

Introduction to Computer Networks



Transport Layer Protocols

© All rights reserved. No part of this publication and file may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without prior written permission of Professor Nen-Fu Huang (E-mail: nfhuang@cs.nthu.edu.tw).

Outline

- **Introduction to end-to-end protocols**
- **Simple Demultiplexer protocol (UDP)**
- **Reliable Byte Stream protocol (TCP)**

End-to-end Protocols

- A transport protocol is usually expected to provide
 - **Guaranteed** message delivery
 - Delivers messages in the **same order** they were sent
 - Delivers **at most one copy** of each message
 - Supports arbitrarily large messages
 - Supports **synchronization** between the sender and the receiver
 - Allows the receiver to apply **flow control** to the sender
 - Supports multiple application processes on each host

End-to-end Protocols

- Typical limitations of the network service (like IP of Internet) on which transport protocol will operate
 - Drop messages
 - Reorder messages
 - Deliver duplicate copies of a given message
 - Limit messages to some finite size
 - Deliver messages after an arbitrarily long delay
- Unreliable service

End-to-end Protocols

- Challenge for Transport Protocols
 - Develop algorithms that **turn** the **unreliable service** of the underlying network **into** the **service** required by application programs
 - *Unreliable service → Unreliable service (UDP)*
 - *Unreliable service → Reliable service (TCP)*

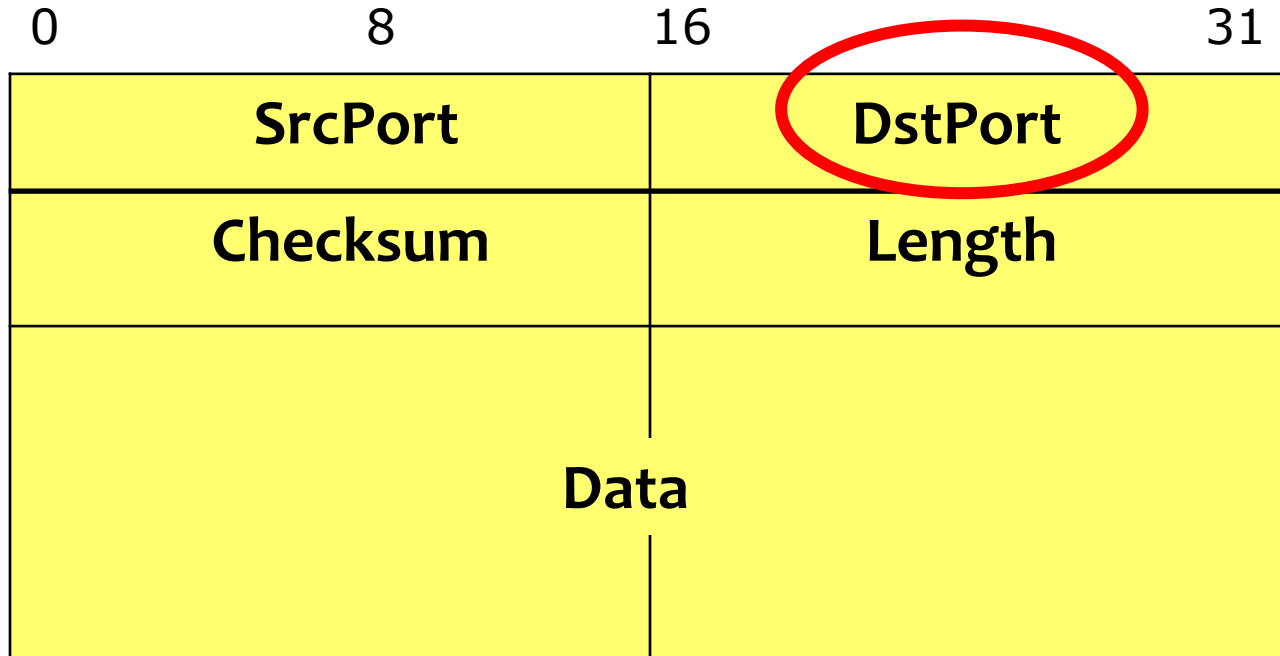
Outline

- Introduction to end-to-end protocols
- **Simple Demultiplexer protocol (UDP)**
- Reliable Byte Stream protocol (TCP)

Simple Demultiplexer (UDP)

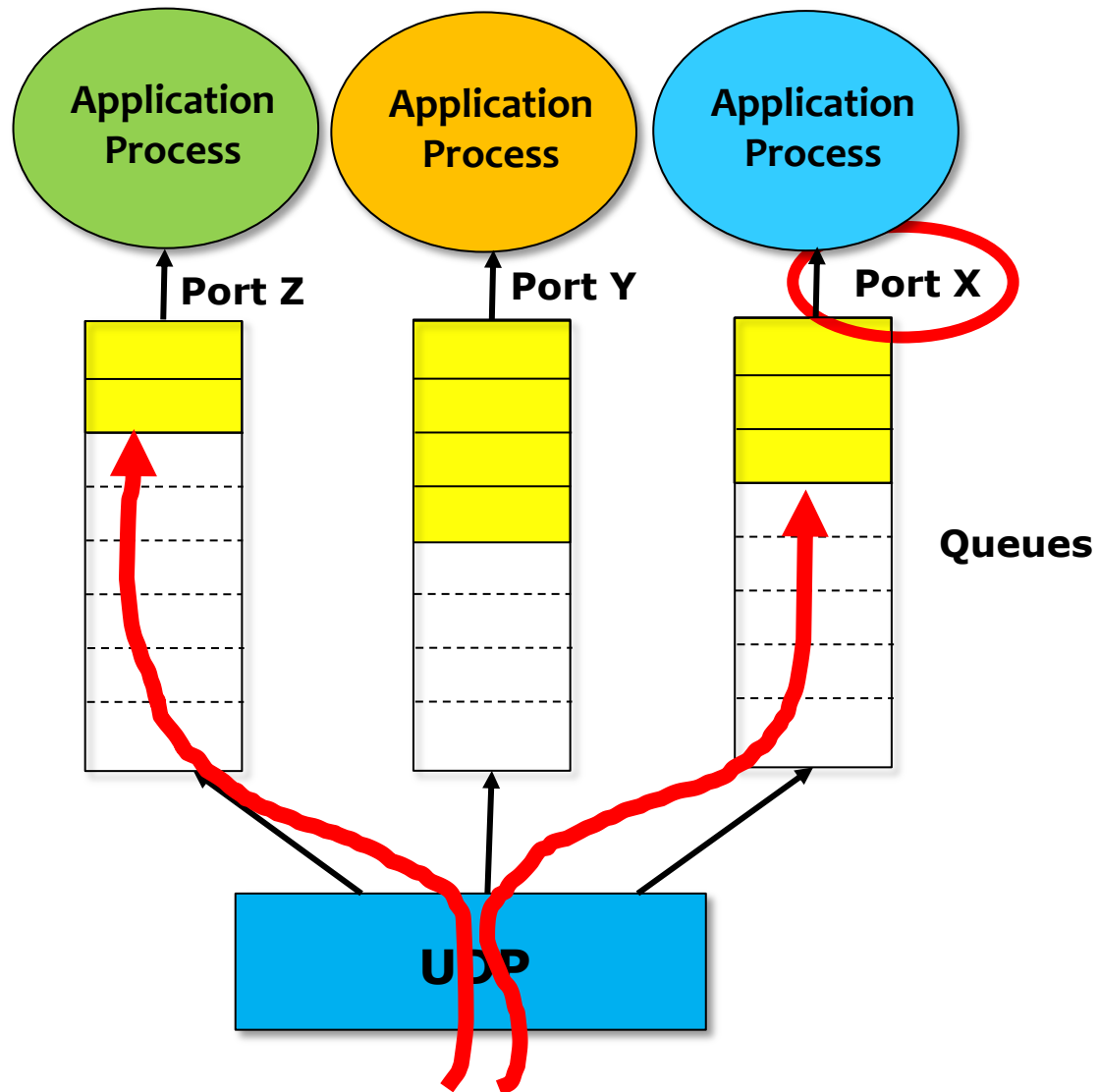
- Extends host-to-host delivery service of the underlying network into a **process-to-process** communication service
- Adds a level of **demultiplexing** which allows multiple application processes on each host to share the network

Simple Demultiplexer (UDP)



Format for UDP header

Simple Demultiplexer (UDP)



UDP Packet Demultiplexer

Outline

- Introduction to end-to-end protocols
- Simple Demultiplexer protocol (UDP)
- **Reliable Byte Stream protocol (TCP)**

Reliable Byte Stream (TCP)

- In contrast to UDP, Transmission Control Protocol (TCP) offers the following services
 - Reliable
 - Connection oriented
 - Byte-stream service

Flow control VS Congestion control

- **Flow control** involves preventing senders from overrunning the capacity of the receivers
- **Congestion control** involves preventing too much data from being injected into the network, thereby causing routers/switches or links to become overloaded

End-to-end Issues

- TCP runs **over the Internet** rather than a point-to-point link
- **The TCP sliding window algorithm** need to consider:
 - TCP supports **logical connections** between processes that are running on two different computers in the Internet
 - TCP connections are likely to have widely **different RTT times**
 - Packets may get **reordered** in the Internet

End-to-end Issues

- TCP needs a mechanism using which each side of a connection will **learn what resources** the other side offers to the connection
- TCP needs a mechanism using which the sending side will **learn the capacity** of the network

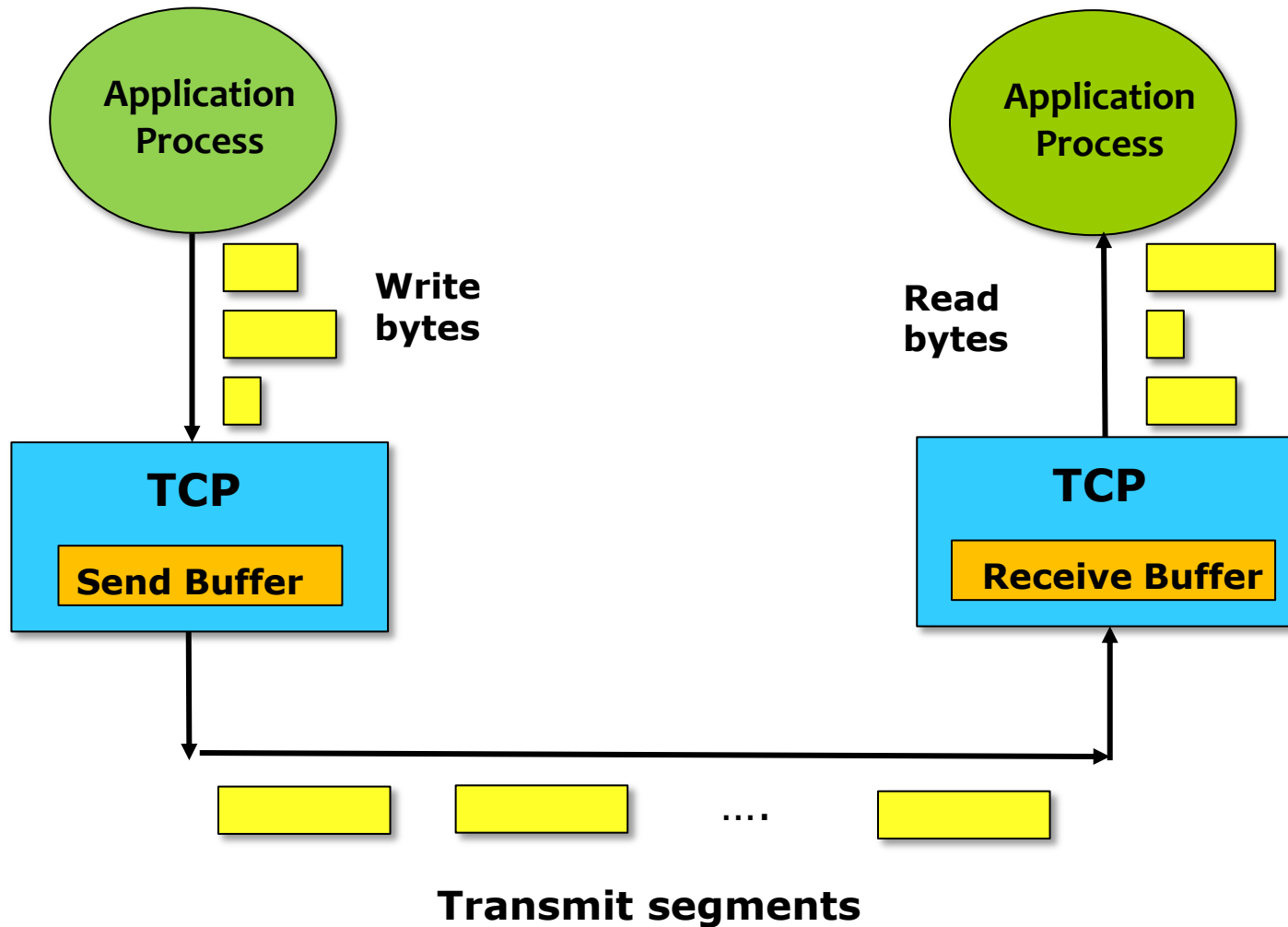
TCP Segment

- TCP is a **byte-oriented protocol**
- The **sender writes bytes** into a TCP connection and the **receiver reads bytes** out of the TCP connection.
- However, TCP does **not transmit individual bytes** over the Internet.

TCP Segment

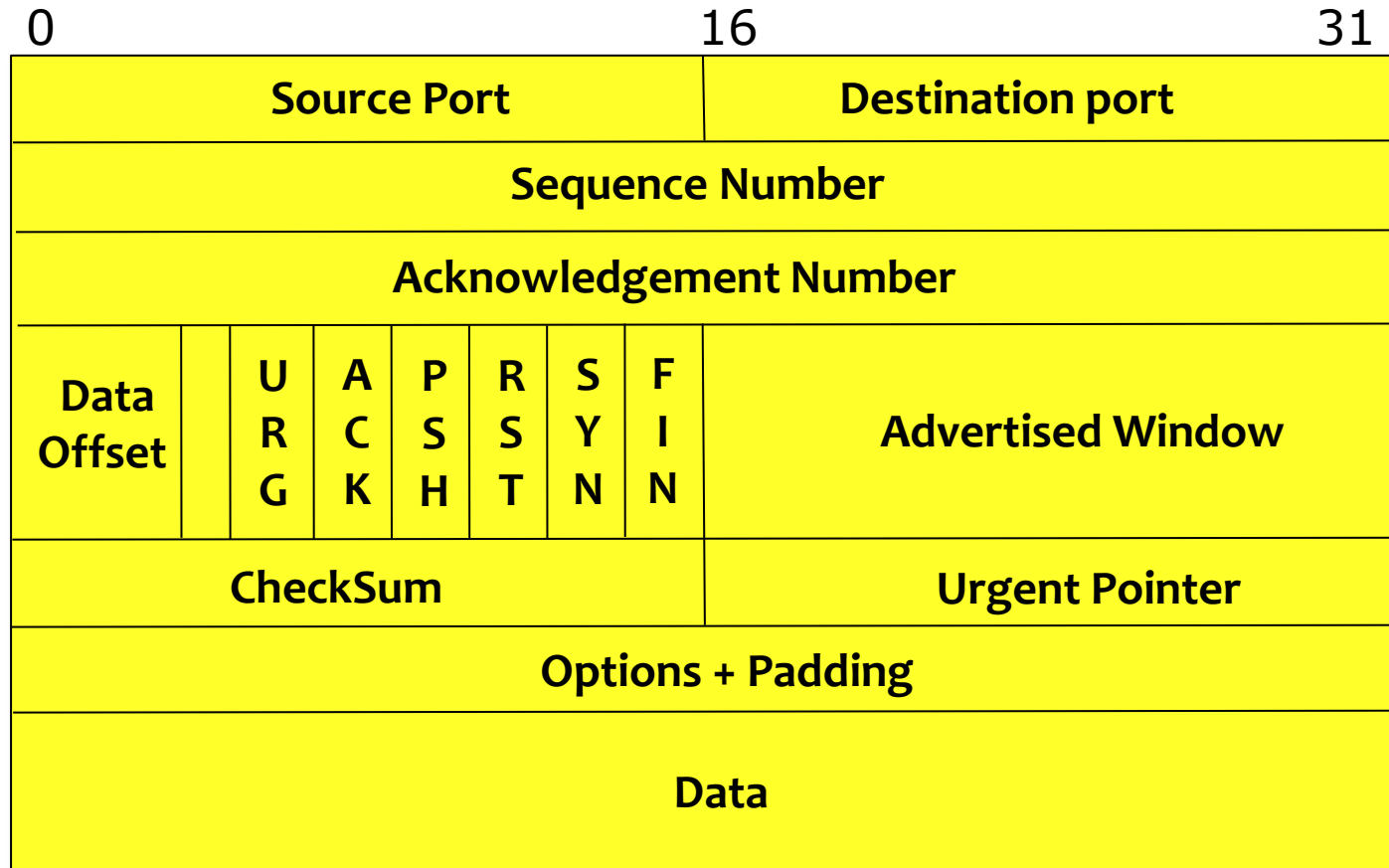
- The source TCP **buffers enough bytes from the sending process** to fill a reasonably sized packet and then sends this packet to its peer on the destination host.
- The destination TCP then puts the contents of the packet into a **receive buffer**, and the receiving process reads from this buffer.
- The packets exchanged between TCP peers are called **segments**.

TCP Segment



How TCP manages a byte stream.

TCP Header



TCP Header Format

TCP Header

- The **SrcPort** and **DstPort** fields identify the source and destination ports, respectively.
- The **Acknowledgment**, **SequenceNum**, and **AdvertisedWindow** fields are all involved in TCP's sliding window algorithm.
- Because 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 that segment.
- The Acknowledgment and AdvertisedWindow fields carry **information about the flow of data** going in the other direction.

TCP Header

- The 6-bit Flags field is used to **relay control information** between TCP peers.
- The possible flags include **SYN, FIN, RESET, PUSH, URG,** and **ACK**.
- The **SYN and FIN flags** are used when establishing and terminating a TCP connection, respectively.
- The **ACK flag** is set any time the Acknowledgment field is valid, implying that the receiver should pay attention to it.

TCP Header

- The **URG flag** signifies that this segment contains urgent data. When this flag is set, the **UrgPtr field** indicates where the nonurgent data contained in this segment begins.
- The urgent data is contained **at the front of the segment** body, up to and including a value of UrgPtr bytes into the segment.
- The **PUSH flag** signifies that the sender invoked the **push operation**, which indicates to the receiving side of TCP that it should notify the receiving process of this fact.

TCP Header

- The **RESET flag** signifies that the receiver has become confused, it received a segment it did not expect to receive—and so wants to abort the connection.
- Finally, the **Checksum field** is used in exactly the same way as for UDP—it is computed over the TCP header, the TCP data, and the **pseudoheader**, which is made up of the source address, destination address, and length fields from the **IP header**.

TCP Connection Management

- TCP sender, receiver **establish “connection”** before exchanging data segments
- initialize TCP variables:
 - Sequence numbers
 - Buffers, flow control info (e.g. RcvWindow)
- **Client:** connection initiator

```
Socket clientSocket = new Socket("hostname", "port number");
```
- **Server:** contacted by client

```
Socket connectionSocket = welcomeSocket.accept();
```

TCP Connection Management

Three-way handshake:

Step 1: Client sends TCP **SYN** segment to server

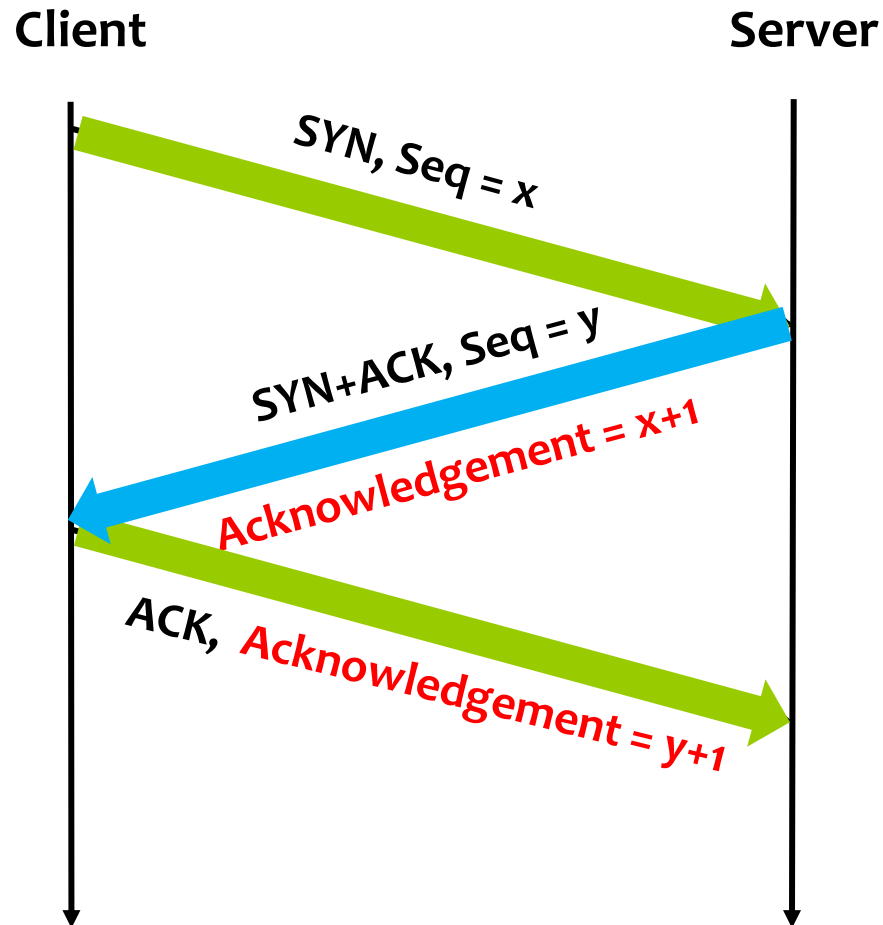
- specifies initial seq #
- no data

Step 2: Server receives SYN, replies with **SYN/ACK** segment

- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYN/ACK, replies with **ACK** segment, which may contain data

Connection Establishment in TCP



Timeline for **three-way handshake** algorithm

TCP Connection Management (cont.)

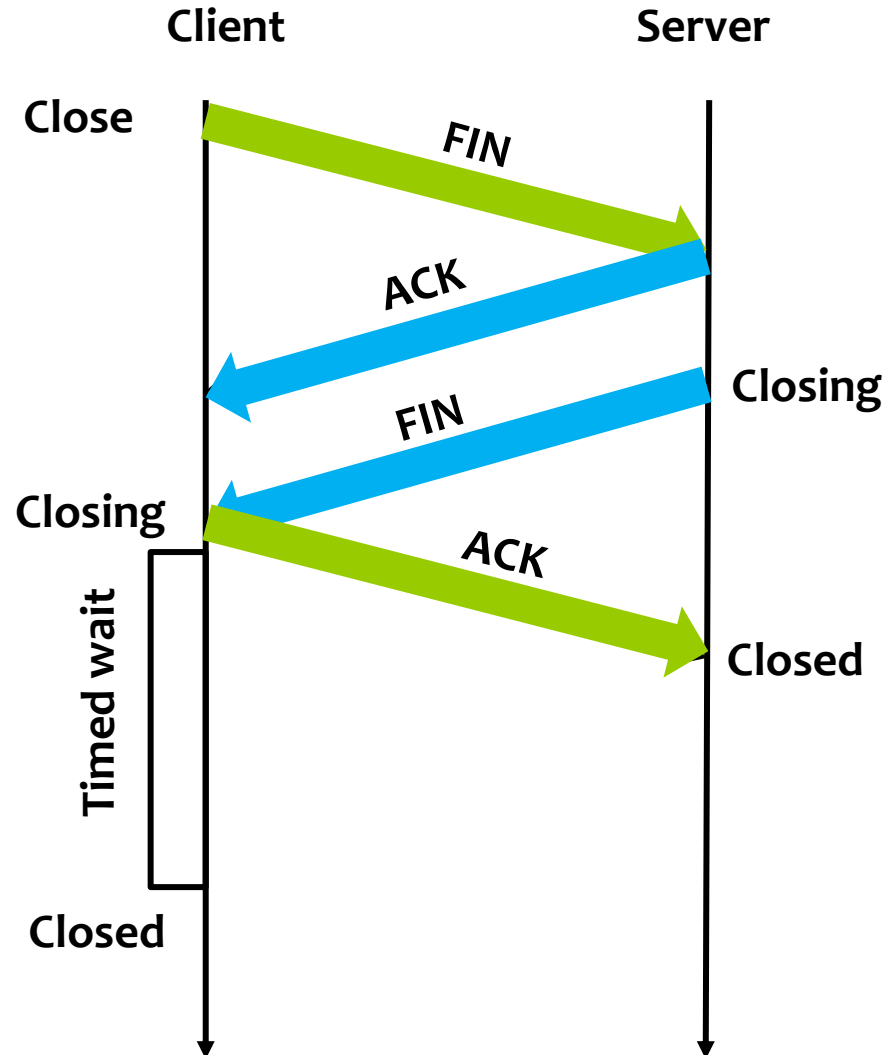
Closing a connection:

client closes socket:

```
clientSocket.close()  
;
```

Step 1: Client sends TCP FIN control segment to server

Step 2: Server receives FIN, replies with ACK. Closes connection, sends FIN.



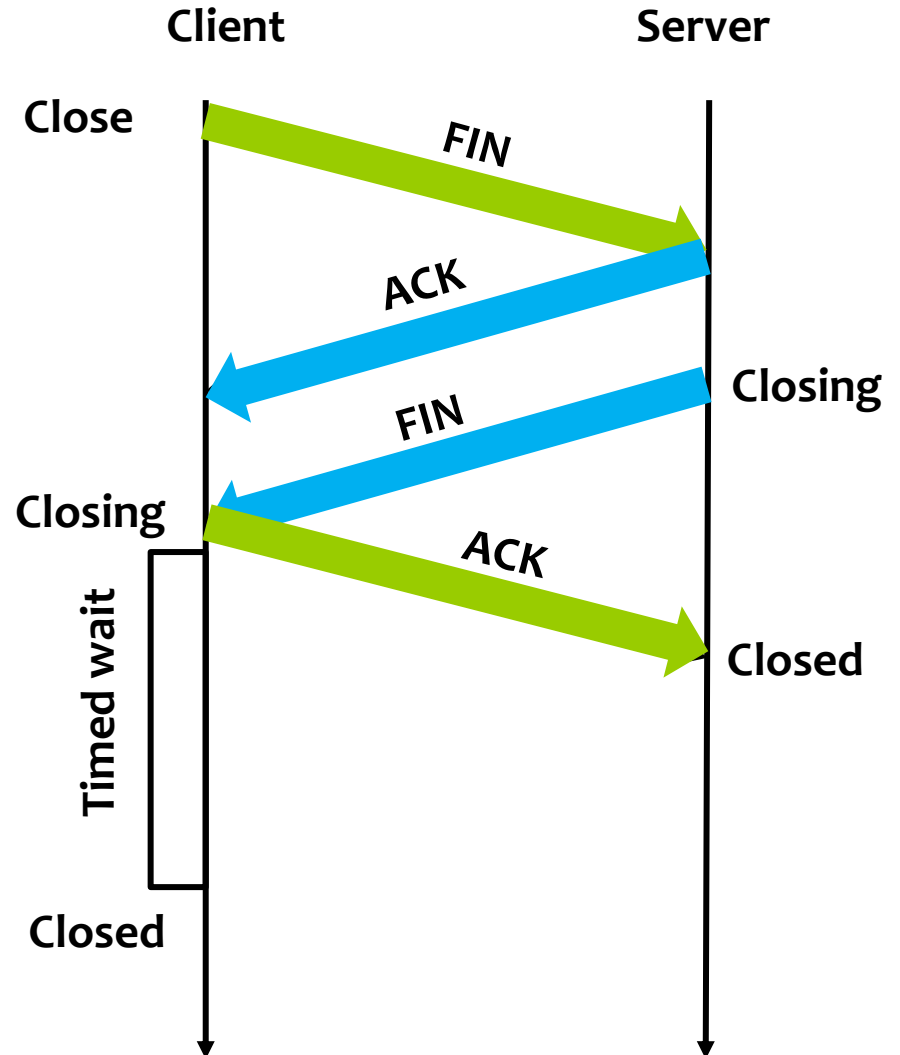
TCP Connection Management (cont.)

Step 3: Client receives FIN, replies with ACK.

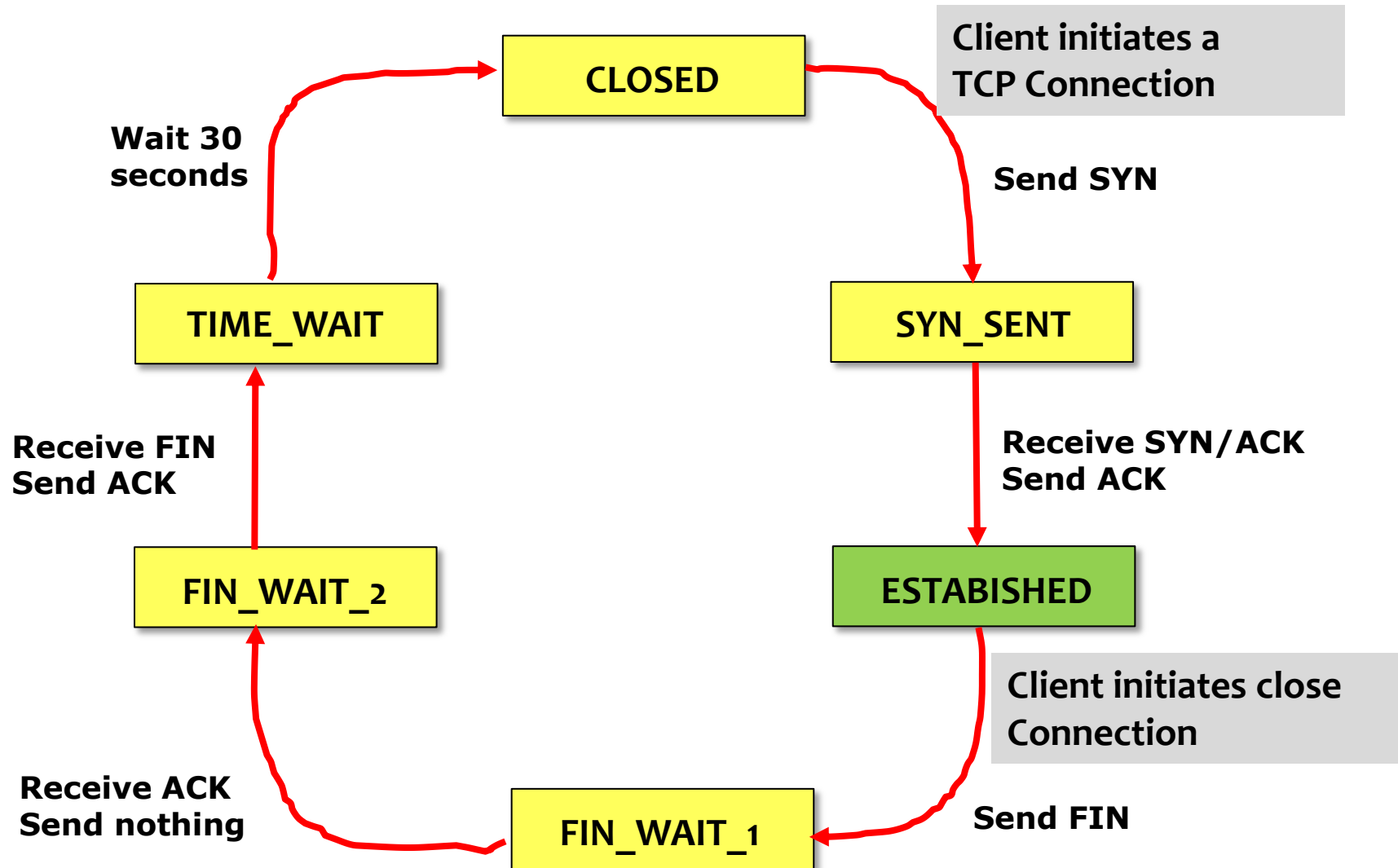
- Enters “timed wait” - will respond with ACK to received FINs

Step 4: Server receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.

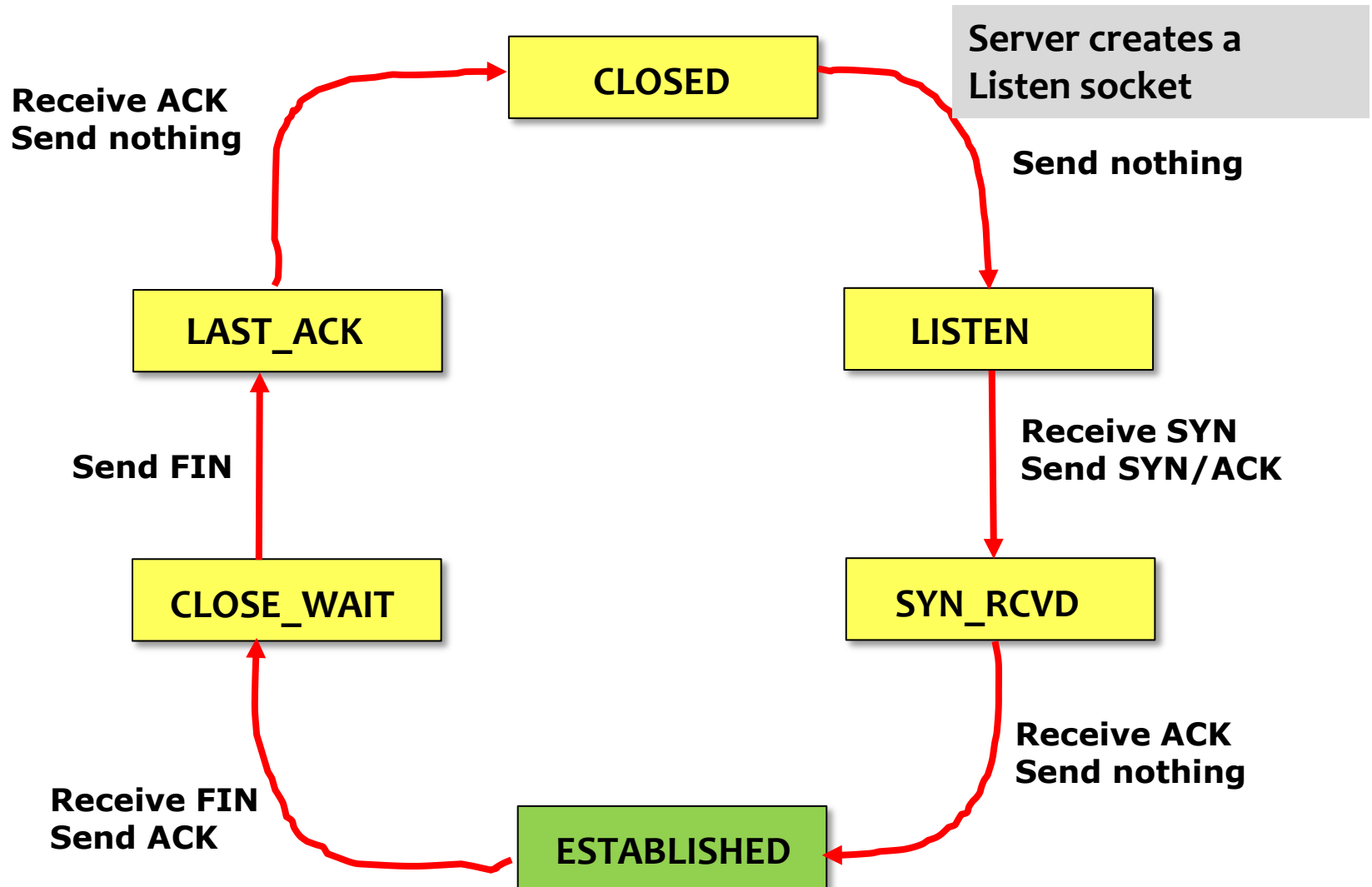


TCP Connection Management (cont)



TCP client state diagram

TCP Connection Management (cont)



TCP server state diagram

Timeout value for Retransmission

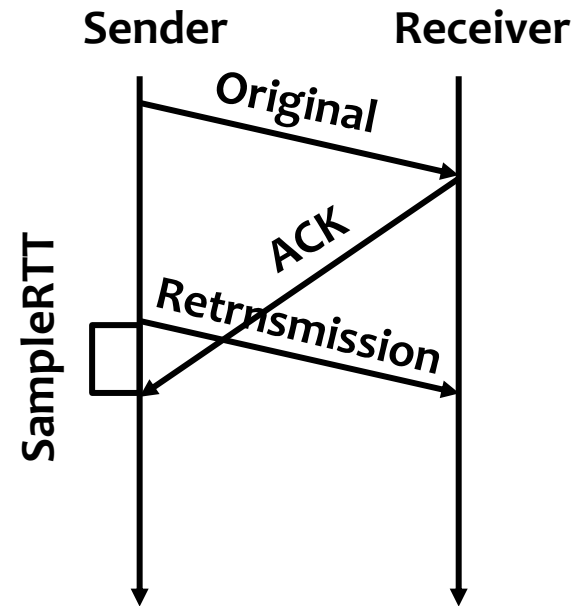
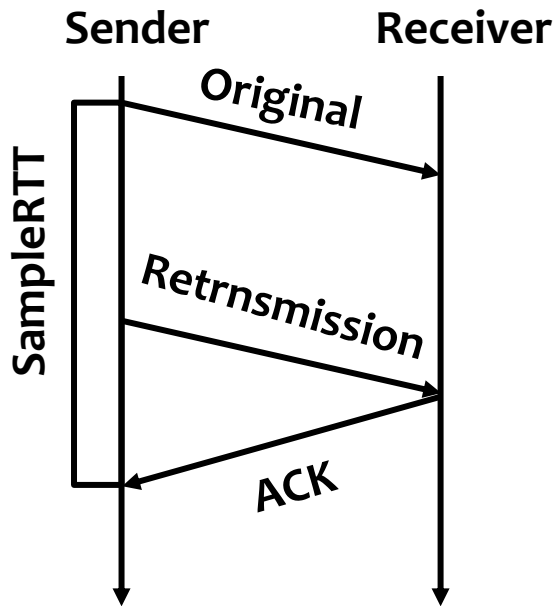
■ Original Algorithm

- Measure **SampleRTT** for each **segment/ ACK pair**
- Compute **weighted average of RTT**
 - ▶ $\text{EstRTT} = \alpha \times \text{EstRTT} + (1 - \alpha) \times \text{SampleRTT}$
 - α between 0.8 and 0.9
- Set timeout based on EstRTT
 - ▶ **TimeOut = 2 x EstRTT**

Timeout value for Retransmission

- Problem of calculating the SampleRTT
 - When a segment is **retransmitted** and then an ACK arrives at the sender
 - ▶ It is impossible to decide if this **ACK should be associated** with the **first** or the **second transmission** for calculating RTTs

Timeout value for Retransmission



Problems of associating the ACK with

(a) original transmission (**should be retransmission**)

(b) retransmission (**should be original**)

Karn/Partridge Algorithm

- **Do not sample RTT** when retransmitting
- **Double timeout** after each retransmission
- Karn-Partridge algorithm was an improvement over the original approach, but it does not eliminate congestion
- We need to understand **how timeout is related to congestion**
 - If you timeout too soon, you may unnecessarily retransmit a segment which adds load to the network

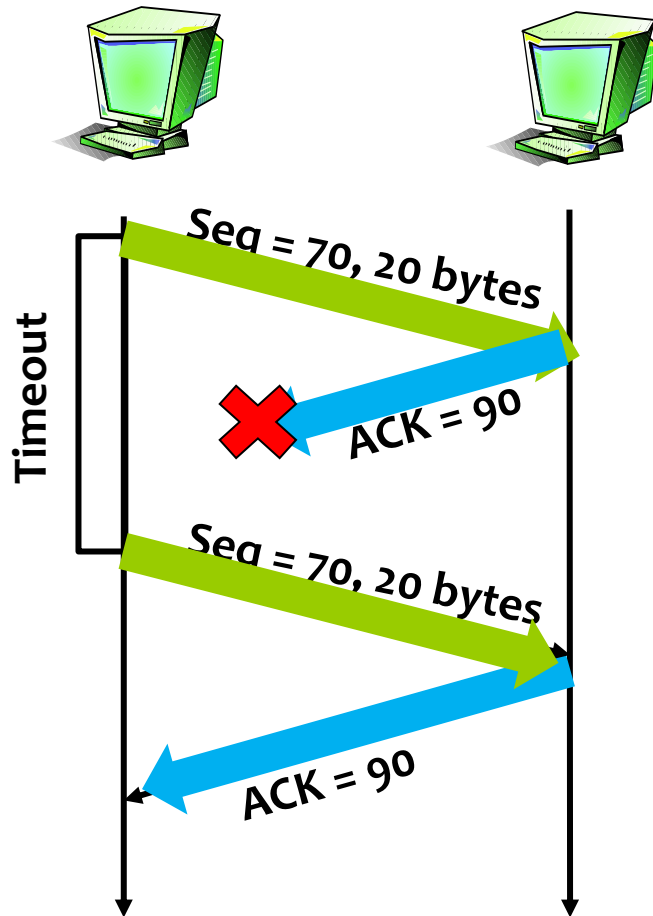
Karn/Partridge Algorithm

- Main problem with the original computation is that it does **not take variance of SampleRTTs** into consideration.
- For **small variance** among SampleRTTs
 - Then the EstimatedRTT can be better trusted
 - There is no need to multiply this by 2 to compute the timeout
- For **large variance** among SampleRTTs
 - The timeout value should not be tightly coupled to the Estimated RTT

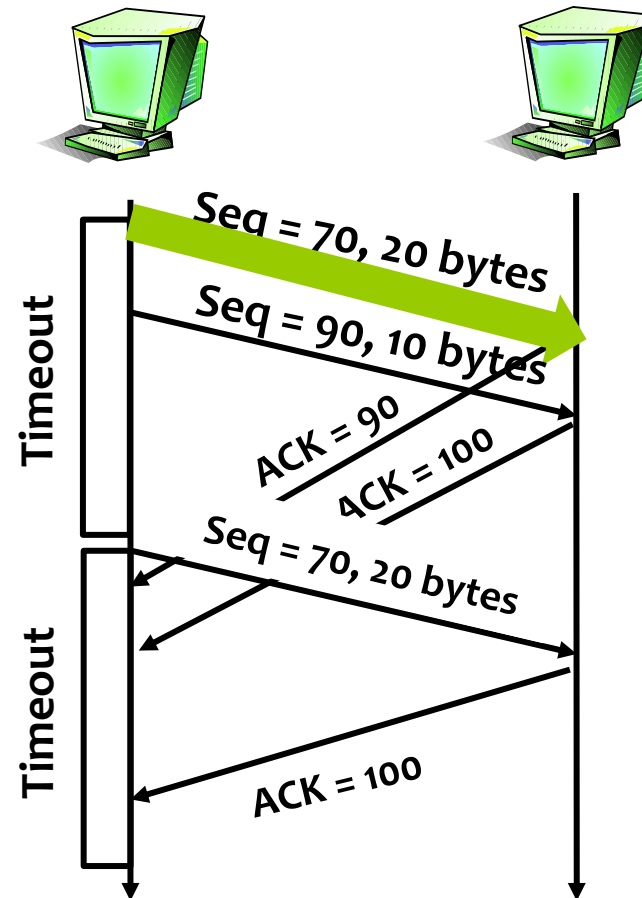
Jacobson/Karels Algorithm

- **Jacobson/Karels** proposed a new scheme for TCP retransmission
- $\text{Difference} = \text{SampleRTT} - \text{EstimatedRTT}$
- $\text{EstimatedRTT} = \text{EstimatedRTT} + (\delta \times \text{Difference})$
- $\text{Deviation} = \text{Deviation} + \delta (|\text{Difference}| - \text{Deviation})$
 - where δ is a factor between 0 and 1
- $\text{TimeOut} = \mu \times \text{EstimatedRTT} + \phi \times \text{Deviation}$
 - where based on experience, $\mu = 1$ and $\phi = 4$.
 - Thus, when the variance is small, TimeOut is close to EstimatedRTT; a large variance causes the deviation term to dominate the calculation.

TCP retransmission scenarios

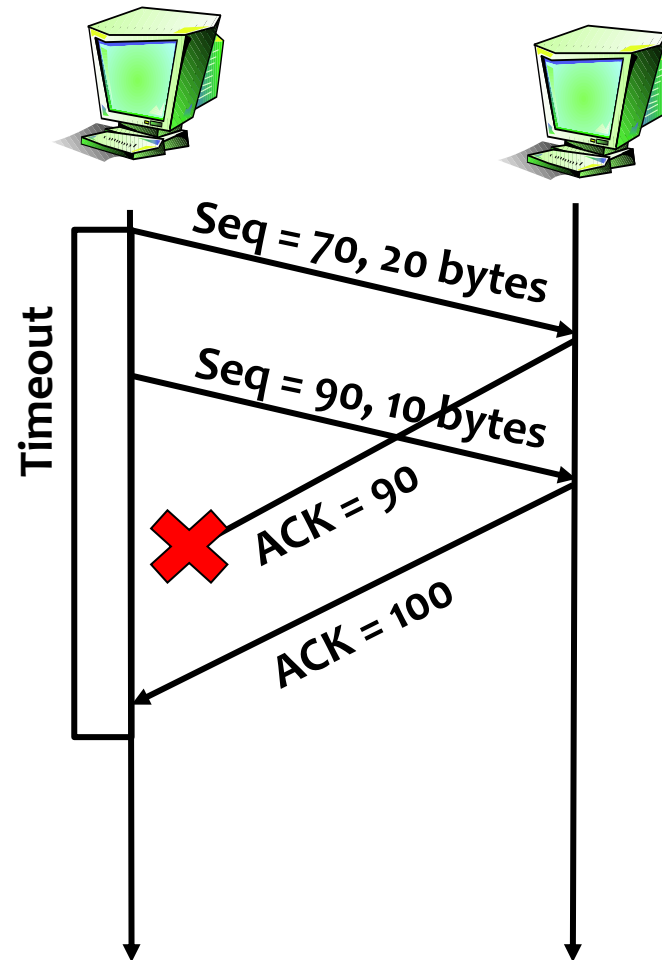


Lost ACK



Delayed ACK

TCP retransmission scenarios (more)



Cumulative ACKs

TCP Fast Retransmission

■ Fast Retransmit

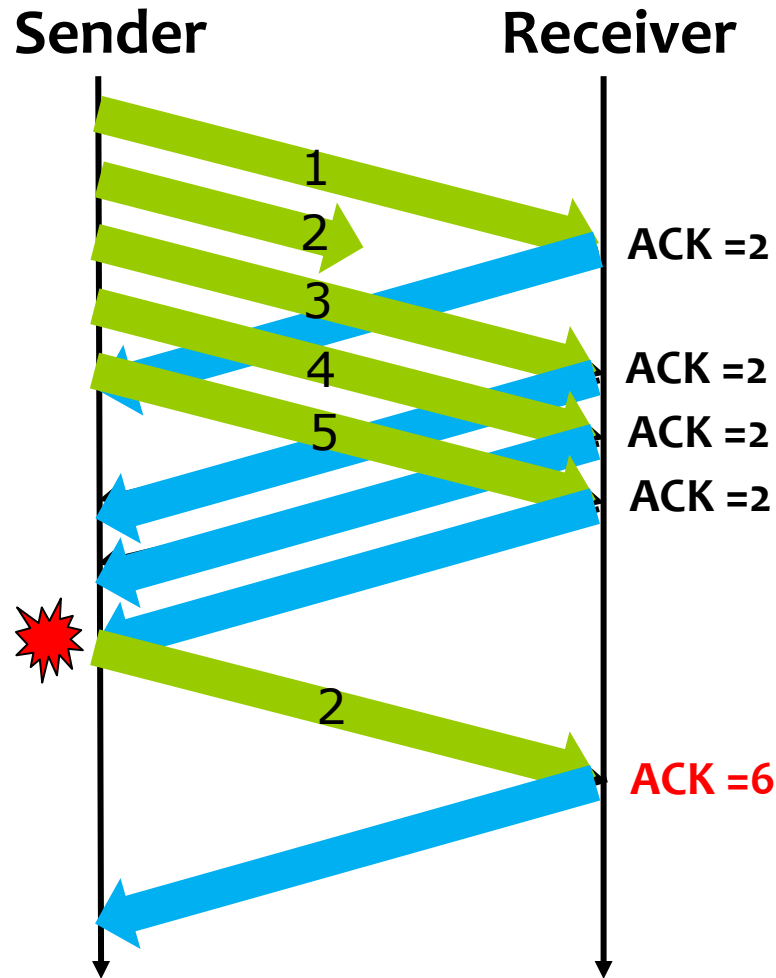
- Every time a data packet arrives at the receiving side, the **receiver responds with an acknowledgment**, even if this sequence number has already been acknowledged.
- Thus, **when a packet arrives out of order—TCP resends the same acknowledgment it sent last time.**
- This second transmission of the same acknowledgment is called a ***duplicate ACK***.

TCP Fast Retransmission

■ Fast Retransmit

- When the sending side sees a duplicate ACK, it knows that the other side must have received a packet out of order, which suggests that **an earlier packet might have been lost**.
- Since it is also possible that the earlier packet has only been delayed rather than lost, the sender waits until it **sees some number of duplicate ACKs** and then retransmits the missing packet.
- TCP waits until it has seen **three duplicate ACKs** before retransmitting the packet.

TCP Fast Retransmission



TCP Congestion Control

- The idea of TCP congestion control is for each source to determine **how much capacity is available in the network**, so that it knows how many packets it can safely have in transit.
- TCP is said to be **self-clocking** by **using ACKs** to pace the transmission of packets.

TCP Congestion Control

■ Additive Increase Multiplicative Decrease (AIMD)

- **CongestionWindow**: used by the source to limit how much data it is allowed to have in transit simultaneously for a connection.
- The **congestion window** is congestion control's counterpart to flow control's **advertised window**.
- The maximum number of bytes of unacknowledged data allowed is now **the minimum of the congestion window and the advertised window**
- Transmission rate

$$\text{Rate} = \frac{\text{CongestionWindow}}{\text{RTT}} \text{ Bytes/sec}$$

TCP Congestion Control

- Additive Increase Multiplicative Decrease (AIMD)
 - TCP's **effective window** is revised as follows:
 - ▶ **MaxWindow** = $\text{MIN}(\text{CongestionWindow}, \text{AdvertisedWindow})$
 - ▶ **EffectiveWindow** = $\text{MaxWindow} - (\text{LastByteSent} - \text{LastByteAked})$.
 - A TCP source is allowed to send **no faster than the slowest component** can accommodate
 - ▶ the network or
 - ▶ the destination host

TCP Congestion Control

- Additive Increase Multiplicative Decrease (AIMD)
 - How to determine the value for **CongestionWindow** ?
 - The **AdvertisedWindow** is sent by the receiver.
 - But no one to send a suitable CongestionWindow to the sending side of TCP.
 - TCP source sets the CongestionWindow **based on the congestion level it observed**.
 - ▶ **Decreasing the congestion window** when the level of congestion goes up and
 - ▶ **Increasing the congestion window** when the level of congestion goes down.
 - Called ***additive increase/multiplicative decrease (AIMD)***

TCP Congestion Control

■ Additive Increase Multiplicative Decrease (AIMD)

- How does the source determine that the **network is congested** and that it should decrease the congestion window?
 - ▶ TCP interprets packet loss (3-duplicate ACK) **as a sign of congestion** and reduces the rate.
 - ▶ Each time a packet loss occurs, the source **sets CongestionWindow to half** of its previous value.
 - ▶ This corresponds to the “**multiplicative decrease**” part of AIMD.

TCP Congestion Control

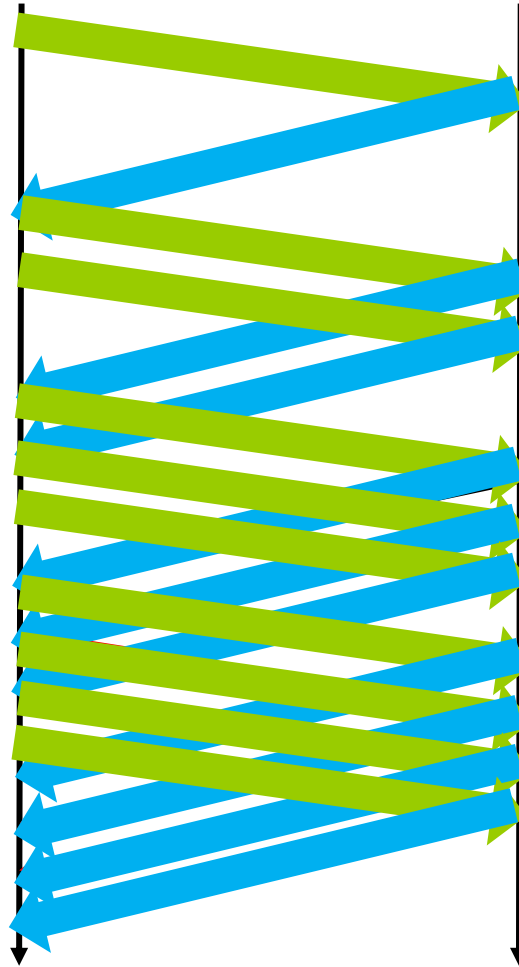
- Additive Increase Multiplicative Decrease (AIMD)
 - Although CongestionWindow is defined in terms of bytes, it is easiest to understand **multiplicative decrease** if we think in terms of whole packets.
 - ▶ For example, suppose the CongestionWindow is currently set to **16 packets**. If a loss is detected, CongestionWindow is set to **8**.
 - ▶ Additional losses cause CongestionWindow to be reduced to **4**, then **2**, and finally to **1 packet**.
 - ▶ CongestionWindow is not allowed to fall below the size of a single packet, or in TCP terminology, the **maximum segment size (MSS)**.

TCP Congestion Control

- Additive Increase Multiplicative Decrease (AIMD)
 - Increase the congestion window when the **newly capacity** of network is available.
 - Every time the source successfully sends a CongestionWindow's worth of packets (all **packets sent out during the last RTT have been ACKed**), it adds the equivalent of 1 packet to CongestionWindow.

TCP Congestion Control

■ Additive Increase Multiplicative Decrease (AIMD)



One more packet is added for each RTT

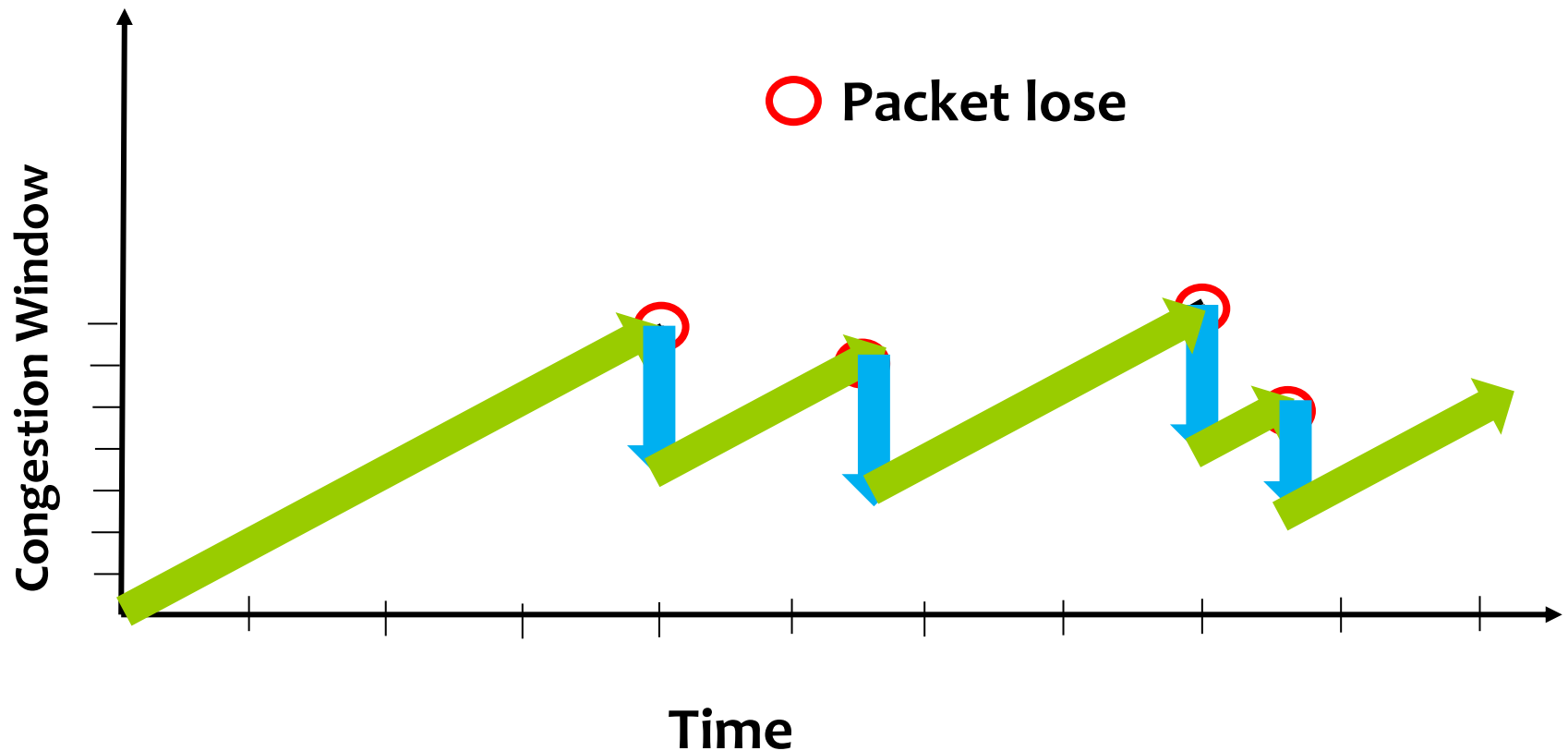
TCP Congestion Control

■ Additive Increase Multiplicative Decrease (AIMD)

- TCP does not wait for an entire window's worth of ACKs to add 1 packet's worth to the congestion window, but instead **increments CongestionWindow by a little for each ACK**.
- 不等所有 ACK 都收到才加 1 (MSS bytes), 每收到一個 ACK 就先加一部分 (CW 可以較早滑動)
- 例如
 - ▶ $CW = 5 \times MSS$, 每收到一個 ACK 就先加 $1/5 MSS$
 - ▶ $CW = 8 \times MSS$, 每收到一個 ACK 就先加 $1/8 MSS$
- **$Increment = MSS \times (MSS / CongestionWindow)$**
- **$CongestionWindow += Increment$**

TCP Congestion Control

- Additive Increase Multiplicative Decrease (AIMD)
- Trace: Sawtooth behavior



TCP Congestion Control

■ Slow Start

- The **additive increase** mechanism is good when the source is operating close to the available capacity of the network, but it **takes too long to ramp up a connection when it is starting from scratch.**
- **Slow start:** to increase the congestion window rapidly from a cold start.
- Slow start effectively increases the congestion window **exponentially, rather than linearly.**

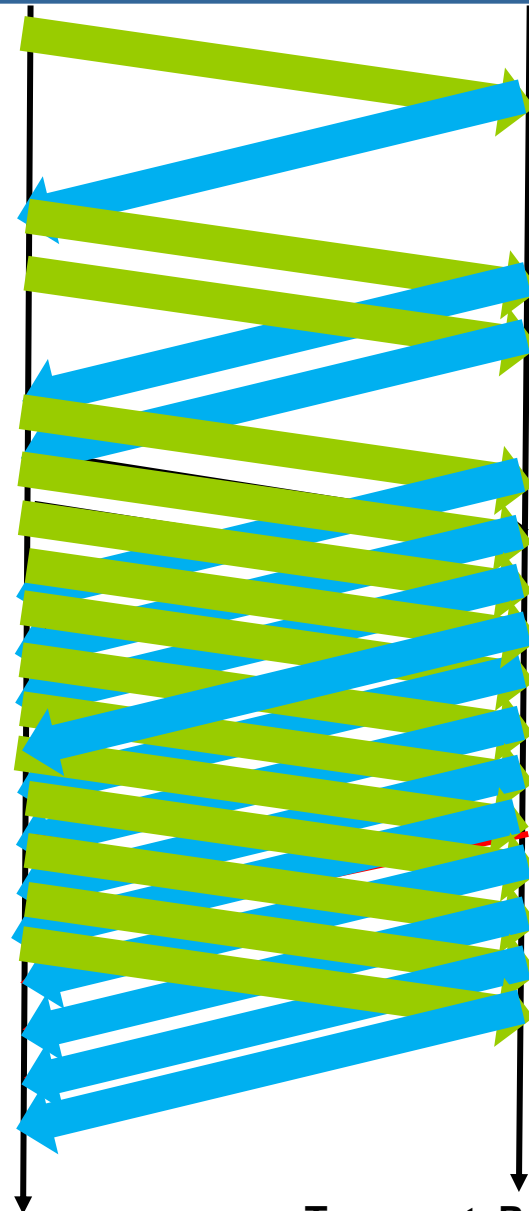
TCP Congestion Control

■ Slow Start

- Initially, the CongestionWindow = 1 packet.
- Example: MSS = 500 bytes, RTT = 200 msec
 - ▶ initial rate = 20 kbps
- When the ACK for this packet arrives, TCP adds 1 to CongestionWindow and then sends two packets.
- 每收到一個 ACK 就加 1 packet (MSS)
- Upon receiving the corresponding two ACKs, TCP increments CongestionWindow by 2— **one for each ACK**—and next sends four packets.
- TCP effectively **doubles the number of packets it has in transit every RTT**.

TCP Congestion Control (Slow Start)

- **Slow Start**
- When connection begins, increase rate exponentially until first loss event:
 - double CongWin every RTT
 - done by incrementing CongWin for every ACK received
- initial rate is slow but ramps up exponentially fast



Packets in transit during slow start

TCP Congestion Control

■ After 3 dup ACKs:

- CongWin is cut in half
- window then grows linearly

■ But after timeout event:

- CongWin instead set to 1 MSS;
- window then grows exponentially
- to a threshold, then grows linearly

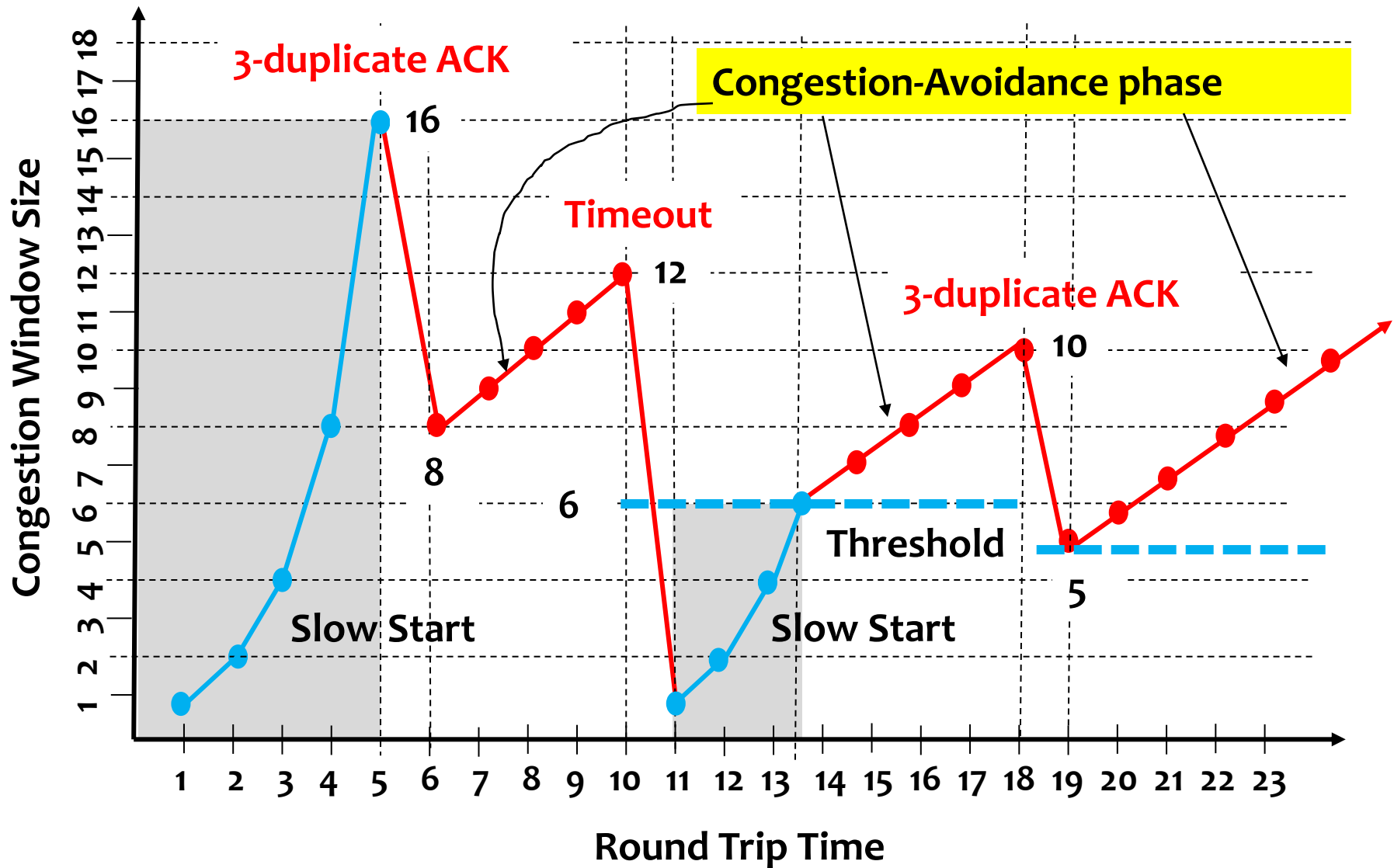
note

- 3 dup ACKs indicates network capable of delivering some segments
- Timeout indicates a “more alarming” congestion scenario

TCP Congestion Control

- Summary :
- When CongWin is below Threshold, sender in **slow-start phase**, window grows exponentially.
- When CongWin is above Threshold, sender is in **congestion-avoidance phase**, window grows linearly.
- When a **triple duplicate ACK** occurs, Threshold set to CongWin/2 and CongWin set to Threshold.
- When **timeout** occurs, Threshold set to CongWin/2 and CongWin is set to 1 MSS.

TCP Congestion Control



TCP Sender Congestion Control

State	Event	TCP Sender Action	Commentary
Slow Start (SS)	ACK receipt for previously unacked data	$\text{CongWin} = \text{CongWin} + \text{MSS}$, If ($\text{CongWin} > \text{Threshold}$) set state to "Congestion Avoidance"	Resulting in a doubling of CongWin every RTT
Congestion Avoidance (CA)	ACK receipt for previously unacked data	$\text{CongWin} = \text{CongWin} + \text{MSS} * (\text{MSS} / \text{CongWin})$	Additive increase, resulting in increase of CongWin by 1 MSS every RTT
SS or CA	Loss event detected by triple duplicate ACK	$\text{Threshold} = \text{CongWin} / 2$, $\text{CongWin} = \text{Threshold}$, Set state to "Congestion Avoidance"	Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.
SS or CA	Timeout	$\text{Threshold} = \text{CongWin} / 2$, $\text{CongWin} = 1 \text{ MSS}$, Set state to "Slow Start"	Enter slow start
SS or CA	Duplicate ACK	Increment duplicate ACK count for segment being acked	CongWin and Threshold not changed

TCP throughput

- What's the average throughput of TCP as a function of **window size** and **RTT** ?
 - Ignore slow start
- Let W be the window size when loss occurs.
- When window is W , throughput is W/RTT
- Just after loss, window drops to $W/2$, throughput to $W/2RTT$.
- Average throughput: **$0.75 W/RTT$**

Summary

- We have introduced how to convert host-to-host packet delivery service to process-to-process communication channel.
- UDP for unreliable transmission service
- TCP for reliable transmission service
 - 3-way handshaking connection establishment
 - TCP connection state diagram
 - TCP timeout value calculation
 - TCP retransmission scenarios
 - TCP fast retransmission

Summary

- **TCP Congestion Control**
 - ▶ **AIMD** (additive Increase Multiplicative Decrease)
 - ▶ **Slow start**
 - ▶ **3-duplicate ACKs** (packet lose)
 - ▶ **Timeout**