TCP
Congestion Control

14-740: Fundamentals of Computer Networks
Credit: Bill Nace, 14-740, Fall 2017

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

Last Lecture

- Reliable Data Transfer Tools
  - Checksum
  - Receiver Feedback
  - Retransmission Timer
  - Sequence Number
  - Window / Pipelining

traceroute

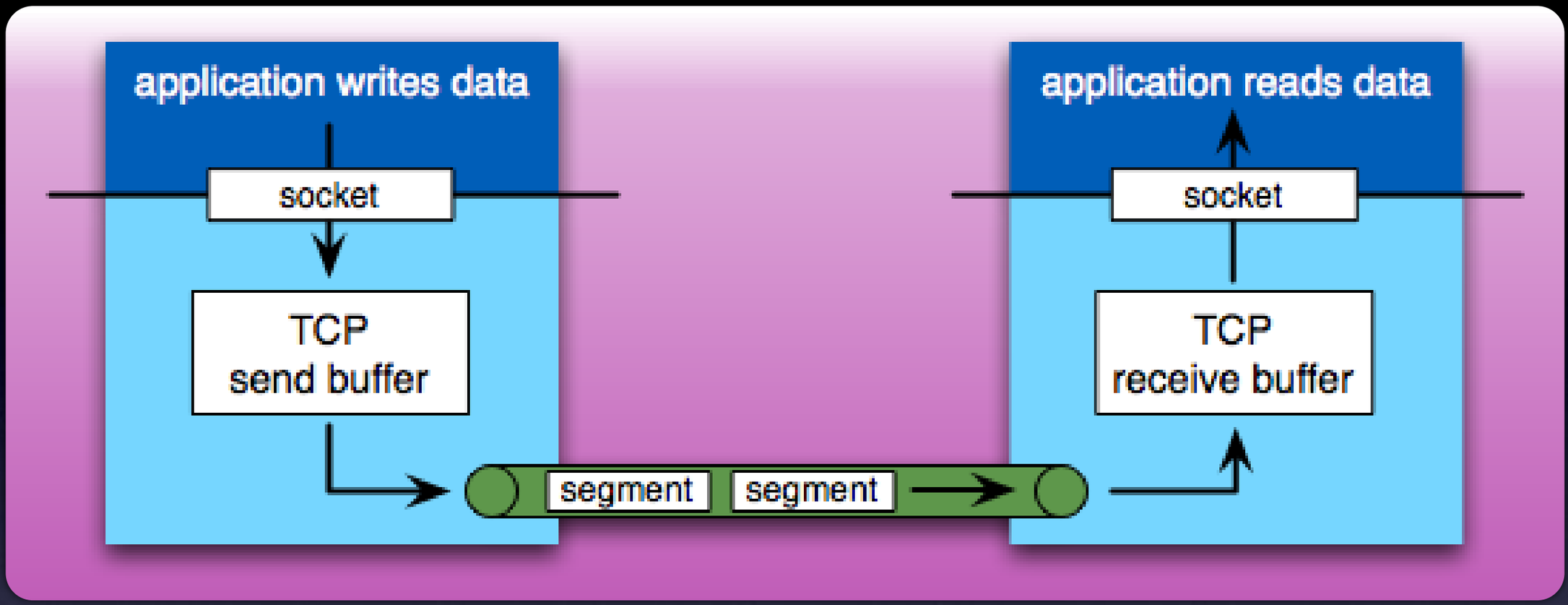
- Connection-oriented Transport: TCP
- Segment Structure
- Connection Management
- Reliable Data Transfer

TCP Overview

- Point-to-point
  - One sender, one receiver
- Reliable
  - Segments delivered in-order without loss
- In-order byte stream
  - No message / record boundaries
- Pipelined
  - Sliding-window type control algorithm

TCP Overview

- Full duplex data
  - bi-directional data flow in the same connection
- Connection-oriented
  - handshaking (exchange of control messages) initializes sender & receiver state before data exchange



- Both sides have buffers
  - Lots of “Producer-Consumer” coordination problems to overcome

Transmission Control

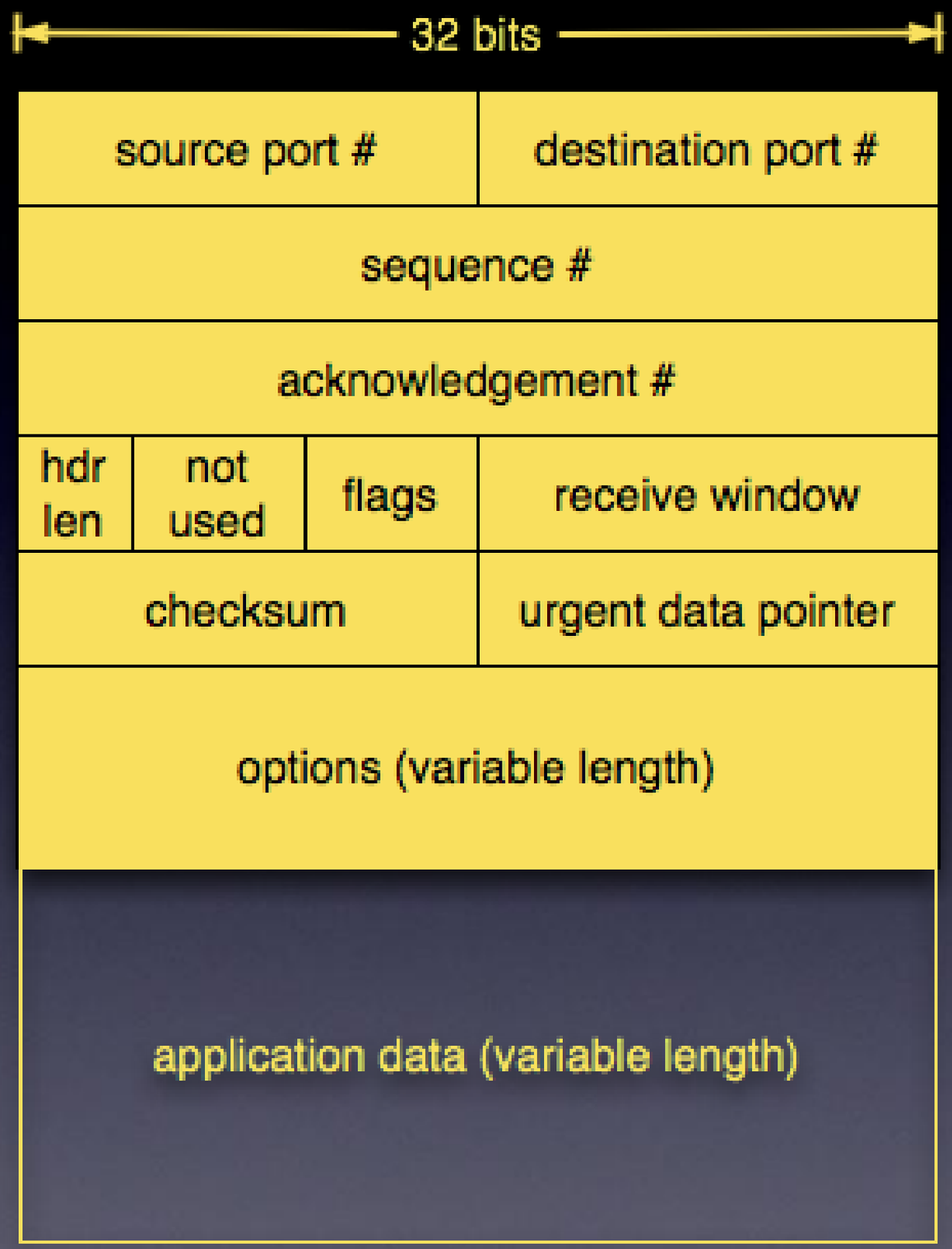
- Flow Control
  - Sender will not overwhelm receiver
- Congestion Control
  - Various algorithms employed to limit sending of segments
  - Don’t want to overwhelm the network

traceroute

- Connection-oriented Transport: TCP
- Segment Structure
- Connection Management
- Reliable Data Transfer

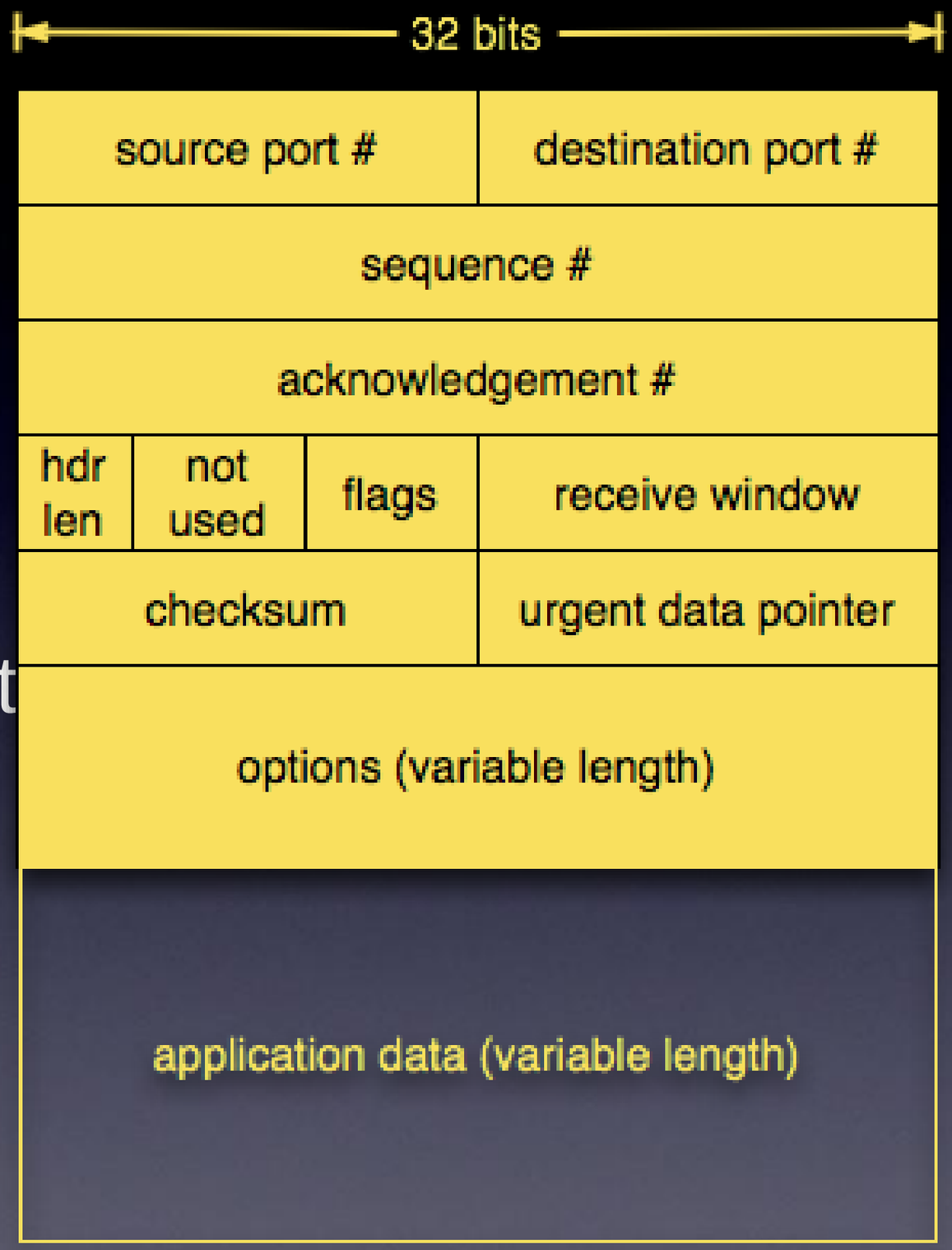
Format

- Port numbers
  - like UDP
  - 16-bit, some are "well-known"
  - not same as UDP



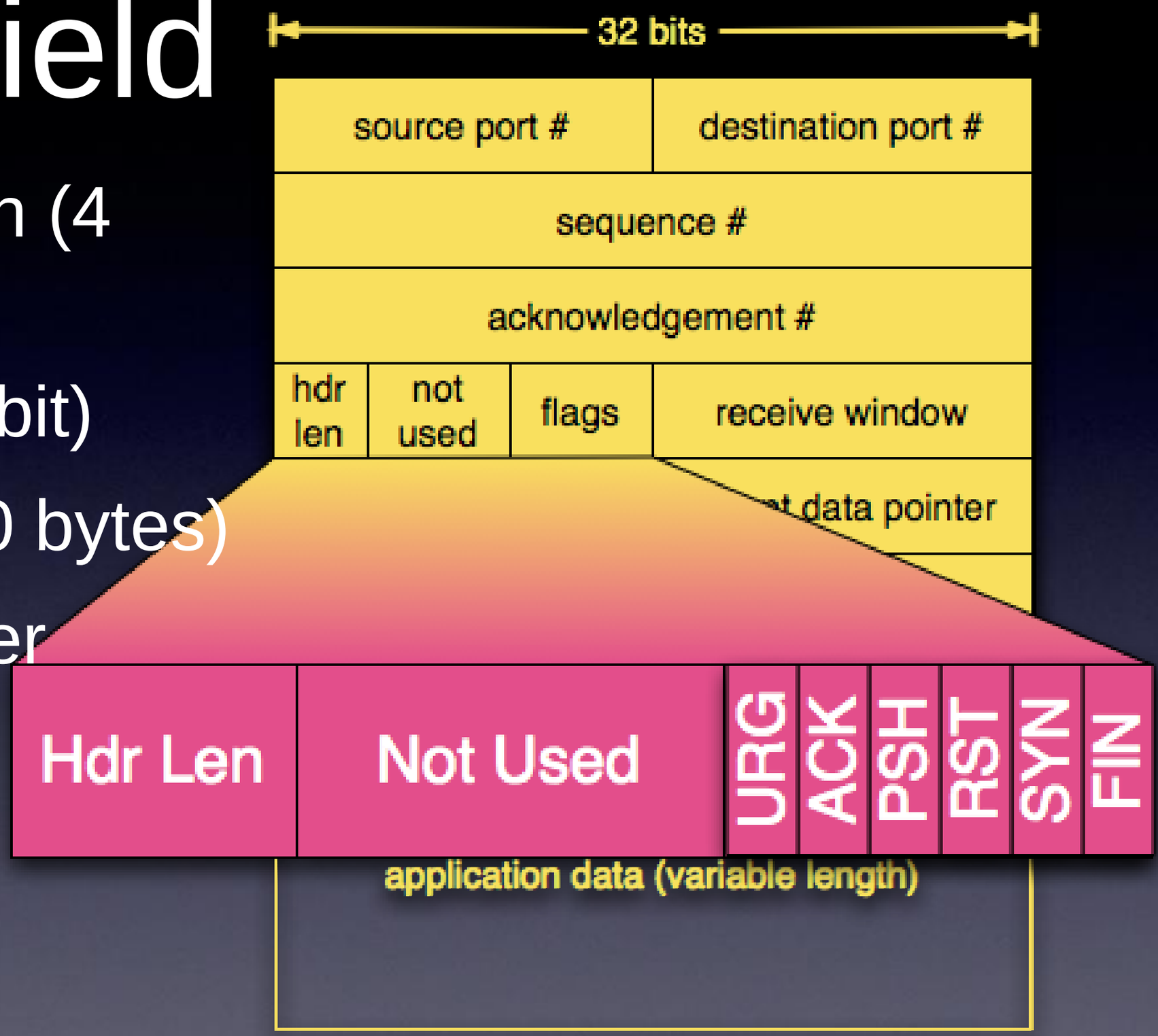
Format

- Seq# and Ack#
  - counted by bytes of data, not segments
- Seq: number of the first byte in this segment
- Ack: number of the NEXT expected byte
  - Acks are cumulative



Flag Field

- Header Length (4 bits)
  - # words (32-bit)
  - Usually 5 (20 bytes)
  - Can be longer with options

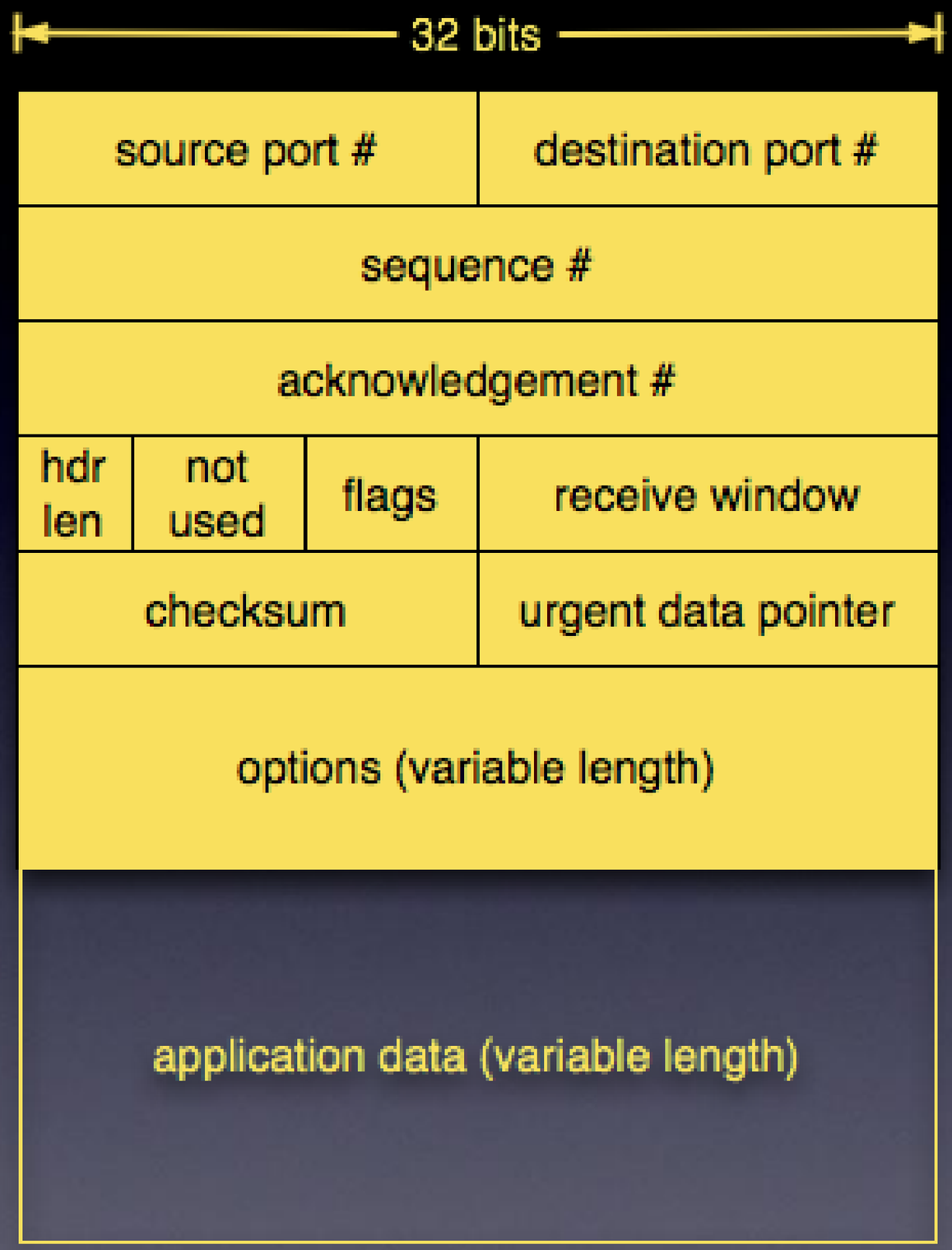


Flag Field

- ACK: Segment acknowledges receipt of another segment (i.e. ack seq# is good)
- Setup and teardown signals
  - SYN: Synchronize seq#
  - FIN: No more data from sender
  - RST: Reset connection
- Rarely used
  - PSH: Push the data to app-layer immediately
  - URG: Urgent data, indicated by Urgent Data Pointer

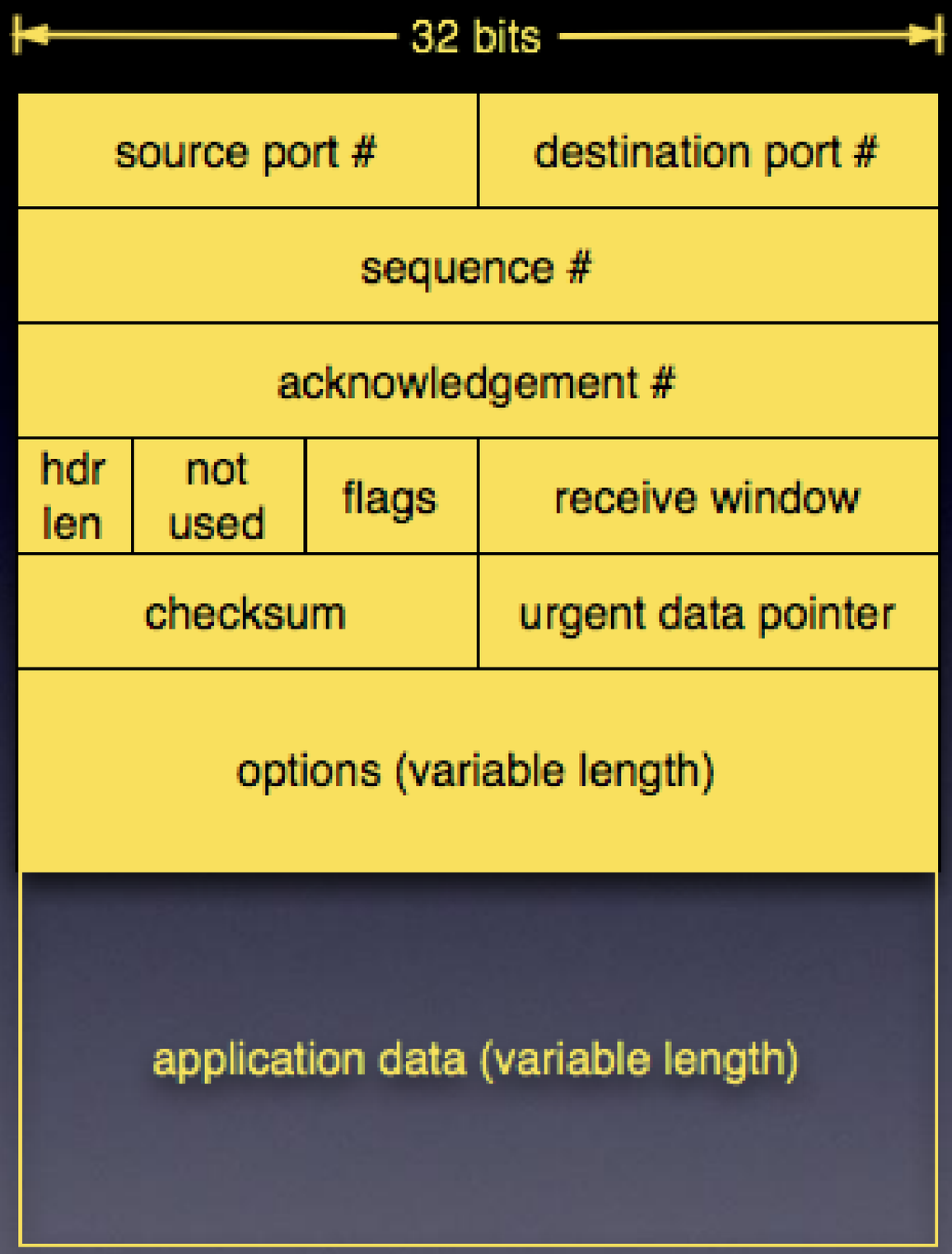
Format

- Receive window
  - #bytes receiver is willing to accept
- For flow-control



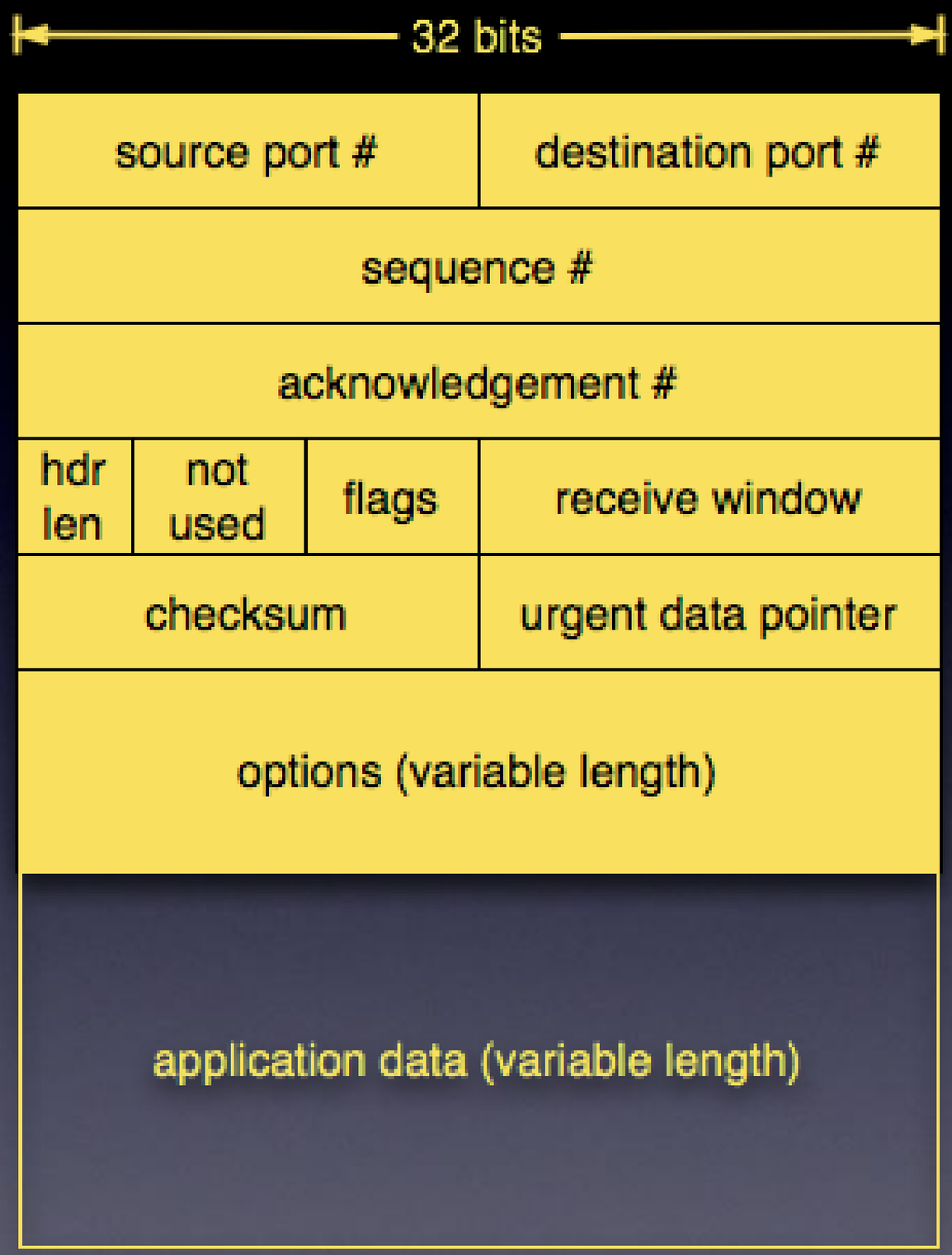
Format

- Checksum
  - Same as UDP
- Urgent data pointer
  - Offset in data field
- Options
  - Time stamps
  - Window scaling factors
  - Negotiating MSS



Format

- Application Data
  - Size limited by MSS
- MSS=Maximum Segment Size
  - Despite name, MSS is most app data that can be carried
- MTU (Max Trans Unit) of lower level generally drives MSS
  - App data + TCP header + IP overhead must fit in MTU
  - MSS is often 1460, 536 or 512 bytes



Question

- Huge file of L bytes, MSS of 1200 bytes, TCP sequence number field of 4 bytes
- What is the maximum value of L such that TCP sequence numbers are not reused?

- Kind of a trick question…
- TCP sequence number is based on number of bytes, not segments → MSS is irrelevant
- Sequence number field is 32-bits, but all sequence number arithmetic is modulo 2^{32} so a theoretical file of L = ∞ bytes is possible

traceroute

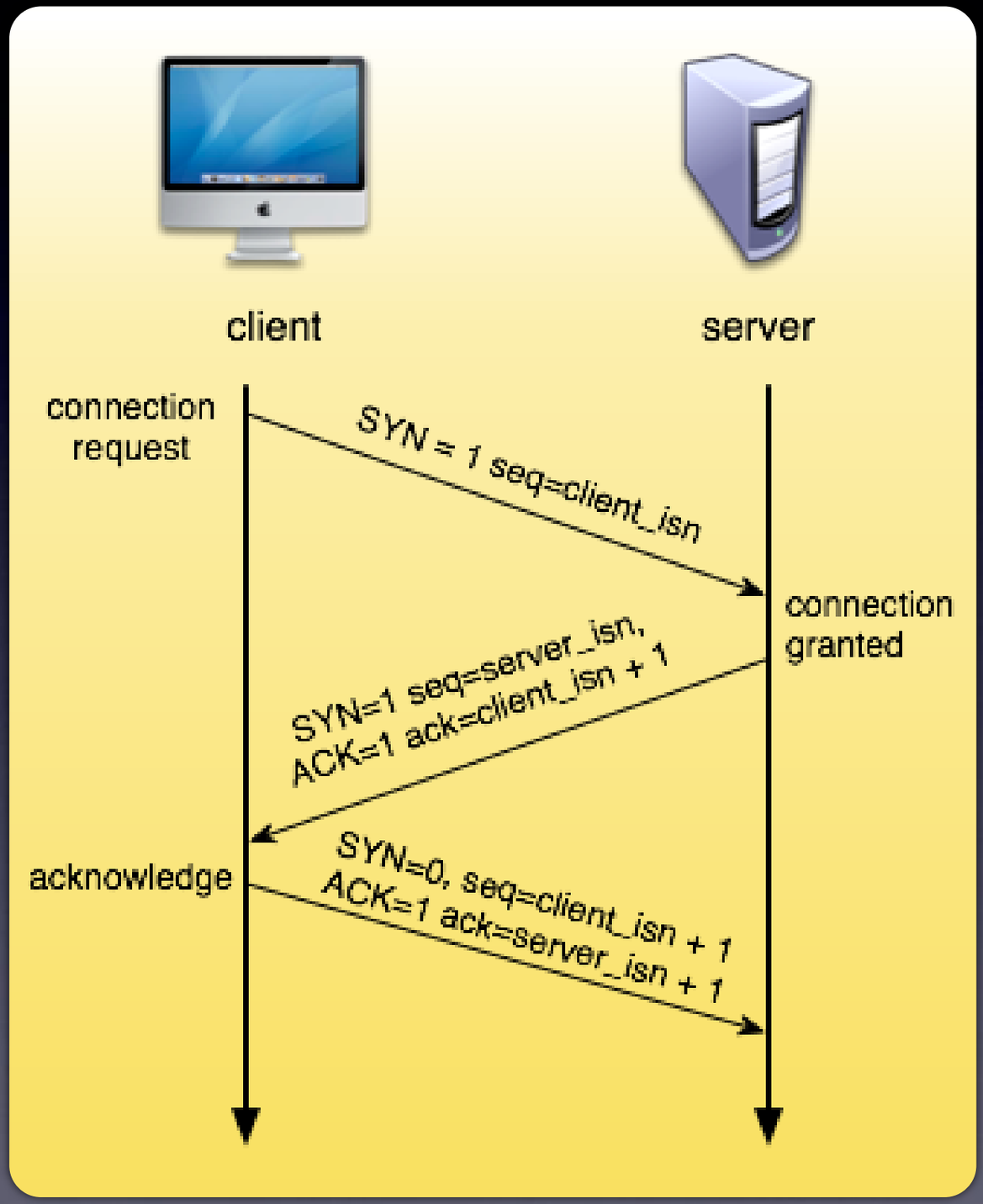
- Connection-oriented Transport: TCP
- Segment Structure
- Connection Management
- Reliable Data Transfer

TCP Connection Management

- Why connection establishment?
  - TCP sender, receiver setup state before exchanging data segments
    - Initialize TCP variables:
      - seq#s
      - buffers, flow control info (RcvWindow)
- Why connection teardown?
  - Free up state

Three-way Handshake

- Step 1: client sends TCP SYN segment to server
  - specifies initial seq#
  - holds no data
- Step 2: server responds with SYNACK segment
  - server allocates buffers
  - specifies initial seq#
- Step 3: Client replies with ACK
  - may contain data



Question

- TCP specification requires each side of a connection to select an initial starting sequence number at random[1]. Why?
- Let’s say TCP does not exchange initial sequence numbers, and just use 0 as the starting point. What can happen?
- An earlier incarnation of the same connection can interfere with a later one

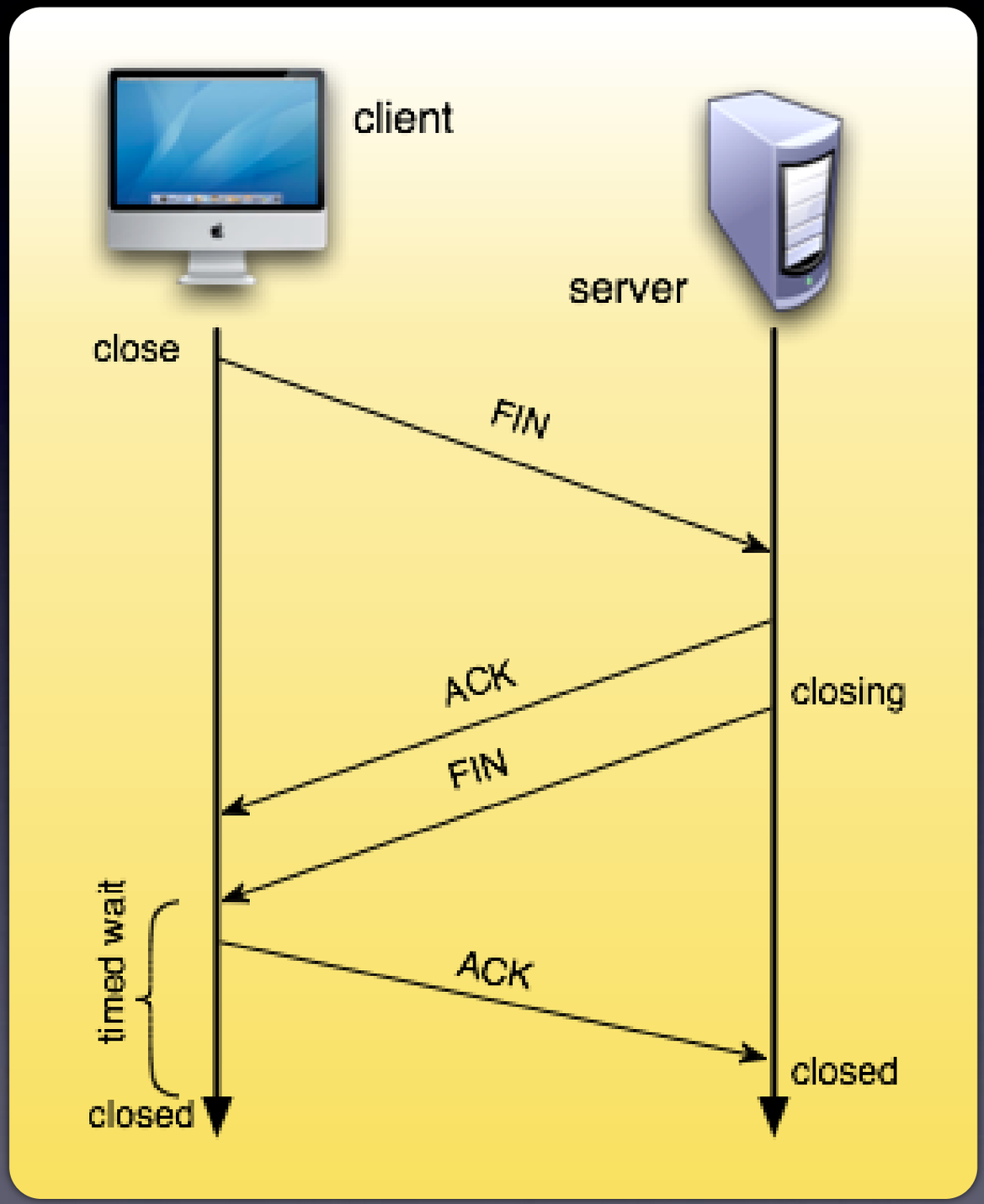
[1]Not exactly, but it's complicated. Let's pretend it is random.

Go Away, Dude

- If a host receives a TCP SYN segment to a closed port, it responds with a RST segment
- UDP sends a special ICMP packet in this situation

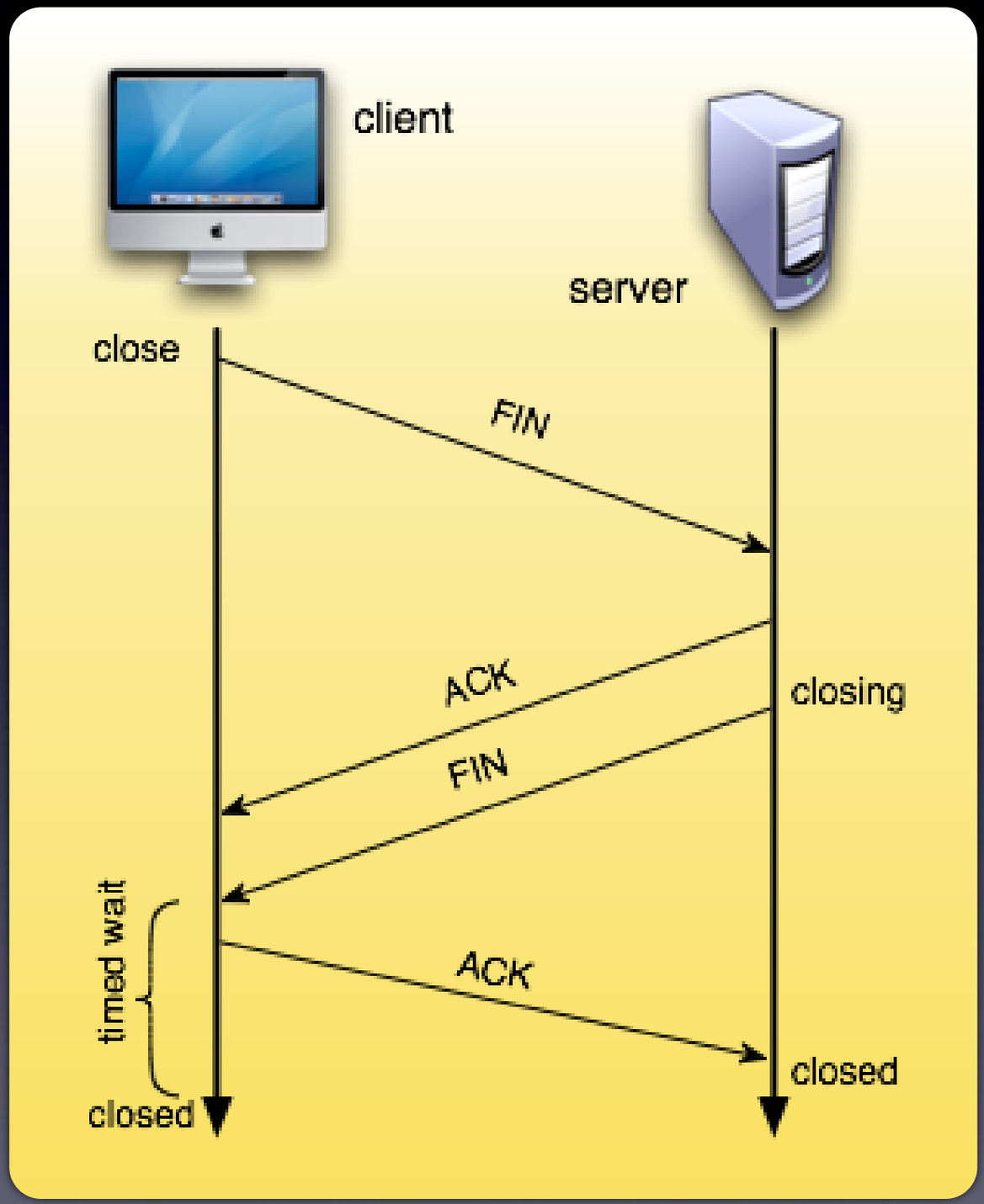
Closing a Connection

- Step 1: Client sends TCP FIN segment to server
- Step 2: Server receives FIN, responds with ACK
  - Closes connection
  - Sends FIN
- Recall: Connection is bi-directional, needs to be shut down from each side



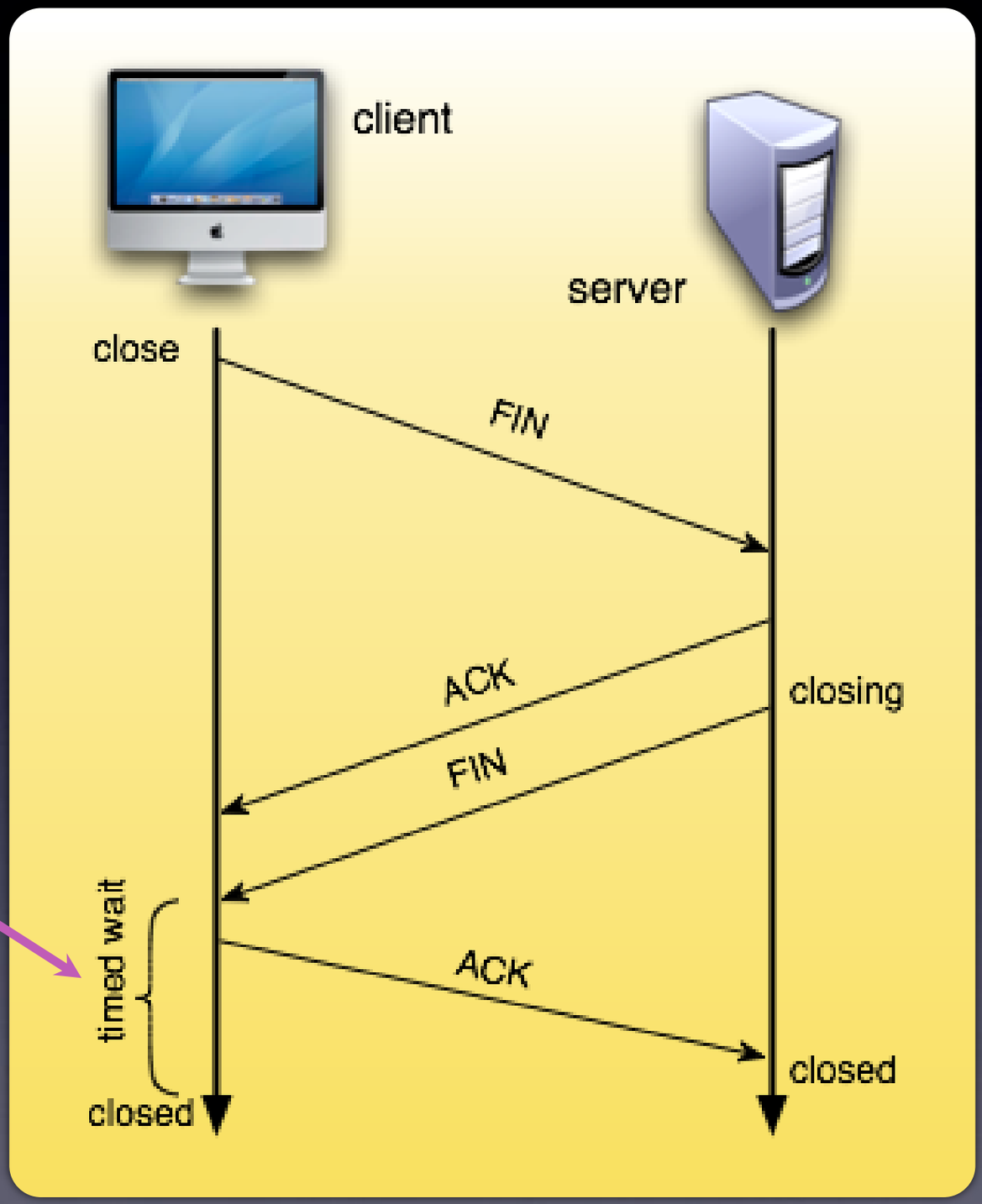
Closing a Connection

- Step 3: client receives FIN, replies with ACK
  - Client enters “timed wait”
  - Will respond to FIN with ACK
  - 240 seconds
    - (2x max time a segment can exist in internet)
- Step 4: server receives ACK, closes connection



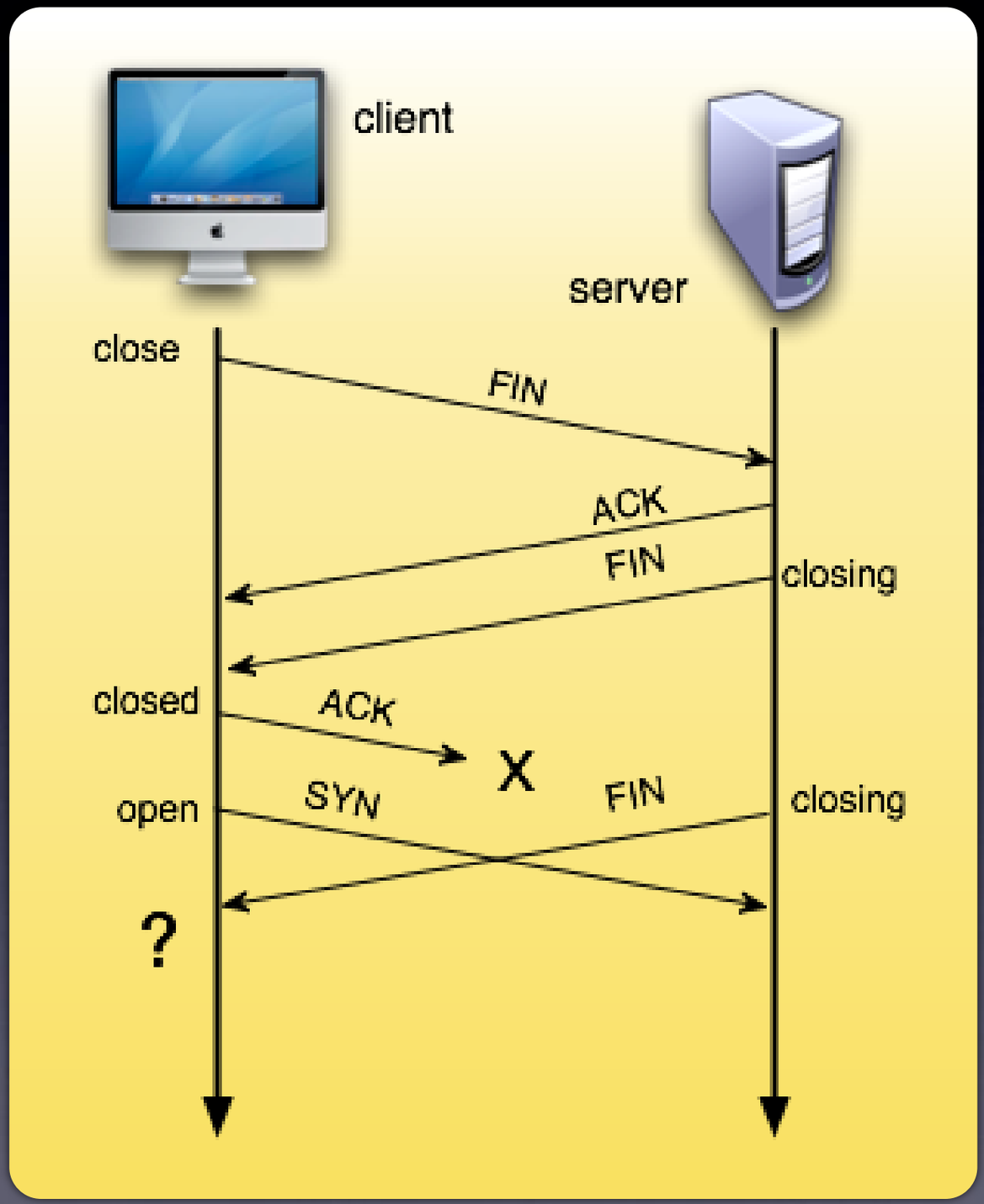
Question

- Why does client enter “timed wait” state before “closed,” even after receiving FIN from server?



Answer

- What happens if ACK from the client is lost, and the client does not wait?
- Client may open the same connection again (same pair of port #s)
- Receives FIN from earlier incarnation of connection
- Immediately initiate closing of the later incarnation



traceroute

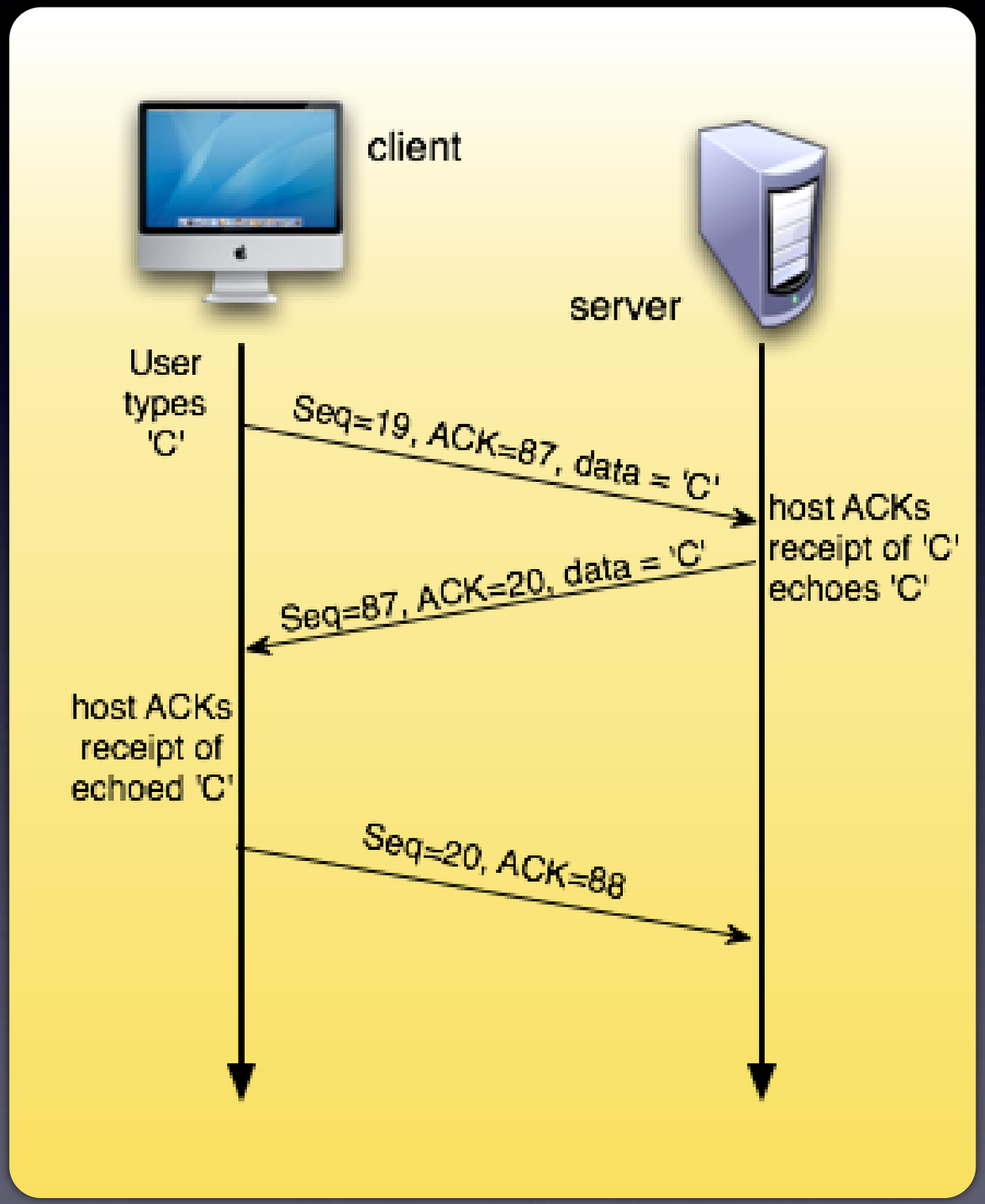
- Connection-oriented Transport: TCP
- Segment Structure
- Connection Management
- Reliable Data Transfer

TCP RDT

- TCP creates RDT service on top of IP's unreliable service
  - Pipelined segments
  - Cumulative acks
  - Retransmission timer
- Retransmissions are triggered by:
  - timeout events
  - duplicate acks
- Initially, we consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control
  - assume RTT is estimated somehow

TCP Seq# and ACKs

- Sequence numbers:
  - byte stream “number” of first byte in segment’s data
- ACKs:
  - seq # of next byte expected from other side
- cumulative ACK
  - acknowledges bytes up to the first missing byte in the stream
- piggybacked
  - with data, if possible



TCP Sender Events

- Data received from app:
  - Create segment
  - seq# is byte-stream number of first data byte in segment
  - Send, if allowed by congestion & flow-control
  - start timer if not already running (think of timer as for oldest unacked segment)
  - expiration interval: TimeOutInterval
- Timeout:
  - retransmit segment that caused timeout
  - restart timer
- ACK received:
  - If acknowledges previously unACKed segments
    - update what is known to be ACKed
    - start timer if there are outstanding segments

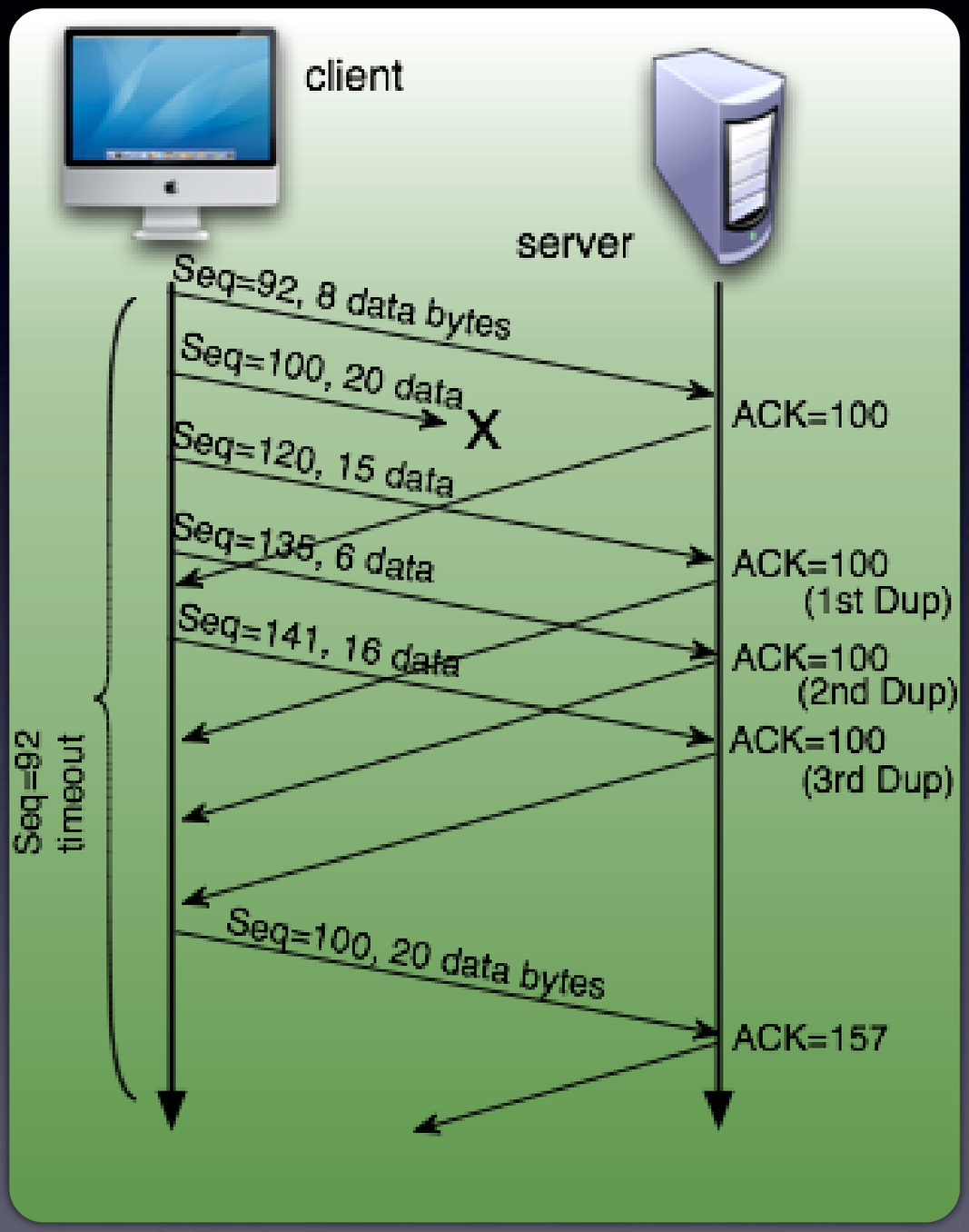
ACK Generation

Event at receiver	Receiver Action
Arrival of in-order segment with expected seq#. All data up to seq# already ACKed	*Delayed ACK.* Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq#. One other segment has ACK pending	Immediately send single cumulative ACK for both in-order segments
Arrival of out-of-order segment higher than expected seq#. Gap detected	Immediately send *duplicate ACK,* indicating seq# of next expected byte
Arrival of segment that partially or completely fills gap	Immediately send ACK, provided that segment starts at lower end of gap

Fast Retransmit

- Time-out period often relatively long:
  - Long delay before resending lost segment
- Detect lost segments via duplicate ACKs
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs
- If sender receives 3 duplicate ACKs, it supposes that segment after ACKed data was lost:
  - Fast retransmit: resend segment before timer expires

Scenario

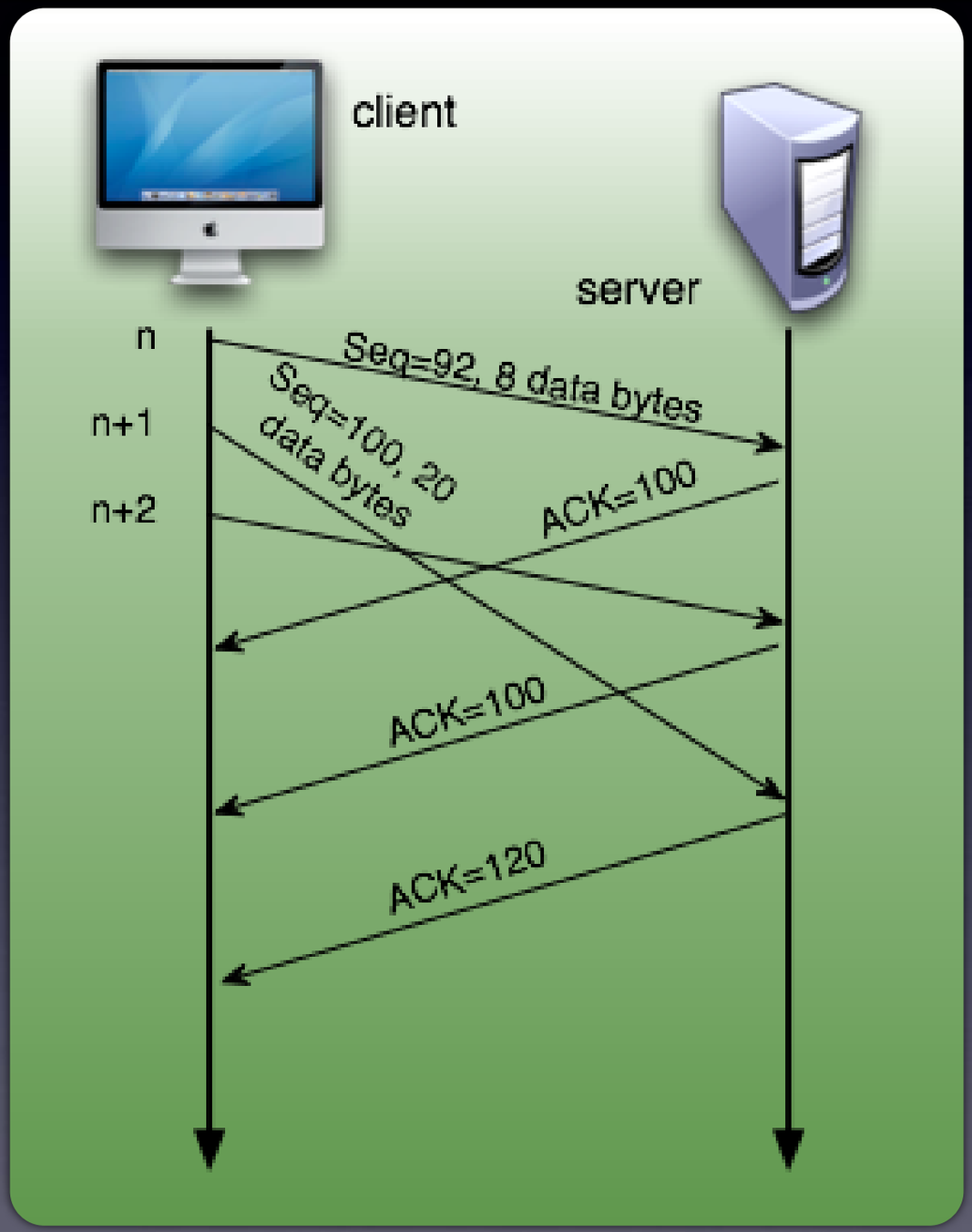


Fast Retransmission

- Sender doesn't have to wait for a timeout to notice probable loss of seq=100 segment
- Sort of a NACK

Why 3?

- Why 3 duplicate ACKs? Why not do fast retransmit after the first duplicate ACK for a segment is received?
- If n+1 and n+2 (or n+3) are just reordered, then waiting for 2 duplicate ACKs will not retrigger retransmission
  - Voodoo constant



Lesson Objectives

- Now, you should be able to:
  - describe the requirements and features of TCP
  - describe the segment format of TCP
  - calculate MSS from the relationship of MTU, Network and Transport header sizes

- You should be able to:
  - describe the operations behind establishing and tearing-down a TCP connection
  - describe the operation of sender and receiver in reliably transferring data across the TCP connection. This description should include events occurring at the sender (including fast retransmission optimizations) and receiver, as well as scenarios whereby error conditions are overcome

Flow and Congestion Control (Hosts)

14-740: Fundamentals of Computer Networks

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

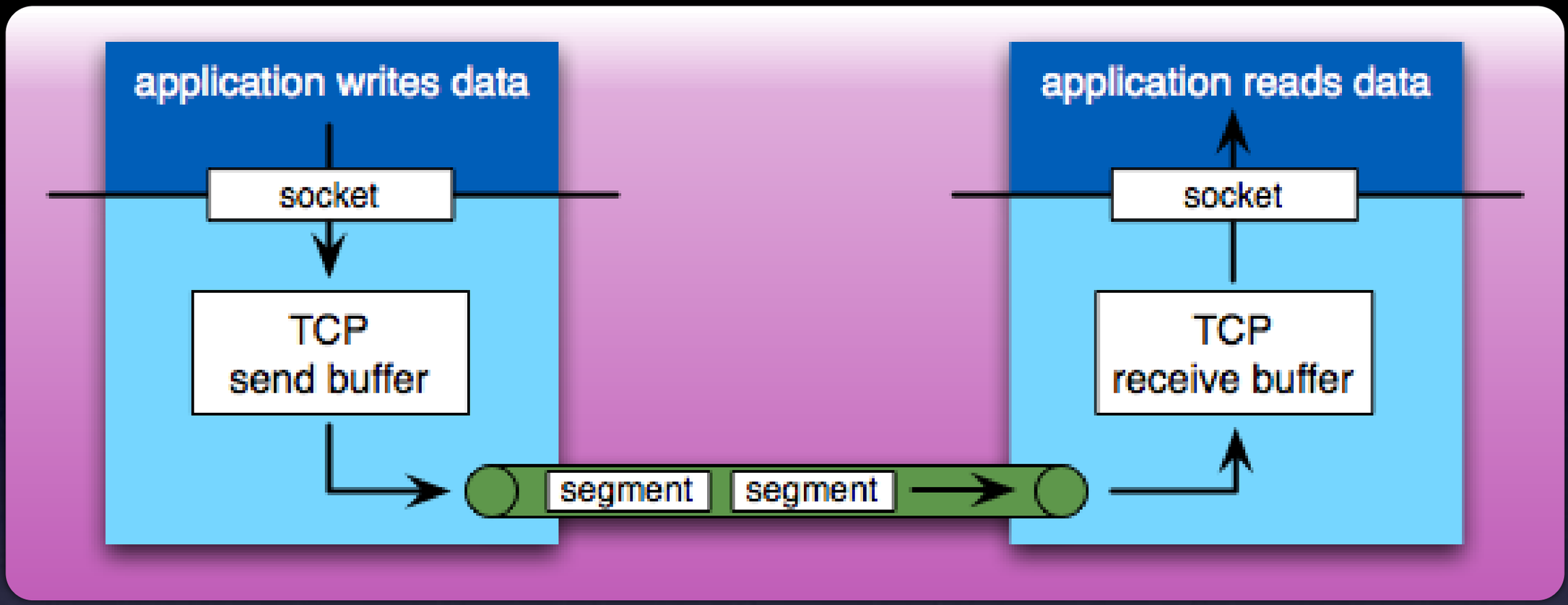
traceroute

- 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 Window
  - Congestion Control Window
- Sender limited by smallest window

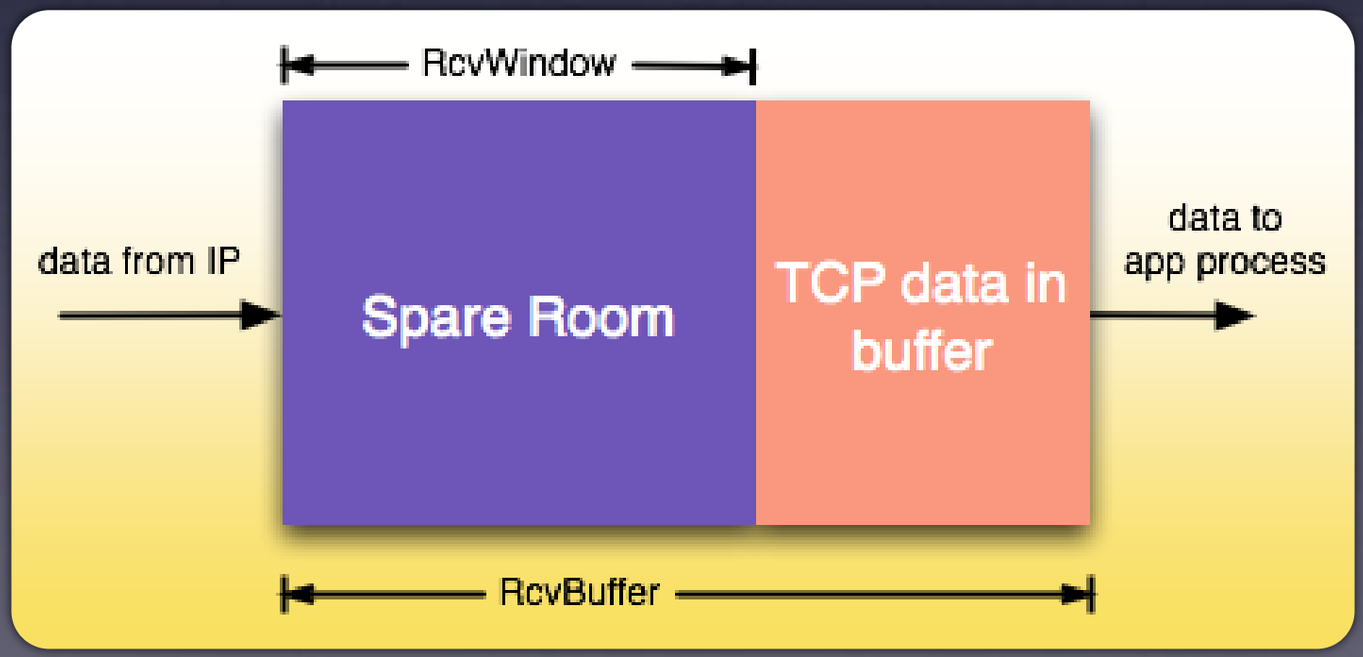
Flow Control != Congestion Control



- Both sides have buffers
  - Lots of “Producer-Consumer” coordination problems to overcome

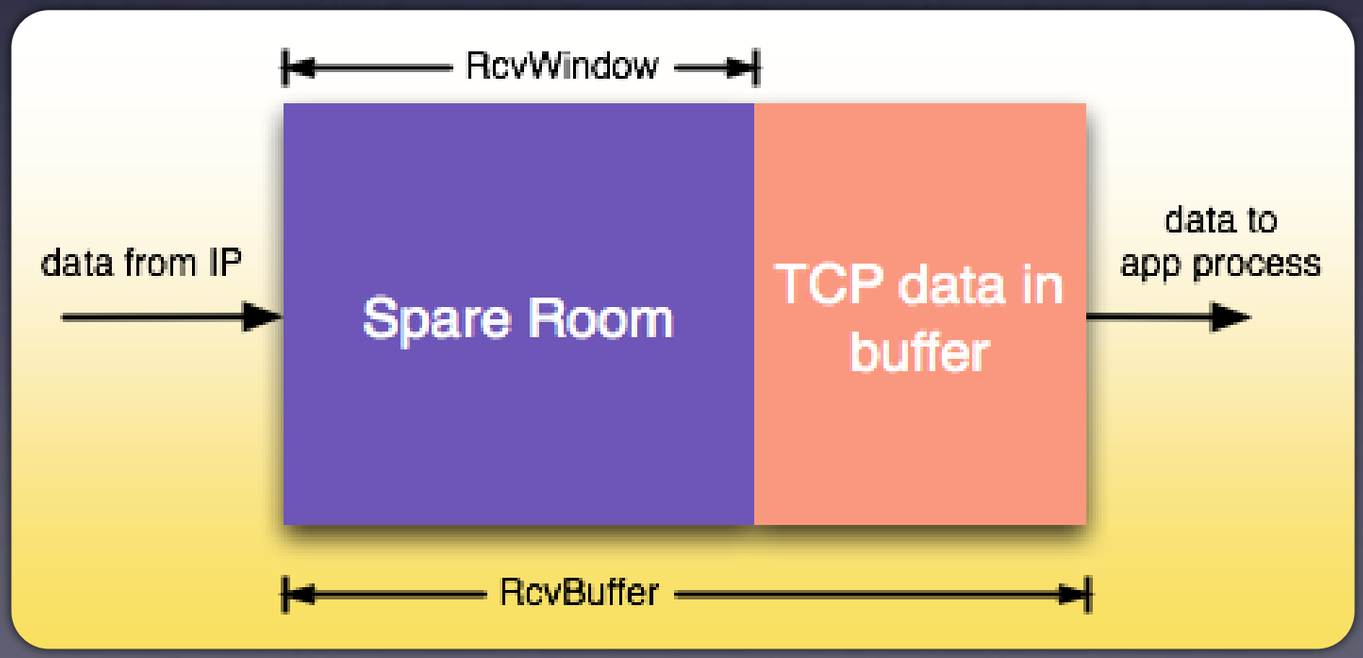
Flow Control

- 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



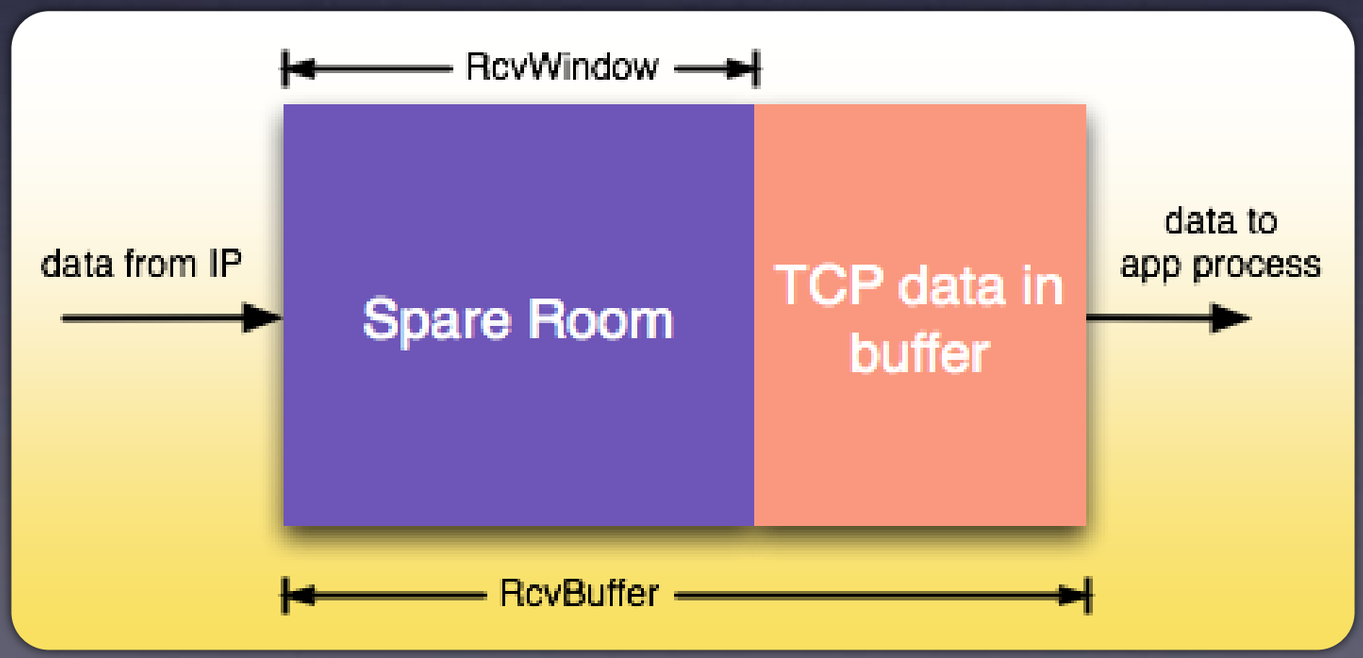
Mechanism

- Spare room in buffer

= RcvWindow

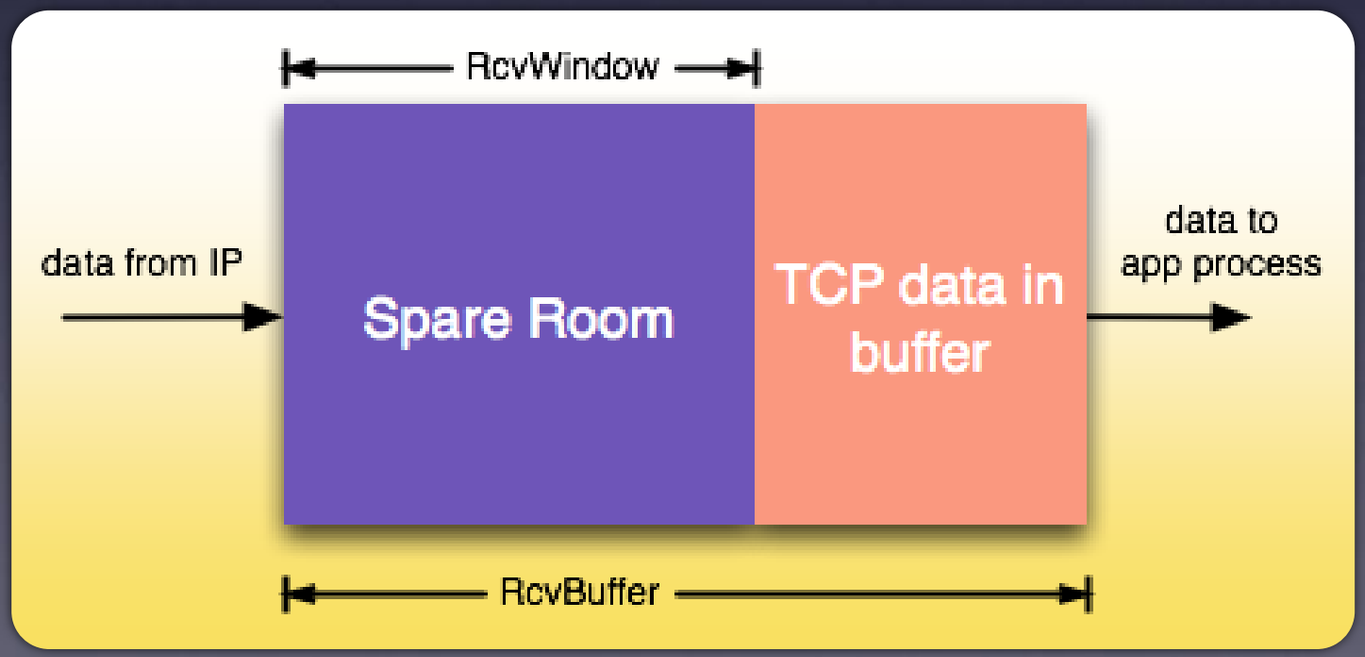
= RcvBuffer - [LastByteRcvd - LastByteRead]

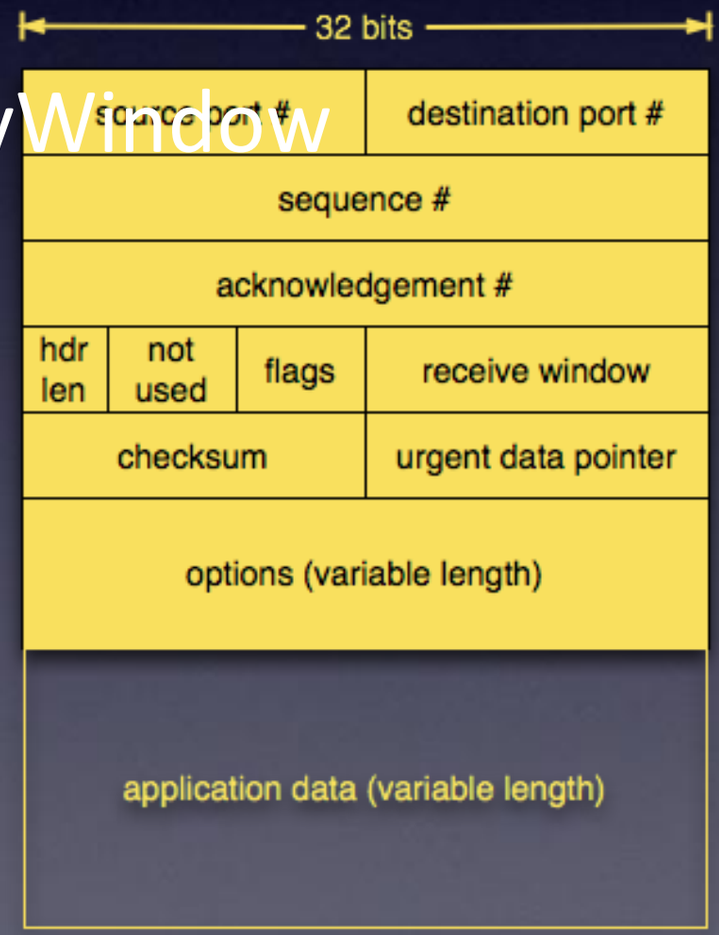
Yes, this assumes receiver discards out-of-order segments. Easy enough to program around simplification



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

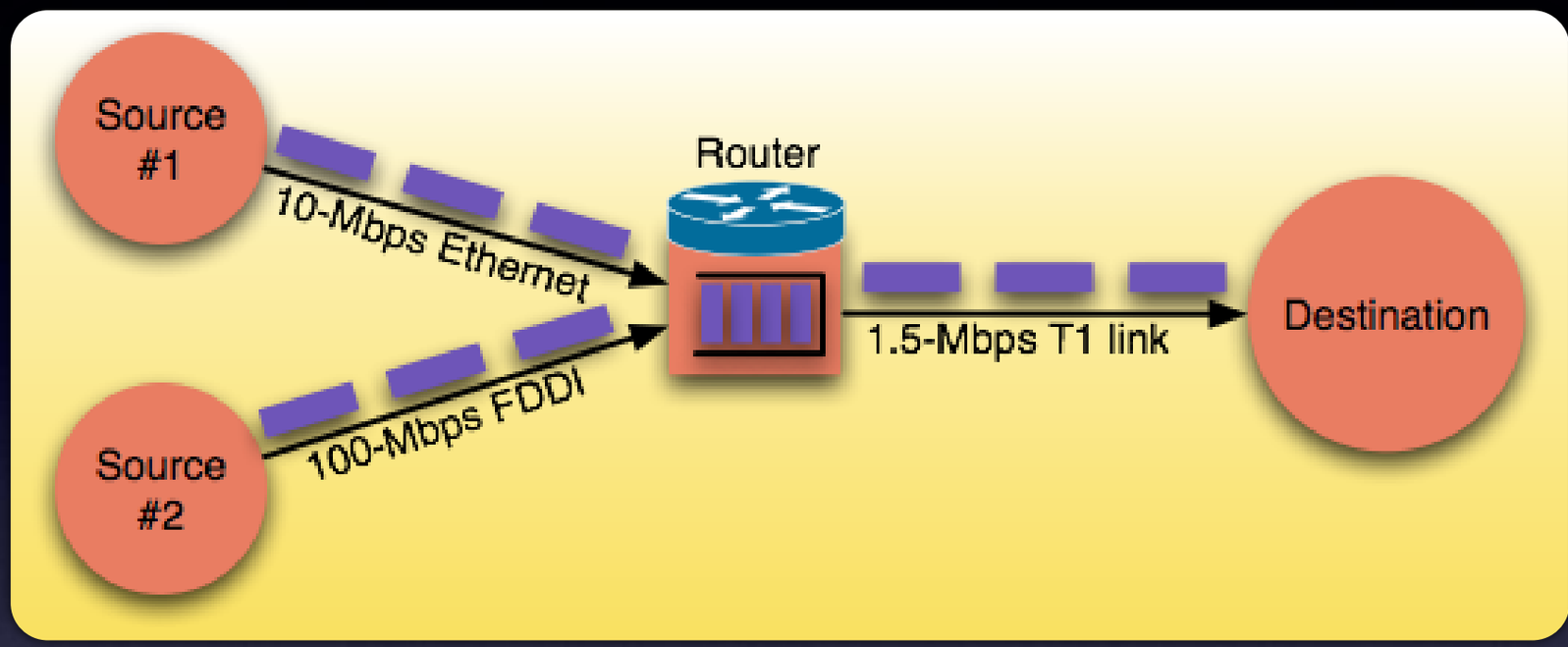




traceroute

- 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

Costs of Congestion

- Packets can experience large queuing delays
  - Even if not dropped (adv topic: Bufferbloat)
- Packets can be dropped due to buffer overflow
  - Sender must perform retransmission of lost segments
  - Retransmission might not be necessary but was triggered by premature timeout
  - Wasted network resources

Congestion

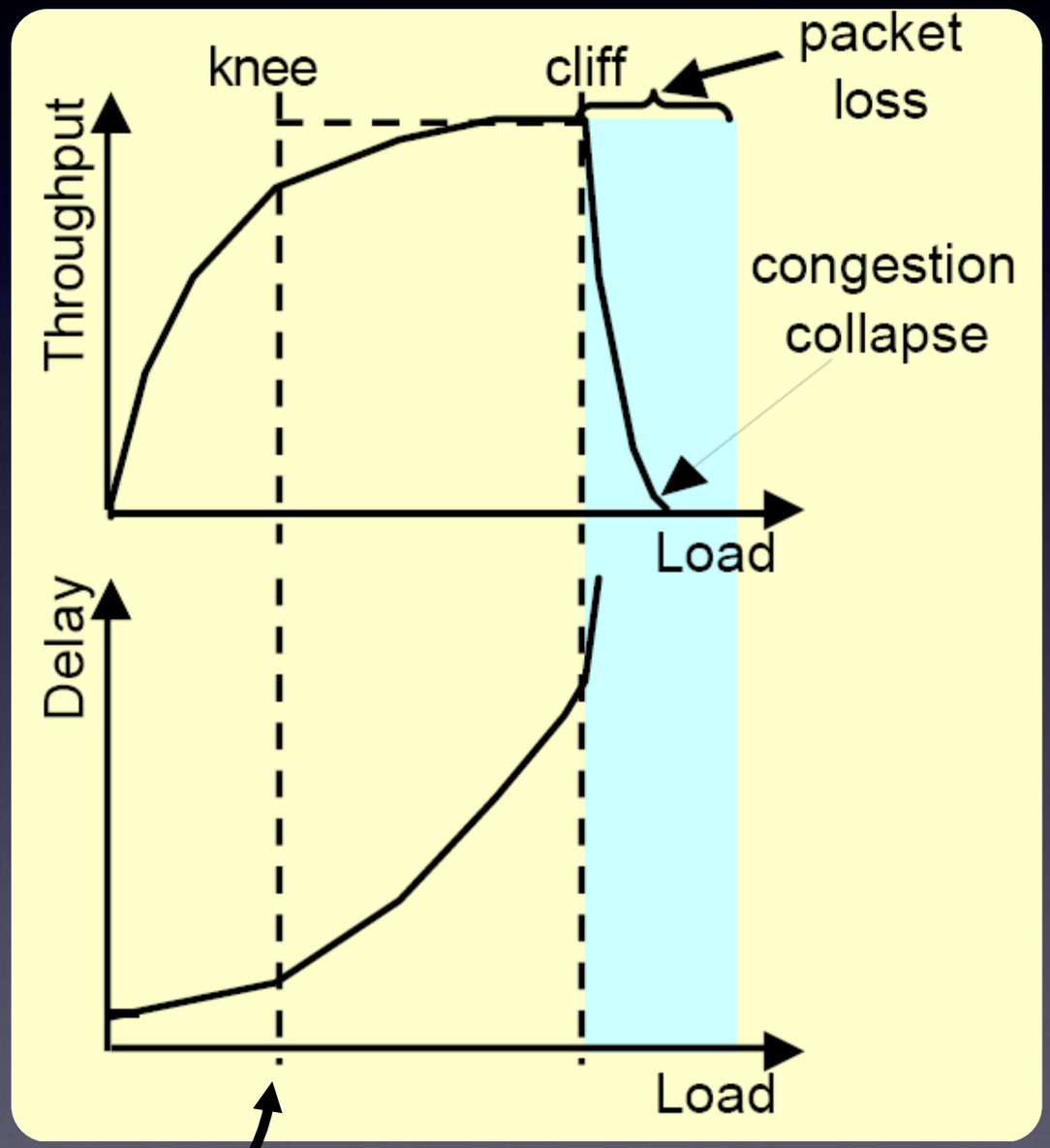
- Too many sources sending too much data too fast for network to handle
  - Different than flow control!
- As network load increases, some router has many packets queued in its buffer, resulting in:
  - Long delays – as packets wait for processing
  - Lost packets – as buffer space overflows, cannot 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?



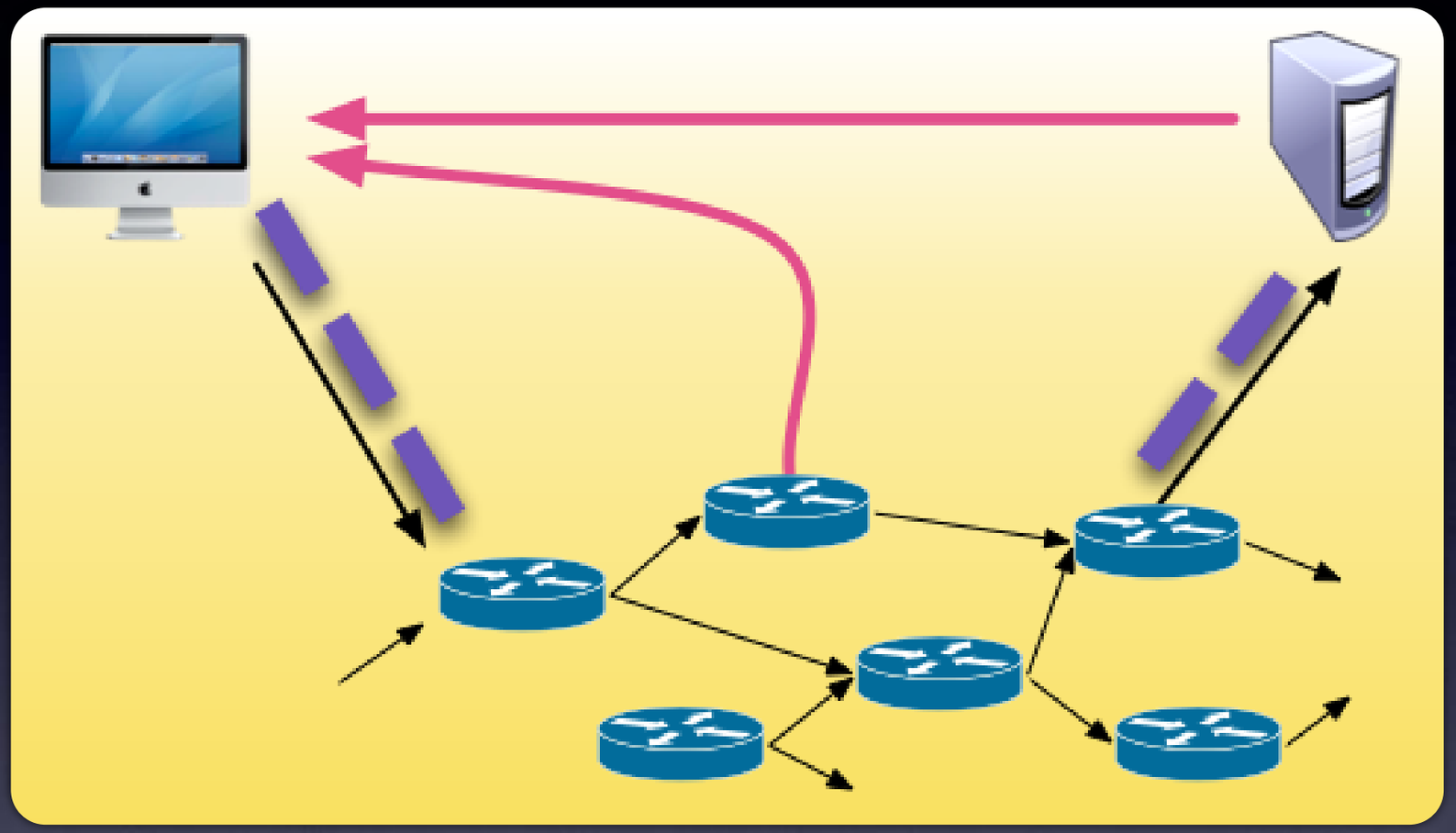
How do we control?

- 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

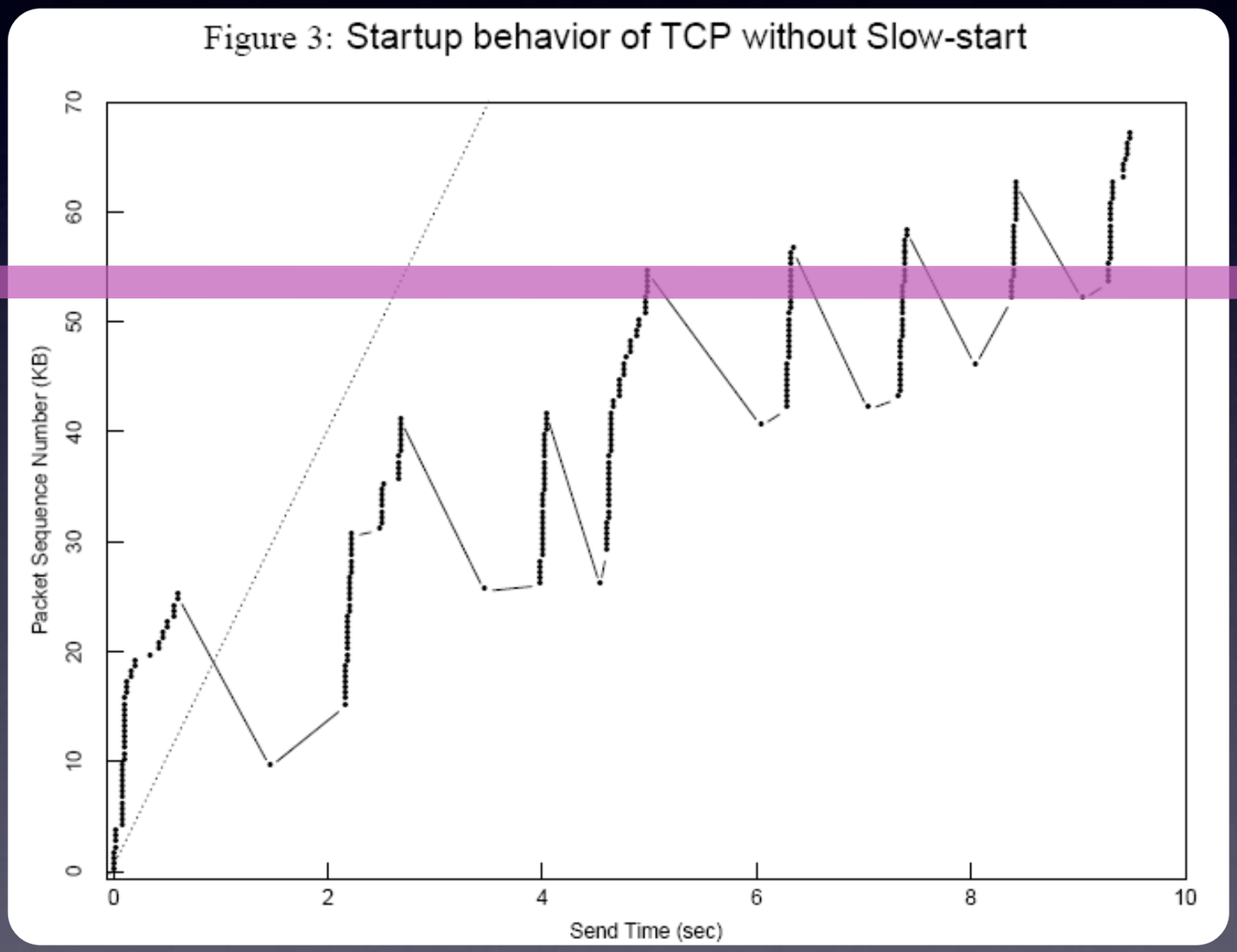
- Slow start – Getting to equilibrium
- Additive-increase, multiplicative-decrease (AIMD) – Adapting to path (avoiding congestion)
- RTT estimation – Conservation at equilibrium
- Reaction to timeout events

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



Some segments got sent 5 times!

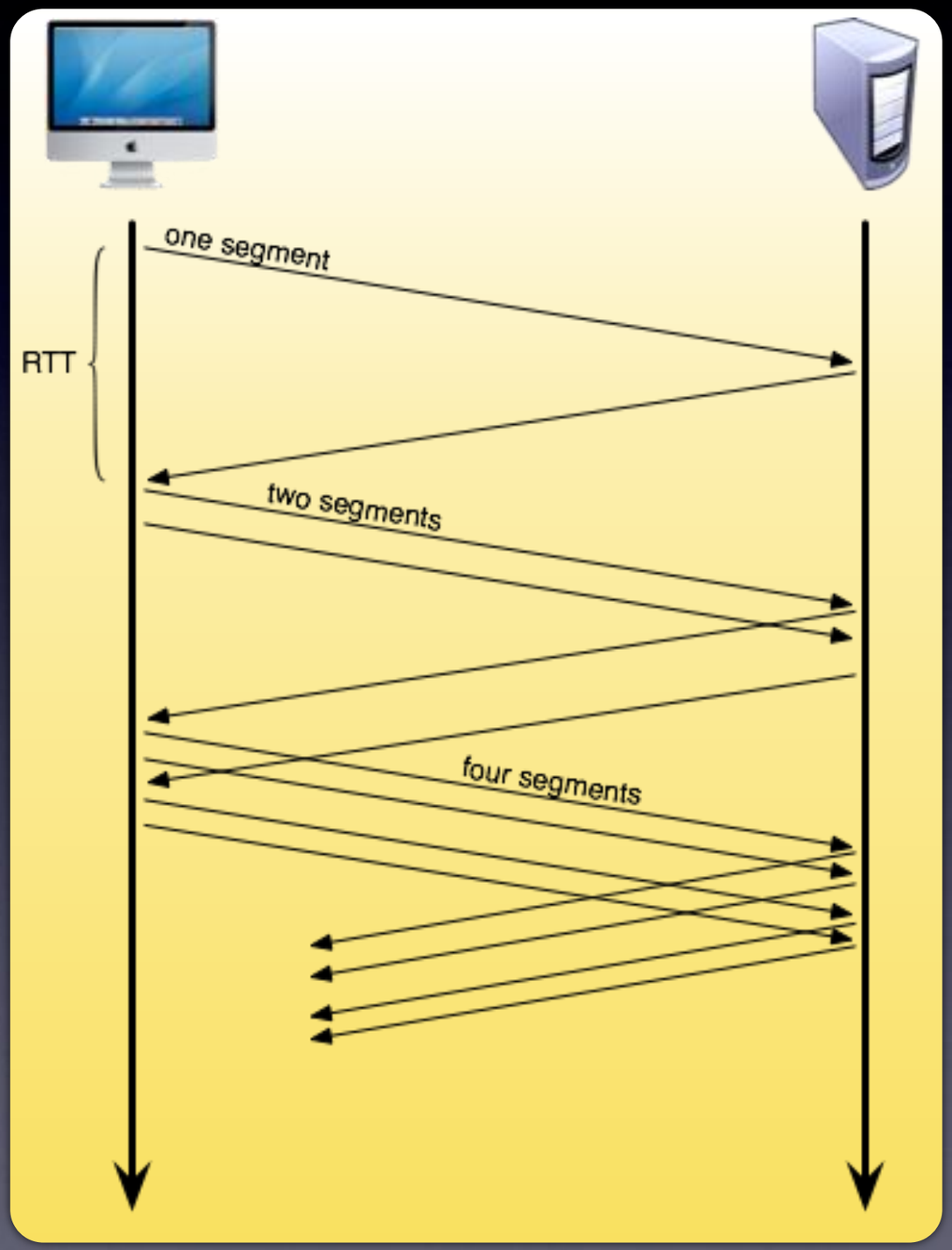
"Nothing in this trace resembles desirable behavior"

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

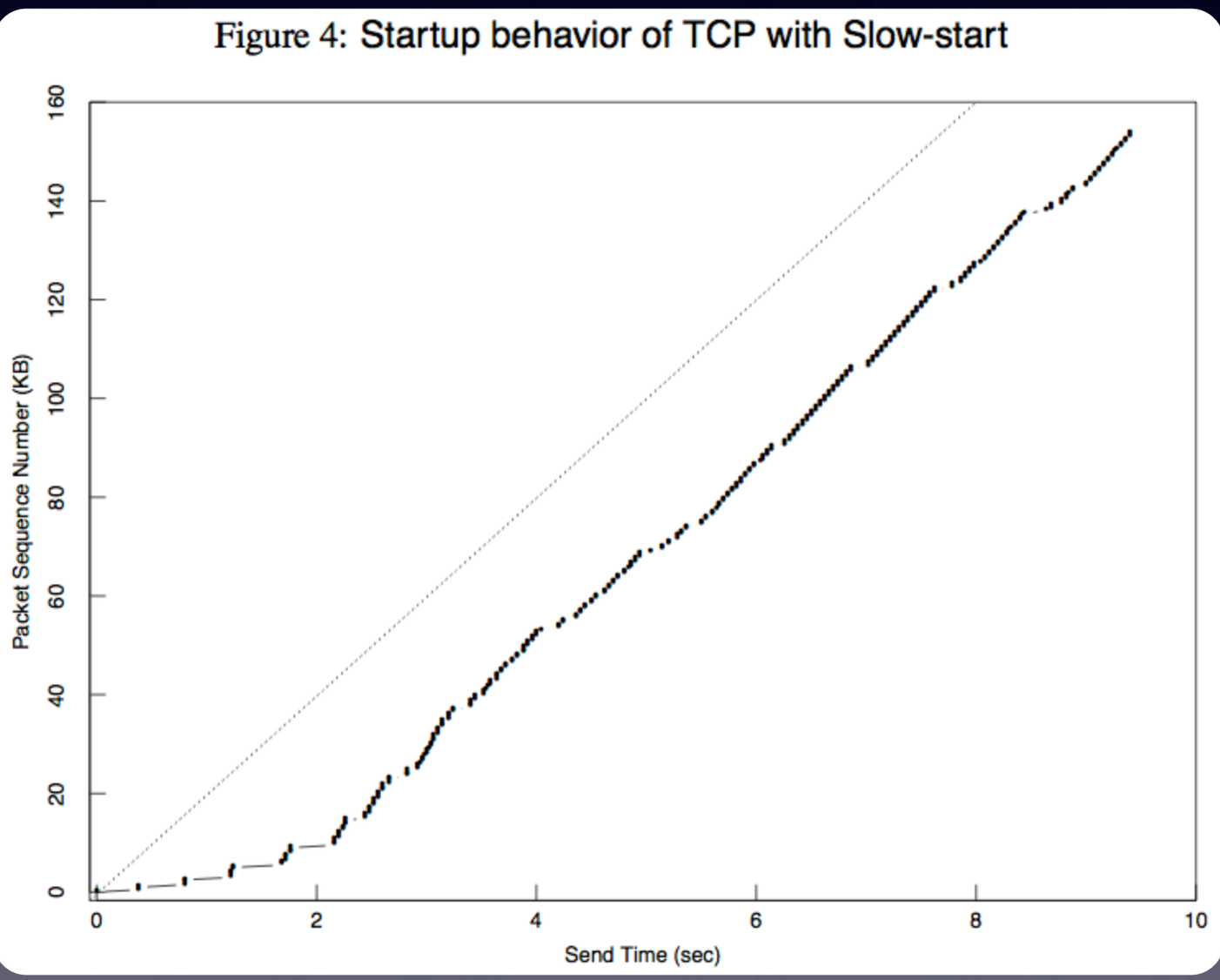
- When connection begins, increase rate exponentially:
  - double **`CongWin`** every RTT
    - done by increasing **`CongWin`** by 1MSS for every ACK received
- Summary: initial rate is slow but ramps up exponentially fast



Behavior of Slow Start

- With Slow Start, no bandwidth wasted on retransmission

Figure 4: Startup behavior of TCP with Slow-start

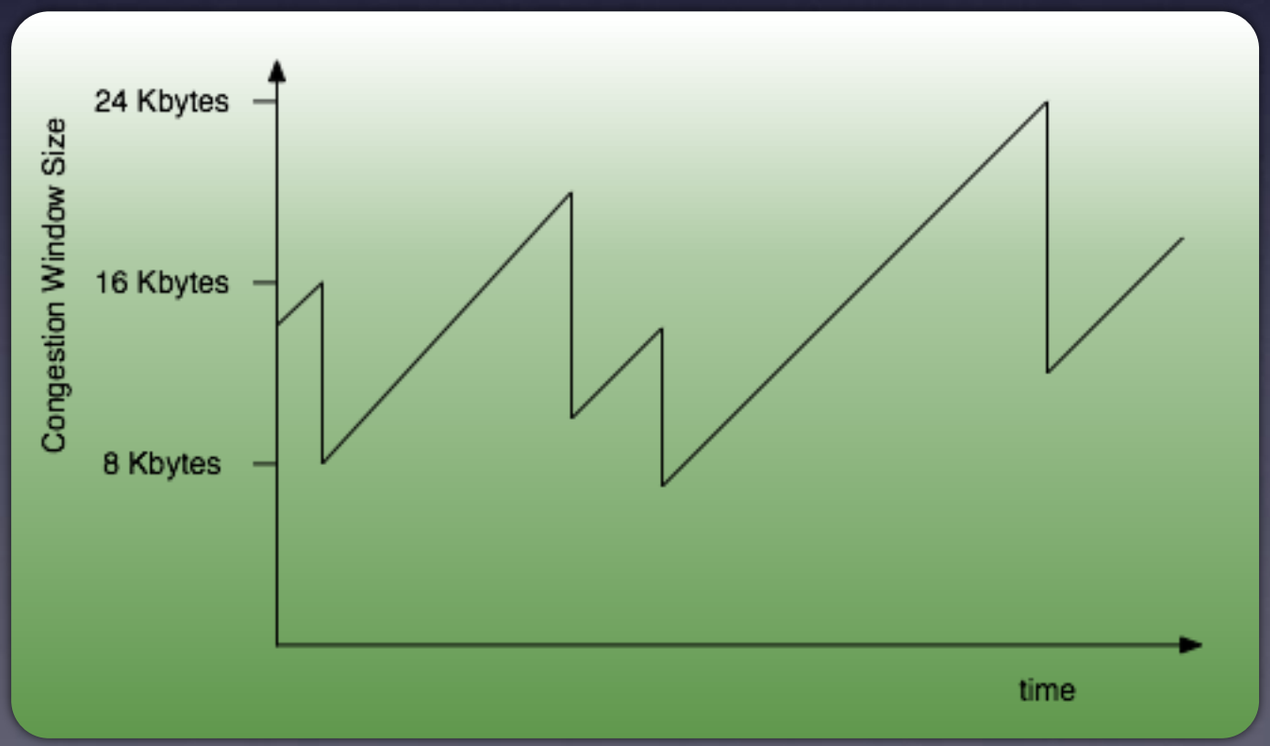


Control at Equilibrium

- 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

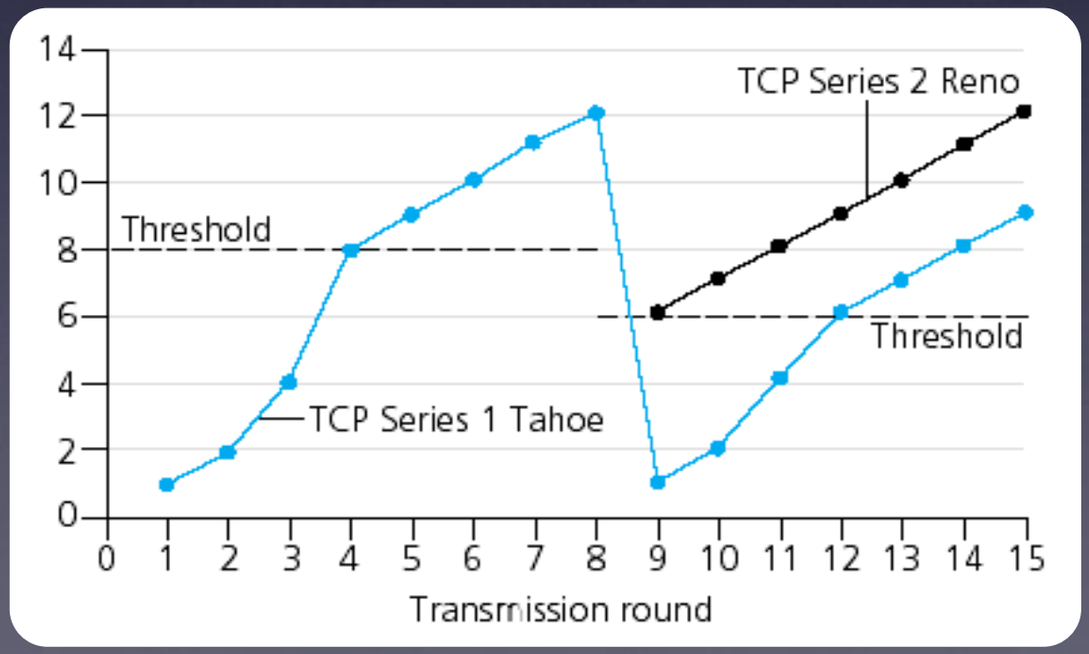
- Additive Increase: Increase **`CongWin`** by 1 MSS every RTT until loss is detected
- Multiplicative Decrease: cut **`CongWin`** in half after a loss event



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

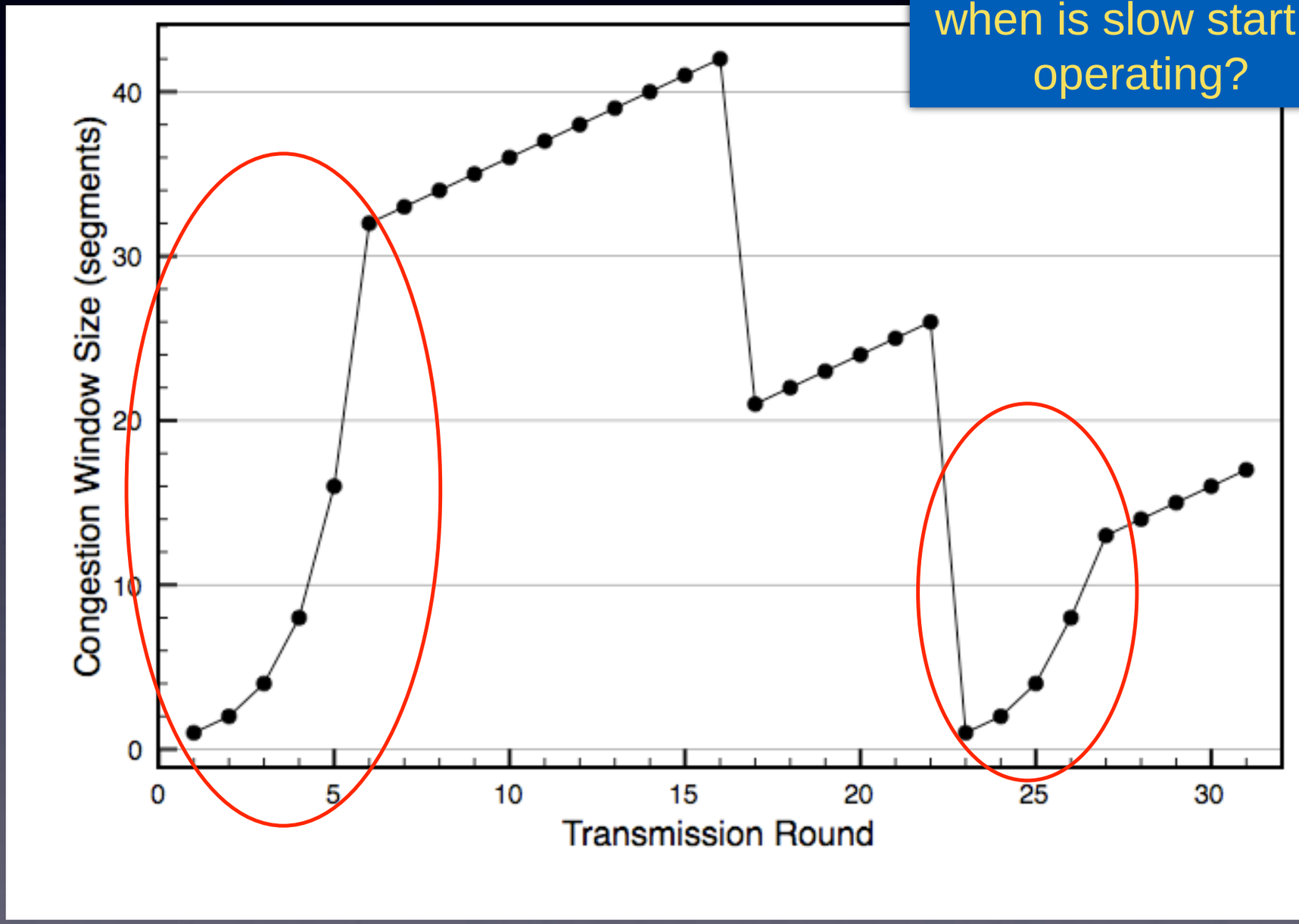
- TCP Reno

Reno's Philosophy

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

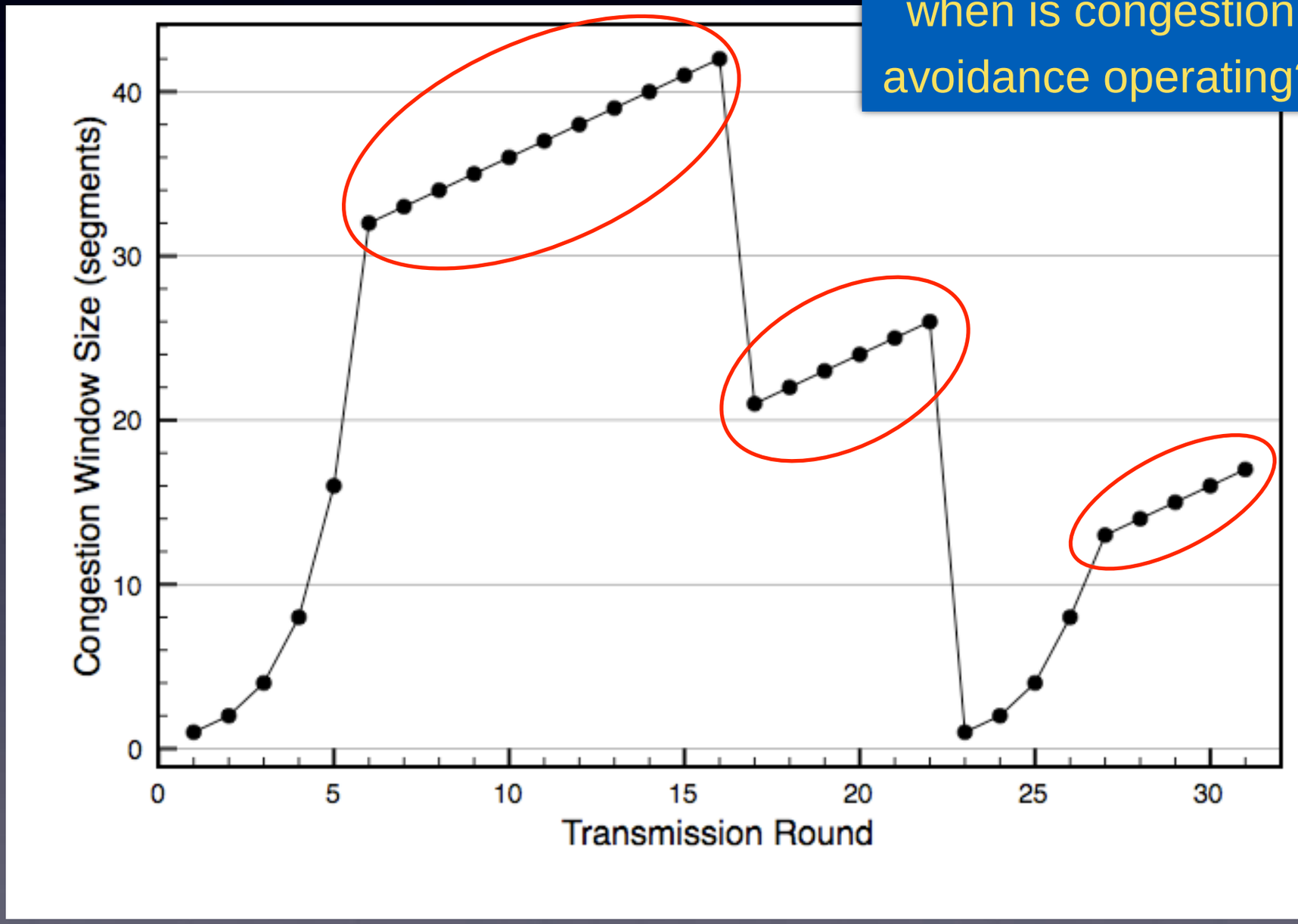
CongWin size plot

when is slow start operating?

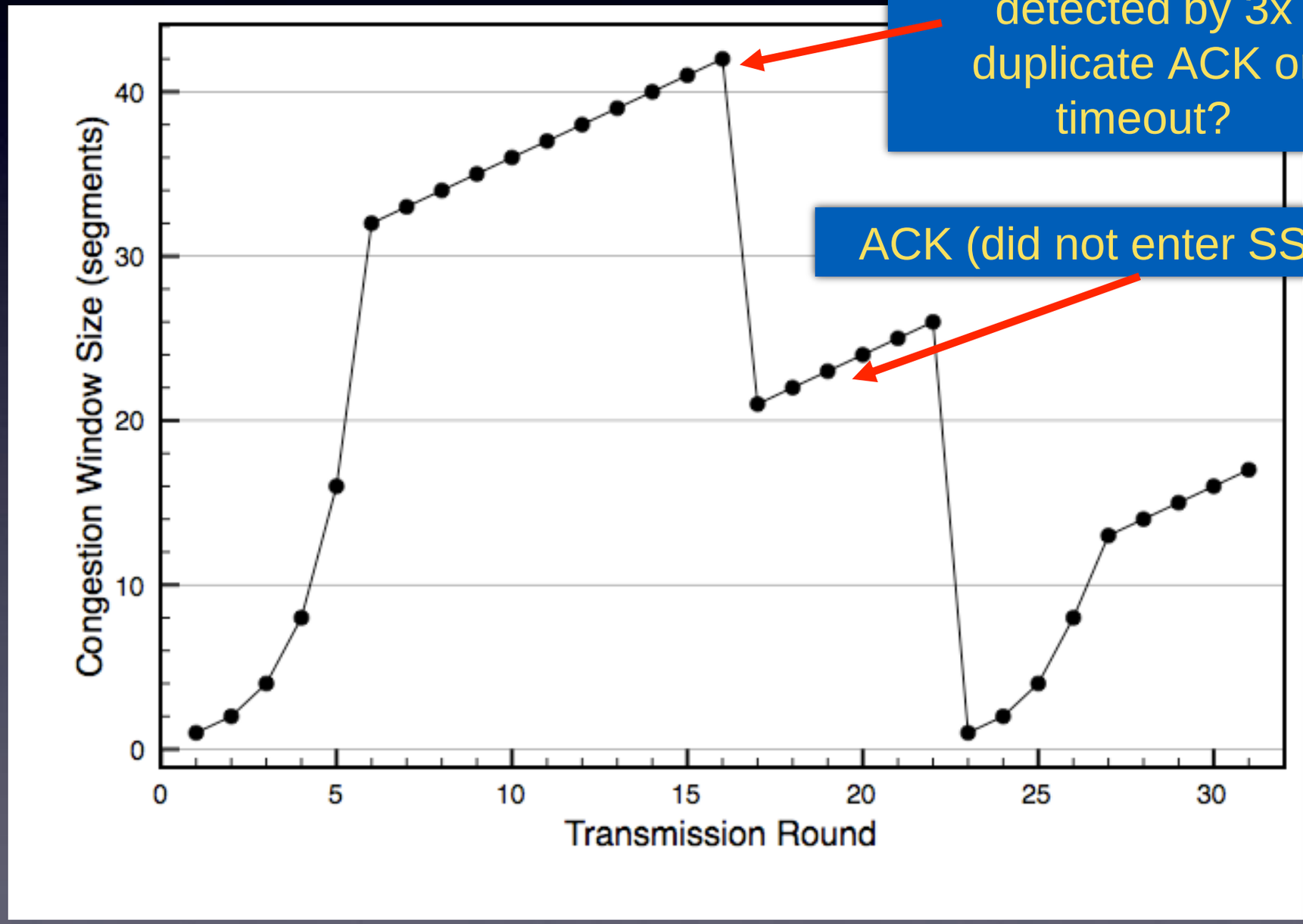


CongWin size plot

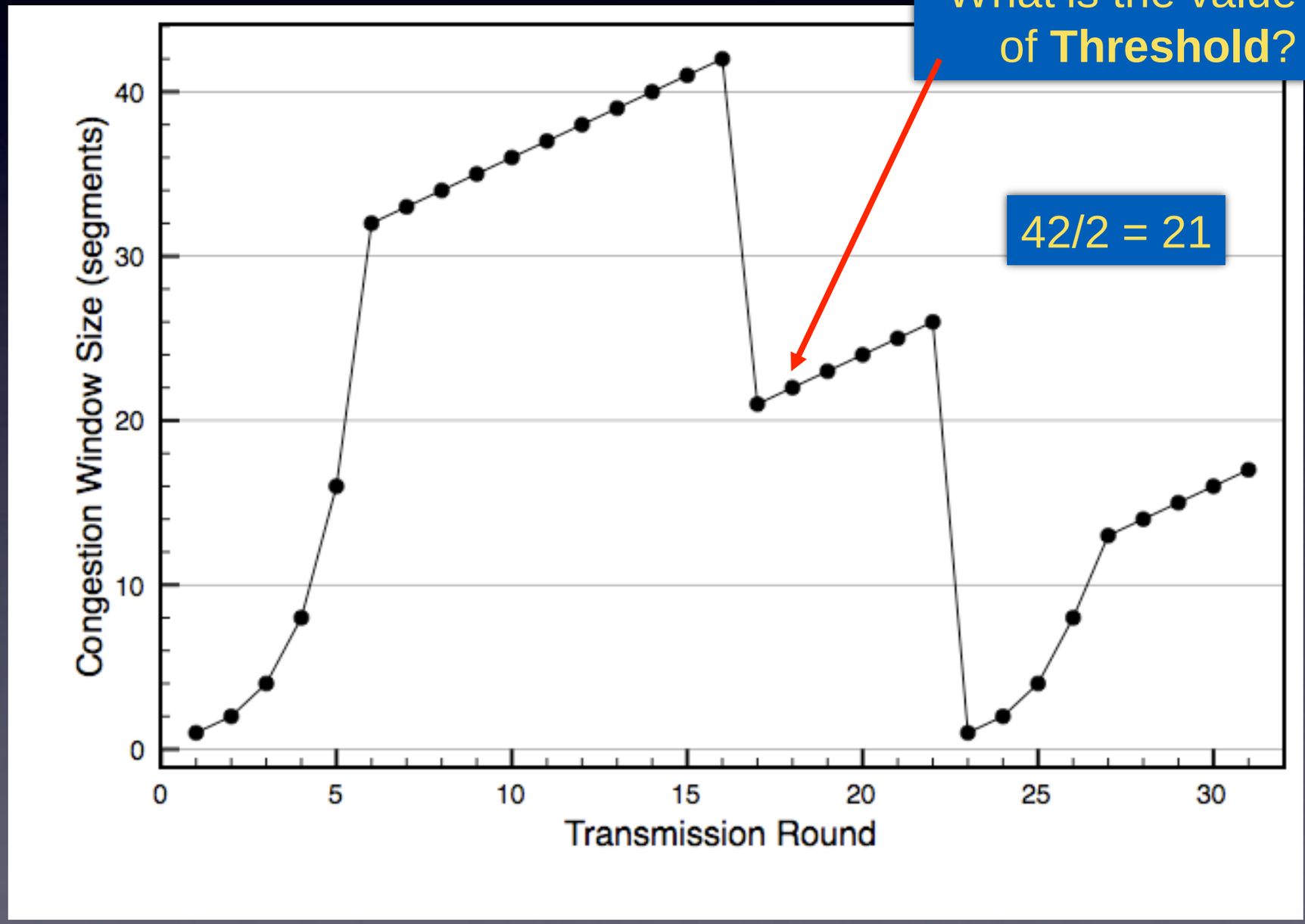
when is congestion avoidance operating?



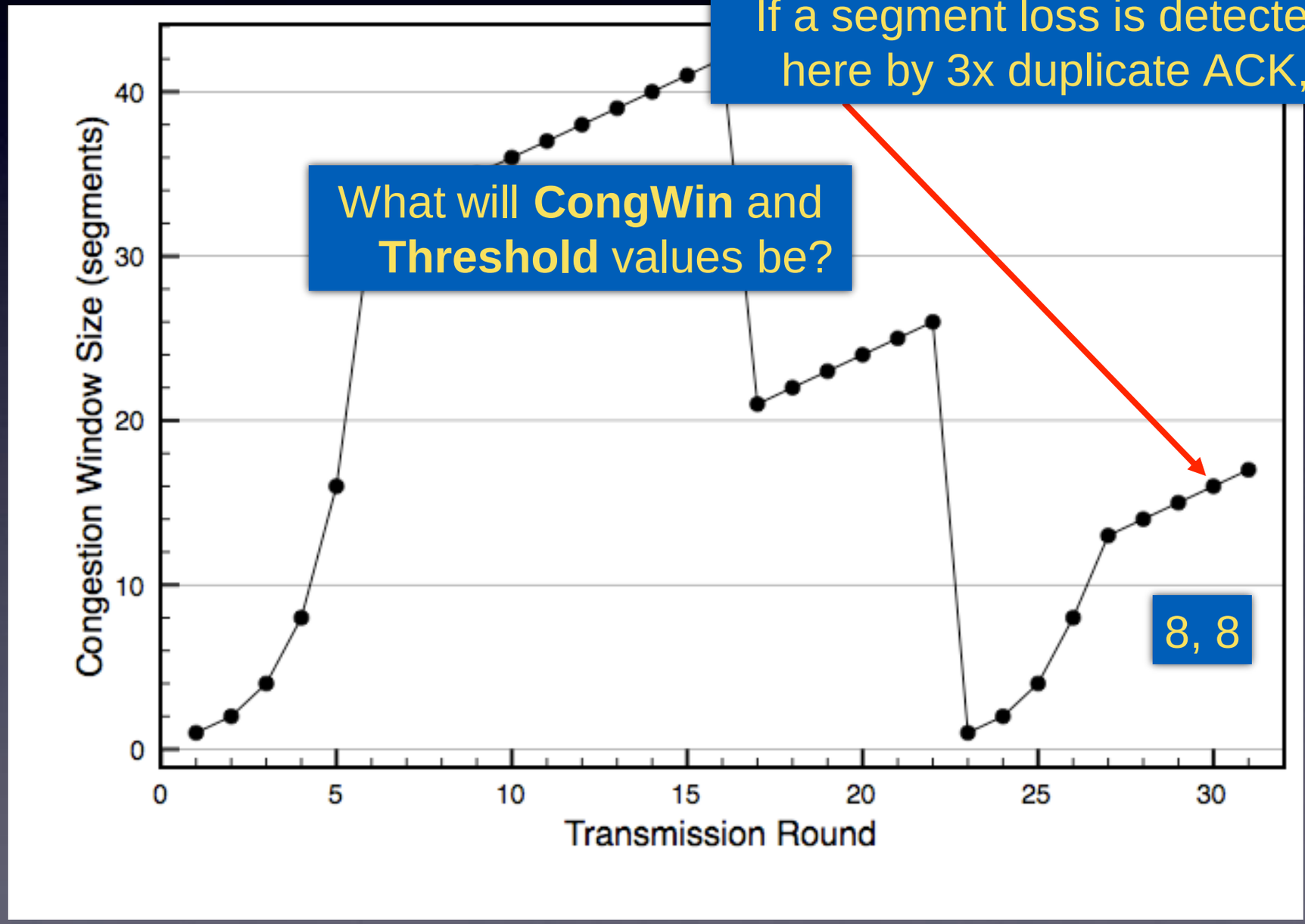
CongWin size plot



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

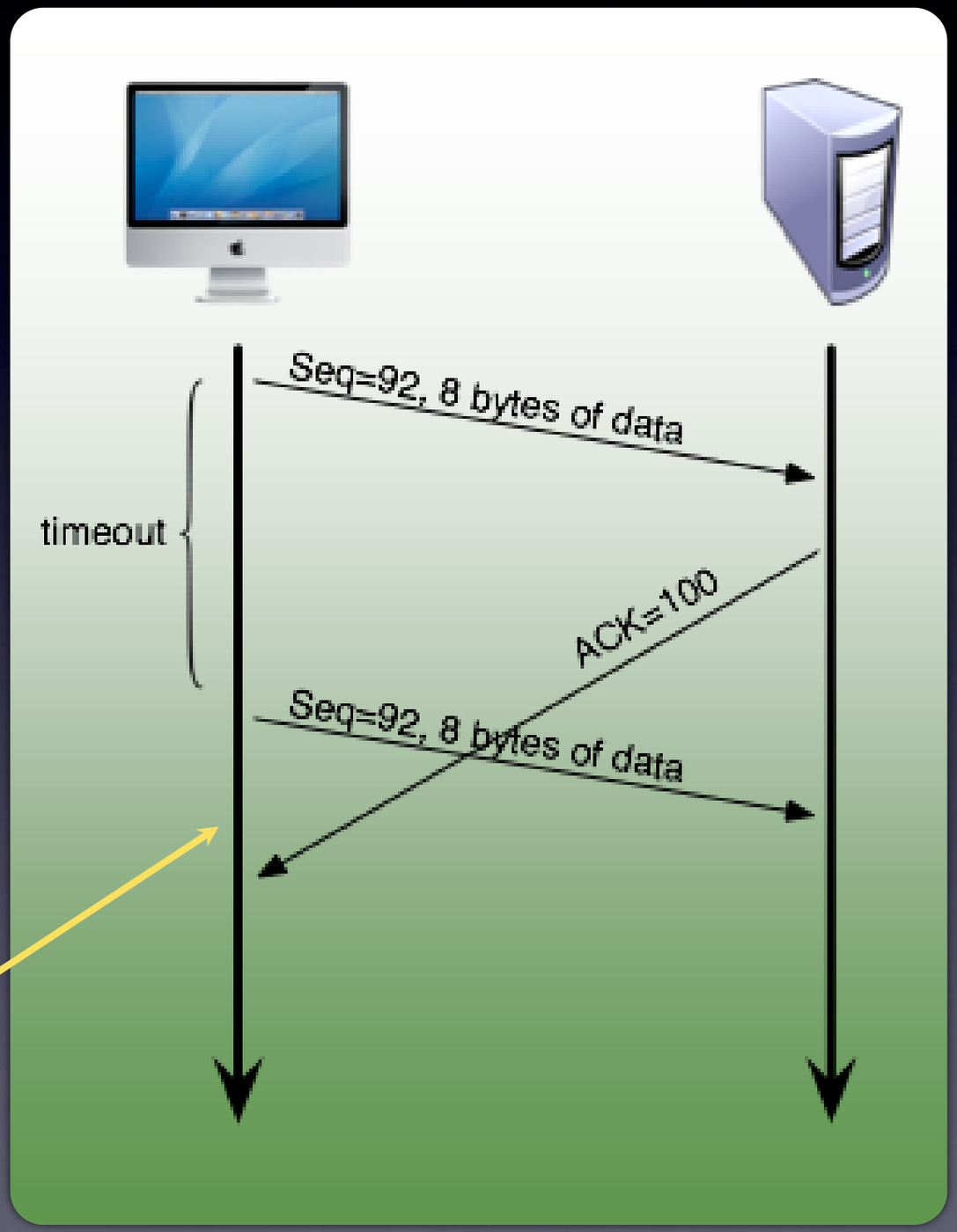
Estimating RTT

- 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?

Whoa! Look how short RTT is!



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