

# TCP

## Congestion Control

Lecture material taken from  
“Computer Networks *A Systems Approach*”,  
Third Ed., Peterson and Davie,  
Morgan Kaufmann, 2003.

# TCP Congestion Control

- **Essential strategy** :: The TCP host sends packets into the network without a reservation and then the host reacts to observable events.
- Originally TCP assumed FIFO queuing.
- **Basic idea** :: each source determines how much capacity is available to a given flow in the network.
- **ACKs** are used to *'pace'* the transmission of packets such that TCP is “self-clocking”.

# AIMD

## (Additive Increase / Multiplicative Decrease)

- CongestionWindow (cwnd) is a variable held by the TCP source for each connection.

MaxWindow :: min (**CongestionWindow** , AdvertisedWindow)

EffectiveWindow = MaxWindow – (LastByteSent - LastByteAacked)

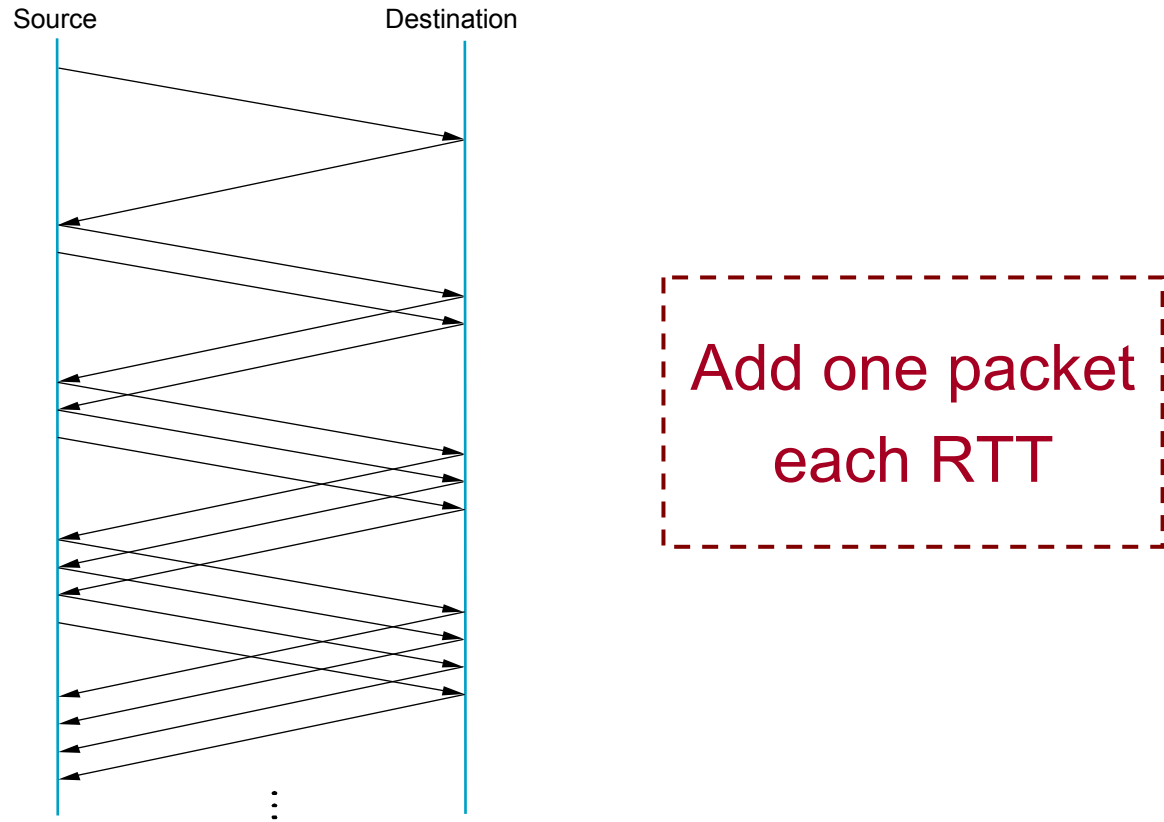
- **cwnd** is set based on the perceived level of congestion. The Host receives *implicit* (packet drop) or *explicit* (packet mark) indications of internal congestion.

# Additive Increase

- Additive Increase is a reaction to perceived available capacity.
- **Linear Increase basic idea**:: For each “cwnd’s worth” of packets sent, increase cwnd by 1 packet.
- In practice, **cwnd** is incremented fractionally for each arriving ACK.

$$\text{increment} = \text{MSS} \times (\text{MSS} / \text{cwnd})$$

$$\text{cwnd} = \text{cwnd} + \text{increment}$$



**Figure 6.8 Additive Increase**

# Multiplicative Decrease

- \* The key assumption is that a dropped packet and the resultant timeout are due to congestion at a router or a switch.

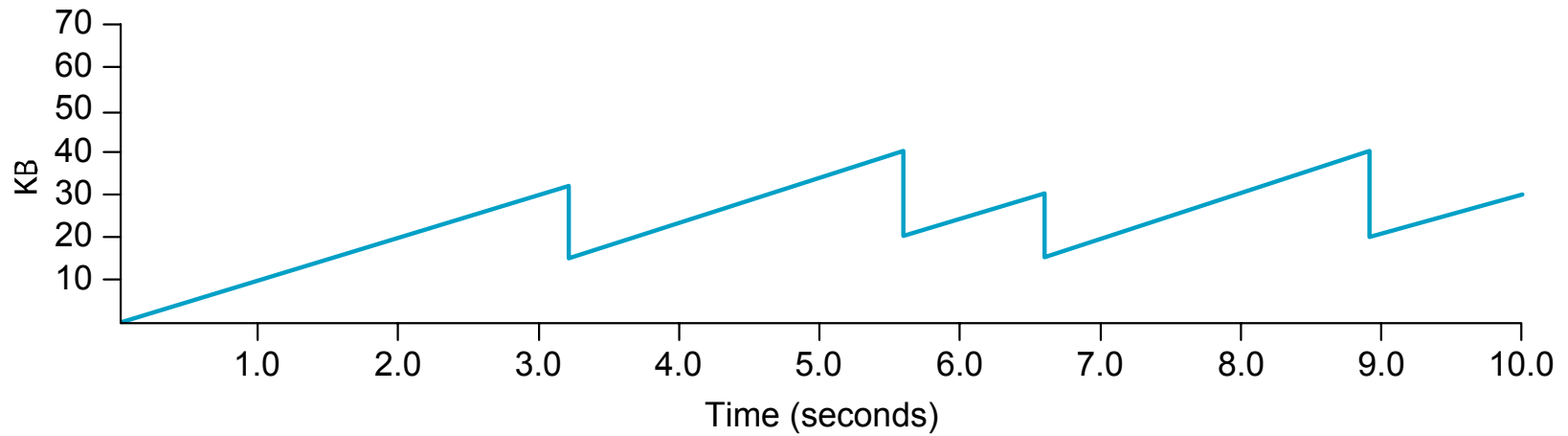
**Multiplicative Decrease::** TCP reacts to a timeout by halving **cwnd**.

- Although **cwnd** is defined in bytes, the literature often discusses congestion control in terms of packets (or more formally in MSS == Maximum Segment Size).
- **cwnd** is not allowed below the size of a single packet.

# AIMD

## (Additive Increase / Multiplicative Decrease)

- It has been shown that AIMD is a necessary condition for TCP congestion control to be stable.
- Because the simple CC mechanism involves timeouts that cause retransmissions, it is important that hosts have an accurate timeout mechanism.
- Timeouts set as a function of average RTT and standard deviation of RTT.
- However, TCP hosts only sample round-trip time once per RTT using coarse-grained clock.



**Figure 6.9 Typical TCP  
Sawtooth Pattern**



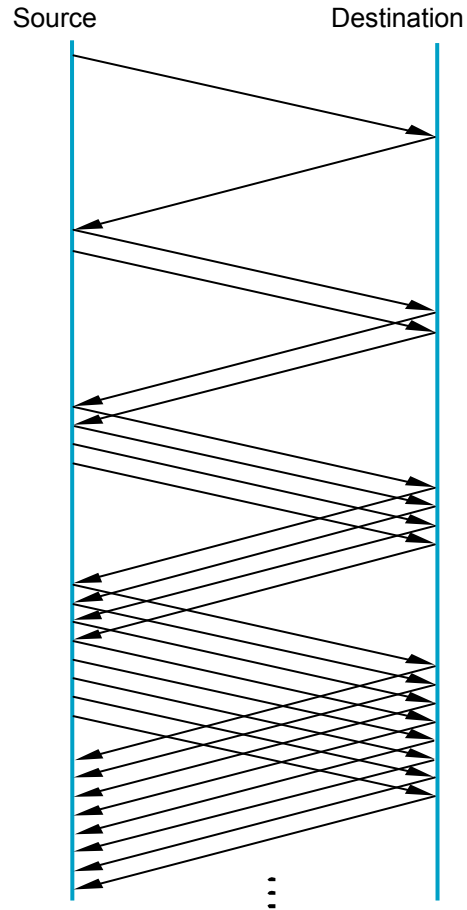
# Slow Start

- Linear additive increase takes too long to ramp up a new TCP connection from cold start.
- Beginning with TCP Tahoe, the **slow start mechanism** was added to provide an initial exponential increase in the size of **cwnd**.

*Remember mechanism by: **slow start prevents a slow start. Moreover, slow start is slower than sending a full advertised window's worth of packets all at once.***

# Slow Start

- The source starts with  $cwnd = 1$ .
- Every time an ACK arrives,  $cwnd$  is incremented.
- $cwnd$  is effectively doubled per RTT “epoch”.
- Two **slow start** situations:
  - At the very beginning of a connection **{cold start}**.
  - When the connection goes dead waiting for a timeout to occur (i.e, the advertized window goes to zero!)

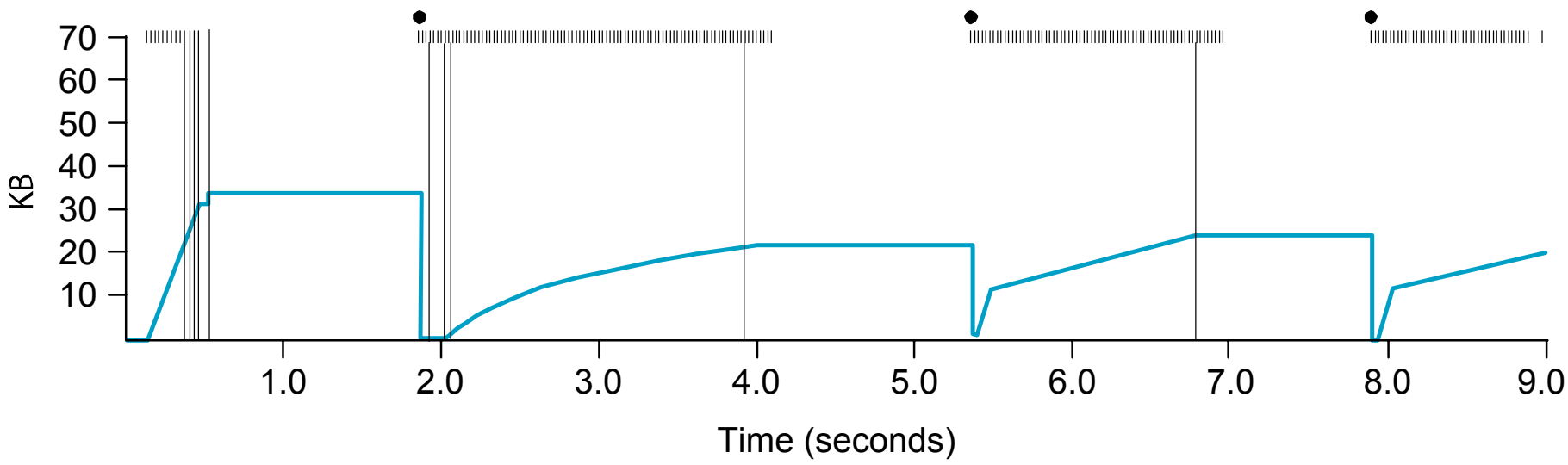


Slow Start  
Add one packet  
per ACK

Figure 6.10 Slow Start

# Slow Start

- However, in the second case the source has more information. The current value of cwnd can be saved as a **congestion threshold**.
- This is also known as the “slow start threshold” **ssthresh**.



**Figure 6.11 Behavior of TCP  
Congestion Control**

# Fast Retransmit

- Coarse timeouts remained a problem, and **Fast retransmit** was added with TCP Tahoe.
- Since the receiver responds every time a packet arrives, this implies the sender will see duplicate ACKs.

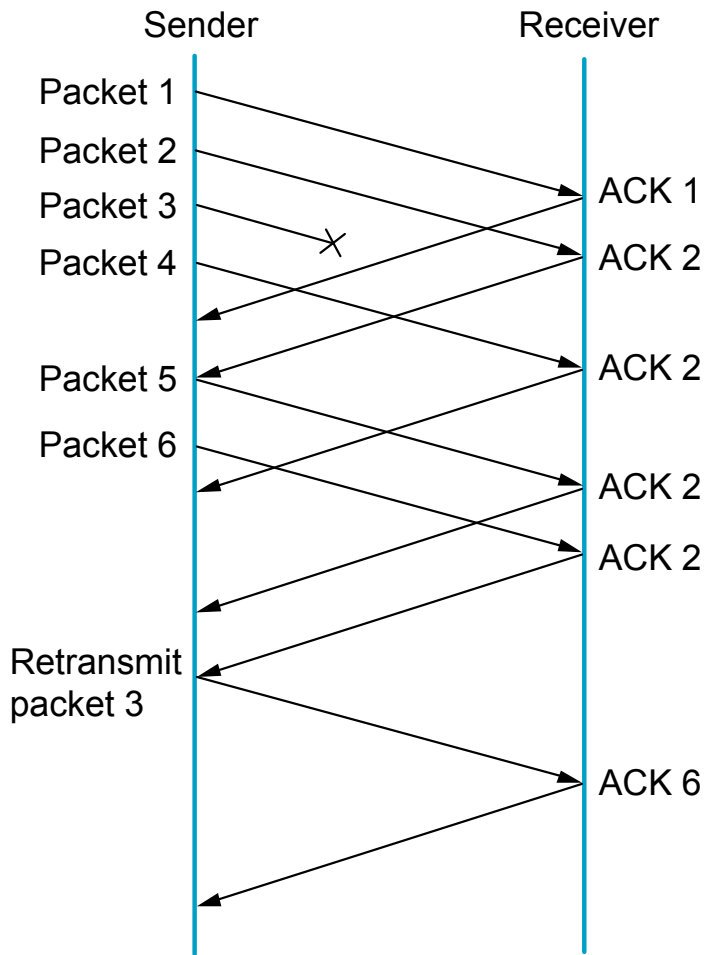
Basic Idea:: *use **duplicate ACKs** to signal lost packet.*

## Fast Retransmit

Upon receipt of *three* duplicate ACKs, the TCP Sender retransmits the lost packet.

# Fast Retransmit

- Generally, **fast retransmit** eliminates about half the coarse-grain timeouts.
- This yields roughly a 20% improvement in throughput.
- Note – **fast retransmit** does not eliminate all the timeouts due to small window sizes at the source.

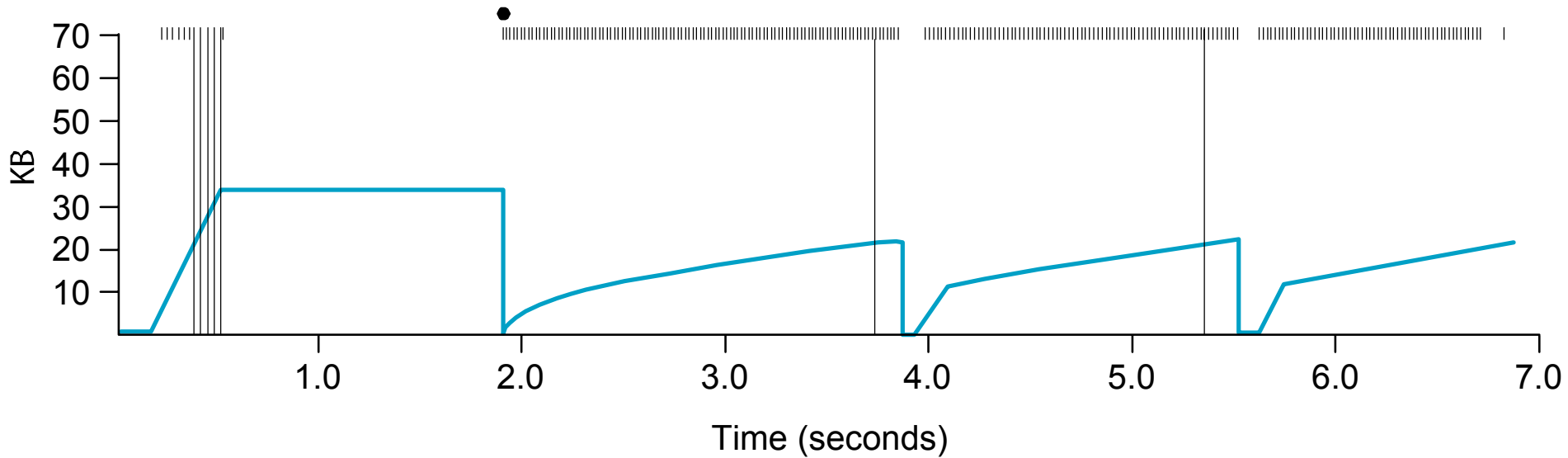


*Fast Retransmit*

Based on three duplicate ACKs

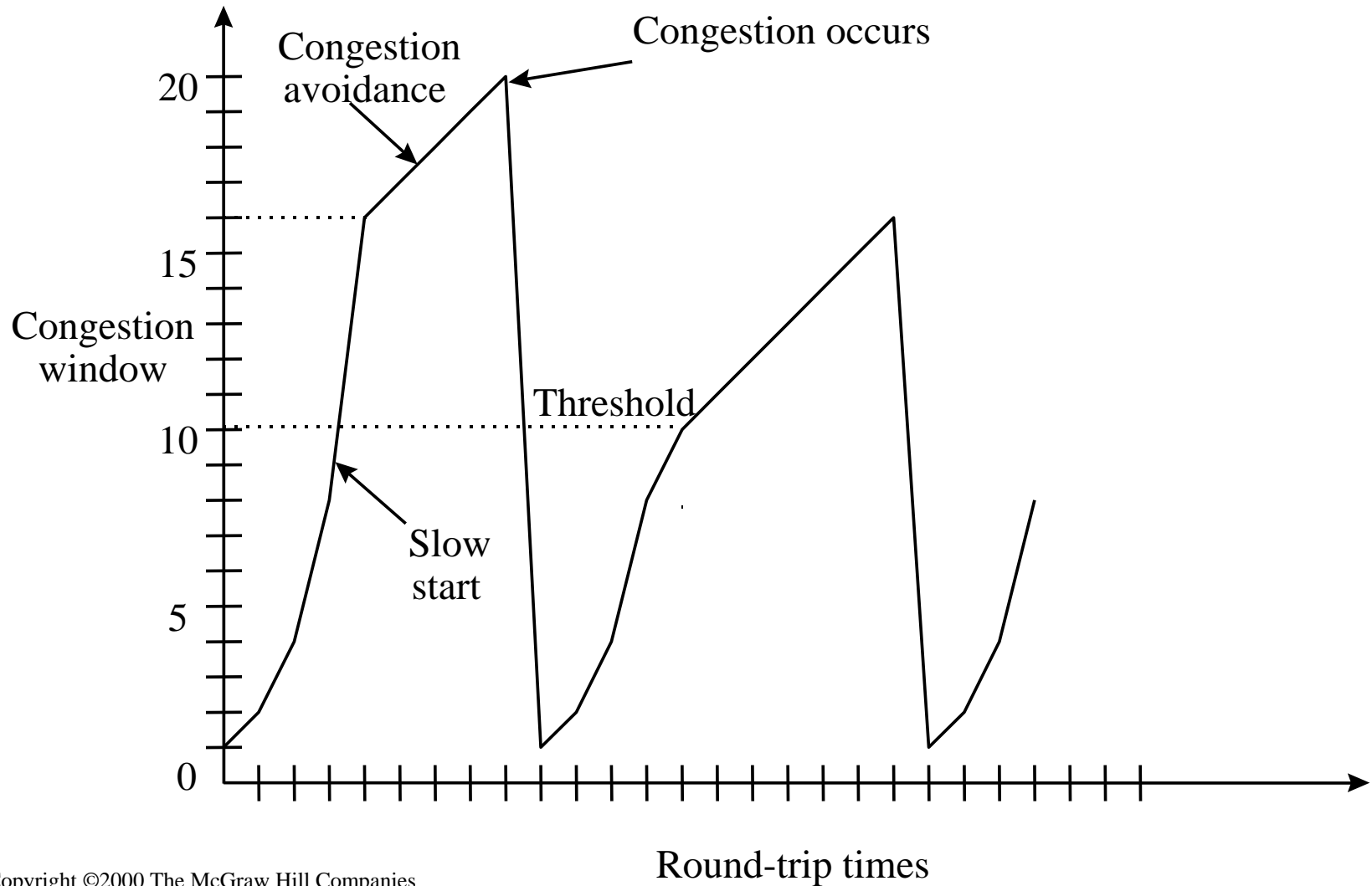
**Figure 6.12 Fast Retransmit**





**Figure 6.13 TCP Fast Retransmit Trace**

# TCP Congestion Control



Copyright ©2000 The McGraw Hill Companies



Leon-Garcia & Widjaja: *Communication Networks*

Advanced Computer Networks: TCP Congestion Control

Figure 7.63

# Fast Recovery

- **Fast recovery** was added with TCP Reno.
- **Basic idea::** When **fast retransmit** detects three duplicate ACKs, start the recovery process from congestion avoidance region and use ACKs in the pipe to pace the sending of packets.

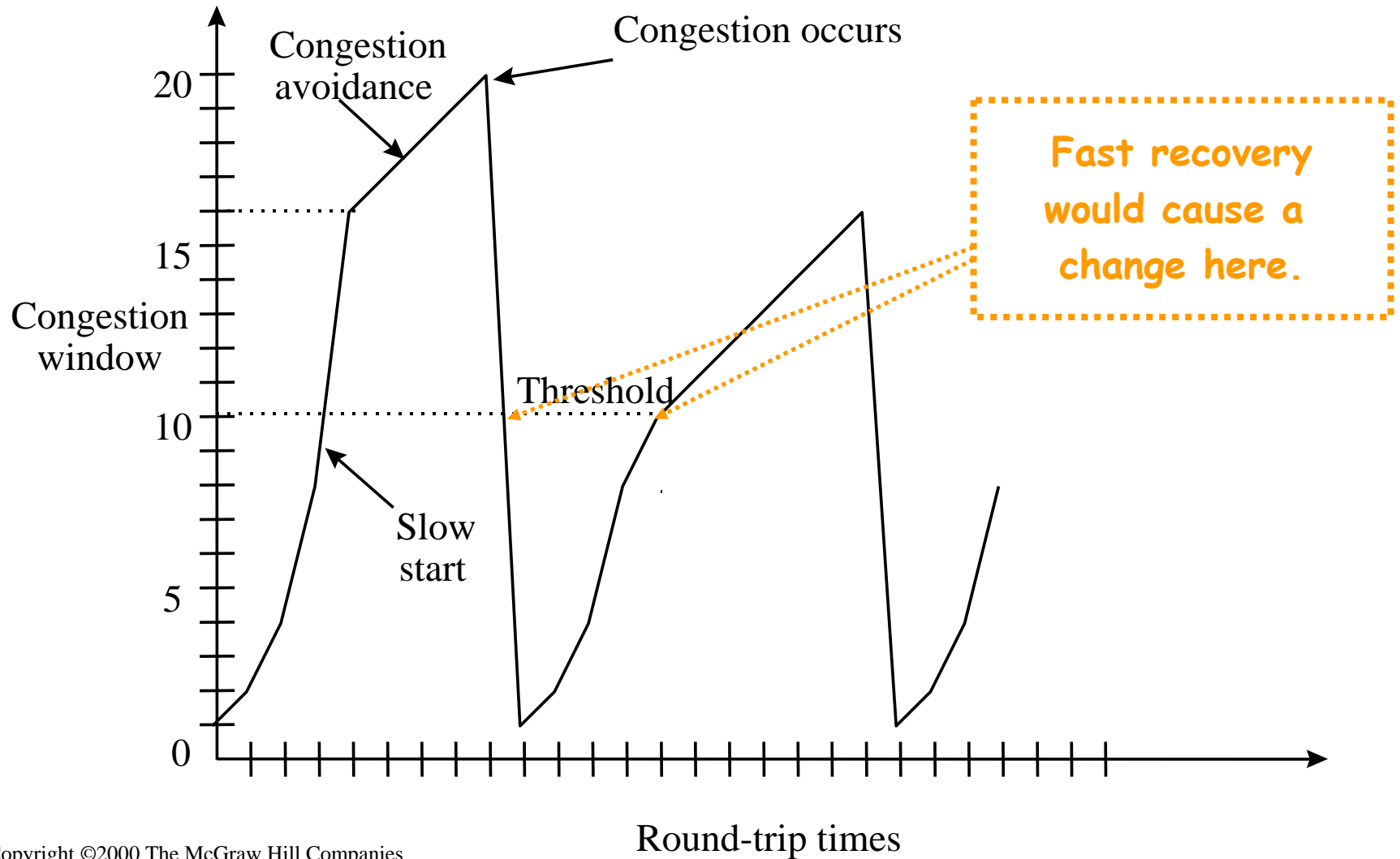
## Fast Recovery

After Fast Retransmit, half **cwnd** and commence recovery from this point using linear additive increase 'primed' by left over ACKs in pipe.

# *Modified Slow Start*

- With **fast recovery**, **slow start** only occurs:
  - At cold start
  - After a coarse-grain timeout
- *This is the difference between TCP Tahoe and TCP Reno!!*

# TCP Congestion Control



Copyright ©2000 The McGraw Hill Companies



Leon-Garcia & Widjaja: *Communication Networks*

Advanced Computer Networks: TCP Congestion Control

Figure 7.63

# TCP Congestion Control



# Adaptive Retransmissions

RTT:: Round Trip Time between a pair of hosts on the Internet.

- How to set the TimeOut value?
  - The timeout value is set as a function of the expected RTT.
  - Consequences of a bad choice?

# Original Algorithm

- Keep a running average of RTT and compute TimeOut as a function of this RTT.
  - Send packet and keep timestamp  $t_s$  .
  - When ACK arrives, record timestamp  $t_a$  .

$$\text{SampleRTT} = t_a - t_s$$



# Original Algorithm

Compute a weighted average:

$$\text{EstimatedRTT} = \alpha \times \text{EstimatedRTT} + (1 - \alpha) \times \text{SampleRTT}$$

Original TCP spec:  $\alpha$  in range (0.8,0.9)

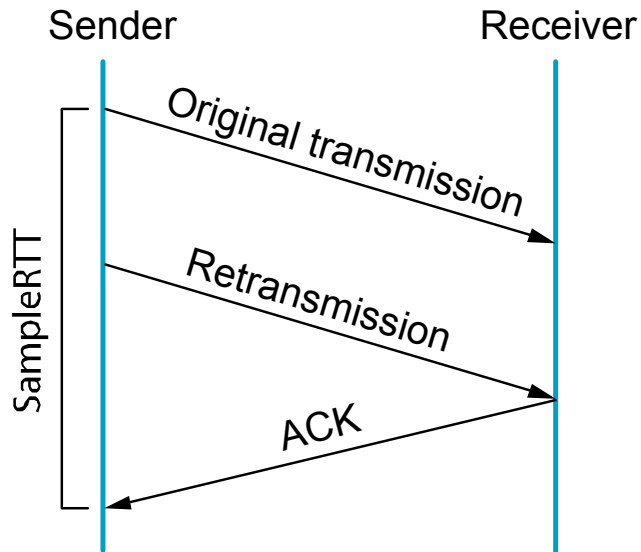
$$\text{TimeOut} = 2 \times \text{EstimatedRTT}$$

# Karn/Partridge Algorithm

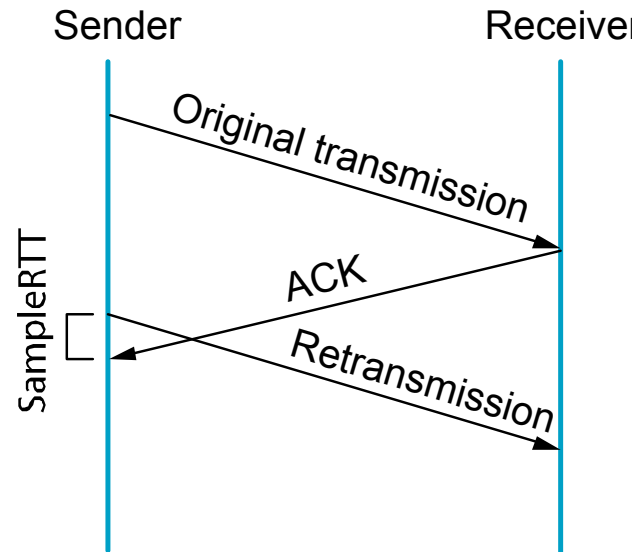
An obvious flaw in the original algorithm:

Whenever there is a retransmission it is impossible to know whether to associate the ACK with the original packet or the retransmitted packet.

# Figure 5.10 Associating the ACK?



(a)



(b)

# Karn/Partidge Algorithm

1. Do not measure **SampleRTT** when sending packet more than once.
2. For each retransmission, set **TimeOut** to **double** the last **TimeOut**.

{ Note – this is a form of exponential backoff based on the believe that the lost packet is due to **congestion**. }

# Jacobson/Karels Algorithm

*The problem with the original algorithm is that it did not take into account the variance of SampleRTT.*

Difference = SampleRTT – EstimatedRTT

EstimatedRTT = EstimatedRTT +

( $\delta$  x Difference)

Deviation =  $\delta$  (|Difference| - Deviation)

where  $\delta$  is a fraction between 0 and 1.

# Jacobson/Karels Algorithm

TCP computes timeout using both the mean and variance of RTT

$$\text{TimeOut} = \mu \times \text{EstimatedRTT} + \Phi \times \text{Deviation}$$

where based on experience  $\mu = 1$  and  $\Phi = 4$ .