

# TCP Congestion Control



# Principles of Congestion Control

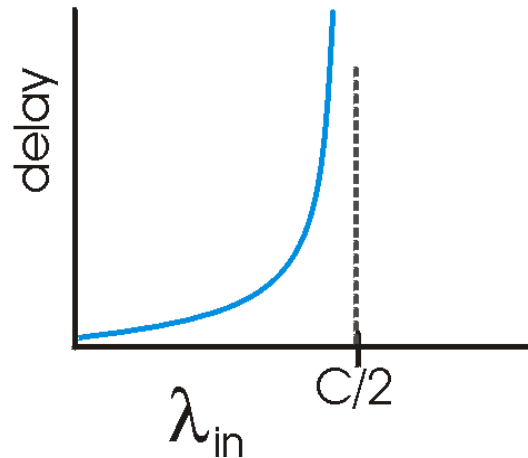
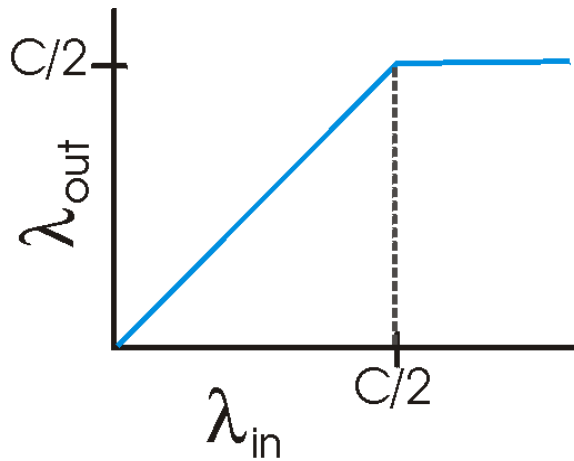
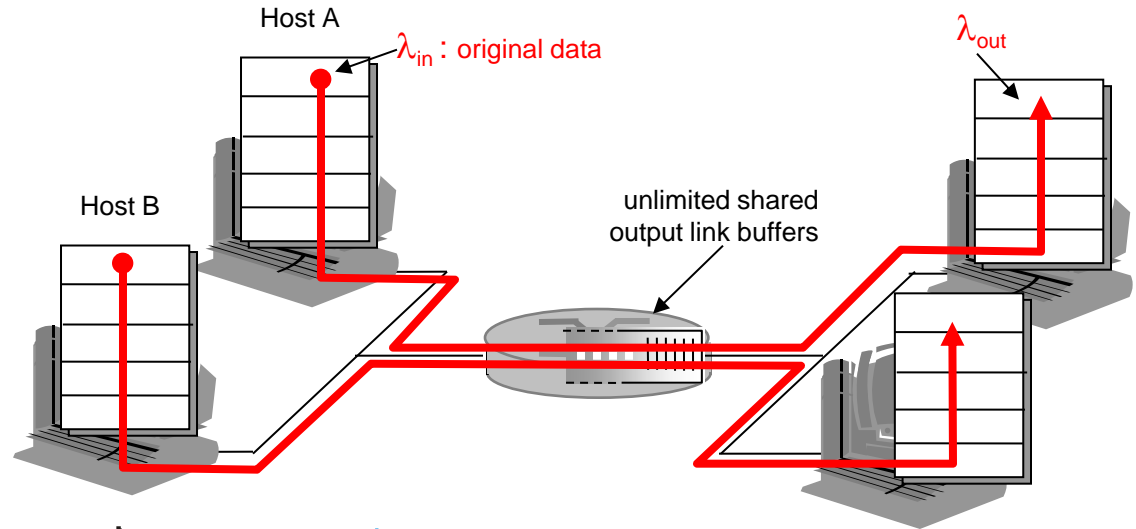
## Congestion:

- informally: “too many sources sending too much data too fast for the **network** to handle”
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a major problem in networking!

# Causes/Costs of Congestion

## Scenario 1

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

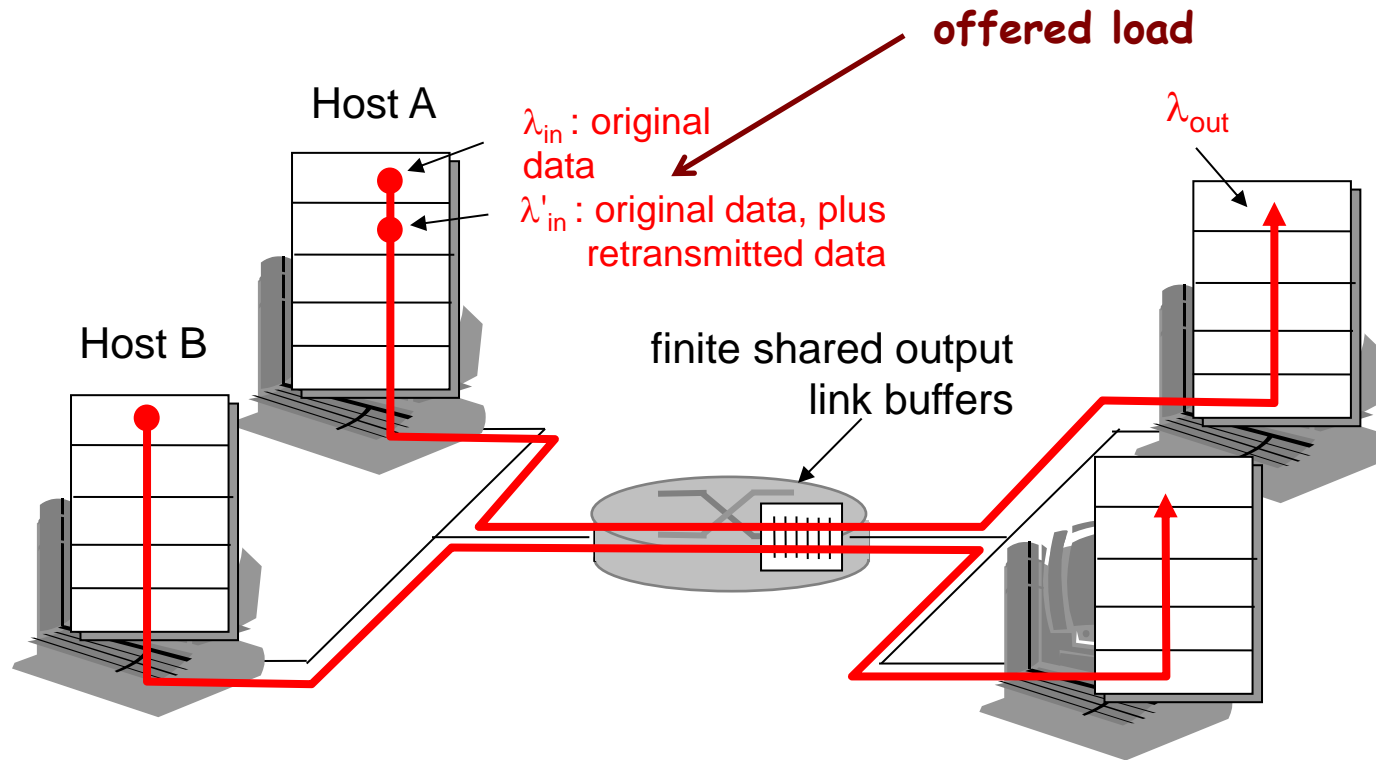


- large delays when congested
- maximum achievable throughput

# Causes/Costs of Congestion

## Scenario 2

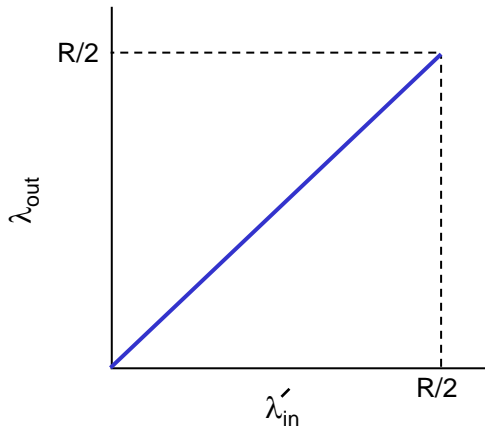
- one router, **finite** buffers
- sender retransmits lost packets



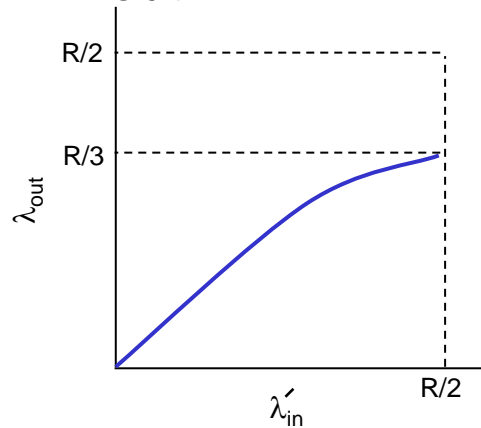
# Causes/Costs of Congestion

## Scenario 2

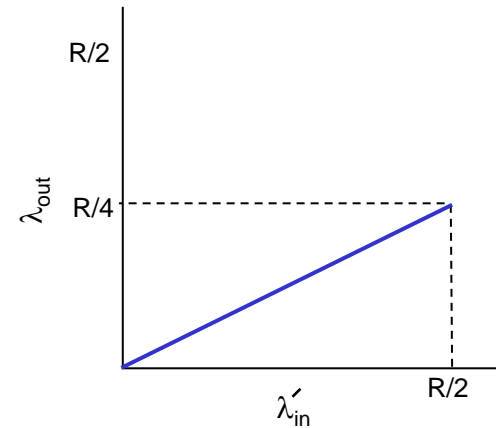
- always:  $\lambda_{in} = \lambda_{out}$  (goodput)
- “perfect” retransmission only when loss:  $\lambda'_{in} > \lambda_{out}$
- retransmission of delayed (not lost) packet makes  $\lambda'_{in}$  larger (than perfect case) for same  $\lambda_{out}$



a.



b.



c.

**“costs” of congestion:**

- more work (retransmissions) for a given “goodput”
- unneeded retransmissions: link carries multiple copies of packet

# Approaches towards Congestion Control

## Two broad approaches towards congestion control:

### end-end congestion control:

- no explicit feedback from network
- congestion **inferred** from end-system observed loss, delay
- approach taken by TCP

### network-assisted congestion control:

- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate sender should use for sending.

# TCP

## Congestion Control

Lecture material taken from  
“Computer Networks *A Systems Approach*”,  
Fourth Edition, Peterson and Davie,  
Morgan Kaufmann, 2007.

# 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”.



# TCP Congestion Control

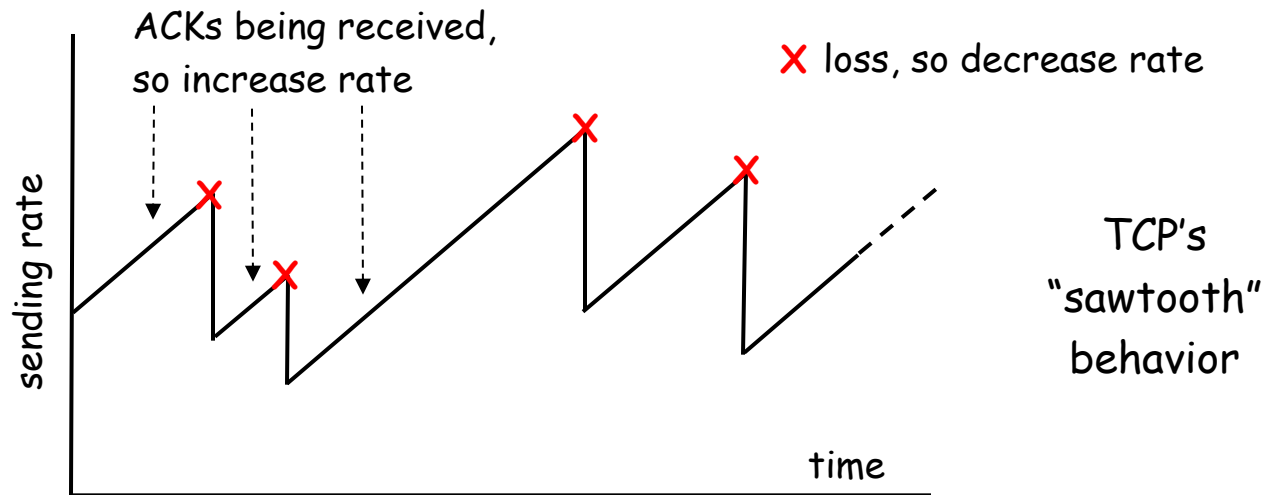
K & R

- **Goal:** TCP sender should transmit as fast as possible, but without congesting network.
  - **issue** - how to find rate *just below* congestion level?
- Each TCP sender sets its window size, based on *implicit* feedback:
  - **ACK** segment received → network is not congested, so increase sending rate.
  - **lost segment** - assume loss due to congestion, so decrease sending rate.

# TCP Congestion Control

K & R

- **“probing for bandwidth”**: increase transmission rate on receipt of ACK, until eventually loss occurs, then decrease transmission rate
  - continue to increase on ACK, decrease on loss (since available bandwidth is changing, depending on other connections in network).



- Q: how fast to increase/decrease?

# 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 - LastByteAcked)

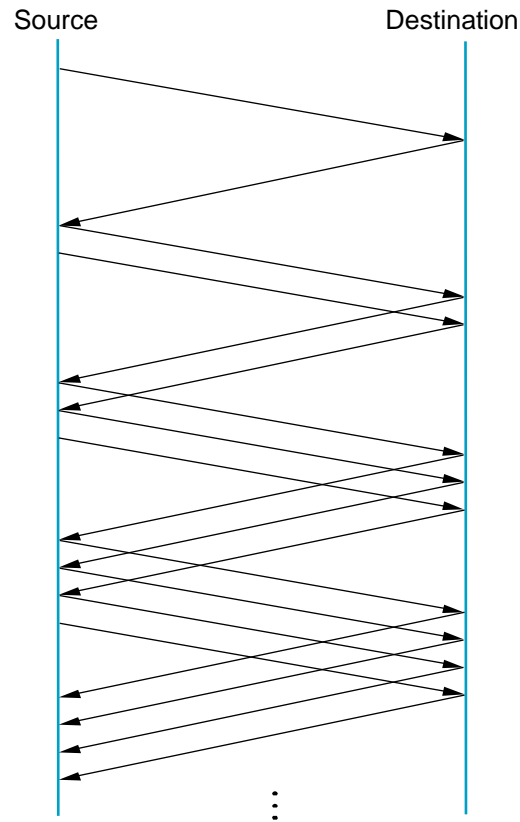
- **cwnd** is set based on the perceived level of congestion. The Host receives *implicit* (packet drop) or *explicit* (packet mark) indications of internal congestion.

# Additive Increase (AI)

- Additive Increase is a reaction to perceived available capacity (referred to as **congestion avoidance** stage).
- Frequently in the literature, additive increase is defined by parameter  $\alpha$  (where the default is  $\alpha = 1$ ).
- **Linear Increase** :: For each “cwnd’s worth” of packets sent, increase cwnd by 1 packet.
- In practice, **cwnd** is incremented fractionally for each arriving ACK.

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

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



Add one packet  
each RTT

# Figure 6.8 Additive Increase

# Multiplicative Decrease (MD)

- \* Key assumption :: a dropped packet and resultant timeout are due to congestion at a router.
- Frequently in the literature, multiplicative decrease is defined by parameter  $\beta$  (where the default is  $\beta = 0.5$ )

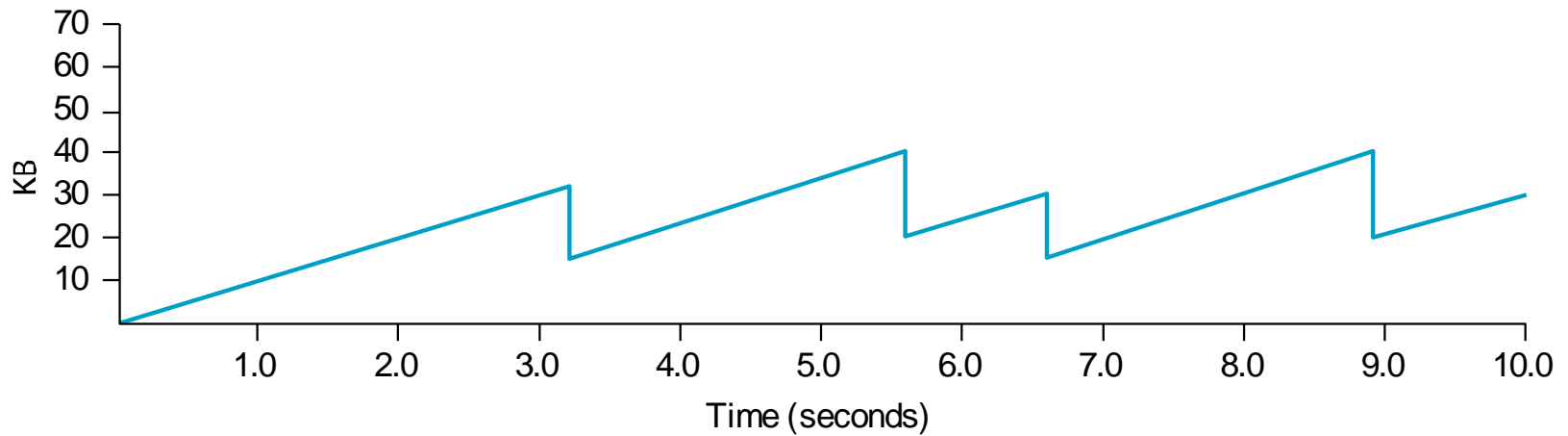
**Multiplicate Decrease**:: TCP reacts to a timeout by halving **cwnd**.

- Although defined in bytes, the literature often discusses **cwnd** 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**



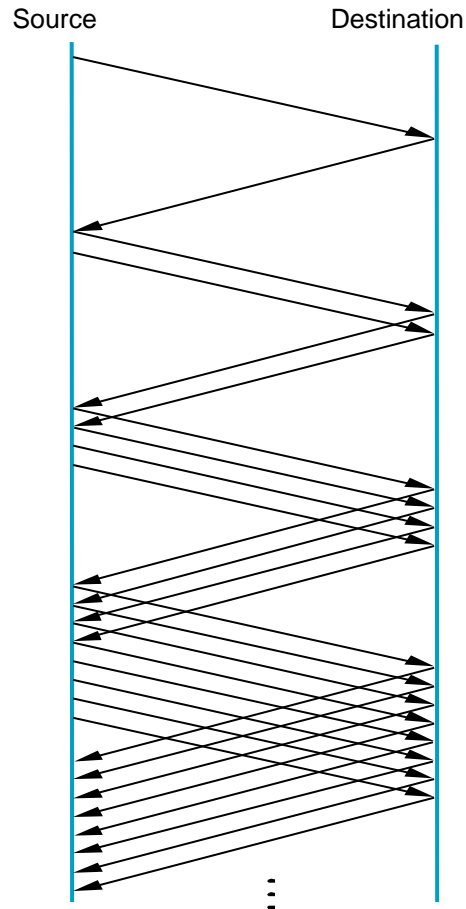
# Slow Start

- 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, when the **advertized window** goes to zero!)



Slow Start  
Add one packet  
per ACK

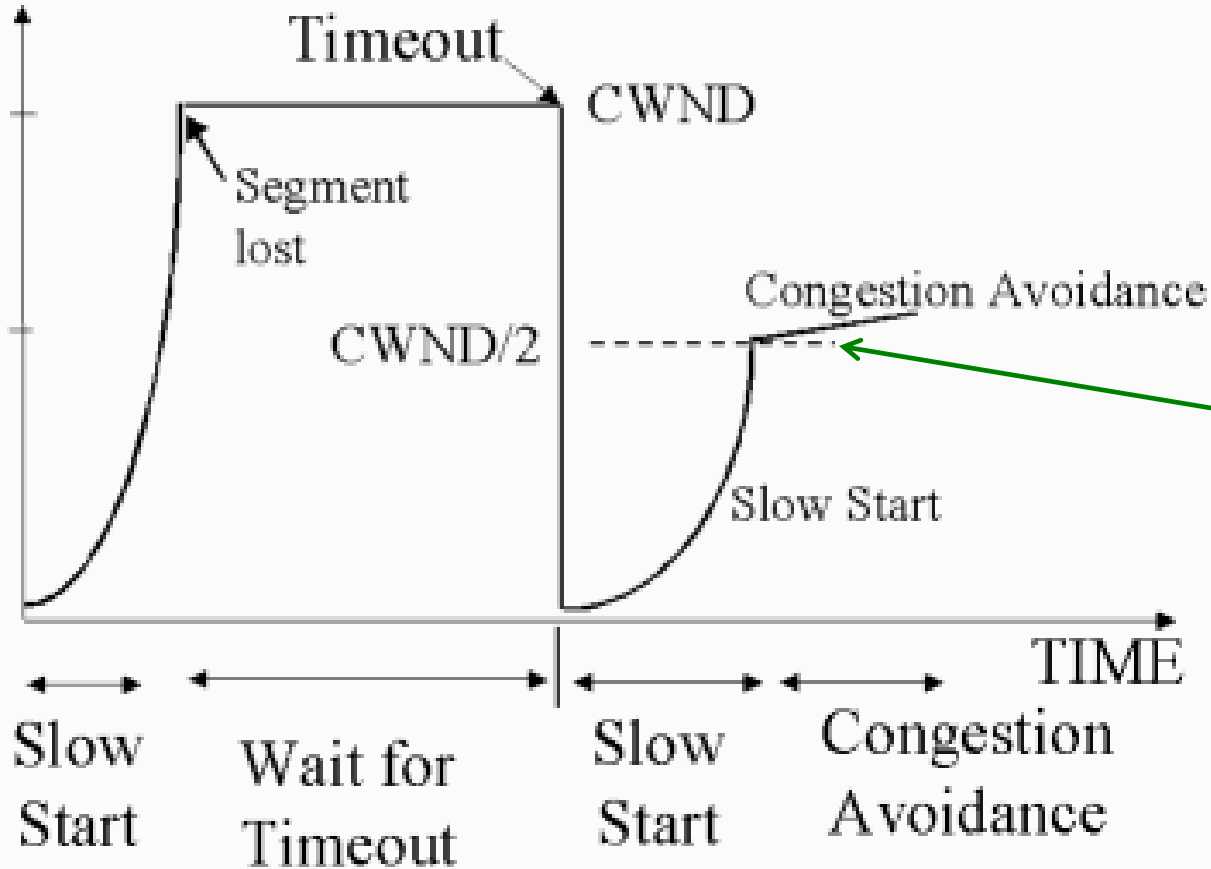
Figure 6.10 Slow Start

# Slow Start

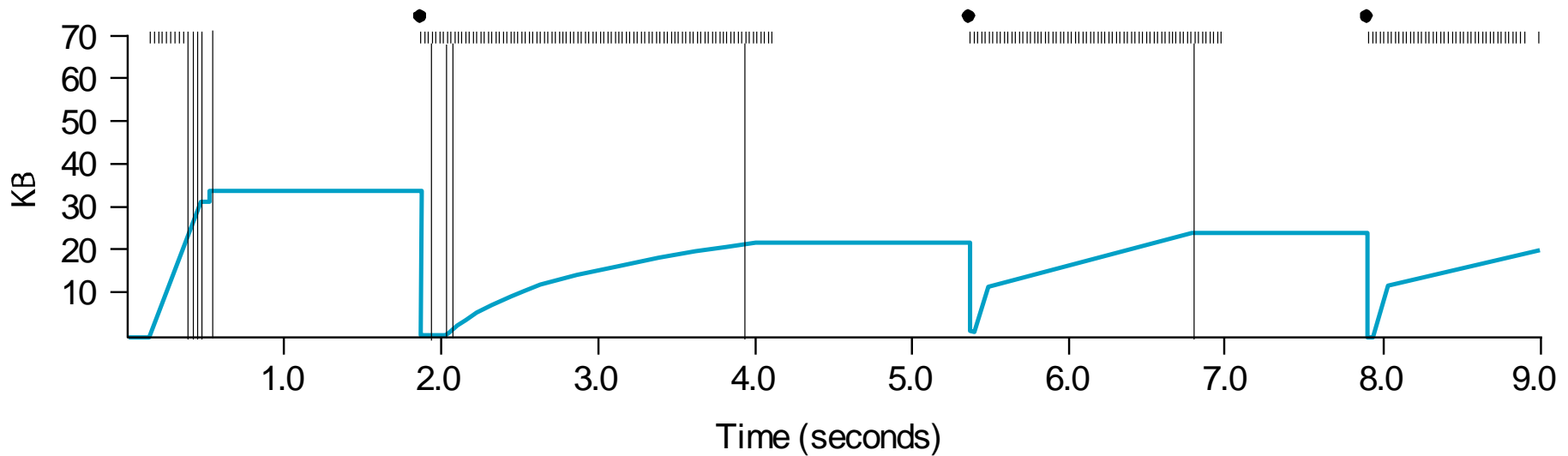
- However, in the second case the source has more information. The current value of cwnd can be saved as a **congestion threshold**.
- This is also known as the “slow start threshold” **ssthresh**.

# Slow Start

Congestion Window



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.*

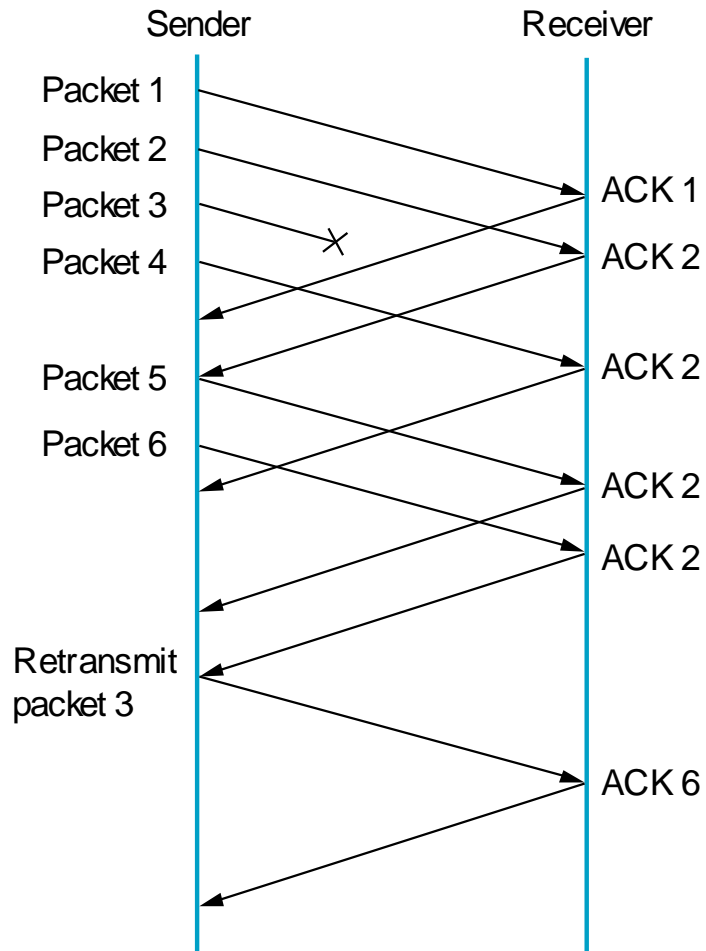
## Fast Retransmit

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

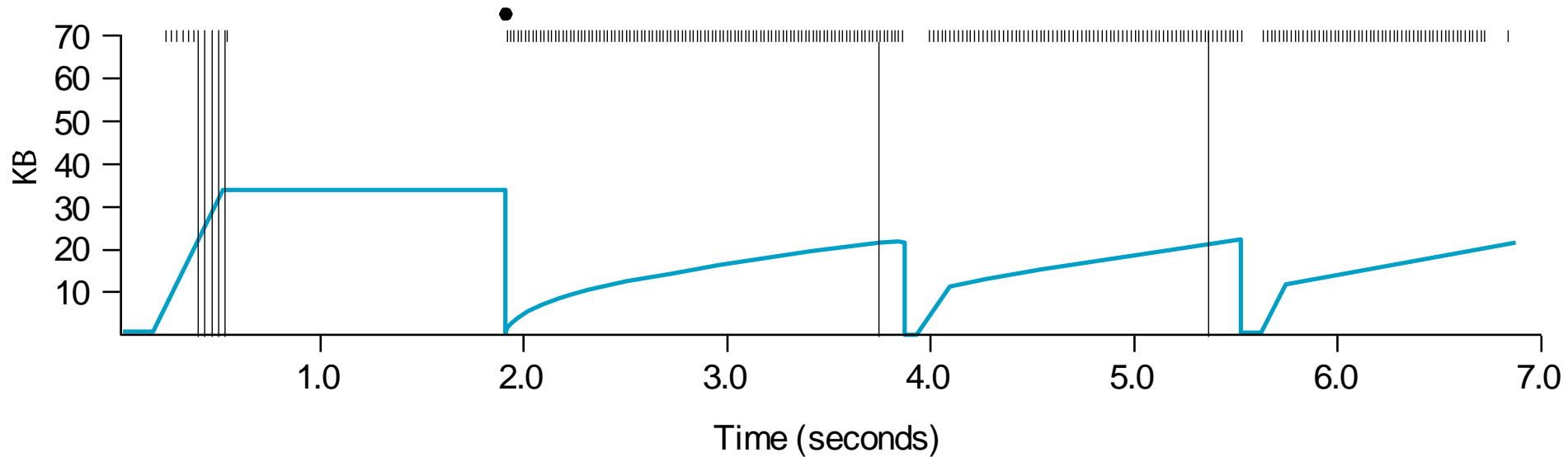


Figure 6.13 TCP Fast Retransmit Trace

# 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!!**

# Many TCP 'flavors'

- TCP New Reno
- TCP SACK
  - requires sender and receiver both to support TCP SACK.
  - possible state machine is complex.
- TCP Vegas
  - adjusts window size based on difference between expected and actual RTT.
- TCP BIC → TCP Cubic {used by Linux}
- TCP Compound {used by Windows}

# TCP New Reno

- Two problem scenarios with TCP Reno
  - bursty losses, Reno cannot recover from bursts of 3+ losses.
  - Packets arriving out-of-order can yield duplicate acks when in fact there is no loss.
- New Reno solution – try to determine the end of a burst loss.

# TCP New Reno

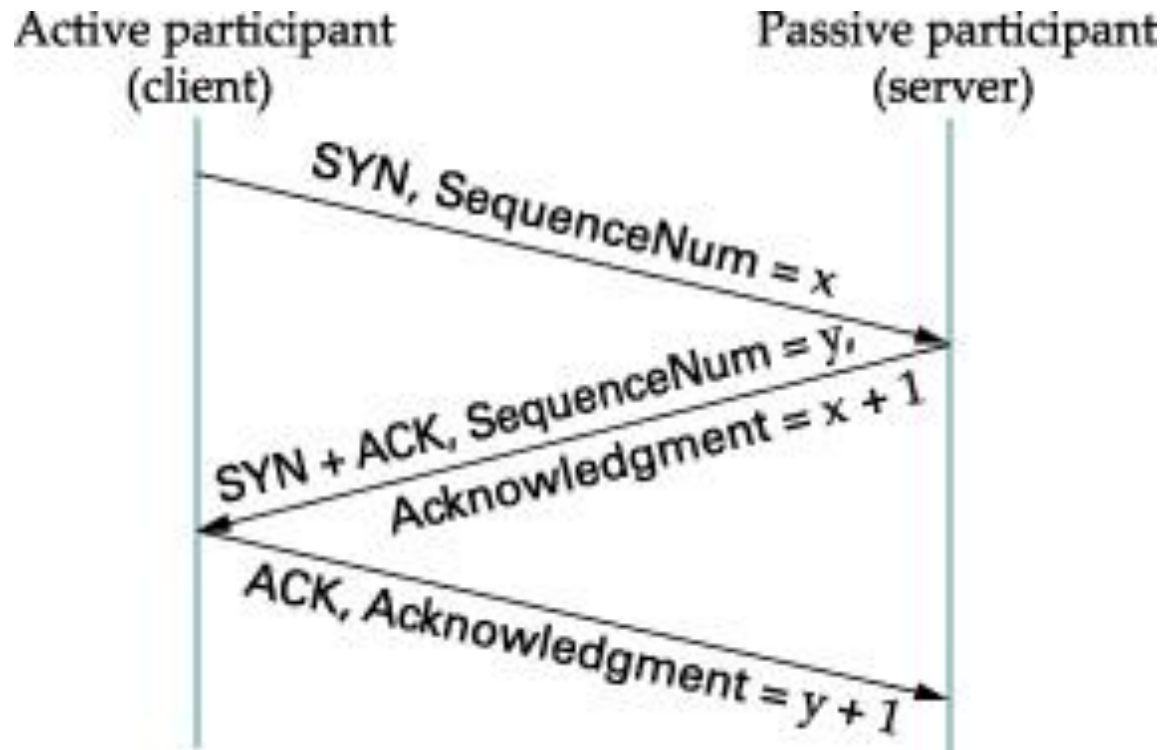
- When duplicate ACKs trigger a retransmission for a lost packet, remember the highest packet sent from window in **recover**.
- Upon receiving an ACK,
  - if  $\text{ACK} < \text{recover} \Rightarrow$  partial ACK
  - If  $\text{ACK} \geq \text{recover} \Rightarrow$  new ACK

# TCP New Reno

- **Partial ACK** implies another lost packet: retransmit next packet, inflate window and stay in **fast recovery**.
- **New ACK** implies fast recovery is over: starting from  $0.5 \times \text{cwnd}$  proceed with congestion avoidance (linear increase).
- New Reno recovers from **n** losses in **n** round trips.



# Figure 5.6 Three-way TCP Handshake



# Adaptive Retransmissions

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

- How to set the Timeout value (RTO)?
  - 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)

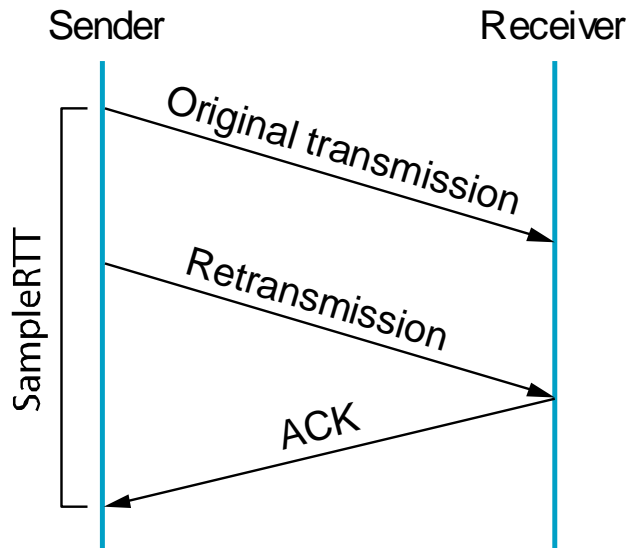
$$\text{TimeOut} = 2 \times \text{EstimatedRTT}$$

# Karn/Partridge Algorithm

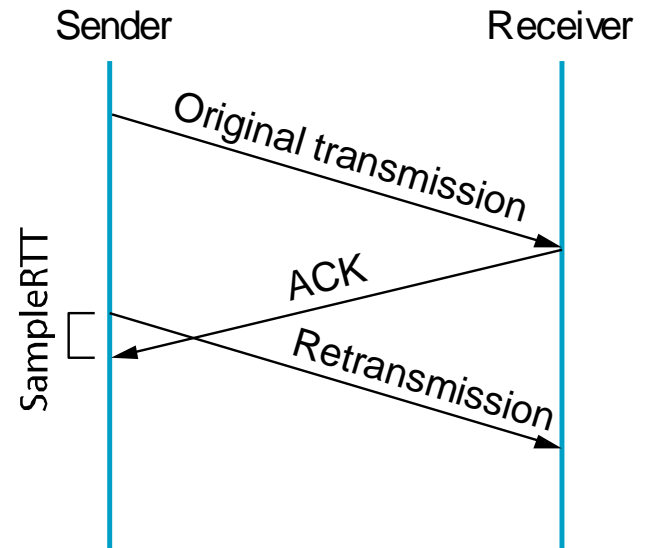
An obvious flaw in the original algorithm:

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  $\Phi = 4$ .

# TCP Congestion Control Summary

- Congestion occurs due to a variety of circumstance.
- TCP interacts with routers in the subnet and reacts to implicit congestion notification (packet drop) by reducing the TCP sender's congestion window **(MD)**.
- TCP increases congestion window using slow start or congestion **avoidance (AI)**.

# TCP Congestion Control Summary

- Important TCP Congestion Control ideas include: **AIMD, Slow Start, Fast Retransmit and Fast Recovery.**
- Currently, the two most common versions of TCP are Compound (Windows) and Cubic (Linux).
- TCP needs rules and an algorithm to determine **RIO and RTO.**