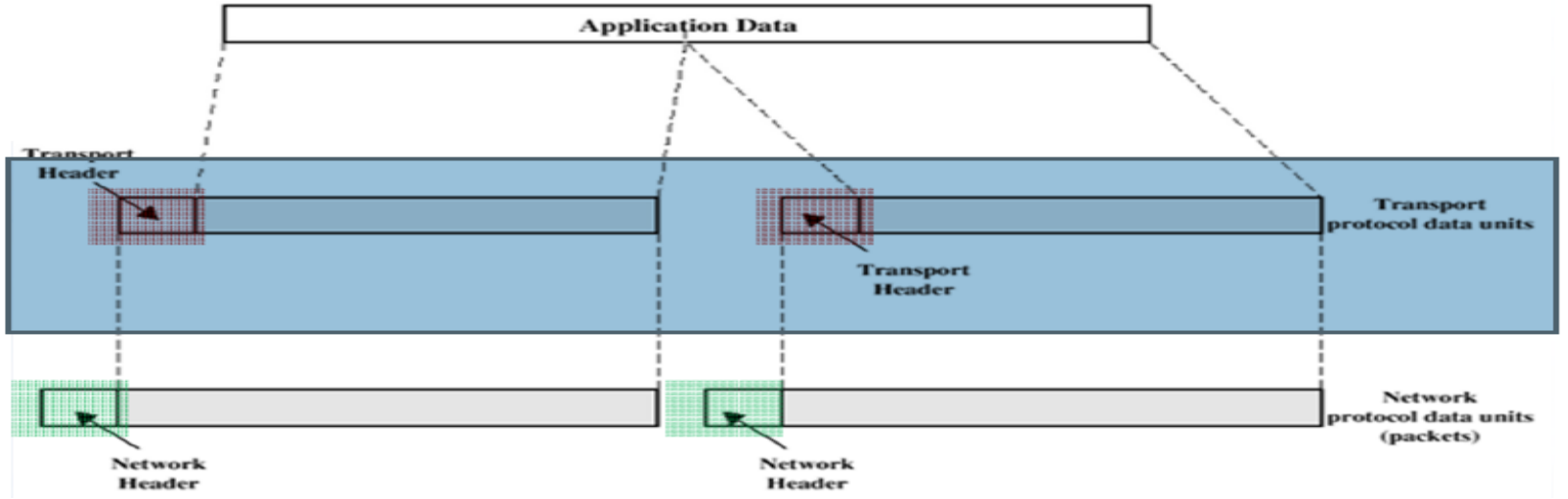
2025

Transmission Control Protocol(TCP)

- TCP provides the reliable data transfer
  - Reliable transmission
  - Congestion control
  - Flow control

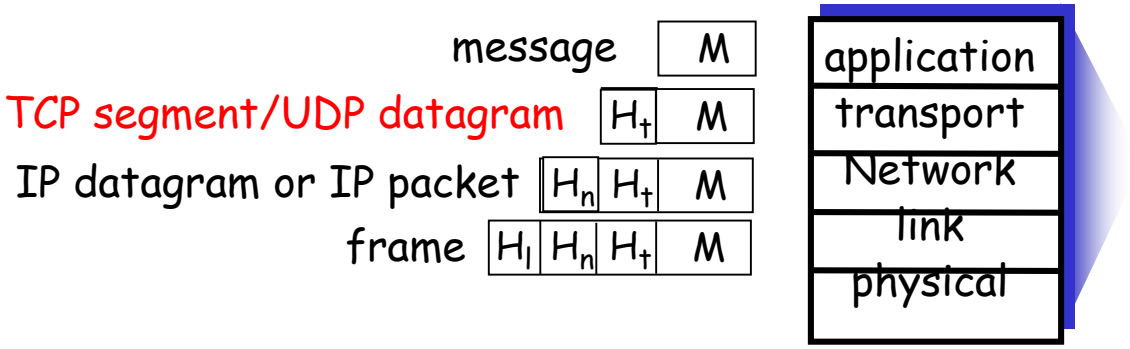
Protocol Data Unit (PDU)

- In order to achieve the communications between two end systems, some control information must be transmitted as well as user data.
- When the data is sent from higher layer to lower layer, data will get formatted and inserted with a header, according to the protocol used
- The combination of data from the higher layer and control information is known as protocol data unit (PDU)



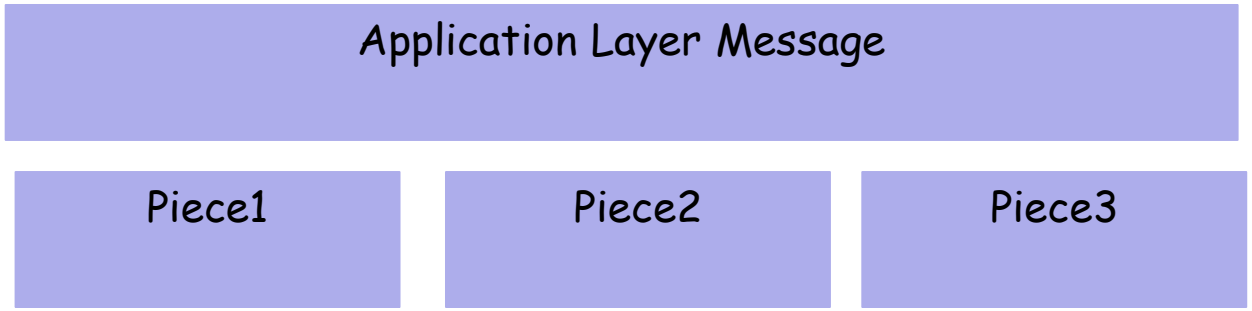
Transport Layer

- In the Internet terminology, we give the protocol data unit (i.e., the transmitting unit) at different layers a different name.
- PDU at the transport layer is called TCP segment or UDP datagram。
- Transport layer is a central piece of the layered network architecture.
- It provides data transmission between processes running on different hosts.



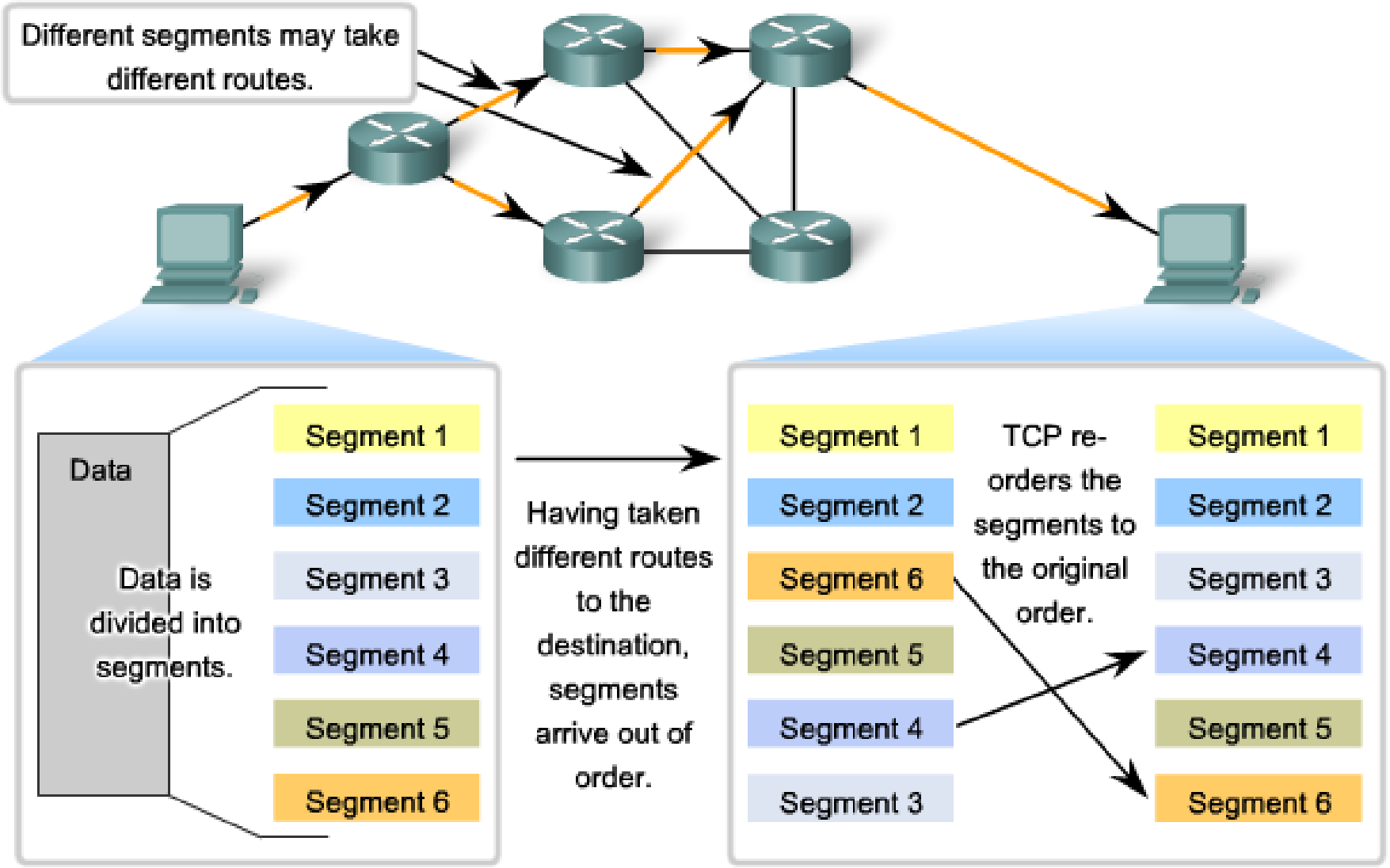
Segmentation

- Transport Layer Segmentation: Dividing data into small parts, and sending them from the source to the destination,



TCP Segment:

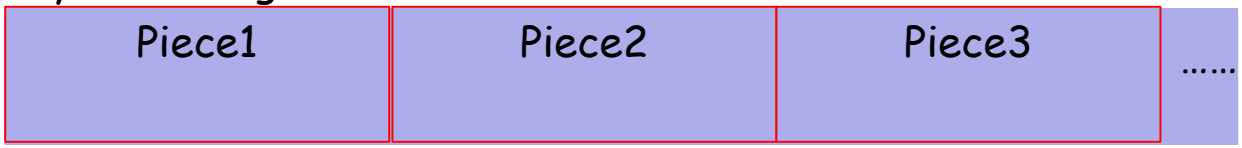
Re-assembling segments(with sequence number)



TCP Reliable Data Transfer

- At the sender side: Segmentation
  - The application gives the TCP some data to send.
  - The data is put in a send buffer; TCP breaks the data into chunks, adds TCP header to each chunk of data to form a TCP segment, then passes them down to the network layer.

Application Layer Message



TCP Segment:

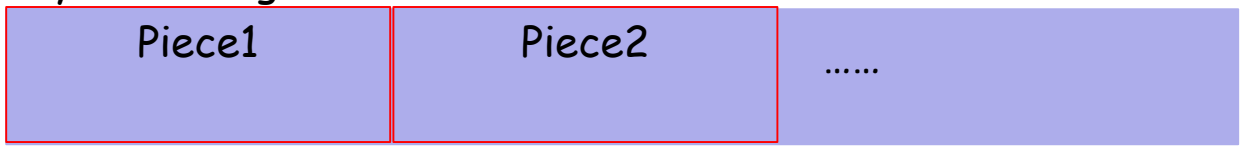
TCP Reliable Data Transfer

- At the receiver side: Re-assembling segments and buffer the incoming data
  - Receive buffer: The receiver allocates buffer to hold the received data.
  - The TCP does not know when the application will ask for the received data. TCP buffers incoming data so it is ready when the applications require them.

TCP Reliable Data Transfer

- How to form TCP segment at the sender
  - How to break the data in the send buffer into chunks?
  - How to decide the Sequence No.?

Application Layer Message



TCP Segment:

Segment 1

Segment 2

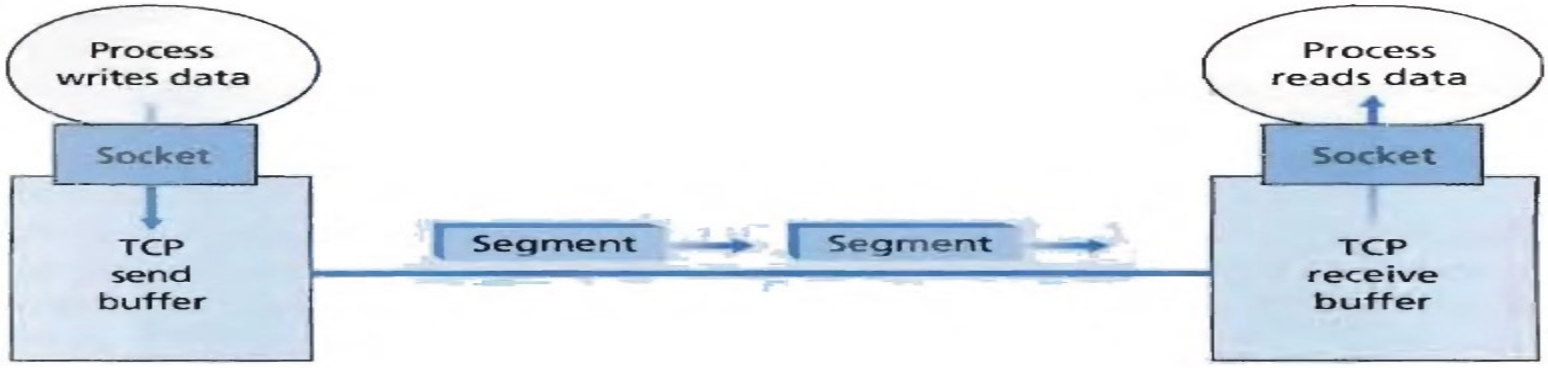
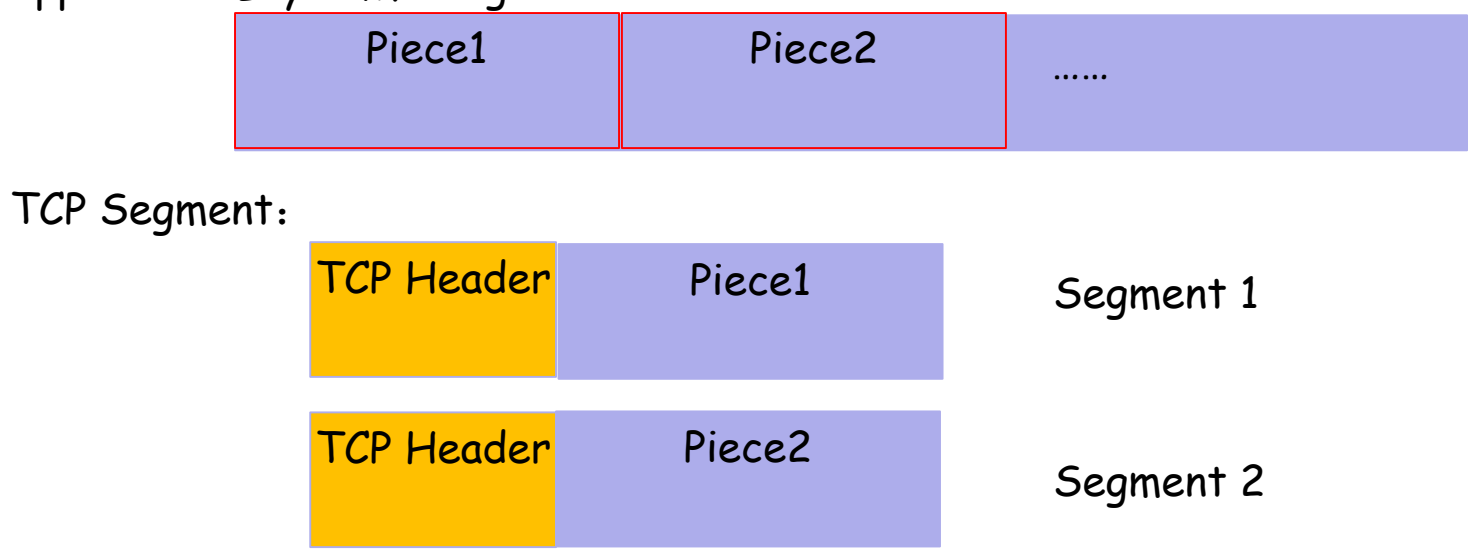


Figure 3.28 ♦ TCP send and receive buffers

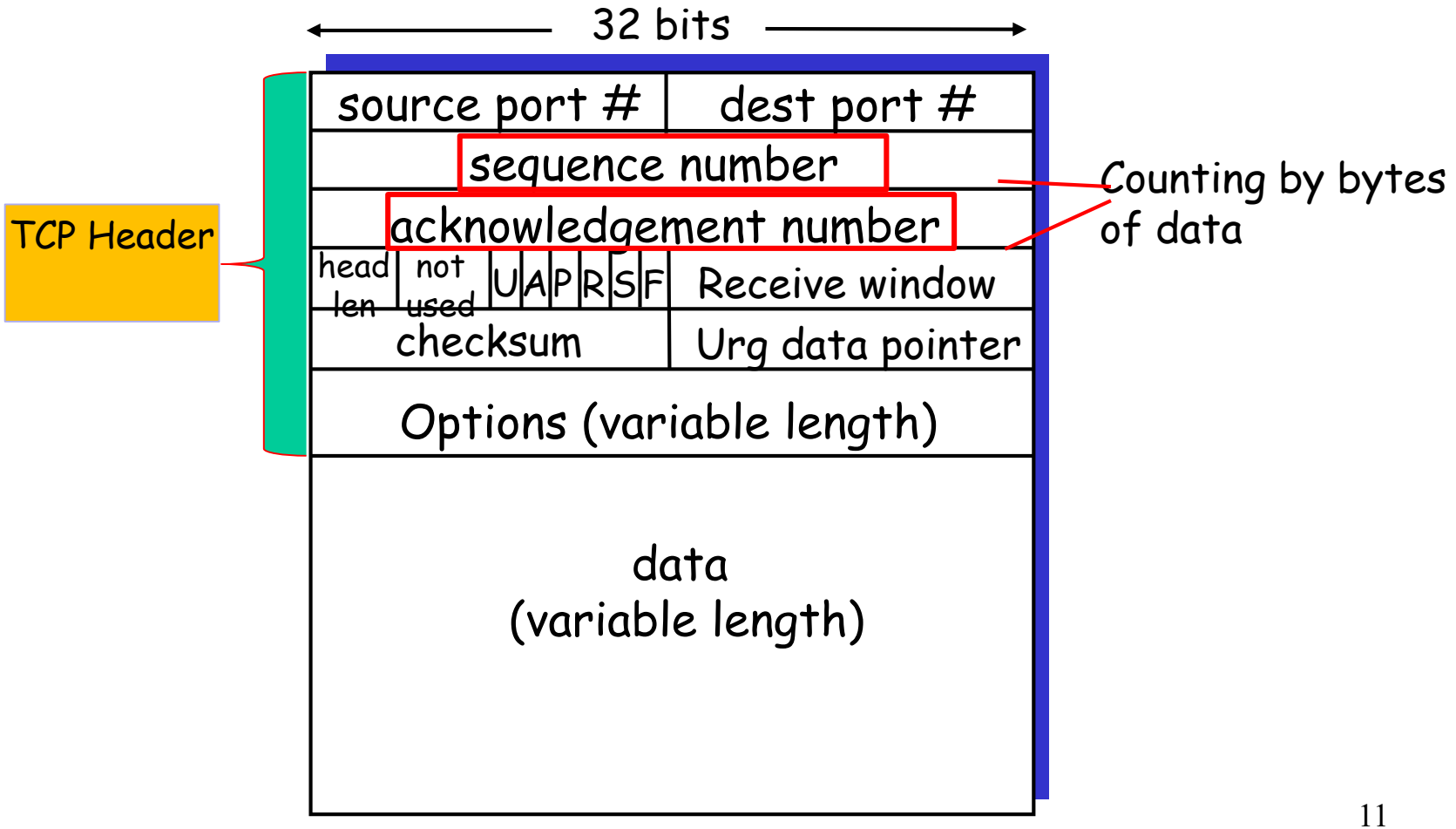
How to break the data into chunks?

- The data breaking is based on the maximum segment size (MSS).

- Maximum segment size (MSS):
  - It is the maximum amount of data (in bytes) that can be placed in a segment. Note that the MSS is the maximum amount of application-layer data in the segment, not the maximum size of the TCP segment including headers (i.e.,20bytes).



TCP segment structure



How to break the data into chunks?

- The data breaking is based on the maximum segment size (MSS).

- Maximum segment size (MSS):
  - It is the maximum amount of data (in bytes) that can be placed in a TCP segment exclusive of TCP header. That is, the MSS is the maximum amount of application-layer data in the segment, not the maximum size of the TCP segment including headers (i.e.,20bytes).

How to Decide the Sequence No.

- TCP views data as unstructured, but ordered stream of bytes.
- We label these bytes with integer numbers.
- Sequence number is the number of the first data in the segment in unit of bytes.
- Sequence numbers are over bytes, not segments.
- Example:
  - The data file consisting of 500,000 bytes, MSS is 1000bytes, the first segment gets assigned sequence number 0.
  - TCP constructs 500 segments; the sequence number set in the first, second, third segments is 0, 1000, 2000, respectively.

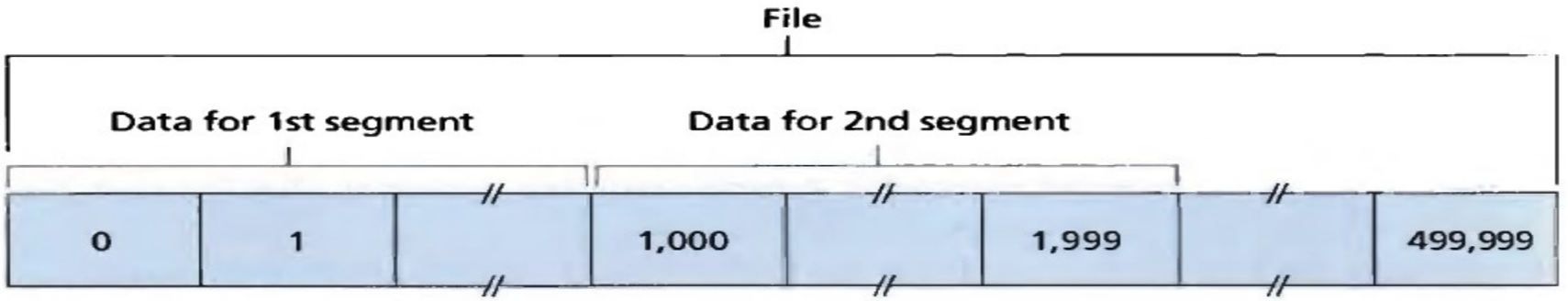


Figure 3.30 ♦ Dividing file data into TCP segments

How to Decide the ACK No.

- Acknowledgement number – At the receiver side, ACK number is the sequence number of the next byte/segment that the receiver node is expecting to receive from the sender.
- Example:
  - Host B received the 1st segment from Host A and is waiting for all the subsequent segments.
  - In this case, Host B puts 1000 in the ACK number field of the segment it sends to Host A.

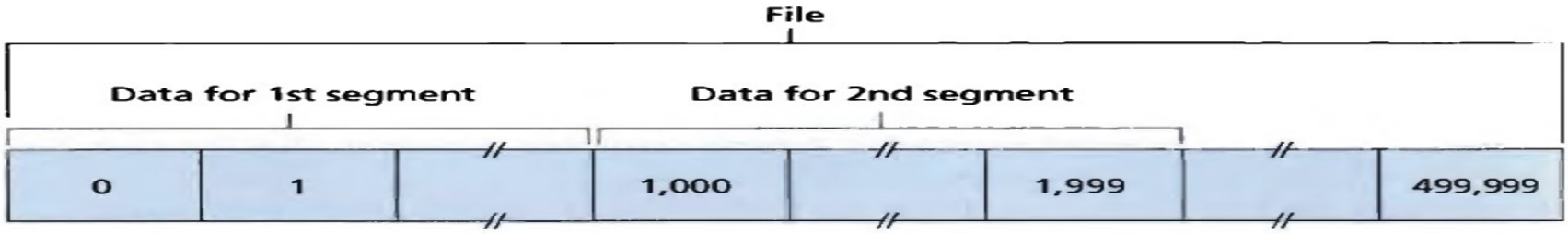


Figure 3.30 ♦ Dividing file data into TCP segments

TCP Reliable Data Transfer

- ACKs are cumulative
  - TCP uses cumulative ACK: putting the sequence number of the next in-order segment expected to receive in ACK.
  - Correctly received but out-of-order segments are not individually ACKed by the receiver, but they will be buffered at the receiver side.

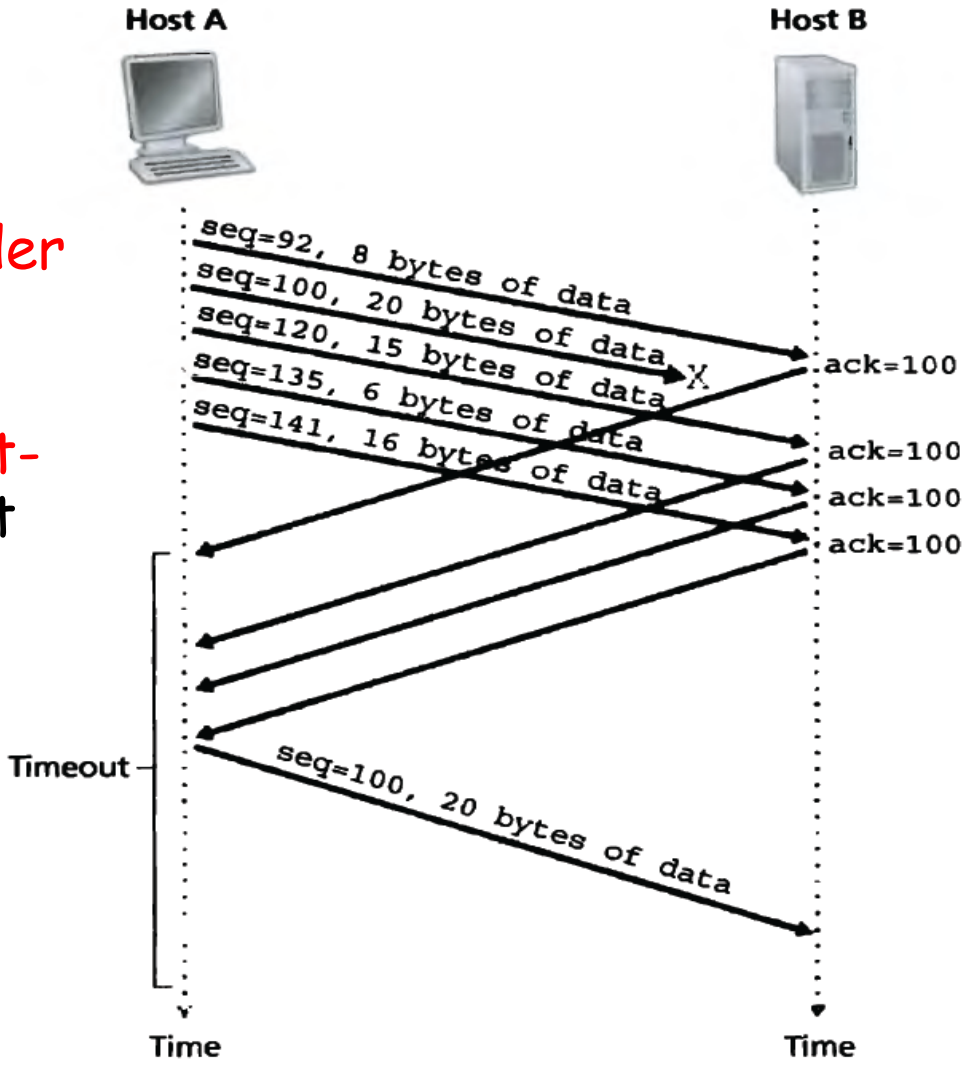


Figure 3.37 ♦ Fast retransmit: retransmitting the missing segment bef
the segment's timer expires.

TCP Reliable Data Transfer

- Is TCP a selective repeat protocol?
  - No. In selective repeat, all received packets are ACKed separately.
- Is TCP is a GBN protocol?
  - No. In GBN, all out-of-order packets are deleted.

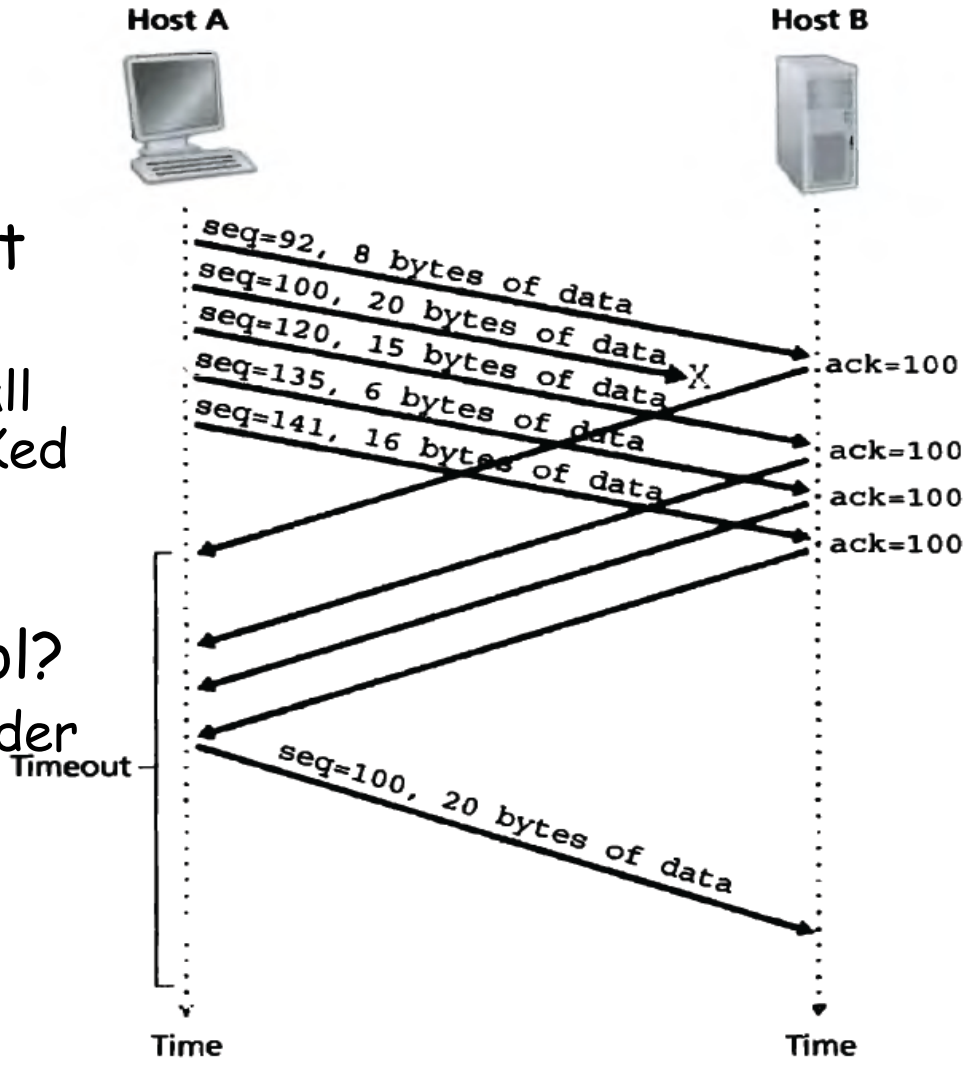


Figure 3.37 ♦ Fast retransmit: retransmitting the missing segment before the segment's timer expires.

Example

- Suppose Host A sends three TCP segments back to back to host B over a TCP connection. The first segment has sequence number 90; the second one has sequence number 110; the third one has sequence number 120
  - How many bytes of data is in the first segment?
    - 20 bytes

Timer and Timeout

- TCP uses a single timer
- Restart timer is triggered when
  - A segment is sent and the timer is not running for any other segment.
  - ACK is received.
  - Timeout event happens.

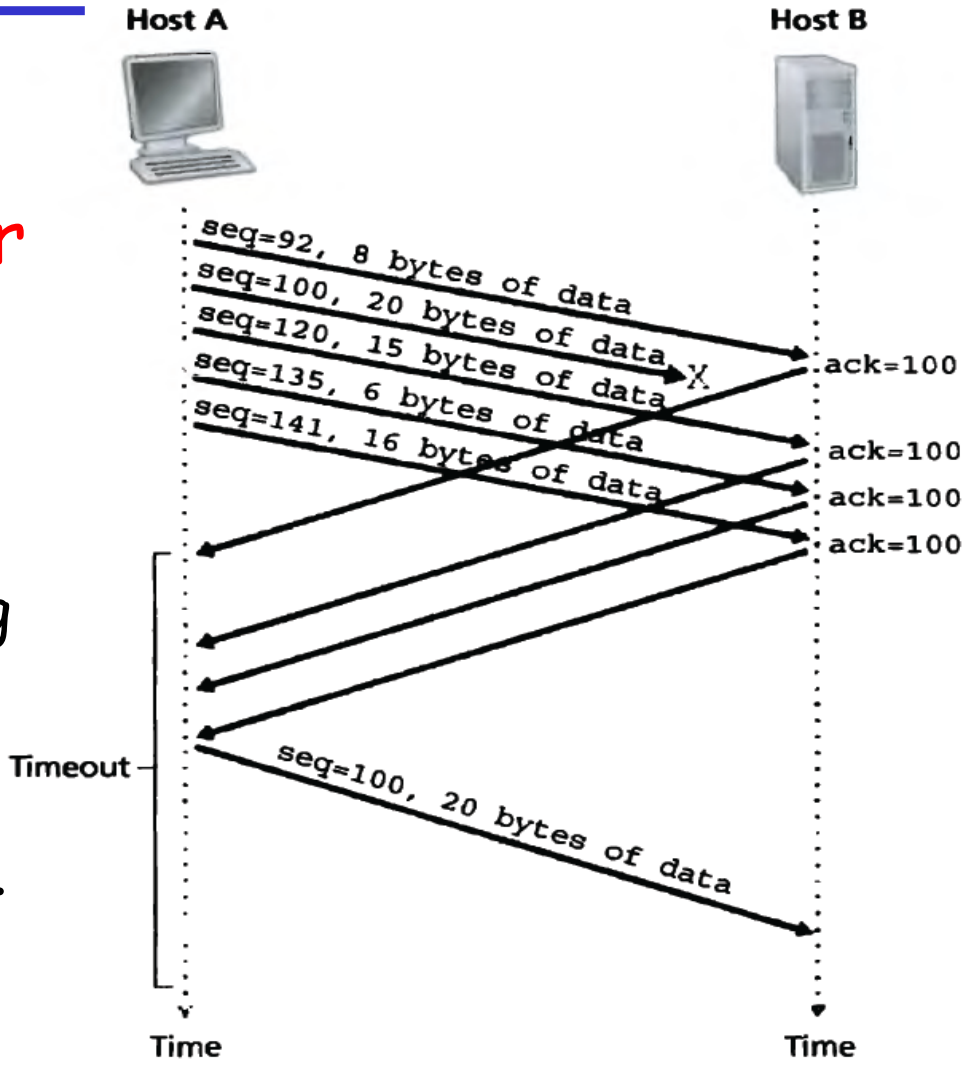
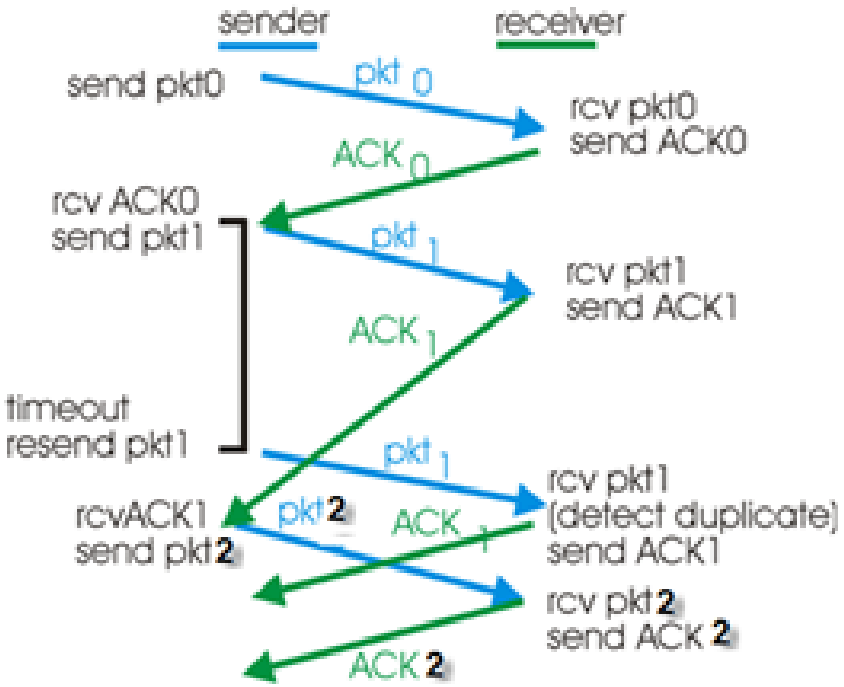


Figure 3.37 ♦ Fast retransmit: retransmitting the missing segment bef
the segment's timer expires.

How to Set TCP Timeout Value?

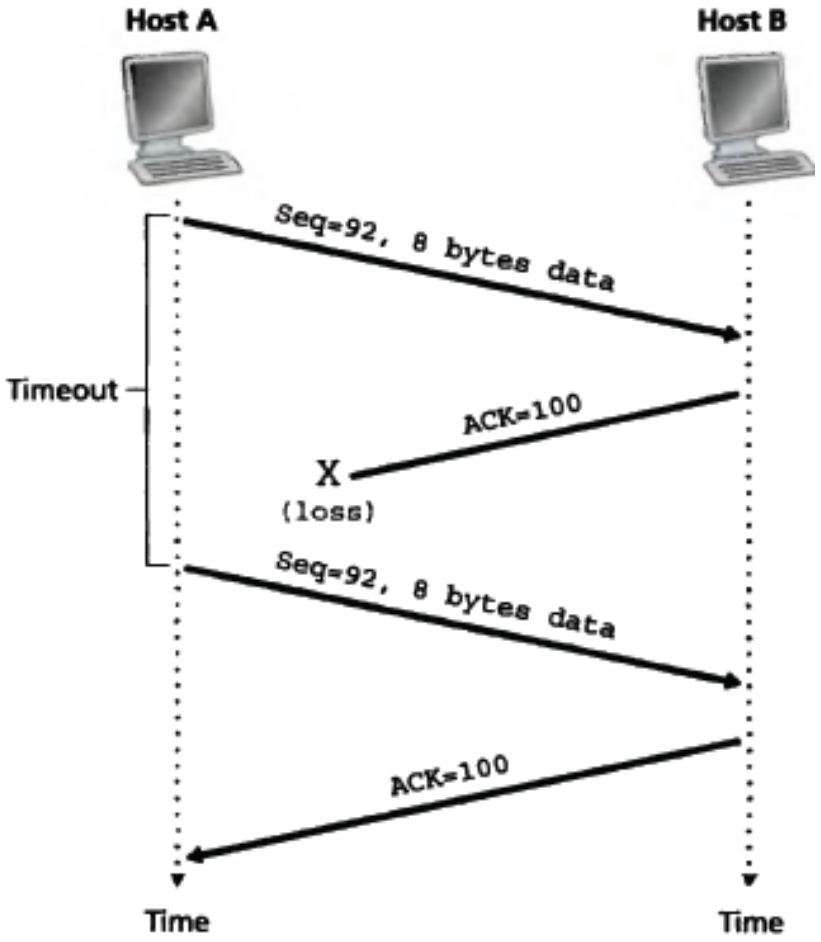
- longer than RTT (Round-Trip Time)
  - but RTT varies
- too short: premature timeout
  - Lead to unnecessary retransmission
- too long
  - Lead to slow reaction to segment/ACK loss, lead to large data transfer delay.

Example of premature timeout



(d) premature timeout

Example of timeout setting



How to Estimate RTT?

- **SampleRTT**: measured time from segment transmission until ACK receipt

- **SampleRTT** will vary. We want the estimated RTT to be "smoother"
  - average several recent measurements, not just current **SampleRTT**
- Exponential weighted moving average(EWMA)

```
EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT
```

  - Put different weight on recent samples and old samples
  - Influence of past sample decreases exponentially fast

How to Estimate RTT?

- Exponential weighted moving average(EWMA)

  EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT

  - Put different weight on recent samples and old samples
  - Influence of past sample decreases exponentially fast

- Time slot 1: EstimatedRTT_1=SmapleRTT_1;
- Time slot 2: EstimatedRTT_2

  =(1-a)EstimatedRTT_1 + a * SampleRTT_2

  =(1-a)SampleRTT_1 + a * SampleRTT_2;
- Time slot 3: EstimatedRTT_3

  =(1-a)* EstimatedRTT_2 + a * SampleRTT_3

  = (1-a) * [(1-a)SampleRTT_1 + a * SampleRTT_2] + a * SampleRTT_3

  $=(1-a)^2$ * SampleRTT_1 + (1-a) * a *SampleRTT_2+ a * SampleRTT_3

Let a=0.6, we have

EstimatedRTT_3=**0.16**SampleRTT_1+**0.24**SampleRTT_2+**0.6**SampleRTT_3

Example

- Please calculate the estimated RTT using the exponential weighted moving average (EWMA) and fill in the following table, where the parameter α=0.125. Assume that the value of EstimatedRTT was 100 ms just before the first of these samples was obtained.

Pkt #	SampleRTT (ms)	EstimatedRTT(ms)
1	106	
2	120	
3	140	
4	90	
5	115	

Solution

- Calculate the EstimatedRTT after obtaining the first SampleRTT.

*EstimatedRTT = (1- α) * EstimatedRTT+α * SampleRTT*

*EstimatedRTT = (1-0.125) * 100+0.125 * 106*

*=0.875 * 100+0.125* 106*

=87.5+13.25

=100.75ms

Pkt #	SampleRTT (ms)	EstimatedRTT(ms)
1	106	100.75
2	120	
3	140	
4	90	
5	115	

Solution

- Calculate the EstimatedRTT after obtaining the second SampleRTT.

*EstimatedRTT = (1- α) * EstimatedRTT+α * SampleRTT*

*EstimatedRTT =(1-0.125) * 100.75+0.125 * 120*

*=0.875 * 100.75+0.125* 120*

= 88.15625+15

=103.15625ms

Pkt #	SampleRTT (ms)	EstimatedRTT(ms)
1	106	100.75
2	120	103.15625
3	140	
4	90	
5	115	

Solution

- Calculate the EstimatedRTT after obtaining the third SampleRTT.

*EstimatedRTT = (1- α) * EstimatedRTT +α * SampleRTT*

EstimatedRTT = (1-0.125) * 103.15 +0.125 * 140

= 0.875 * 103.15 +0.125* 140

= 90.26 +17.5

=107.75*ms*

Pkt #	SampleRTT (ms)	EstimatedRTT(ms)
1	106	100.75
2	120	103.15625
3	140	107.75
4	90	
5	115	

Solution

- Calculate the EstimatedRTT after obtaining the fourth SampleRTT.

*EstimatedRTT = (1- α) * EstimatedRTT + α * SampleRTT*

*EstimatedRTT =(1-0.125) * 107.75 +0.125 * 90*

*= 0.875 * 107.75 +0.125* 90*

=+94.28 +11.25

=105.53ms

Pkt #	SampleRTT (ms)	EstimatedRTT(ms)
1	106	100.75
2	120	103.15625
3	140	107.75
4	90	105.53
5	115	

Solution

- Calculate the EstimatedRTT after obtaining the fourth SampleRTT.

*EstimatedRTT = (1- α) * EstimatedRTT +α * SampleRTT*

*EstimatedRTT = (1-0.125) * 105.53 +0.125 * 115*

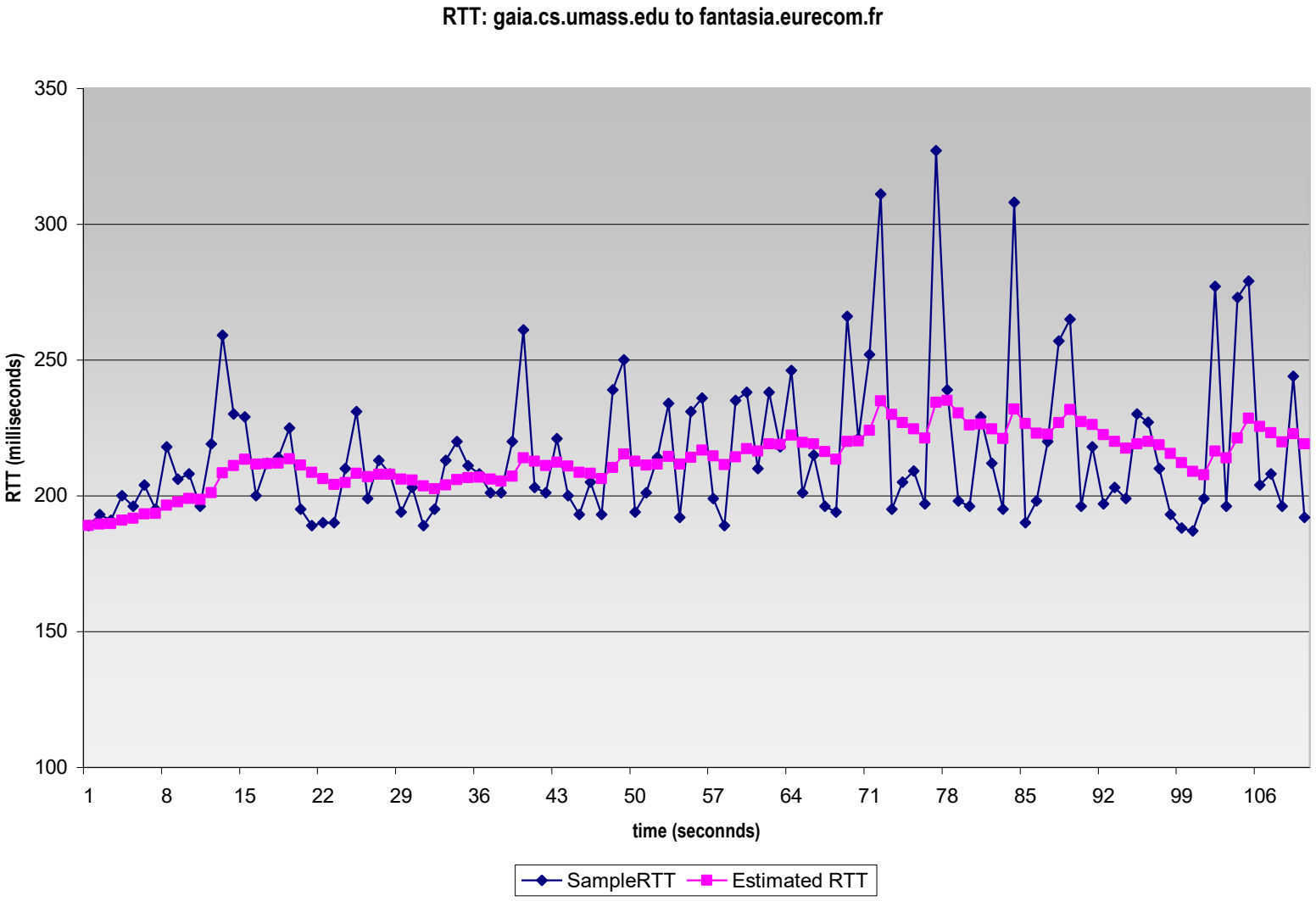
*= 0.875 * 105.53 +0.125* 115*

= 92.34+14.375

=106.715ms

Pkt #	SampleRTT (ms)	EstimatedRTT(ms)
1	106	100.75
2	120	103.15625
3	140	107.75
4	90	105.53
5	115	106.715

Example RTT estimation:



TCP Round Trip Time and Timeout

- RTT variation DevRTT
  - Measure the variability of the RTT.
  - It is an estimation of how much SampleRTT typically deviates from EstimateRTT
- Exponential weighted moving average(EWMA)

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

  - Put different weight on recent samples and old samples
  - Influence of past sample decreases exponentially fast

Example

- Please calculate the DevRTT after each sample is obtained, using the exponential weighted moving average (EWMA), where the parameter β=0.25. Assume that the value of DevRTT was 5ms just before the first of these samples was obtained.

Pkt #	SampleRTT (ms)	EstimatedRTT(ms)	DevRTT(ms)
1	106	100.75	
2	120	103.15625	
3	140	107.75	
4	90	105.53	
5	115	106.715	

Solution

- Calculate the RevRTT after obtaining the first SampleRTT.

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

= (1-0.25) *5 + 0.25 * |106-100.75|

=0.75 * 5 + 0.25 *5.25

=3.75 + 1.3125

=5.0625*ms*

Pkt #	SampleRTT (ms)	EstimatedRTT(ms)	DevRTT(ms)
1	106	100.75	5.0625
2	120	103.15625	
3	140	107.75	
4	90	105.53	
5	115	106.715	

Solution

- Calculate the RevRTT after obtaining the second SampleRTT.

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

$= (1-0.25) *5.0625 +0.25 * |120-103.15625|$

$=0.75 * 5.0625 +0.25 *16.84$

$=3.79+4.21$

$=8ms$

Pkt #	SampleRTT (ms)	EstimatedRTT(ms)	DevRTT(ms)
1	106	100.75	5.0625
2	120	103.15625	8
3	140	107.75	
4	90	105.53	
5	115	106.715	

Solution

- Calculate the RevRTT after obtaining the third SampleRTT.

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

$=(1-0.25) *8 + 0.25 * |140-107.75|$

$= 0.75 * 8 + 0.25 *32.25$

$=6 + 8.06$

$=14.06ms$

Pkt #	SampleRTT (ms)	EstimatedRTT(ms)	DevRTT(ms)
1	106	100.75	5.0625
2	120	103.15625	8
3	140	107.75	14.06
4	90	105.53	
5	115	106.715	

Solution

- Calculate the RevRTT after obtaining the fourth SampleRTT.

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

$= (1-0.25) *14.06 +0.25 * |90-105.53|$

$=0.75 * 14.06 +0.25 *15.53$

$=10.545+3.88$

$=14.42ms$

Pkt #	SampleRTT (ms)	EstimatedRTT(ms)	DevRTT(ms)
1	106	100.75	5.0625
2	120	103.15625	8
3	140	107.75	14.06
4	90	105.53	14.42
5	115	106.715	

Solution

- Calculate the RevRTT after obtaining the fifth SampleRTT.

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

= (1-0.25) *14.42 +0.25 * |115-106.715|

=0.75 * 14.42 +0.25 *8.285

= 10.815 +2.07

=12.885ms

Pkt #	SampleRTT (ms)	EstimatedRTT(ms)	DevRTT(ms)
1	106	100.75	5.0625
2	120	103.15625	8
3	140	107.75	14.06
4	90	105.53	14.42
5	115	106.715	12.885

TCP Round Trip Time and Timeout

- Setting timeout interval
  - EstimatedRTT plus "safety margin" (RTT variation DevRTT)
  - Large variation in EstimatedRTT → large safety margin

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

Example

- Calculate the TCP Timeout interval after each of these samples is obtained.

Pkt #	SampleRTT (ms)	EstimatedRTT(ms)	DevRTT(ms)	Timeout Interval
1	106	100.75	5.0625	
2	120	103.15625	8	
3	140	107.75	14.06	
4	90	105.53	14.42	
5	115	106.715	12.885	

Solution

- Calculate the Timeout Interval after obtaining the first sample RTT.

TimeoutInterval = EstimatedRTT +4 DevRTT*

= 100.75 + 4 *5.0625

=121ms

Pkt #	SampleRTT (ms)	EstimatedRTT(ms)	DevRTT(ms)	Timeout Interval
1	106	100.75	5.0625	121
2	120	103.15625	8	
3	140	107.75	14.06	
4	90	105.53	14.42	
5	115	106.715	12.885	

Solution

- Calculate the Timeout Interval

TimeoutInterval = EstimatedRTT +4 DevRTT*

Pkt #	SampleRTT (ms)	EstimatedRTT(ms)	DevRTT(ms)	Timeout Interval
1	106	100.75	5.0625	121
2	120	103.15625	8	125.15625
3	140	107.75	14.06	164
4	90	105.53	14.42	163.99
5	115	106.715	12.885	163.21

TCP reliable data transfer

- TCP has 3 main components
  - Flow control
    - Eliminate the probability of the sender overflowing the receiver's buffer.
    - If receiver's buff is filling up, shrink the sending rate.
  - Congestion control
    - Don't send two much data into the network if there is network congestion.
    - Regulate the flow rate based on the degree of the network congestion.
  - Reliable transmission
    - How to deal with timer and timeout.

Buffers and Buffer Overflow

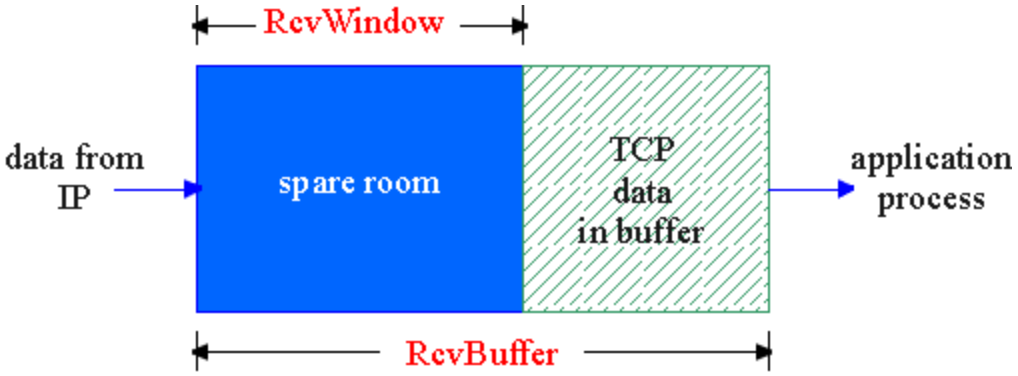
- Both the client and server allocate buffers to hold incoming and outgoing data
- Send buffer:
  - The application gives the TCP some data to send.
  - The data is put in a send buffer.
  - The TCP won't accept data from the application unless there is buffer space.
- Receive buffer
  - Buffer the received data.
  - The TCP does not know when the application will ask for the received data. TCP buffers incoming data so it is ready when the application needs it.

Buffers and Buffer Overflow

- Buffer overflow: the sender may easily overflow the receive buffer if the sender sends too much data too quickly and the application at the receiver is relatively slow at reading the data.

TCP Flow Control

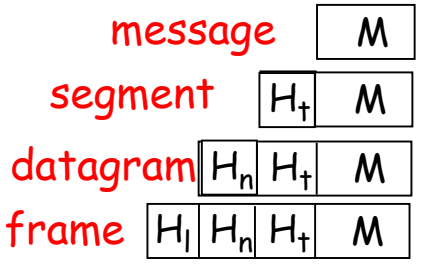
- receiver side of TCP connection has a receiving buffer:

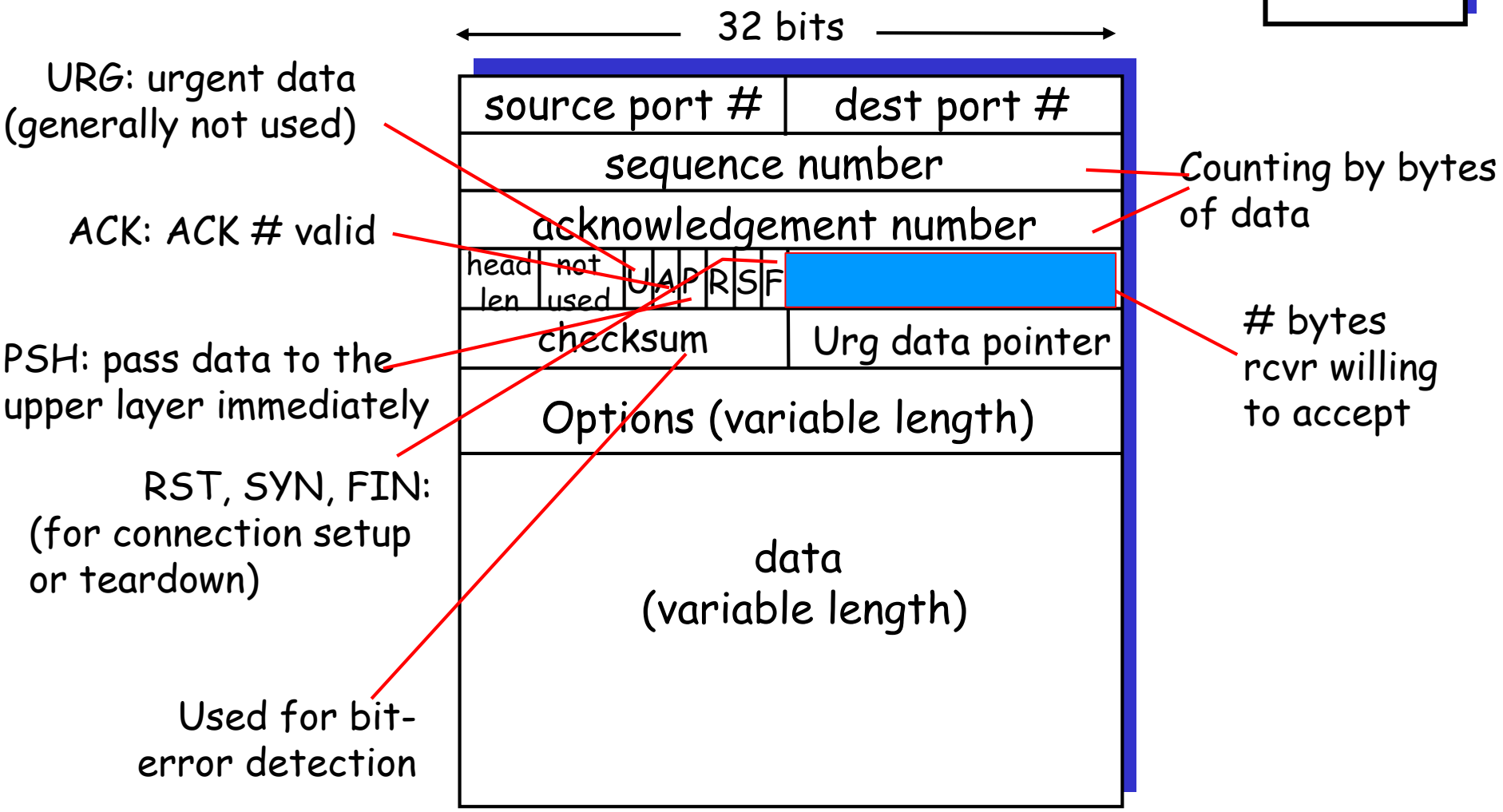


- application process may be slow at reading data from the buffer

- The sender may send the data too fast: speed-matching service is needed to match the send rate to the receiving app's drain rate

TCP segment structure





TCP Flow Control

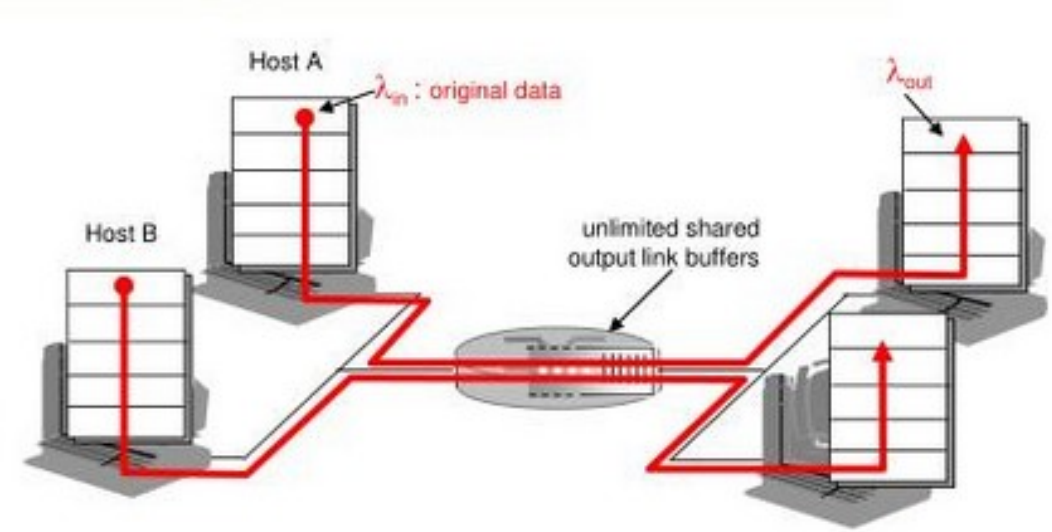
- Sender may overflow receiving buffer by transmitting too much or too fast.
- A process to provide the speed-matching between the sending rate and receive buffer.
- Sender makes sure that unACKed data is less than the amount of data that the receiver can buffer.
- Eliminate the possibility that the sender overflows the receive buffer
- Guarantee that the receive buffer doesn't overflow.

What is Congestion?

- Congestion: informally: "too many sources sending too much data too fast for *network* to handle"
- Congestion occurs when the number of packets being transmitted through the network approaches the packet handling capacity of the network.
- Generally, 80% utilization is critical.
- Congestion results in unfairness and poor utilization of network resources
  - Resources used by dropped packets (before they were lost) are wasted
  - Retransmissions
  - Poor resource utilization at high traffic load

Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission

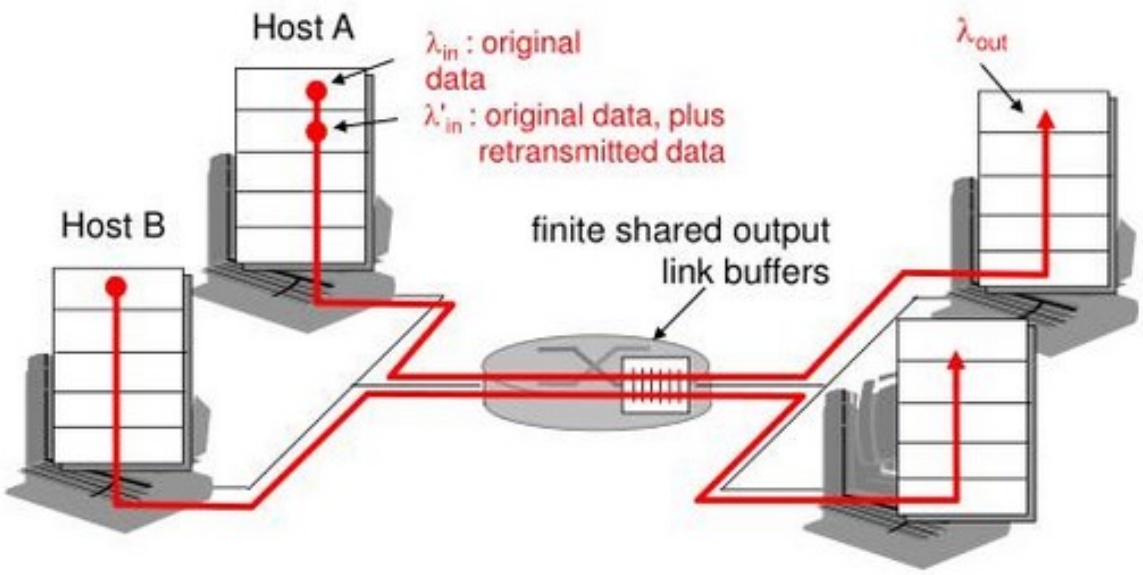


large delays when congested

Causes/costs of congestion: scenario 2

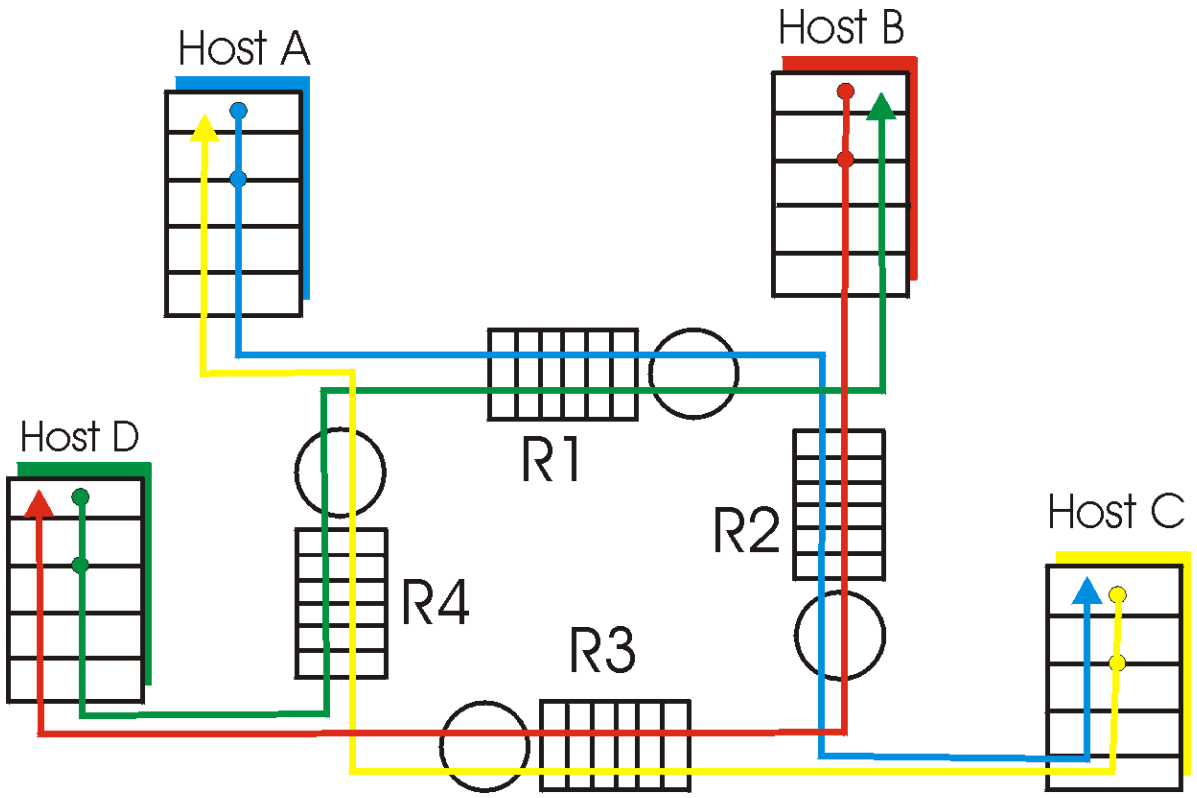
- Two senders and one router with *finite* buffers
- sender retransmission of lost packets

Link carries multiple copies of the same pkt

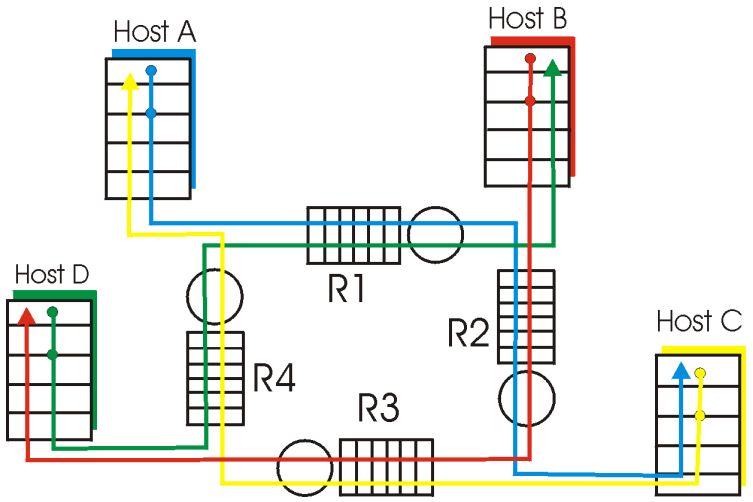


Causes/costs of congestion: scenario 3

- four senders, routers with finite buffer
- multihop paths: A-C path shares router R1 with D-B path, and shares R2 with B-D path.
- timeout/retransmit



Causes/costs of congestion: scenario 3



Another "cost" of congestion:

- when packet is dropped, any "upstream transmission capacity used for that packet is wasted!

Congestion and TCP Congestion Control

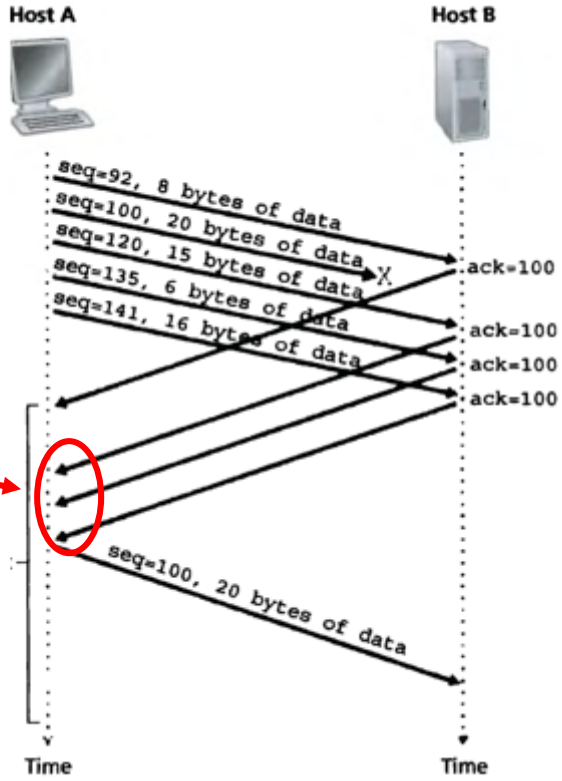
- Congestion control aims to keep the number of packets below level at which performance falls off dramatically.

- Mechanisms are needed to shrink the sending rate in the face of network congestion.

- TCP congestion control mechanism
  - How does a TCP sender perceive the degree of network congestion?
  - How does a TCP sender limit the sending rate?
  - What algorithm should the TCP sender use to adjust the sending rate?

Detecting Packet Loss

- Assumption: packet loss indicates congestion
- Loss event
  - Option 1: time-out
    - Waiting for a time-out may be long!
  - Option 2: duplicate ACKs
- Duplicate ACK
  - It is an ACK that re-acknowledges a segment for which the sender has already ACKed.



How does a TCP sender perceive the degree of network congestion

- **3 duplicate ACKs** indicates that the network is congested, but it still is capable of delivering some segments

- **Timeout** indicates a "more alarming" congestion scenario

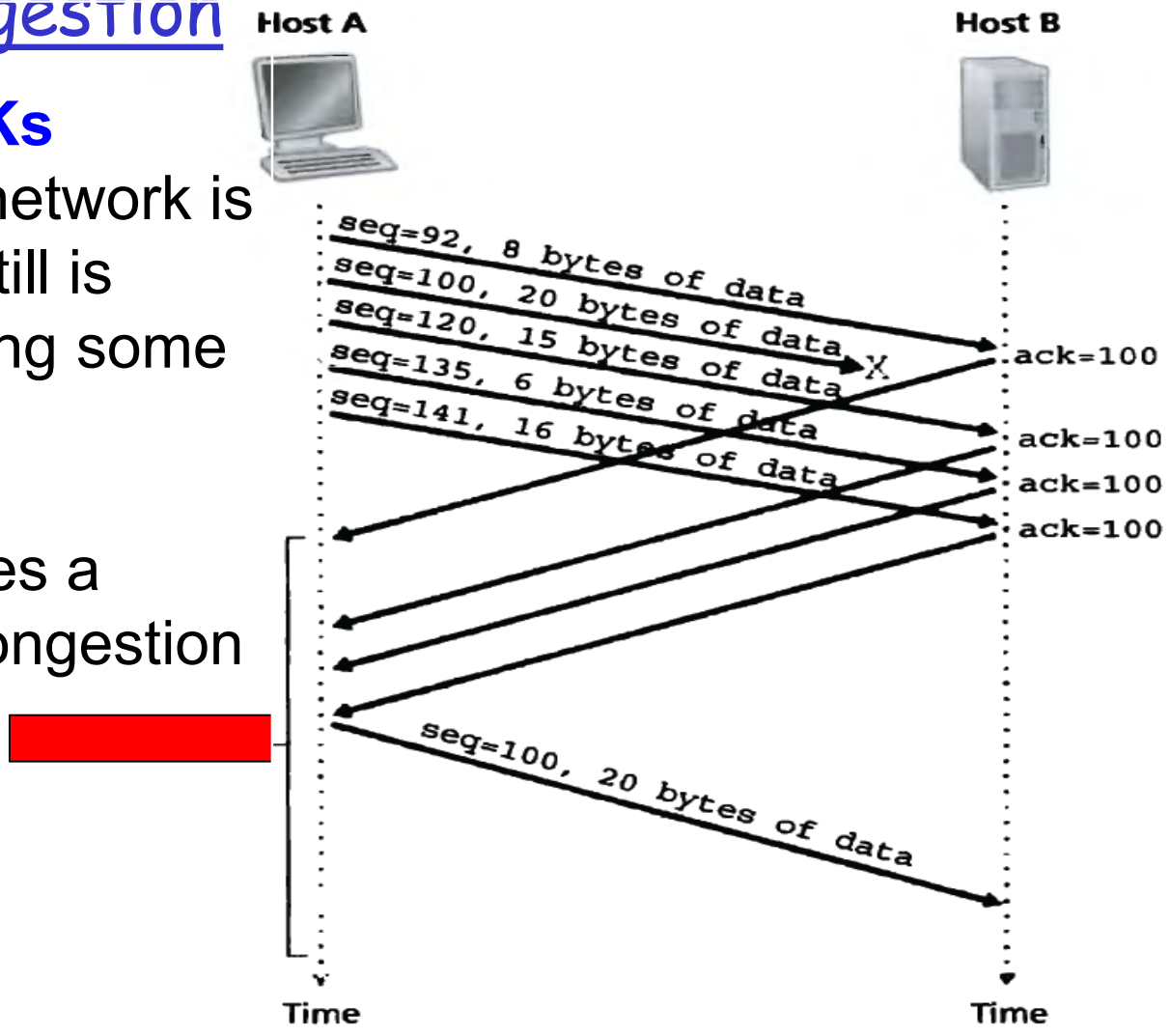


Figure 3.37 ♦ Fast retransmit: retransmitting the missing segment before the segment's timer expires.

TCP Congestion Control

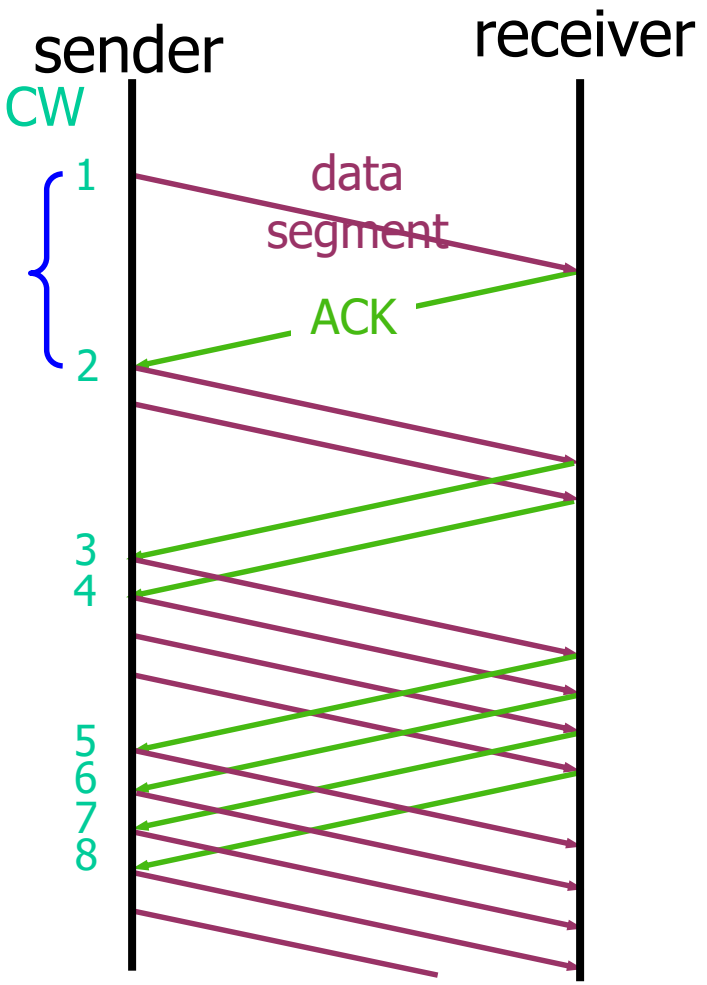
- Congestion window (CW): a parameter to limit the transmission rate
- Sender limits transmission:

  `LastByteSent-LastByteAcked` $\leq$ CW
- CW is dynamic. It is a function of perceived network congestion.

TCP congestion control

- Start from very slow transmission rate

  Initially: *CW = 1*
- Probe for available capacity by increasing CW until packet losses start occurring
  - Slow Start: increase rapidly
  - Congestion avoidance: increase slowly.
  - Slow start threshold (ssthresh): marks transition from rapid to slow increase phase
- When perceiving network congestion, reduce the sensing rate
  - 3 Dup ACKs
  - Time out



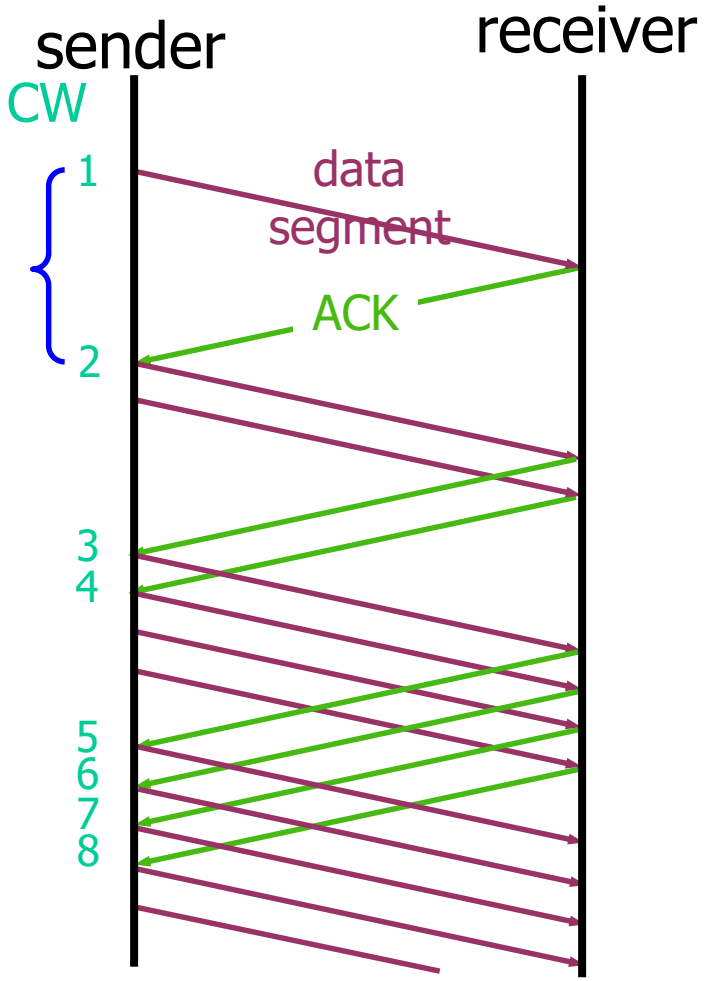
TCP congestion control: Slow Start

- Slow Start: increase rapidly
  - On each successful ACK:

    $CW \leftarrow CW + 1$
  - Exponential growth of CW after each RTT: CW doubles once every RTT.

    $CW \leftarrow 2 \times CW$

- Initial rate is slow but ramps up exponentially fast



double **CongWin** every RTT
done by incrementing **CongWin**
for every ACK received

Congestion Avoidance: Linear Increase Phase or Additive Increase

- Slow down the rate of increase when approaching the most recent point of congestion.
- ssthresh: slow start threshold is used for this purpose. It is the congestion window size after which TCP cautiously probes for available bandwidth.
- Enter CA(Congestion avoidance) when CW >= ssthresh
- On each successful ACK:

  $CW \leftarrow CW + 1/CW$

- Linear growth of CW each RTT: CW approximately increase by one segment every RTT.

  $CW \leftarrow CW + 1$

Congestion Avoidance: Additive Increase, Multiplicative Decrease (AIMD)

- AI: We have "additive increase" during the congestion avoidance in the absence of loss events.
- MD: After loss event, decrease ssthresh by half – "multiplicative decrease"
  - ssthresh = CW/2
  - Enter addition increase (AI) or slow start (SS)

Reaction to Timeout

- Adjusts ssthresh

  ssthresh $\leftarrow$ CW/2

- Enter slow start

  CW $\leftarrow$ 1

Reaction to 3 dupACKs

- After loss event indicated by 3 duplicated ACKs, decrease both ssthresh and congestion window by half
  - Adjusts ssthresh and CW

    $ssthresh \leftarrow CW/2, CW \leftarrow CW/2$
  - Enter additive increase (AI).

- Fast recovery: we call the immediate retransmission after 3 dup ACKs without waiting for timeout as fast retransmission or fast recovery since usually waiting for timeout is quite long.

Fast Retransmit

- Fast retransmission is triggered by three duplicate ACKs

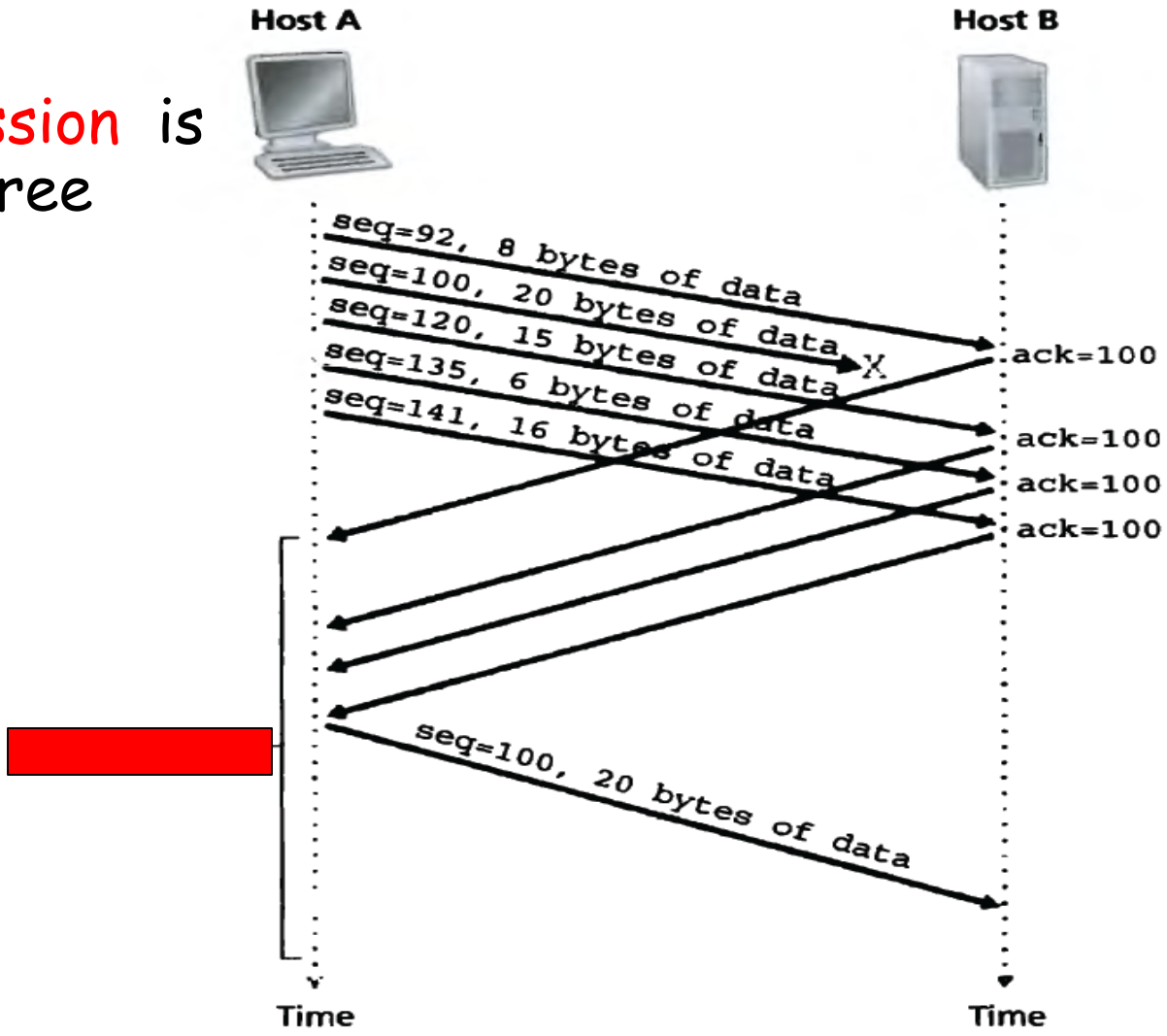


Figure 3.37 ♦ [redacted]: retransmitting the missing segment before the segment's timer expires.

TCP Congestion Control

- How does sender perceive congestion:
  - Timeout
  - 3 duplicate ACKs
- TCP sender reduces rate (i.e., congestion window) after perceiving congestion.
- TCP is self-clocking: TCP uses acknowledgements to trigger (clock) its adjustment of congestion window
- TCP congestion-control algorithm: three mechanisms
  - Slow start (initial phase or after RTO (retransmission timeout)
  - Additive-increase, multiplicative-decrease (AIMD)
  - Reaction to timeout event: conservative after timeout event (CW=1 MSS (maximum segment size))

TCP Congestion Control

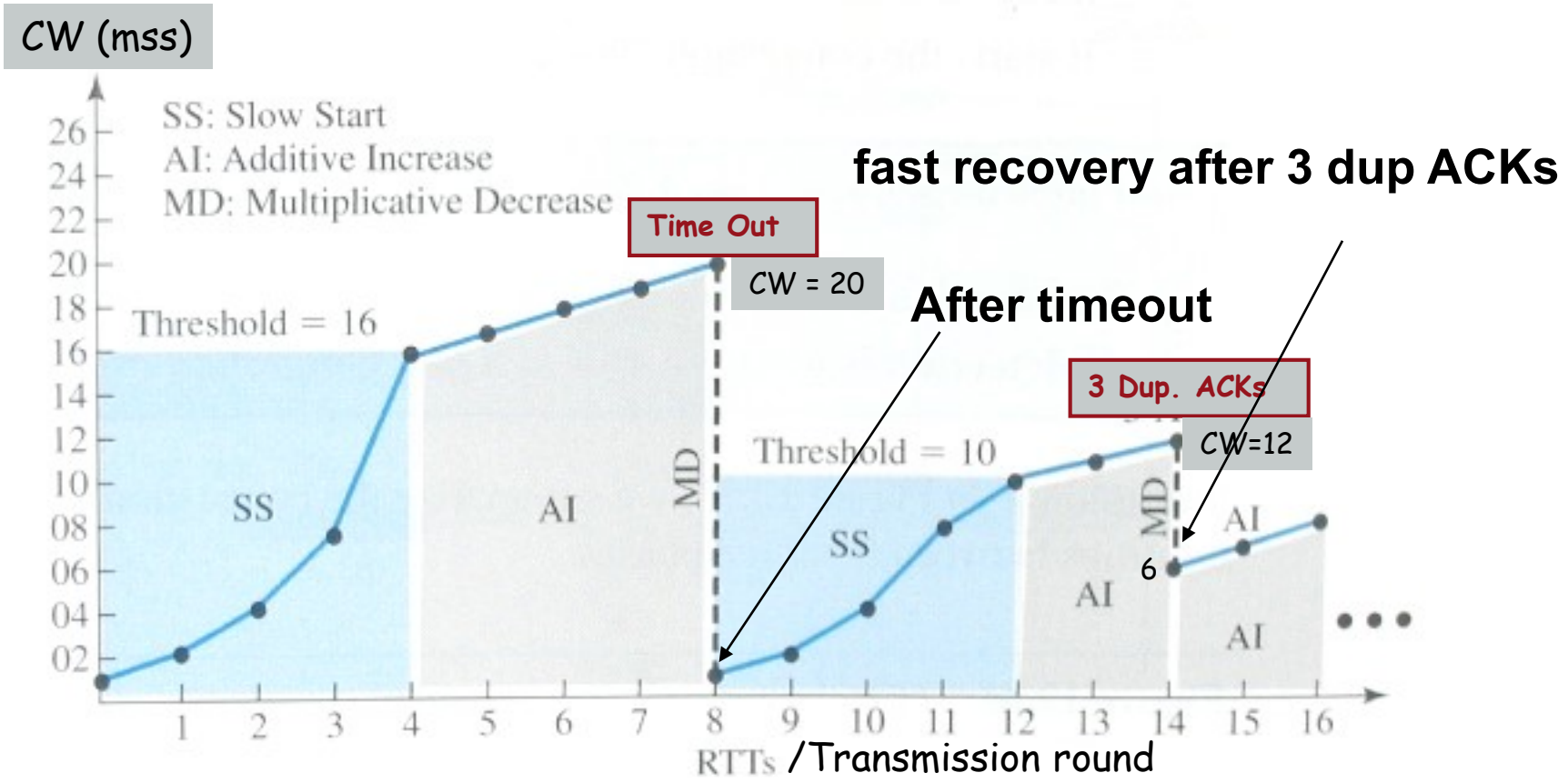
- **after 3 duplicate ACKs:**
  - **CW** is cut in **half**
  - window then grows **linearly**
- **but after timeout event:**
  - **CW** is set to **1**
  - window then grows **exponentially**
  - **Up to the ssthresh**, then grows **linearly**

Philosophy:

- **3 duplicate ACKs** indicates network capable of delivering some segments
- **timeout** indicates a "more alarming" congestion scenario

CW < Threshold: Doubles each RTT (add MSS for each MSS ACKed)
CW >= Threshold: Adds MSS each RTT

Time Out: Threshold = 1/2 CW, CW = 1 (Slow-Start)
3-Dup Ack: Threshold = 1/2 CW, CW = Threshold (Fast Recovery)

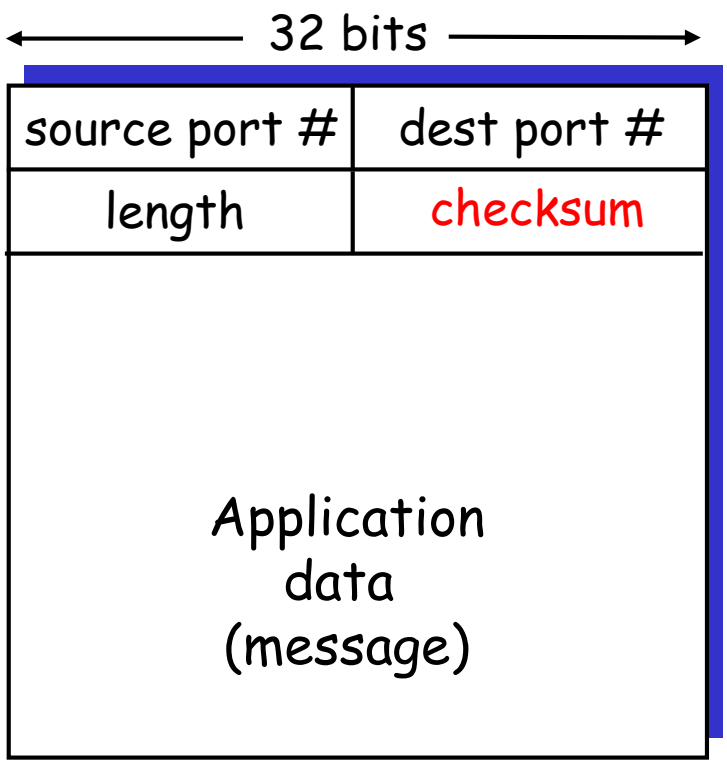


Transport Layer service

- Transport layer: Two transport layer protocols provide different services
  - TCP (Transmission Control Protocol) provides reliable, in-order delivery
    - connection setup
    - flow control, sequence numbers, acknowledgements, timers
    - congestion control
  - UDP(User Datagram Protocol) provides unreliable, unordered delivery
    - error checking: providing integrity checking by including error-detection field in the segments' headers
    - Connectionless: No need to set up a connection before transmitting data; All UDP packets are treated independently by transport layer and they are not related to each other.

UDP Segment Structure

- UDP segment format
  - 8 bytes header
  - Length: specify the length of UDP segment including header, in bytes
  - Checksum: used by the receiving host to check whether errors have been introduced into the segment during the transmission



UDP segment format

UDP: User Datagram Protocol

- UDP only provides essential functions that transport protocol can do (i.e. multiplexing/demultiplexing and some error check).
- UDP segments may be:
  - lost
  - delivered out of order to application
- Why use UDP?
  - No connection establishment cost
  - No connection state
  - Small segment headers (UDP header is only 8 bytes while TCP header is 20 bytes)
- UDP is used for
  - Video streaming: loss tolerant, delay-sensitive
  - DNS: no connection establishment delay