Flow and Congestion Control (Hosts)

14-740: Fundamentals of Computer Networks
Bill Nace

Material from Computer Networking: A Top Down Approach, 6th edition. J.F. Kurose and K.W. Ross

traceroute

- Flow Control

  Principles of Congestion Control

  TCP Congestion Control

Flow Control

In RDT lecture, we discovered windows for managing pipelined transfer

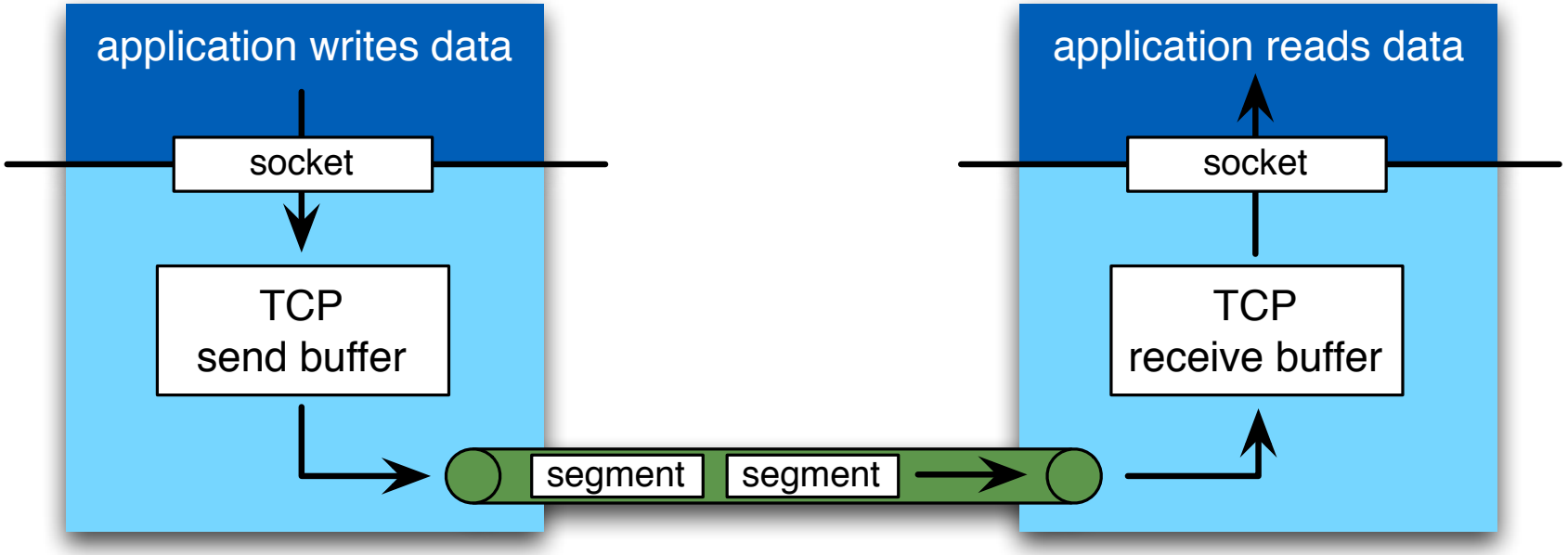
- But, didn't discuss windows (much) in TCP lecture

TCP has 2 windows

Flow Control != Congestion Control

- Flow Control Window
- Congestion Control Window

Sender limited by smallest window



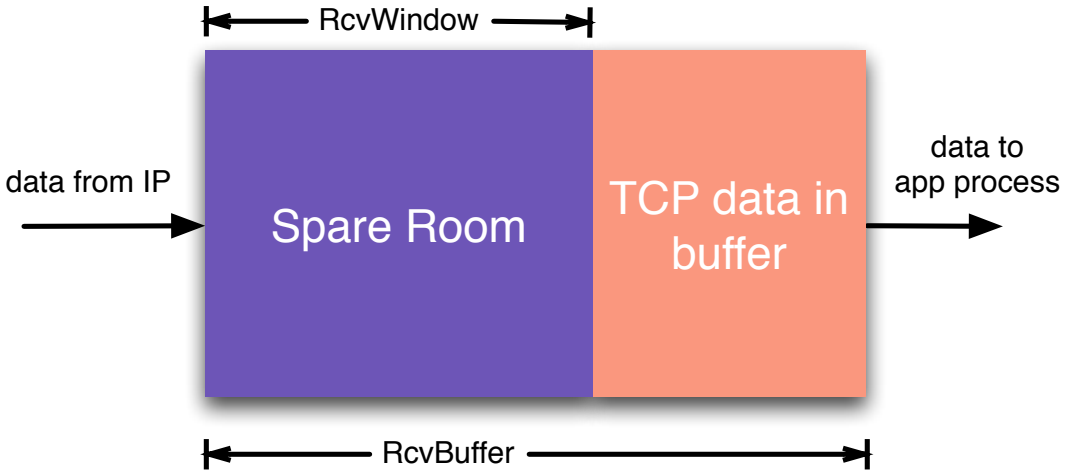
- Both sides have buffers

  Lots of “Producer-Consumer” coordination problems to overcome

Flow Control

Receive side of TCP connection has a receive buffer of size `RcvBuffer`

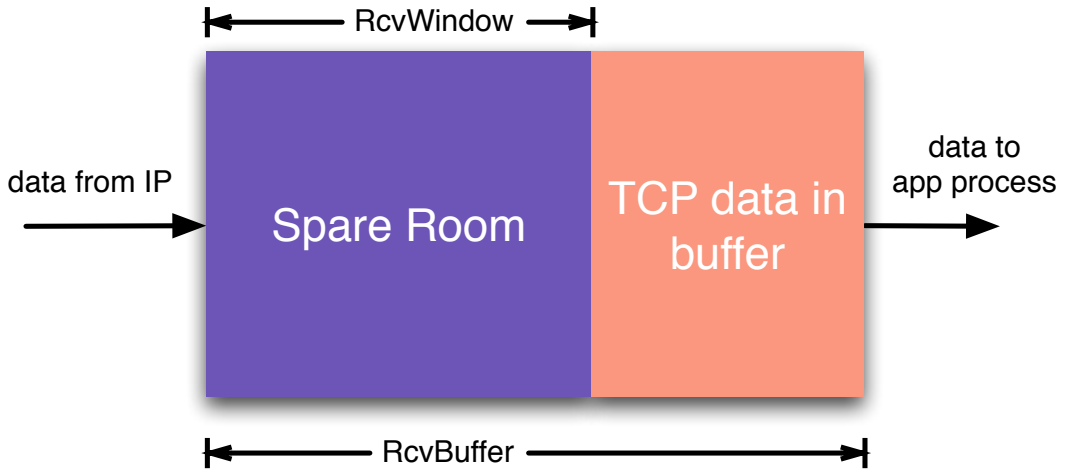
Application process may be slow at reading from the buffer



Flow Control (2)

Flow Control: Sender won't overflow the receiver's buffer by transmitting too much, too fast

Speed-matching service: matching the send rate to the receiving app's drain rate



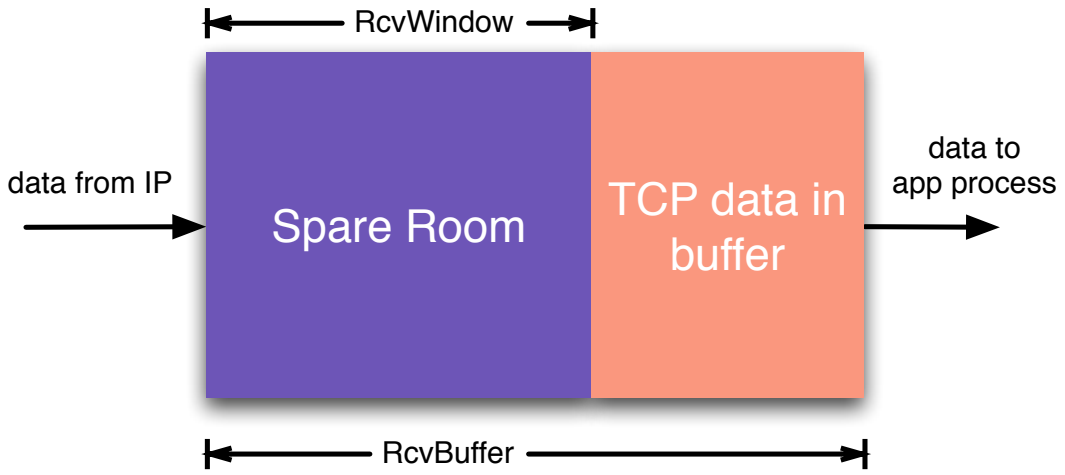
Mechanism

Spare room in buffer

= RcvWindow

= RcvBuffer - [LastByteRcvd - LastByteRead]

Yes, this assumes receiver discards out-of-order segments. Easy enough to program around simplification

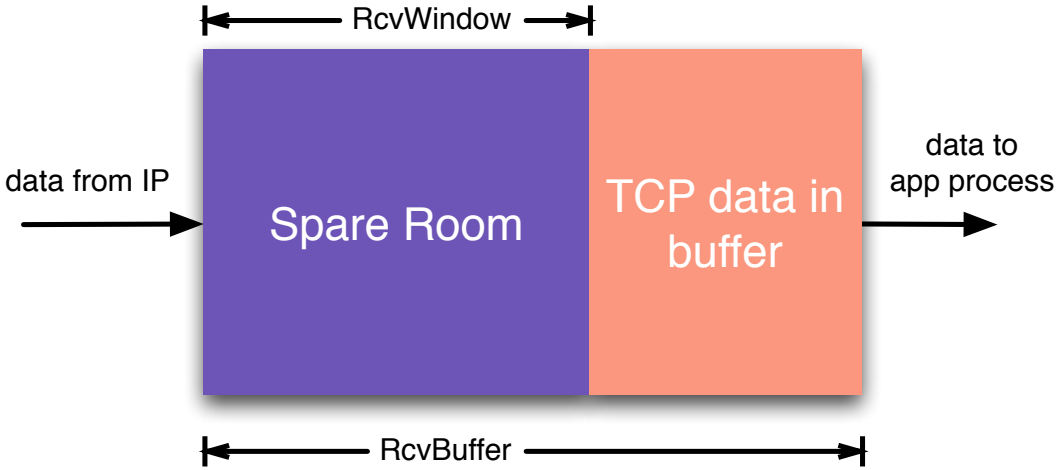


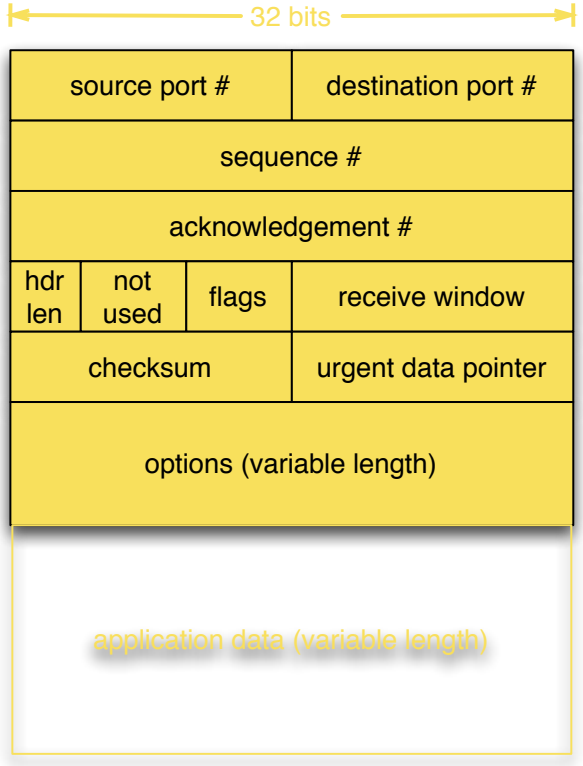
Mechanism (2)

Receiver advertises spare room by including value of `RcvWindow` in ACK segment

Gives sender permission to send this much

Sender limits unACKed data to `RcvWindow` bytes





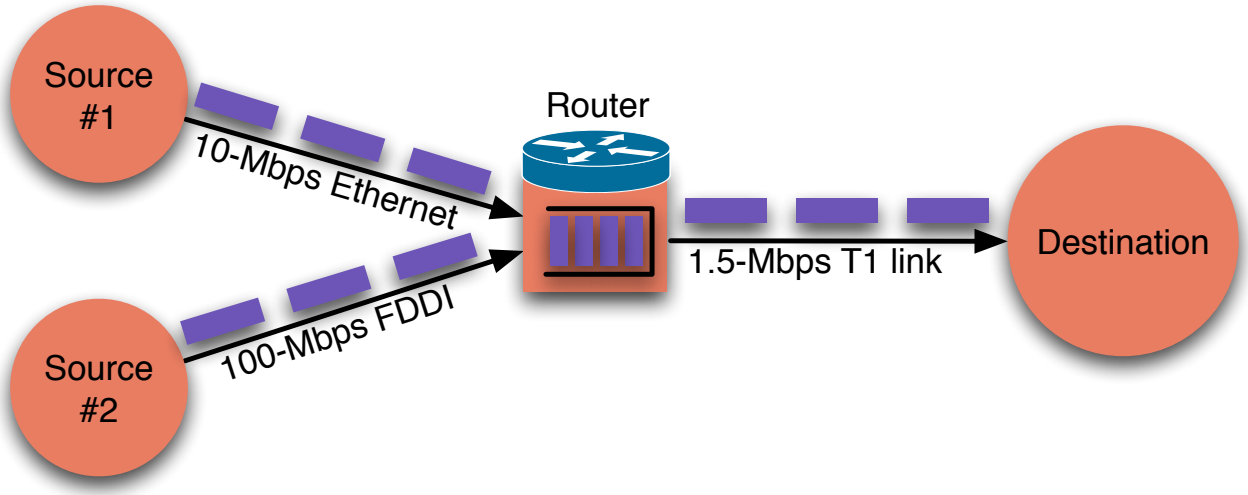
traceroute

Flow Control

- Principles of Congestion Control

TCP Congestion Control

Router View



Router buffer absorbs temporary bursts when input rate > output rate

When buffer is full, router cannot accept more packets and must drop them

Costs of Congestion

Congestion

Too many sources sending too much data too fast for network to handle

- Different than flow control!

As network load increases, some router has many packets queued in its buffer, resulting in:

- Long delays – as packets wait for processing
- Lost packets – as buffer space overflows, cannot handle any more incoming packets!

Congestion Control

Goal: large throughput and small delay

To increase throughput, send more packets

More packets increases queue length at routers -- delay increases

Large throughput != small delay

Congestion Control (2)

Apply some control theory

Region 1: Low throughput

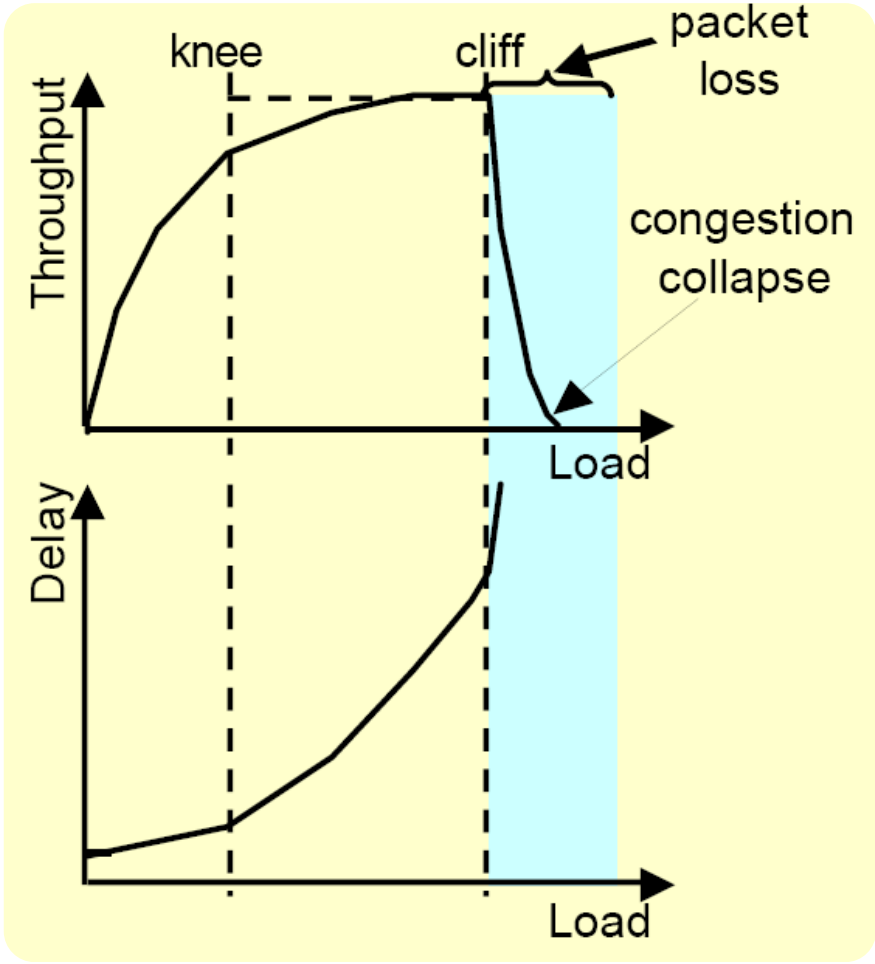
Region 2: High delay

- Throughput increases slowly
- Delay increases quickly

Region 3: Collapse

- Throughput → 0, Delay → ∞

At what load would we like to operate?



How do we control?

Need feedback mechanism!

Detect when network approaches knee point, so countermeasures can be taken

- Slow down sending rate

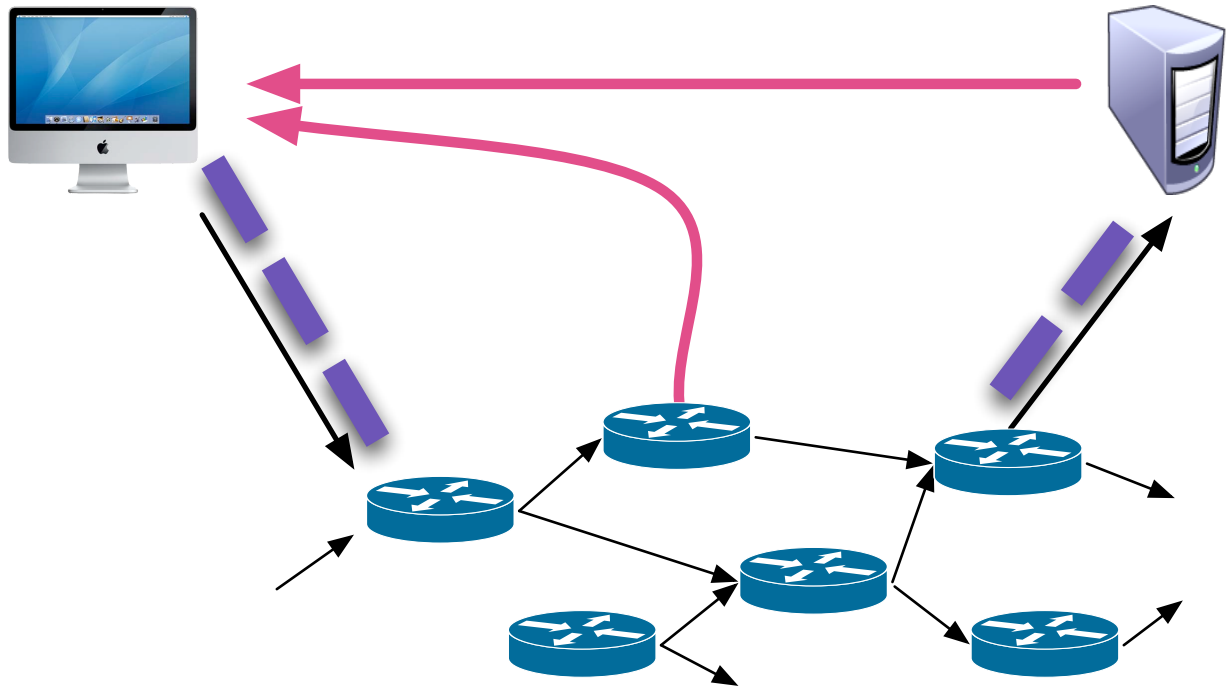
“Conservation of Packets” Law

- When network running in steady state at peak efficiency, don’t put a packet into the network until one leaves

Feedback Approaches

- Network-assisted congestion control (detection):
  - routers provide feedback to sender, or
  - Set single bit in header as it goes by, or ...
    - TCP/IP ECN, ATM, SNA, DECnet
  - tell explicit send rate
- End-end control:
  - no explicit feedback from network
  - congestion inferred from end-system observed loss, delay
  - approach taken by TCP

Feedback Mechanism



Network Assisted: Signals from routers

End-to-end: Messages from receiver

traceroute

Principles of Congestion Control

- TCP Congestion Control

TCP CC: Overview

End-to-End CC: sender limits transmission based on perceived congestion

Uses **`CongWin`** variable -- how much data allowed in-flight at any time

`LastByteSent-LastByteAcked ≤ CongWin`

Roughly, rate = **`CongWin`** / RTT

Congestion Detection

How does sender perceive congestion?

- timeout
- 3 duplicate ACKs

TCP sender reduces rate (**`CongWin`**) after loss event

TCP CC: Components

Slow start – Getting to equilibrium

Additive-increase, multiplicative-decrease (AIMD) – Adapting to path (avoiding congestion)

RTT estimation – Conservation at equilibrium

Reaction to timeout events

TCP CC (2)

TCP is self-clocking

- Uses ACK to trigger (or clock) its increase in congestion window size

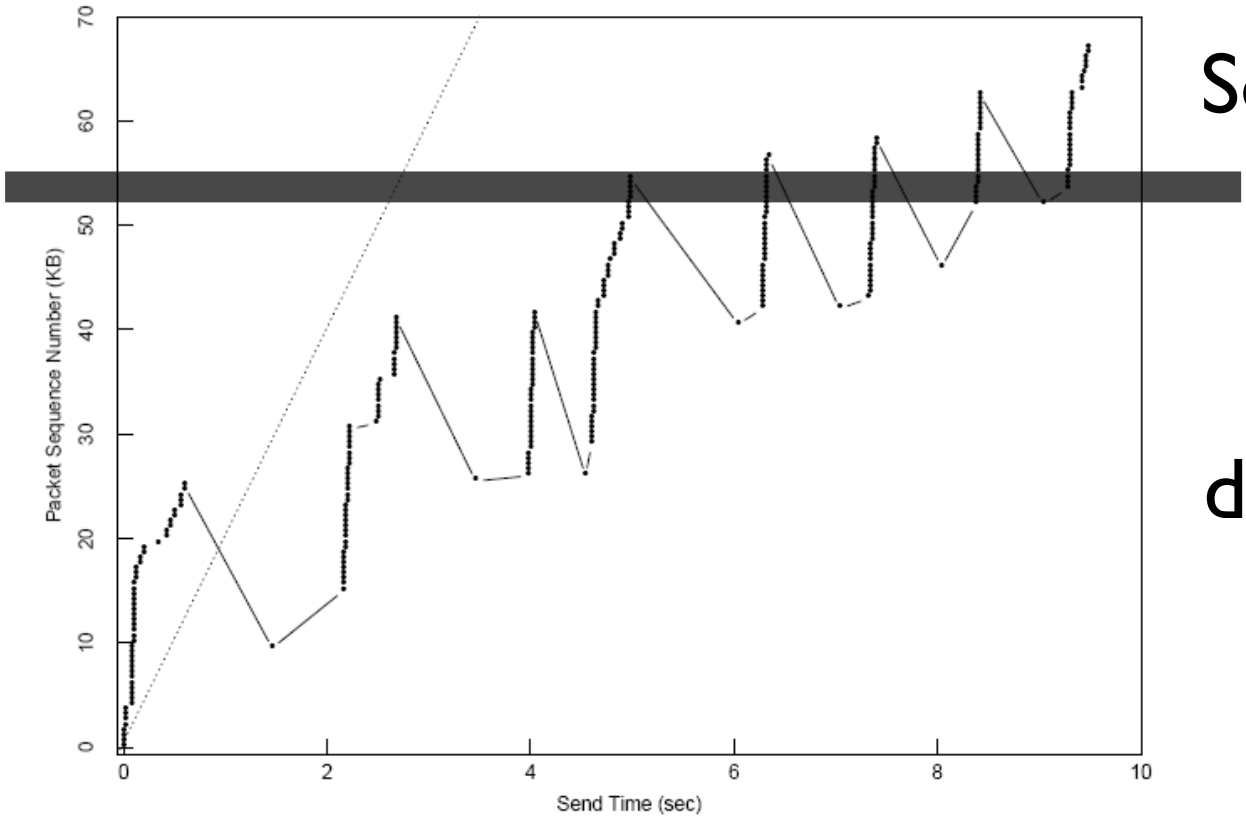
A number of algorithmic varieties:

- TCP Tahoe
- TCP Reno (most widely used)
- TCP Vegas, TCP SACK, etc

Getting Started

Uncontrolled rush to send segments is UGLY

Figure 3: Startup behavior of TCP without Slow-start



Some segments got sent 5 times!

"Nothing in this trace resembles desirable behavior"

Slow Start

- Getting to Equilibrium

  When connection begins, **`CongWin`** = 1 MSS

  Available bandwidth may be >> MSS/RTT

    desire a quick ramp-up to respectable rate

  Therefore, at start, increase rate exponentially fast ...

    until first loss event ...

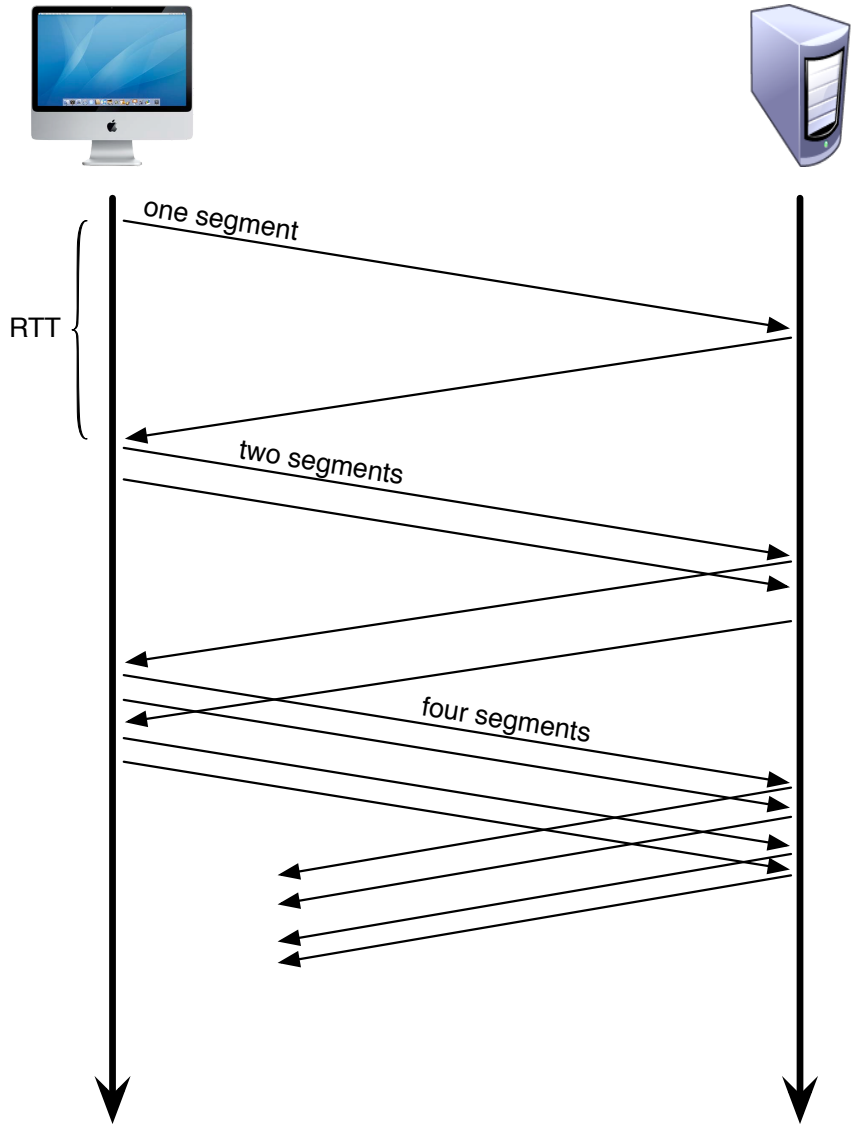
    or **`CongWin`** > **`ssthresh`** (a pre-set threshold)

Slow Start

When connection begins, increase rate exponentially:

- double **`CongWin`** every RTT
- done by increasing **`CongWin`** by 1MSS for every ACK received

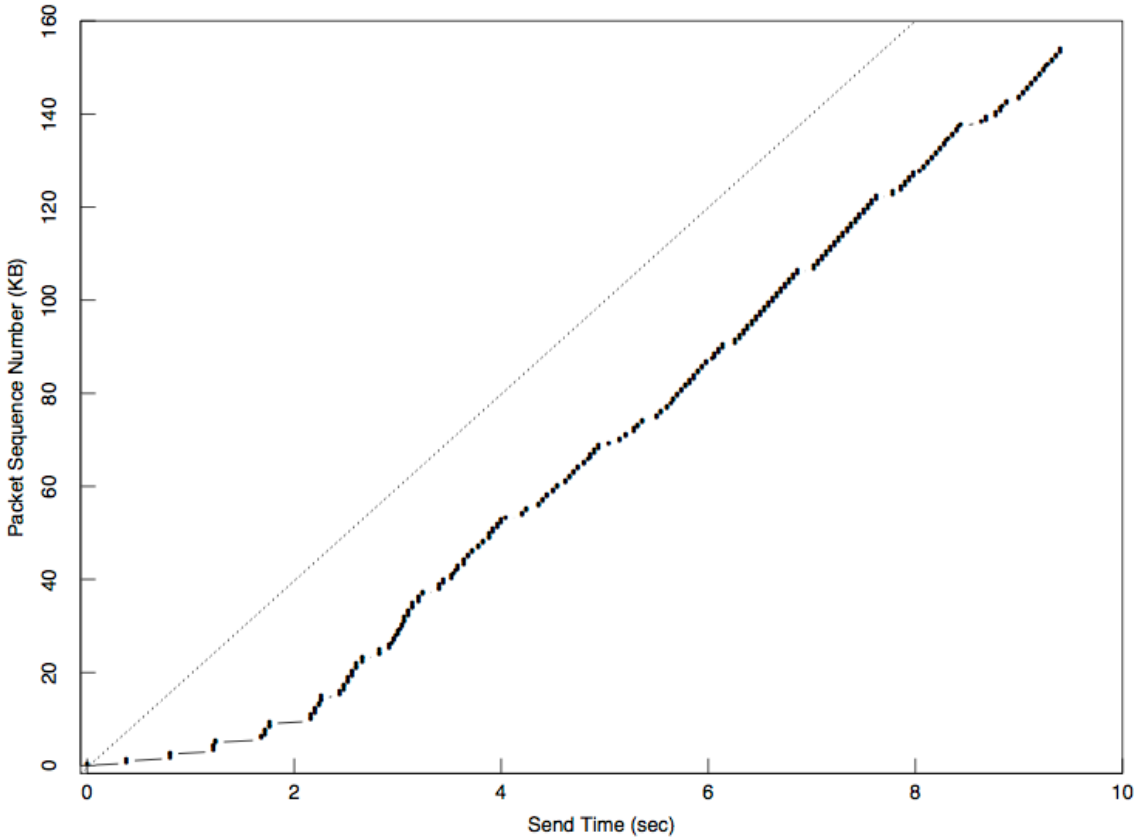
Summary: initial rate is slow but ramps up exponentially fast



Behavior of Slow Start

With Slow Start, no bandwidth wasted on retransmission

Figure 4: Startup behavior of TCP with Slow-start



Control at Equilibrium

Slow start's exponential increase will eventually saturate the network

What happens to keep it in control?

Additive Increase, Multiplicative Decrease (AIMD) algorithm

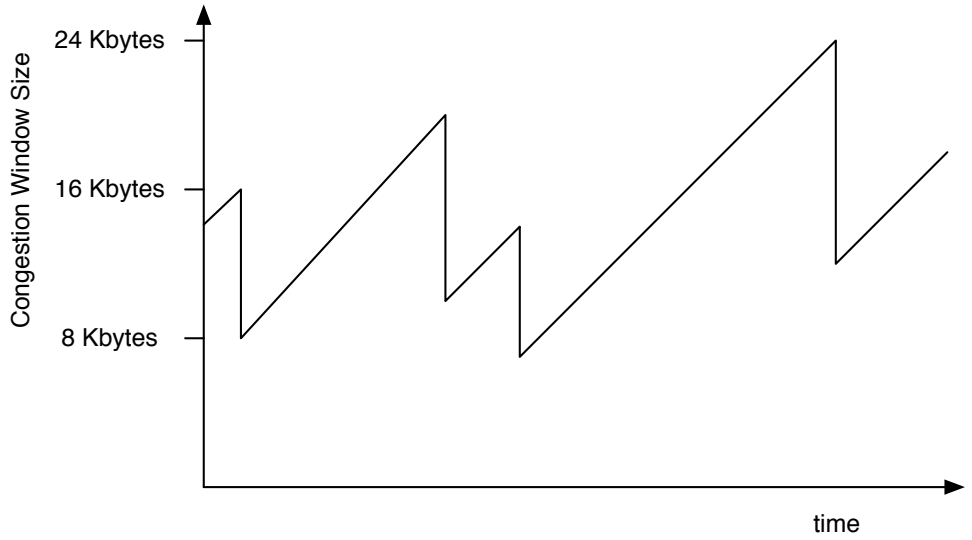
- Backoff quickly when loss occurs
- Continue probing for usable bandwidth

This phase is also called congestion avoidance

AIMD Mechanics

Additive Increase: Increase **`CongWin`** by 1 MSS every RTT until loss is detected

Multiplicative Decrease: cut **`CongWin`** in half after a loss event

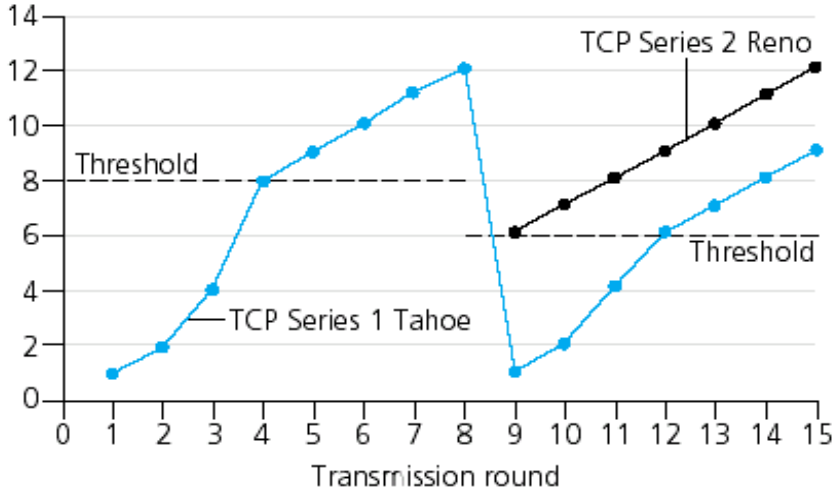


Sawtooth behavior: Probing for bandwidth

Refinements

Under TCP Tahoe, **`CongWin`** set to 1 after a loss, then slow start to **`ssthresh`**, which is set to half of **`CongWin`**'s value at loss

Under TCP Reno, **`CongWin`** is set to **`ssthresh`** and then linear increase



Fast Recovery

TCP Tahoe

After both loss events

- CongWin set to 1 MSS
- Enters Slow Start

Reno's Philosophy

- 3 dup ACKs indicates network capable of delivering some segments
- timeout indicates a "more alarming" congestion scenario ➙ back off!

TCP Reno

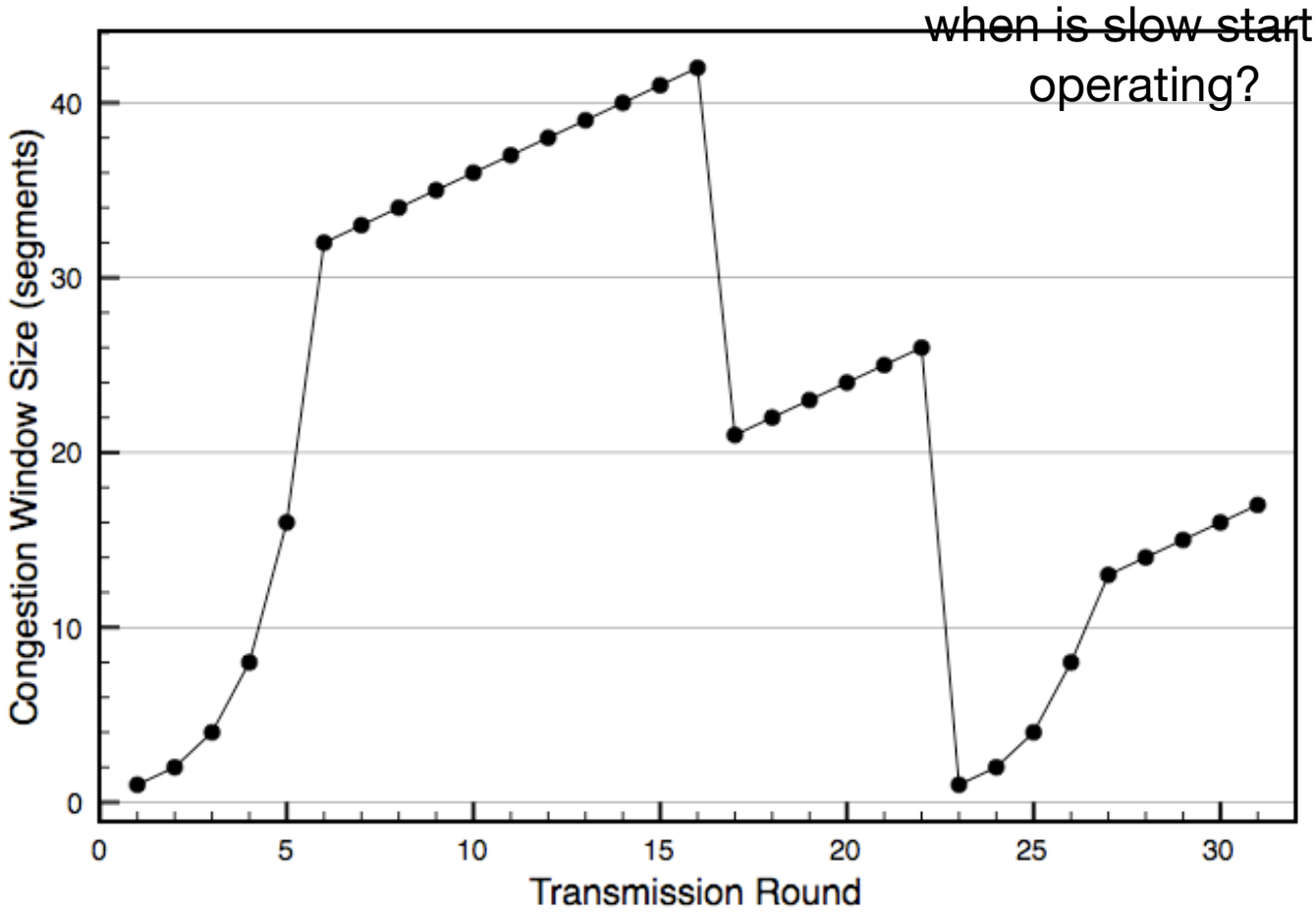
After 3 dup ACKs:

- Cancel Slow Start
- CongWin is cut in half
- window grows linearly

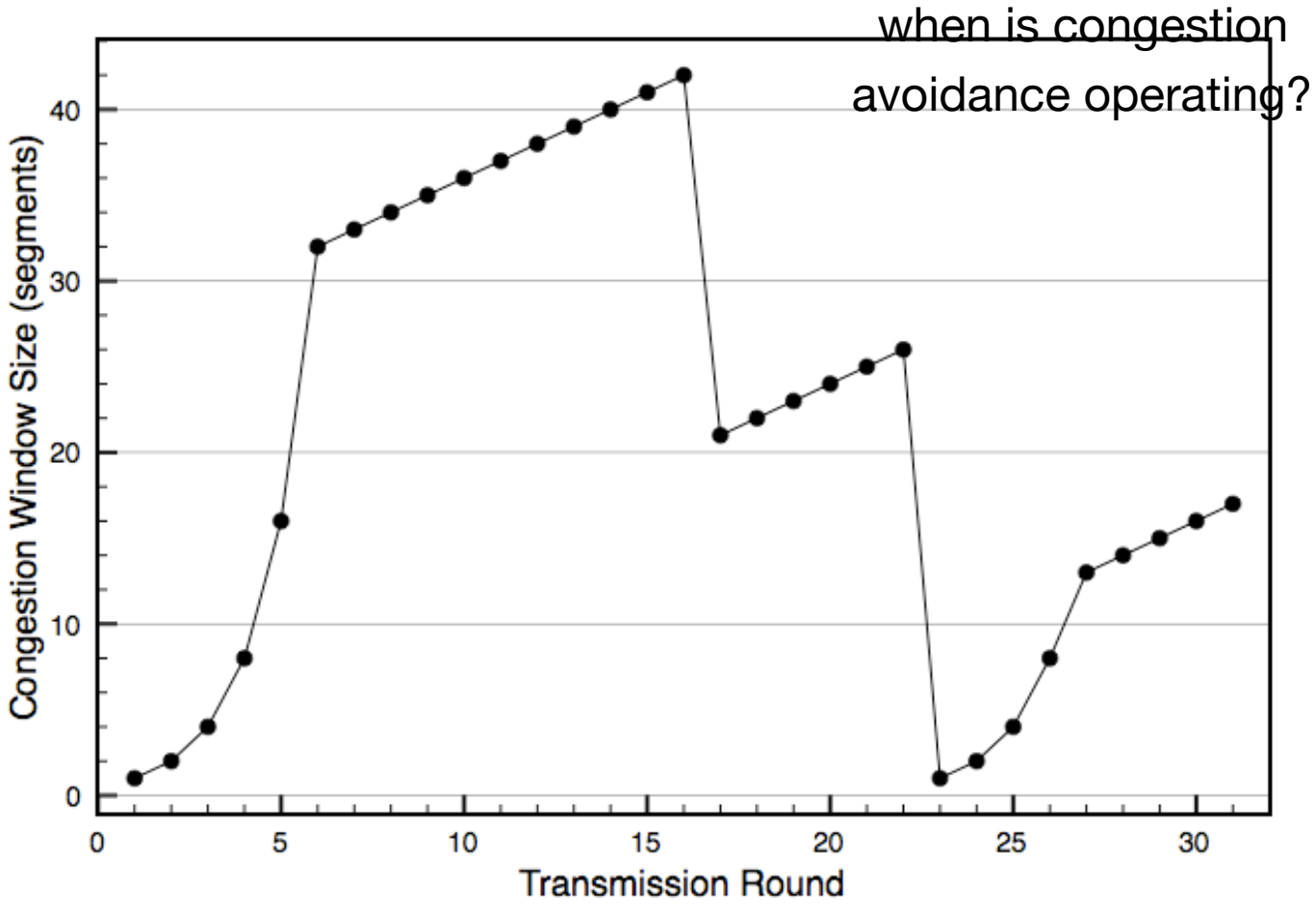
But after timeout event:

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

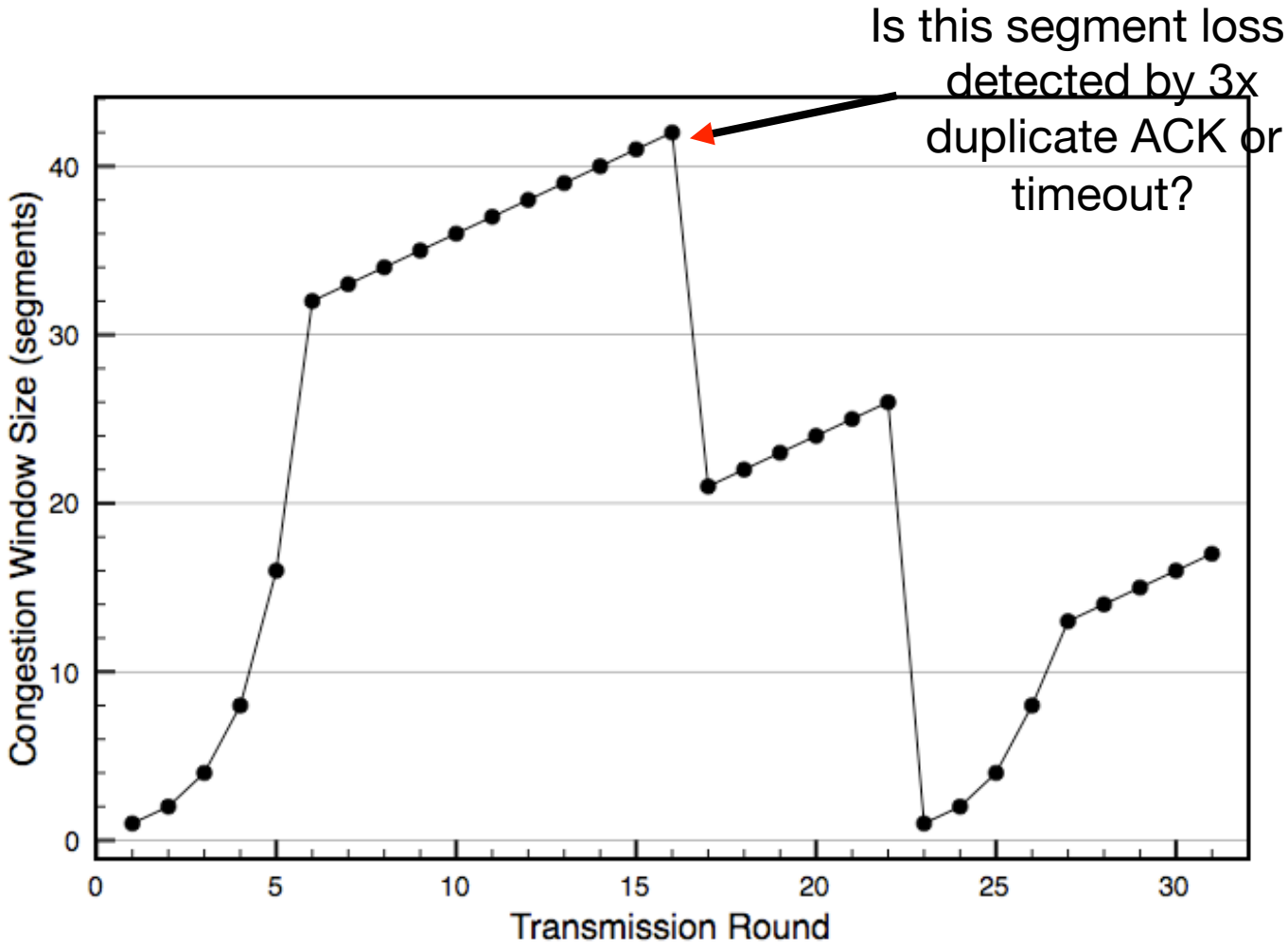
CongWin size plot



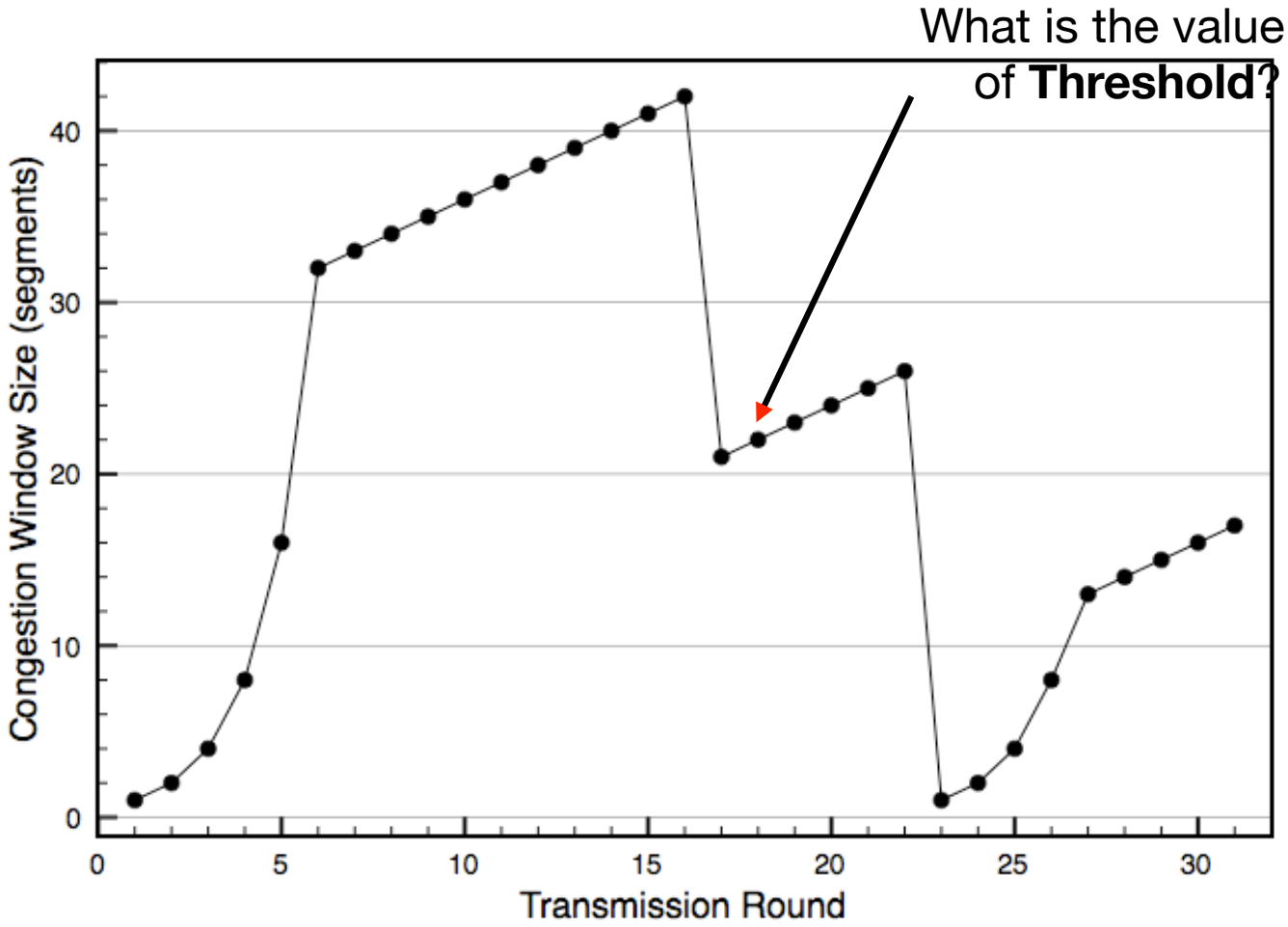
CongWin size plot



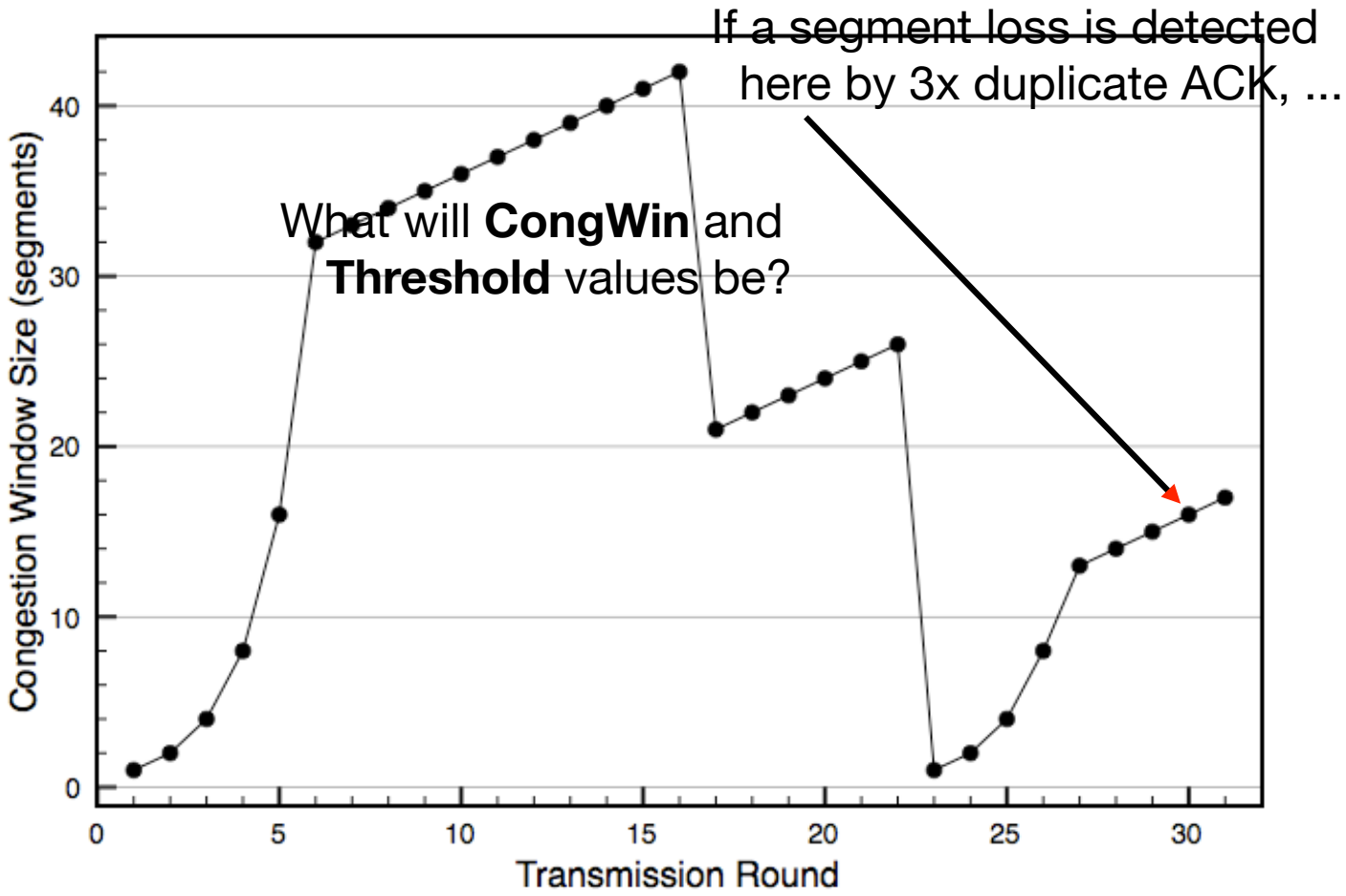
CongWin size plot



CongWin size plot



CongWin size plot



RTT and Timeout

- How should the timeout value be set?

  Must be longer than RTT (which varies)

  Too short: premature timeout

  unnecessary retransmissions

  Too long: slow reaction to segment loss

  Strategy: Measure actual RTTs for baseline

Estimating RTT

Grab some samples: **`SampleRTT`**

Measured time from segment transmission to ACK receipt

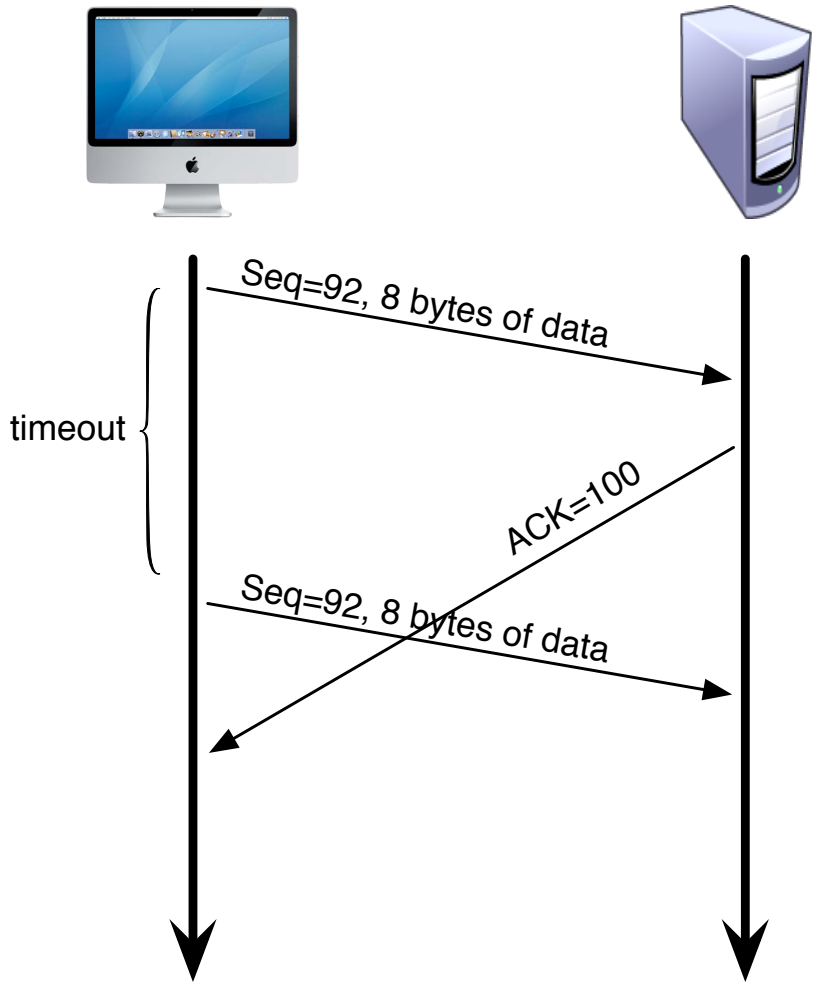
- ignore retransmissions

Sampled values vary, so “smooth”

- average several recent measurements

Why ignore retransmissions?

TCP does not measure SampleRTT for retransmitted segments. Why not?



Smoothing the Samples

To smooth samples, use exponential weighted moving average (EWMA)

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

Influence of past samples decreases exponentially fast

Typical value for **α** is 0.125

Setting the Timeout

Timeout should be EstimatedRTT + “safety margin”

Large variation in EstimatedRTT → larger margin

Use EWMA of deviation in SampleRTT from EstimatedRTT

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

β typically set to 0.25

Then set timeout interval

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

Why worry about Variance?

Why not just estimate RTT (use EWMA), then multiply by a constant “safety margin?”

Older version of TCP did just that:

```
TimeoutInterval = EstimatedRTT*2
```

Not adaptive enough -- Need a larger safety margin when network load is higher, smaller for lower loads

Otherwise, timeout interval is too short – what happens?

- Retransmit segments that are not lost
- Adds more segments to an already congested network!

TCP CC Summary

When **`CongWin`** is below **`Threshold`**, window grows exponentially (slow-start phase)

When **`CongWin`** is above **`Threshold`**, window grows linearly (congestion-avoidance phase)

When a triple duplicate ACK occurs, **`Threshold`** set to **`CongWin/2`** and **`CongWin`** set to **`Threshold`**. Window grows linearly

When timeout occurs, **`Threshold`** set to **`CongWin/2`** and **`CongWin`** is set to 1 MSS. Enters slow-start phase

Lesson Objectives

Now, you should be able to:

- describe the mission, operation and mechanisms for flow control in TCP
- list causes, costs and consequences of network congestion
- describe the operations of, as well as advantages and disadvantages of, different feedback mechanisms

You should be able to:

- describe the overall congestion control mechanisms used in TCP, including the congwin variable, self-clocking nature, and interaction of various phases
- describe the slow start component of TCP congestion control; including starting conditions, reactions to ACKs and ending conditions

You should also be able to:

- describe the congestion avoidance component of TCP congestion control; including starting conditions, ending conditions, reactions to loss, reactions to ACKs and differences between Reno and Tahoe versions
- describe how TCP sets timeout values
- calculate EstimatedRTT, DevRTT and TimeoutInterval