# Chapter 3 Transport Layer

Adapted by RenPing.Liu@uts.edu.au
14 April 2019

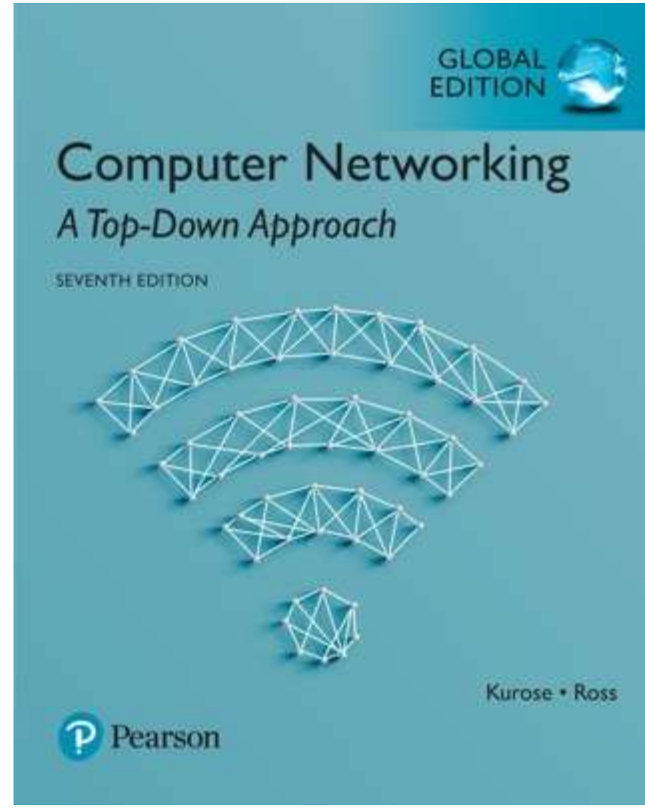
A note on the use of these Powerpoint slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR






*Computer Networking: A Top Down Approach*

$7^{th}$ edition
Jim Kurose, Keith Ross
Pearson/Addison Wesley
April 2016

# Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

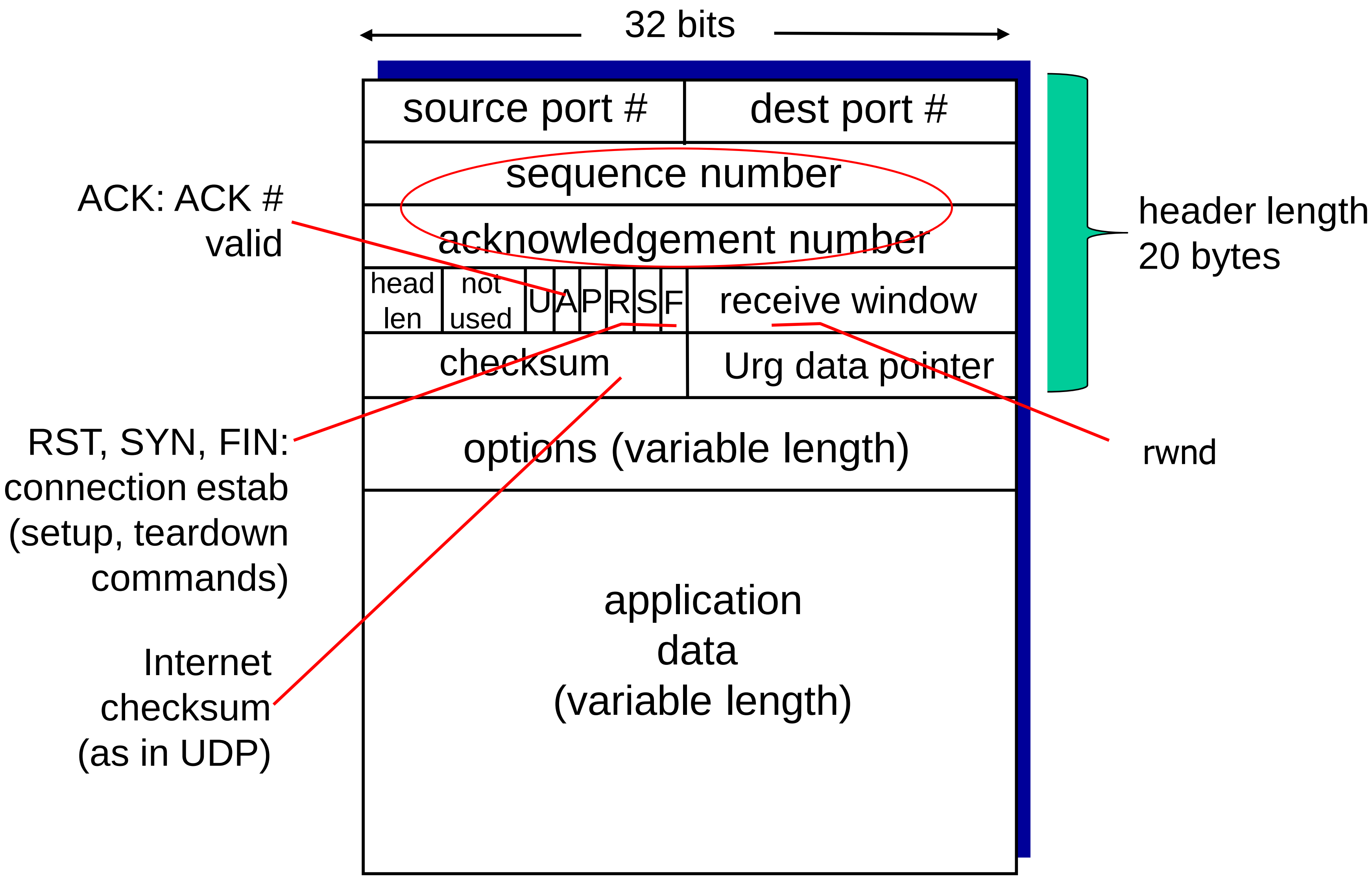
3.6 principles of congestion control

3.7 TCP congestion control

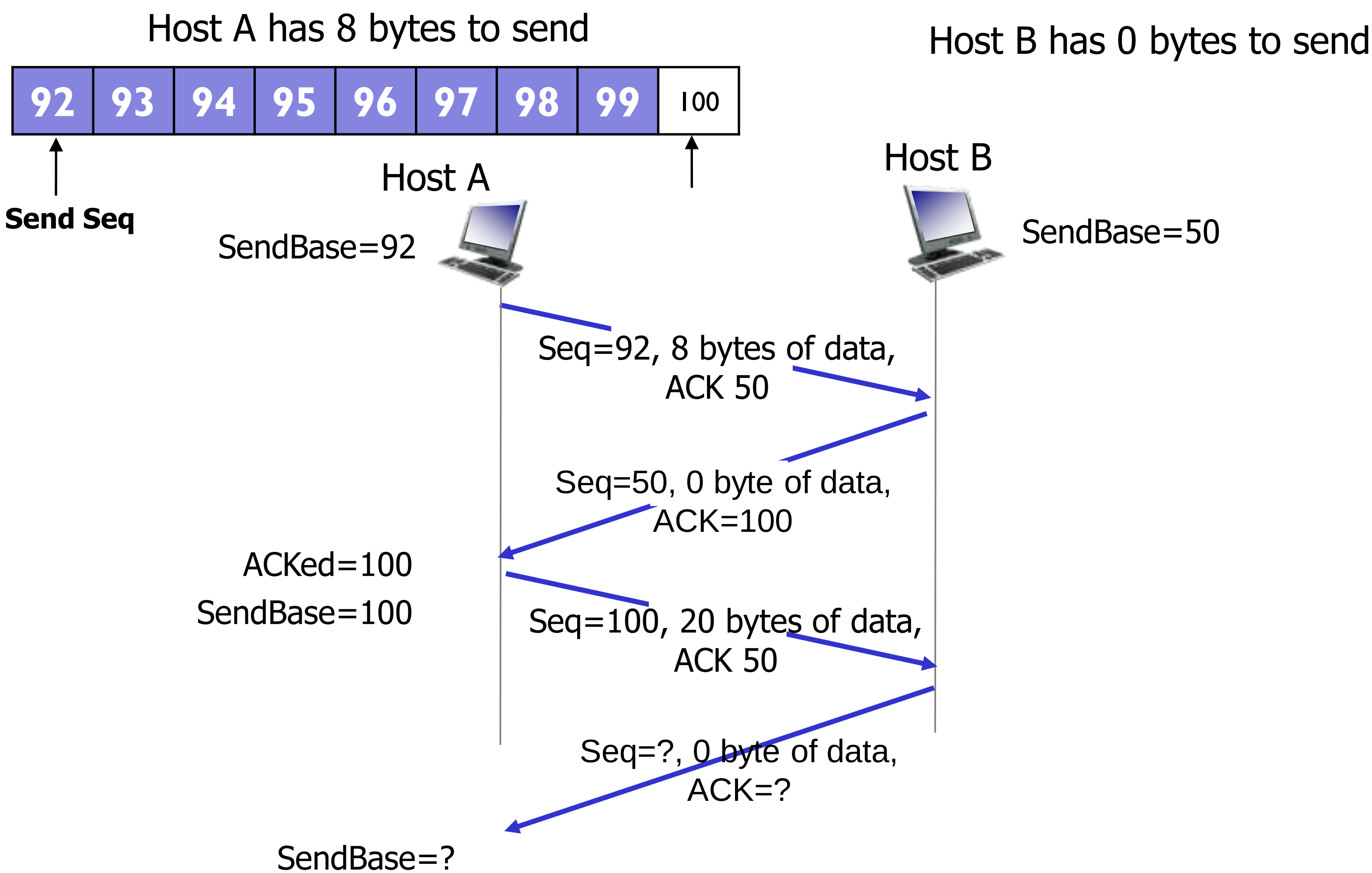
# TCP: Overview RFCs: 793,1122,1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

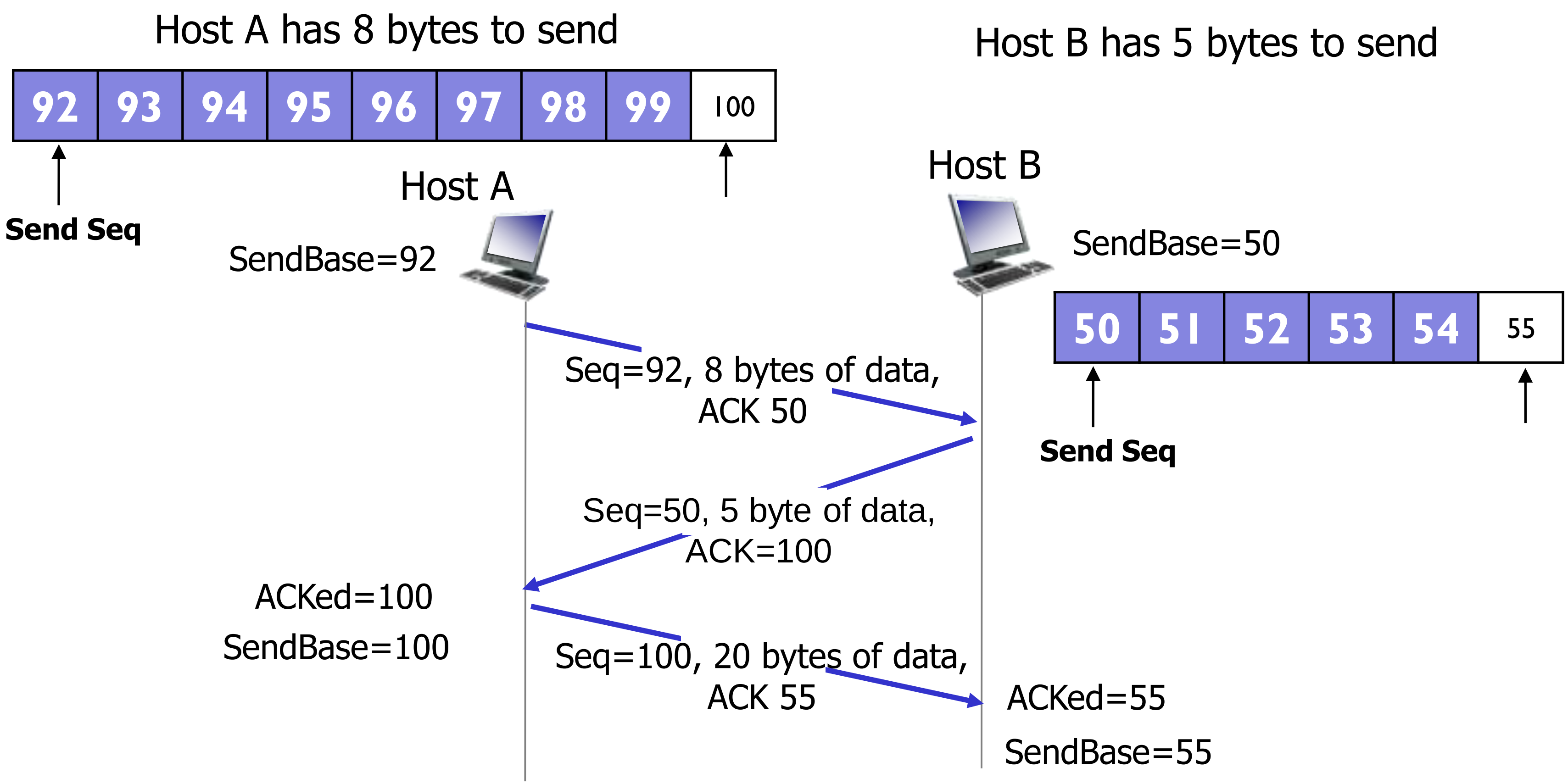
# TCP segment structure



# TCP: seq. ACKs - one way: A→B

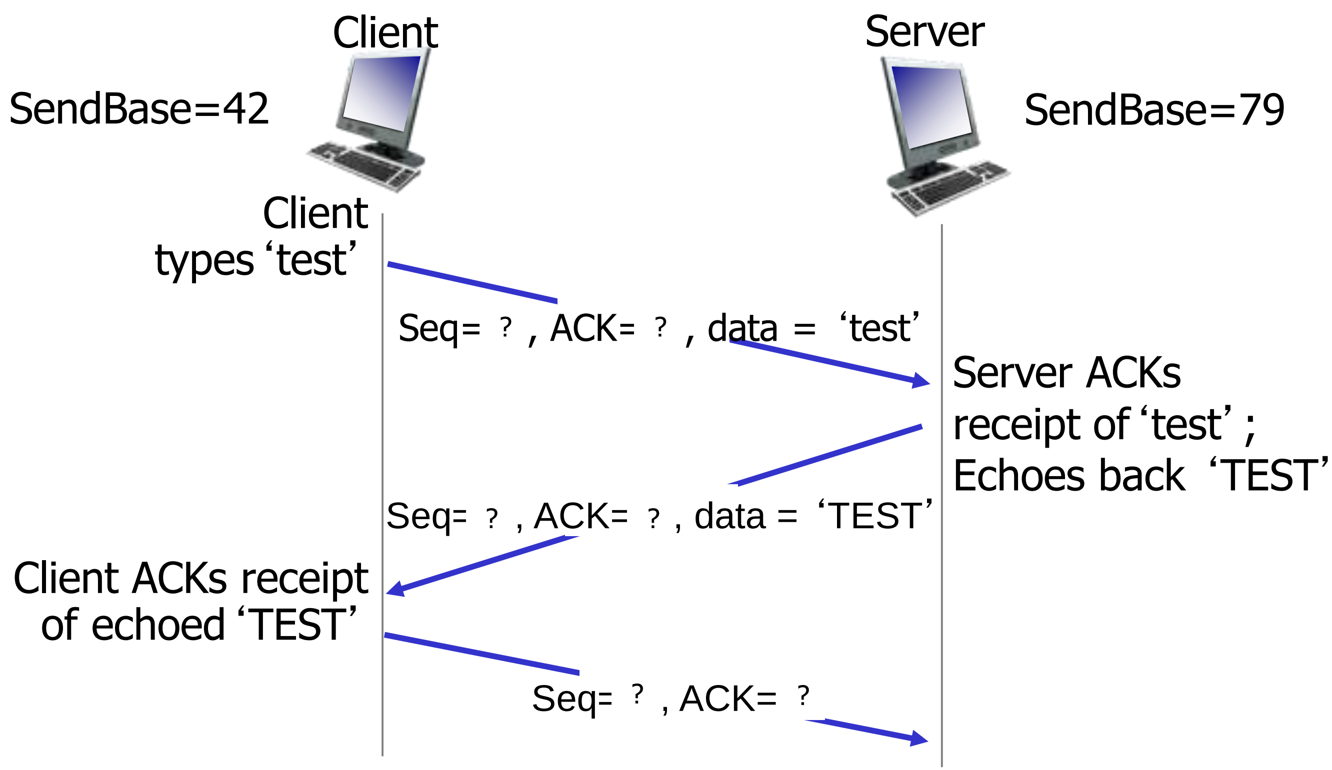


# TCP: seq. Nr, ACK - bidirectional



# TCP seq. numbers, ACKs

Client send 'test' (5Bytes) to Server, Server echo 'TEST' back




simple TCP client / server Python Socket Programming

# Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

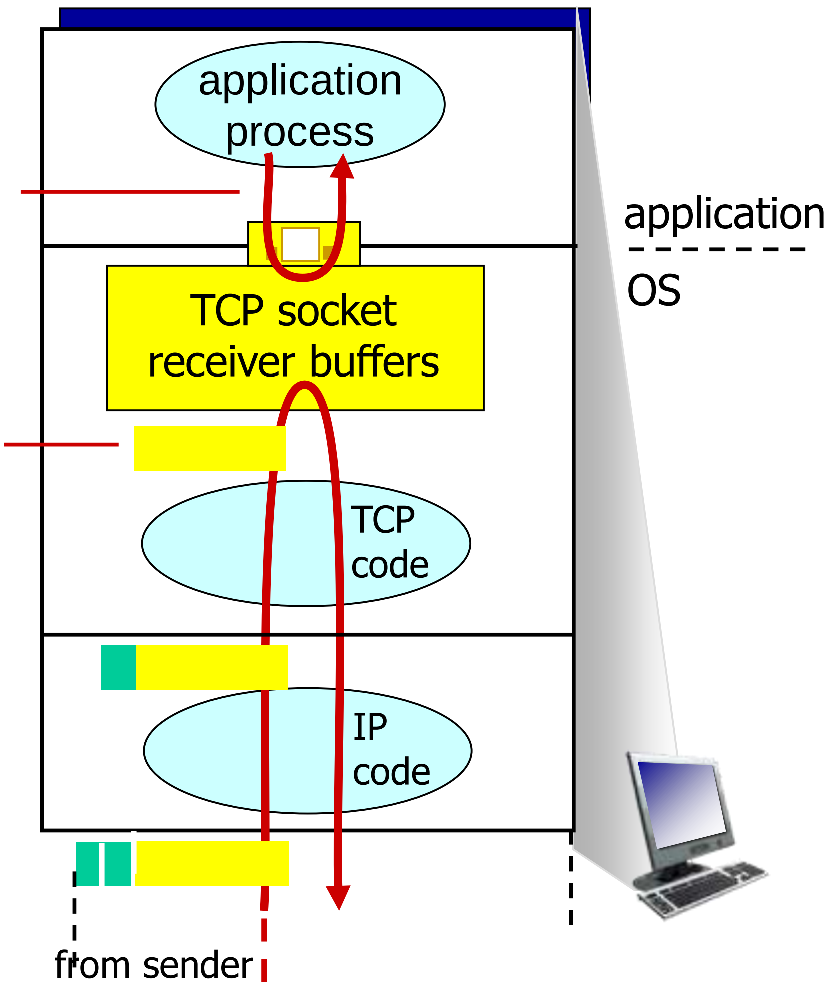
3.7 TCP congestion control

# TCP flow control

application may
retrieve data from
TCP socket buffers ….

*… slower than*

TCP receiver is delivering
(sender is sending)



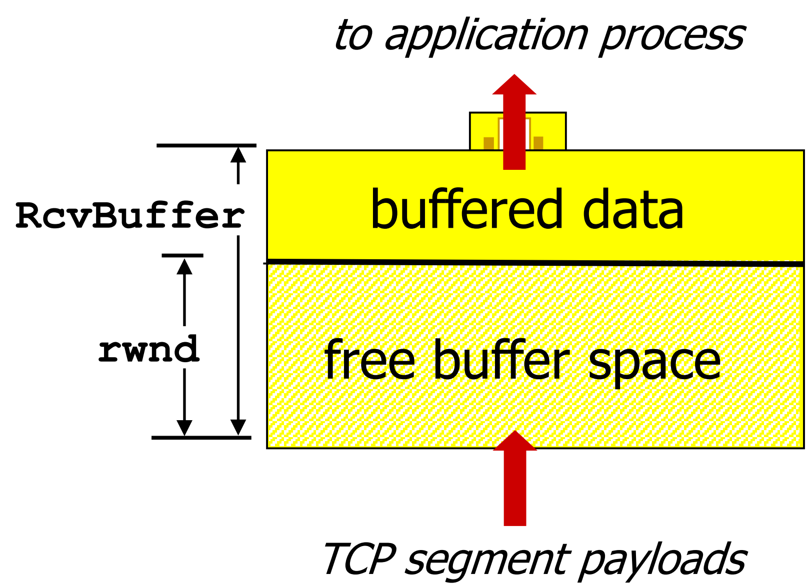

receiver protocol stack

## *flow control*

receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

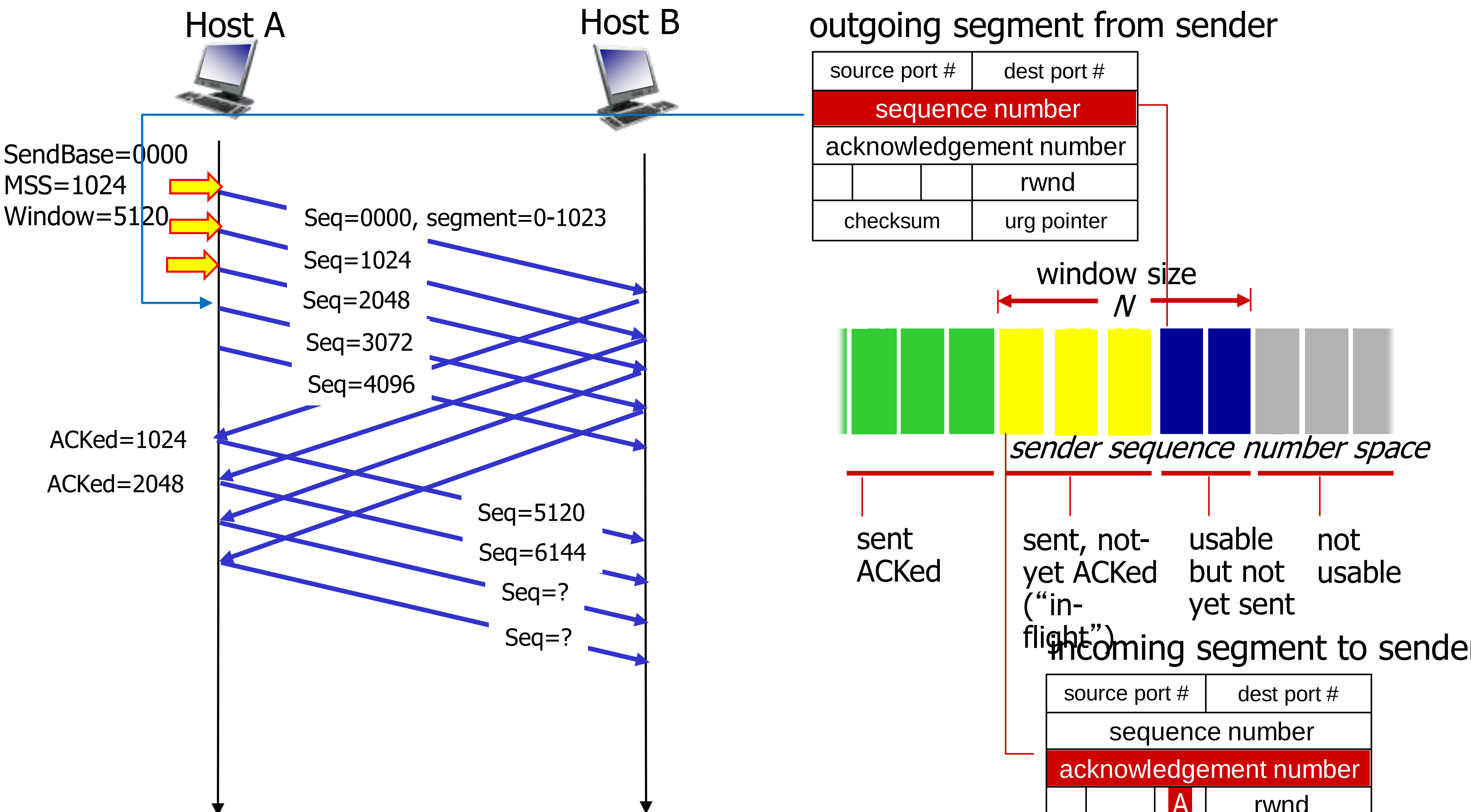
# TCP flow control

- receiver "advertises" free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **RcvBuffer**
- sender limits amount of unacked ("in-flight") data to receiver's **rwnd** value
- guarantees receive buffer will not overflow



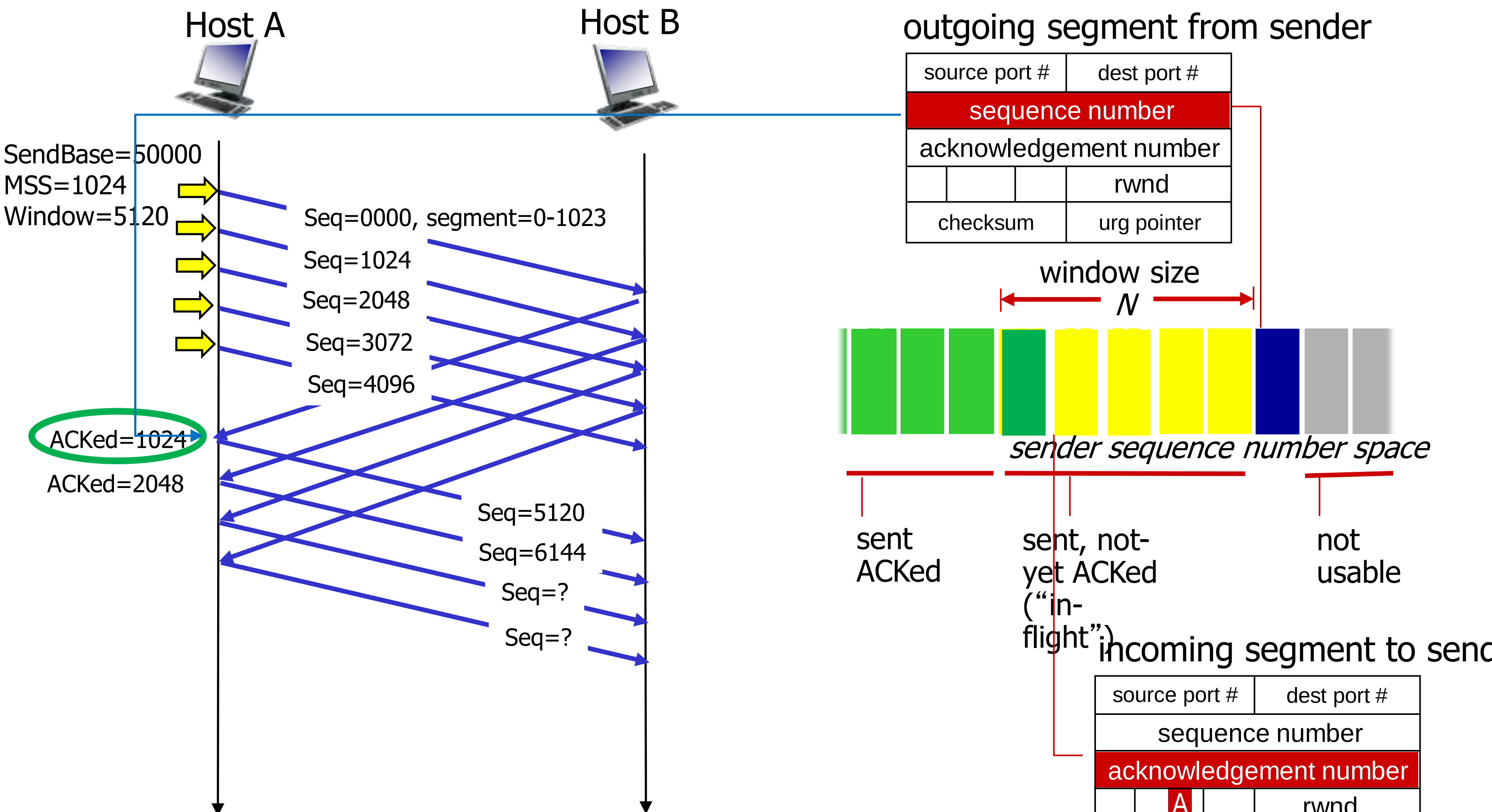

*receiver-side buffering*

# TCP window flow control




fast retransmit after sender receipt of triple duplicate ACK

# TCP window flow control




fast retransmit after sender receipt of triple duplicate ACK

# TCP seq. numbers, ACKs

**sequence numbers:**

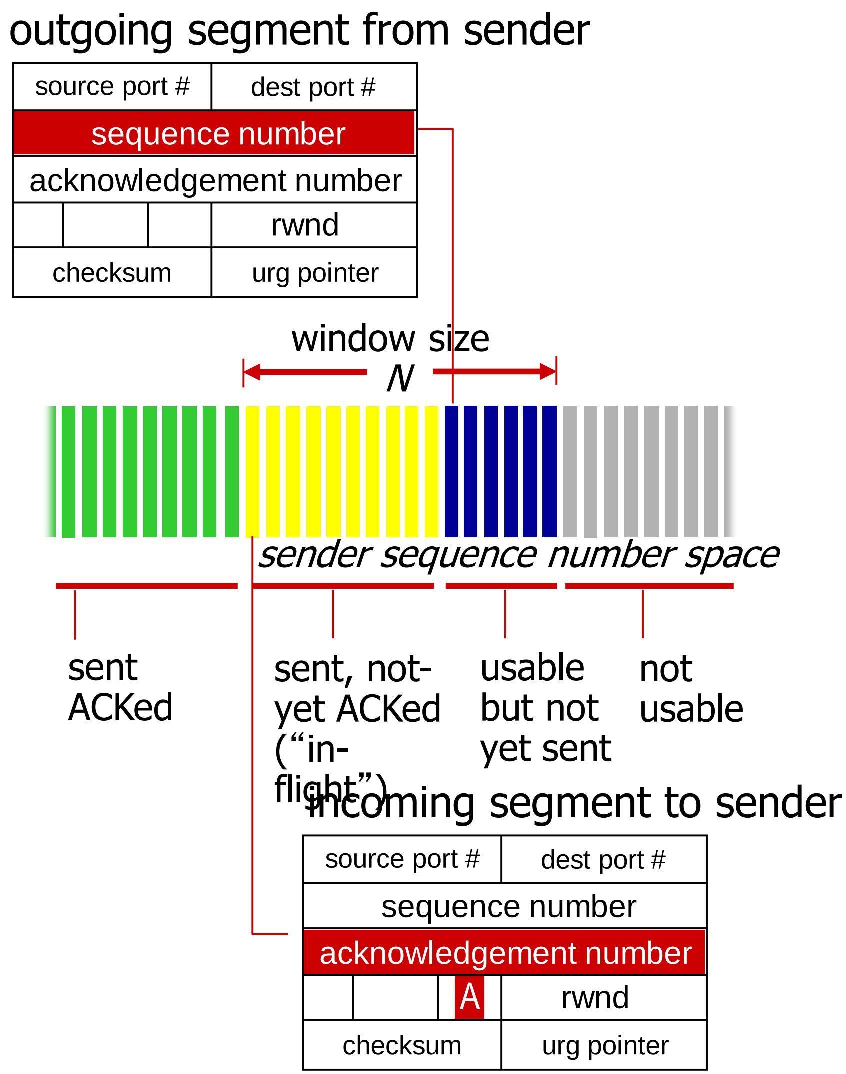
- byte stream "number" of first byte in segment's data

**acknowledgements:**

- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor



# Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

# TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
  - pipelined segments
  - cumulative acks
  - single retransmission timer
- retransmissions triggered by:
  - timeout events
  - duplicate acks

# TCP round trip time, timeout

Q: how to set TCP timeout value?

- longer than RTT
  - but RTT varies
- *too short:* premature timeout, unnecessary retransmissions
- *too long:* slow reaction to segment loss

Q: how to estimate RTT?

- **`SampleRTT`**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - average several *recent* measurements, not just current **`SampleRTT`**

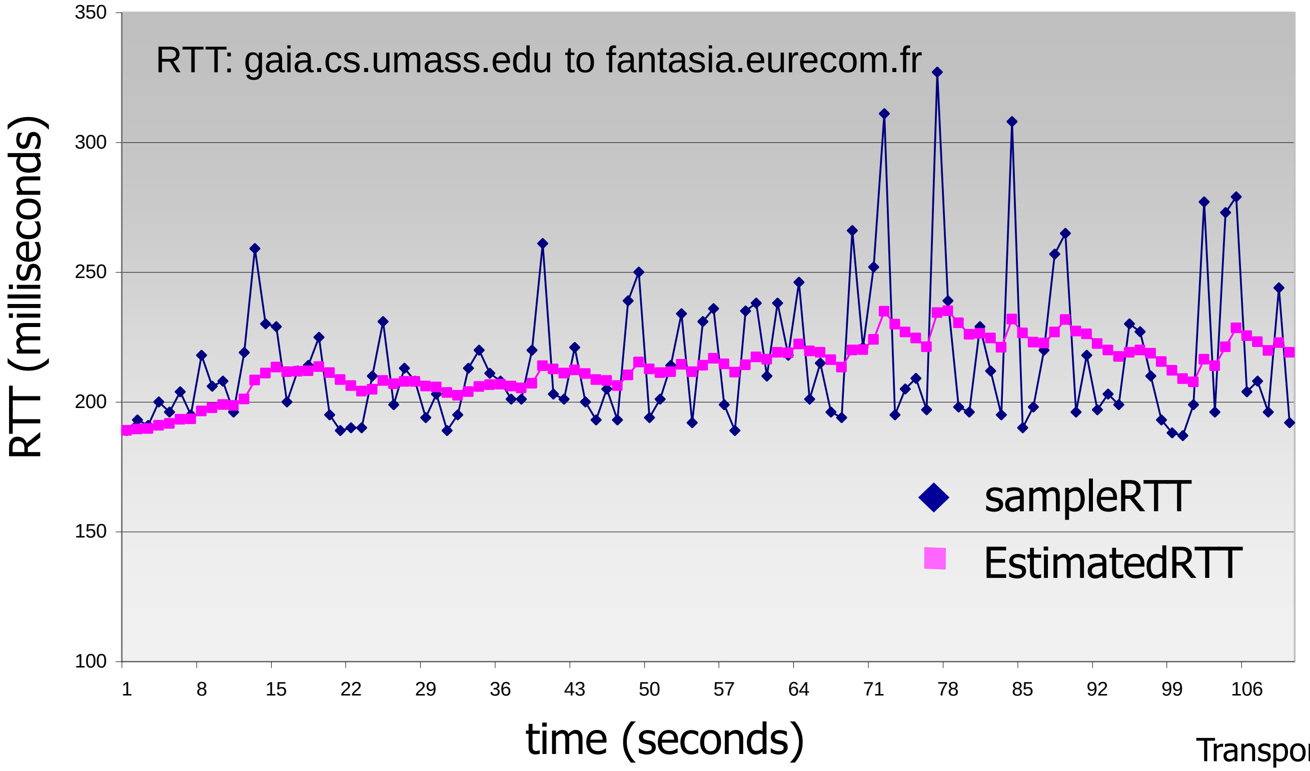
*TCP throughput = rwnd / RTT (in theory!)*

*< rwnd / RTT (in practice … discuss later)*

# TCP round trip time, timeout

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

- exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.125$



# TCP round trip time, timeout

- timeout interval: **EstimatedRTT** plus "safety margin"
  - large variation in **EstimatedRTT** -> larger safety margin
- estimate SampleRTT deviation from EstimatedRTT:

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

              (typically, β = 0.25)
```

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

estimated RTT    "safety margin"

* Check out the online interactive examples for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

# TCP sender events:

*data rcvd from app:*

- create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running
  - think of timer as for oldest unacked segment
  - expiration interval: **`TimeOutInterval`**

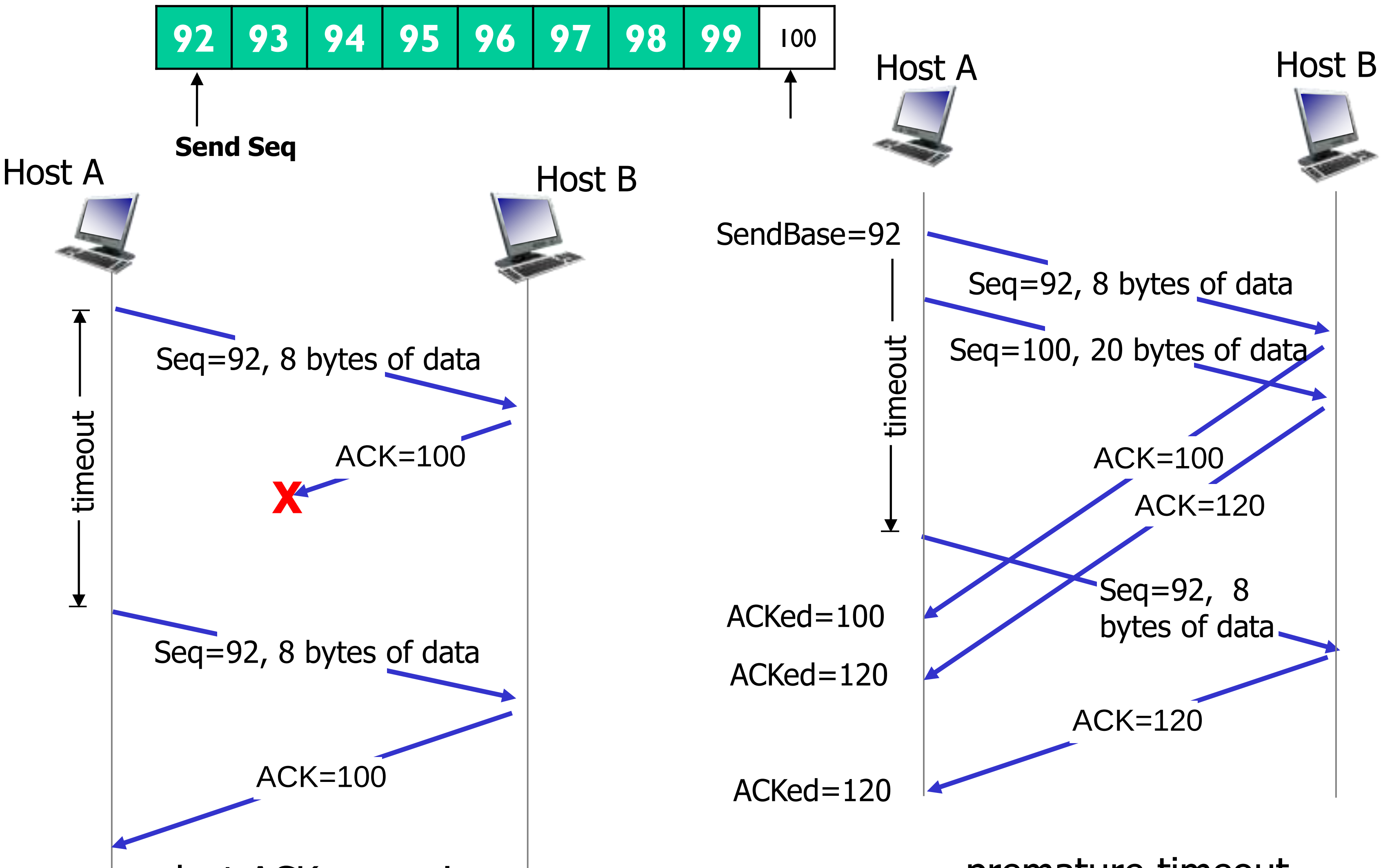
*timeout:*

- retransmit segment that caused timeout
- restart timer

*ack rcvd:*

- if ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments

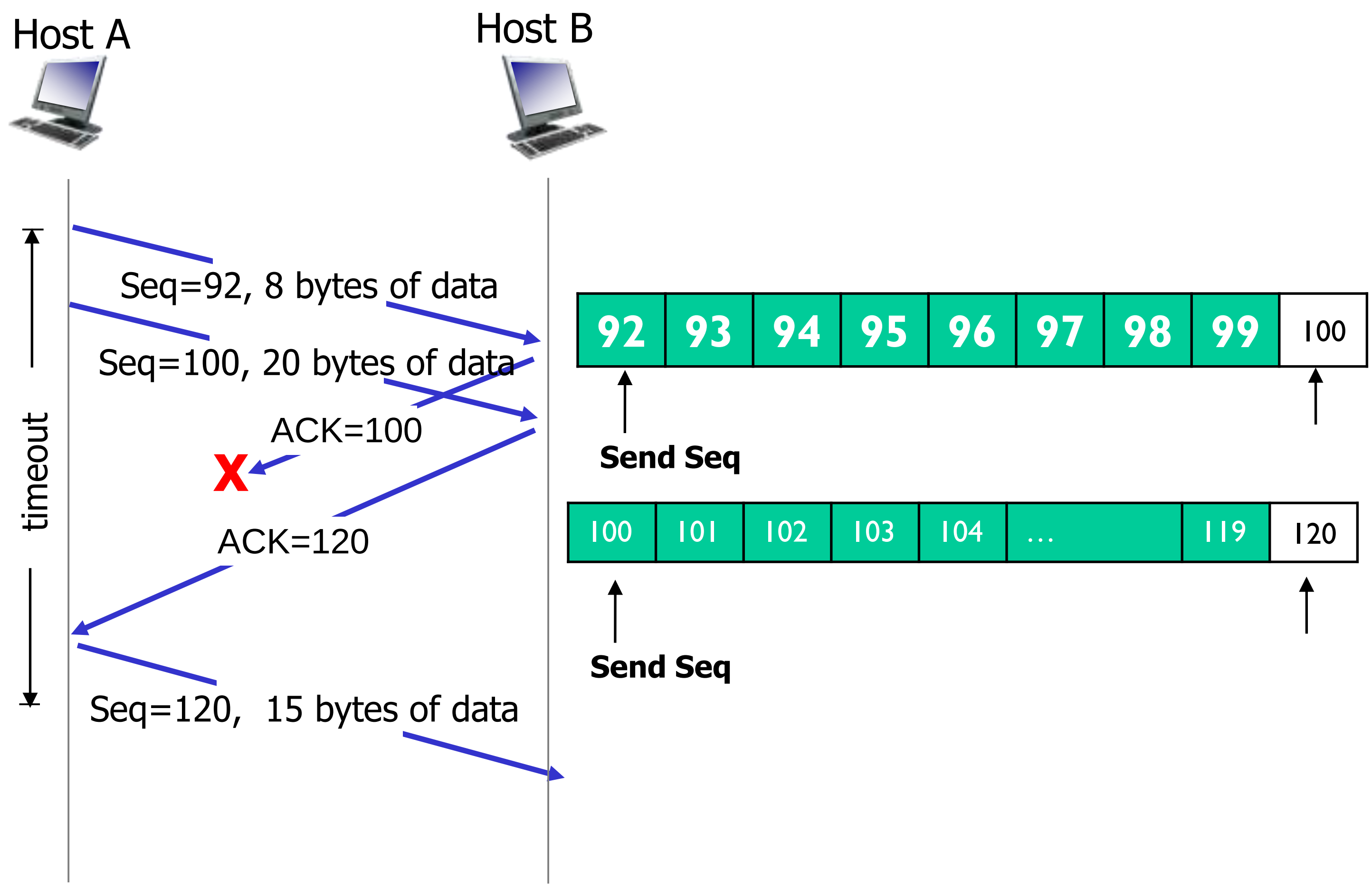
# TCP: retransmission scenarios




lost ACK scenario

premature timeout

# TCP: retransmission scenarios




cumulative ACK

# TCP fast retransmit

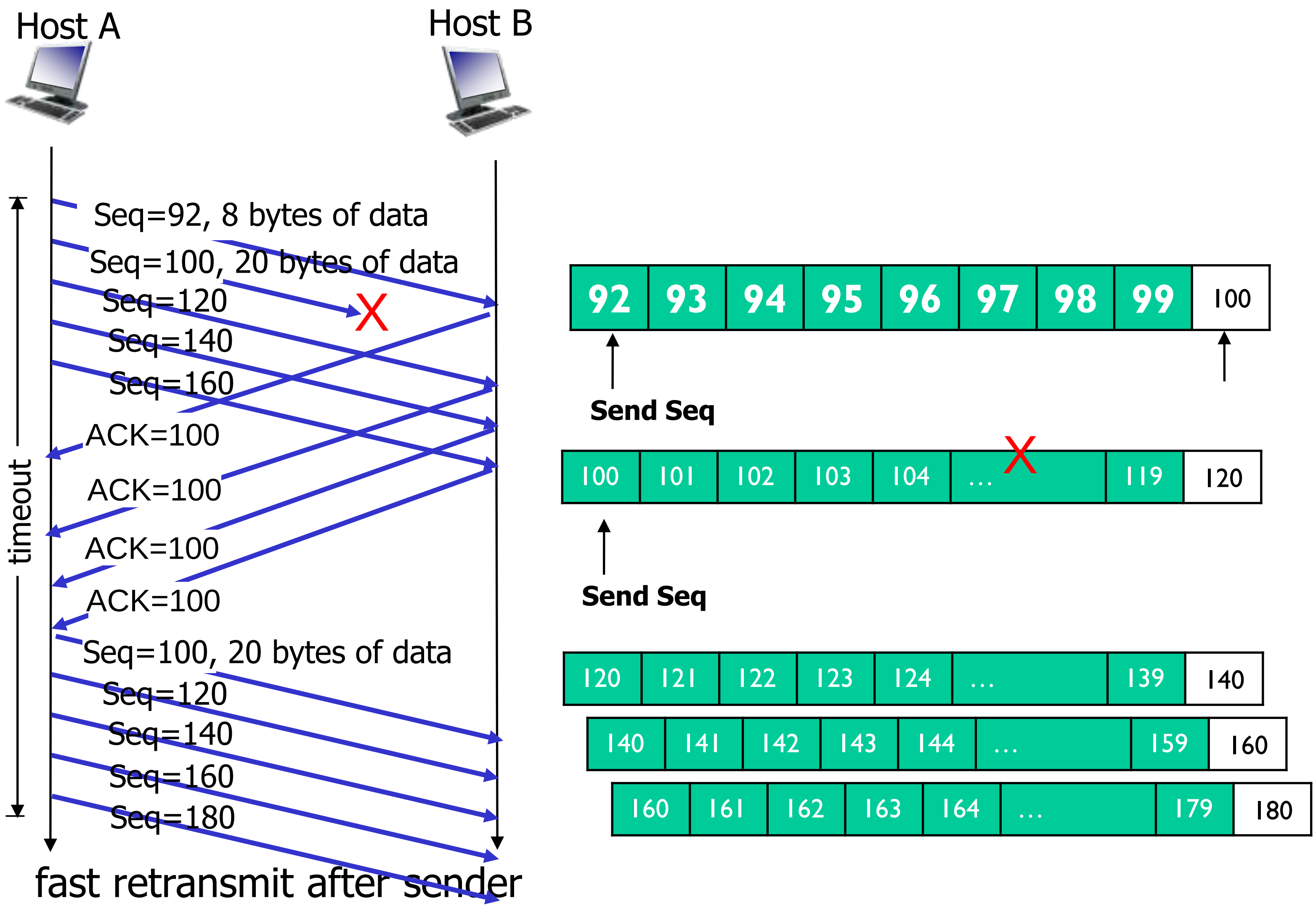
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

*TCP fast retransmit*

if sender receives 3 ACKs for same data ("triple duplicate ACKs"), resend unacked segment with smallest seq #

- likely that unacked segment lost, so don't wait for timeout

# TCP fast retransmit




fast retransmit after sender receipt of triple duplicate ACK

# Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

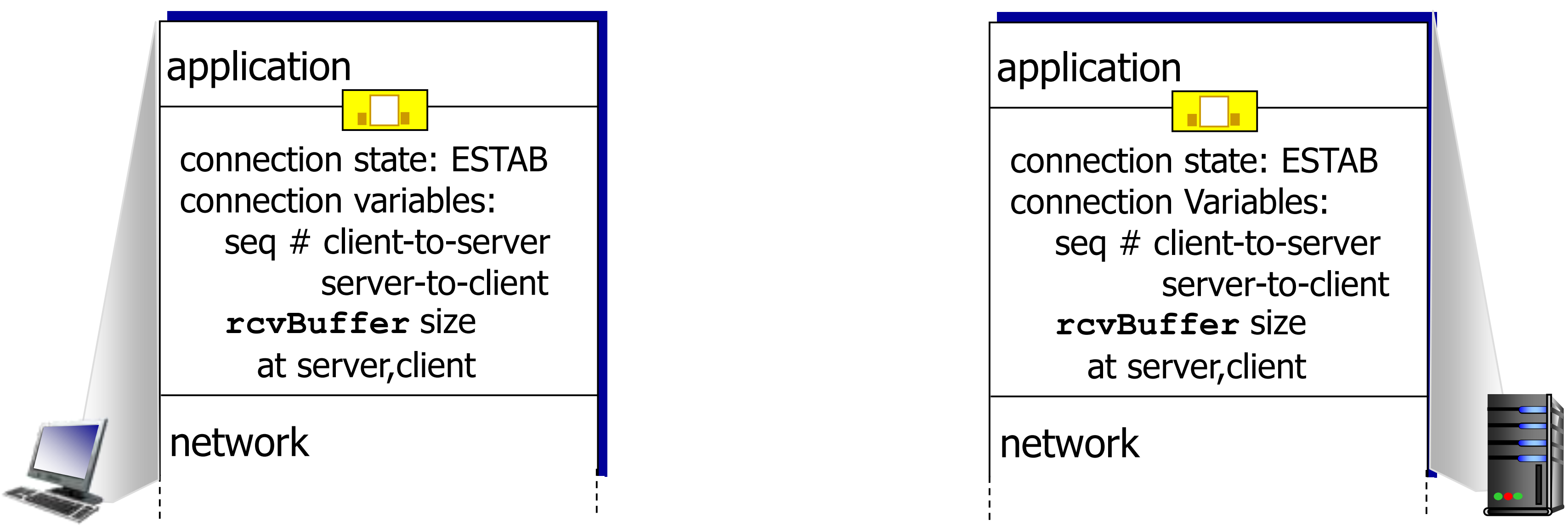
3.6 principles of congestion control

3.7 TCP congestion control

# Connection Management

before exchanging data, sender/receiver "handshake":

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters

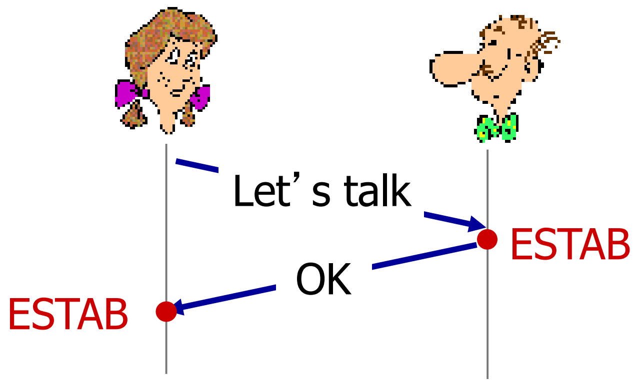


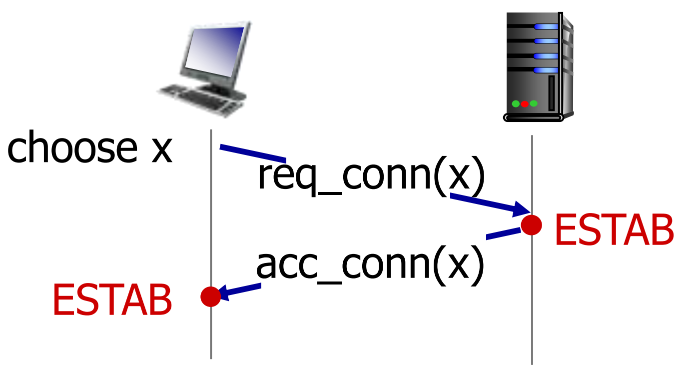

```
clientSocket.connect
("servername","port number");
...
clientSocket.send(…)
```

```
connectionSocket, addr =
   serverSocket.accept();
...
connectionSocket.recv(…)
```

# Agreeing to establish a connection

2-way handshake:

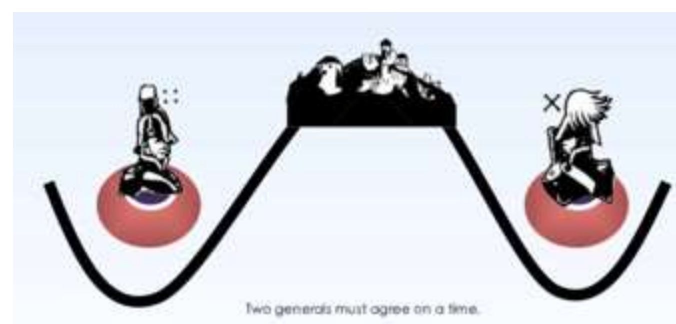


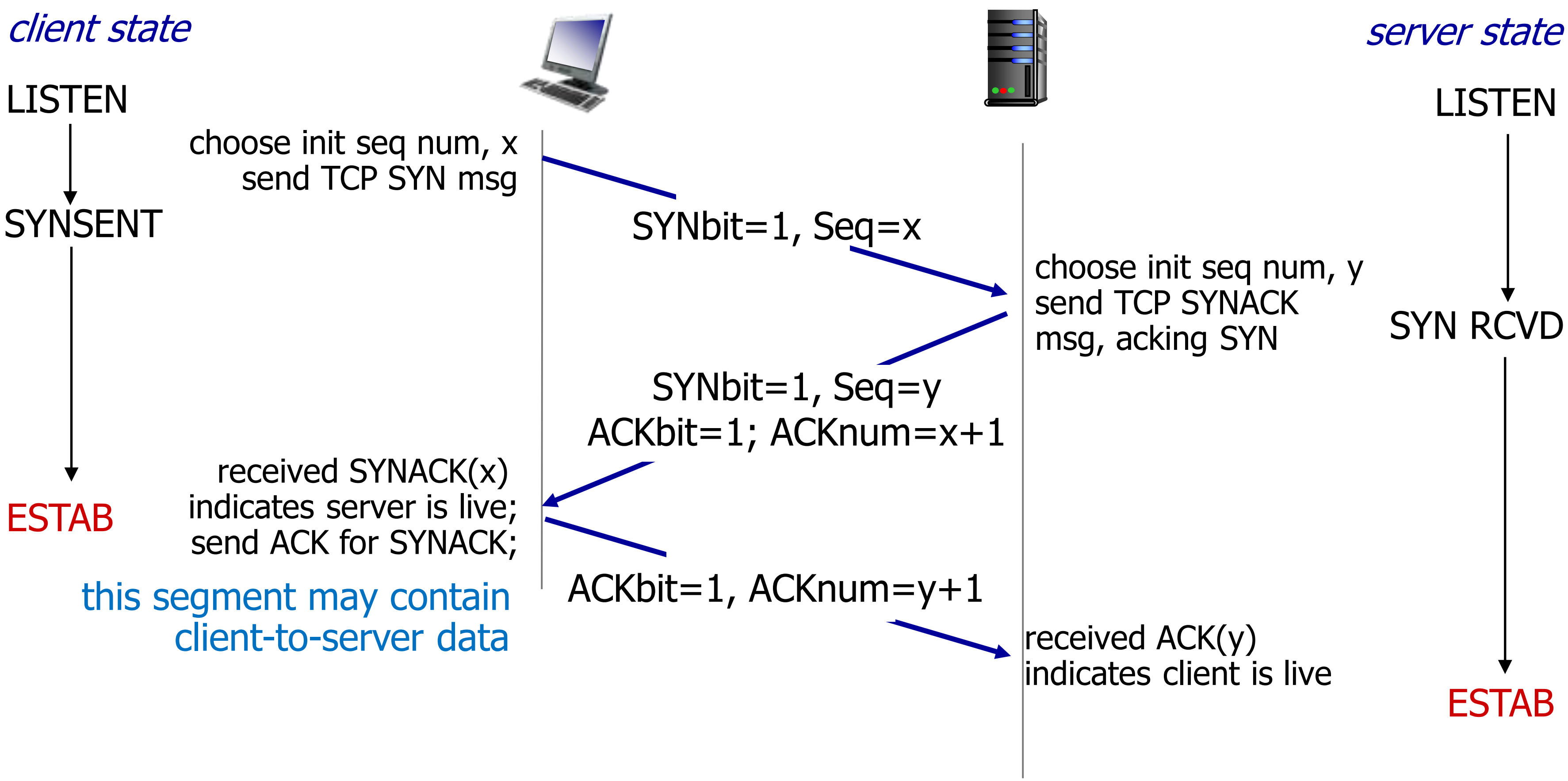

*Q:* will 2-way handshake always work in network?

- unreliable channel
- retransmitted messages (e.g. req_conn(x)) due to loss

Byzantine Generals Problem



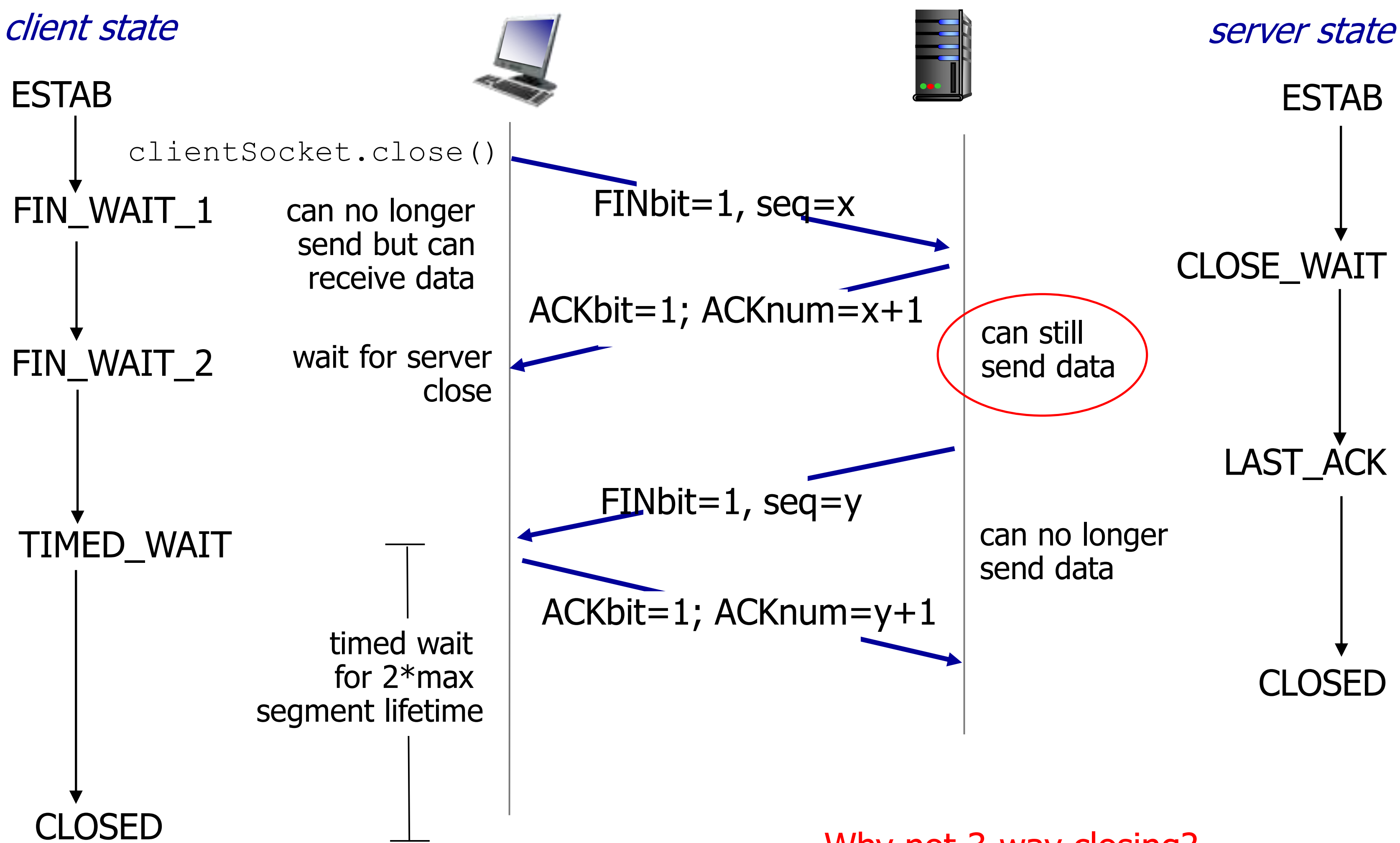
# TCP 3-way handshake



# TCP: closing a connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

# TCP: closing a connection




Why not 3-way closing?

# Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

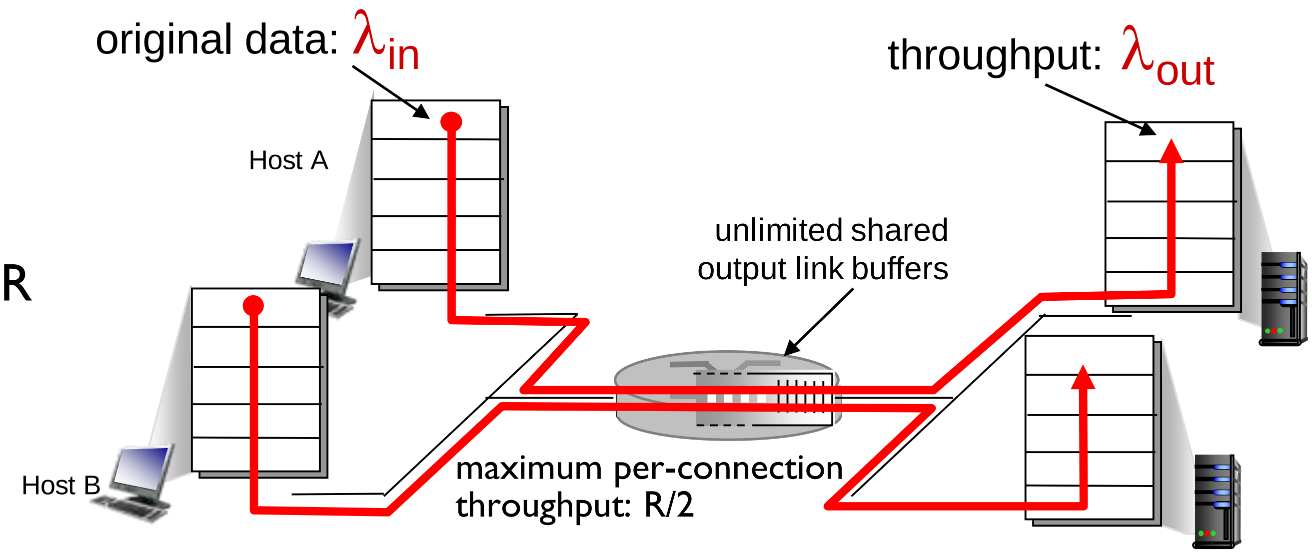
# Principles of congestion control

*congestion:*

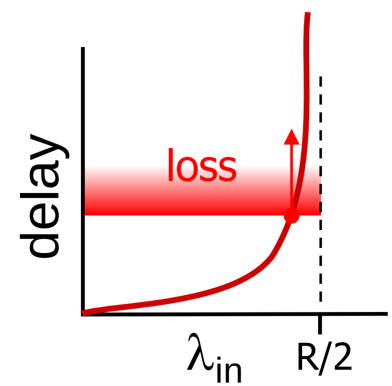
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - long delays (queueing in router buffers)
  - lost packets (buffer overflow at routers)
- a top-10 problem!

# Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router
- output link capacity: R
- As traffic increase:
  - ✗ Delay increase
  - ✗ Packet loss




- How can we?
  - ✓ Avoid congestion
  - ✓ Resolve congestion

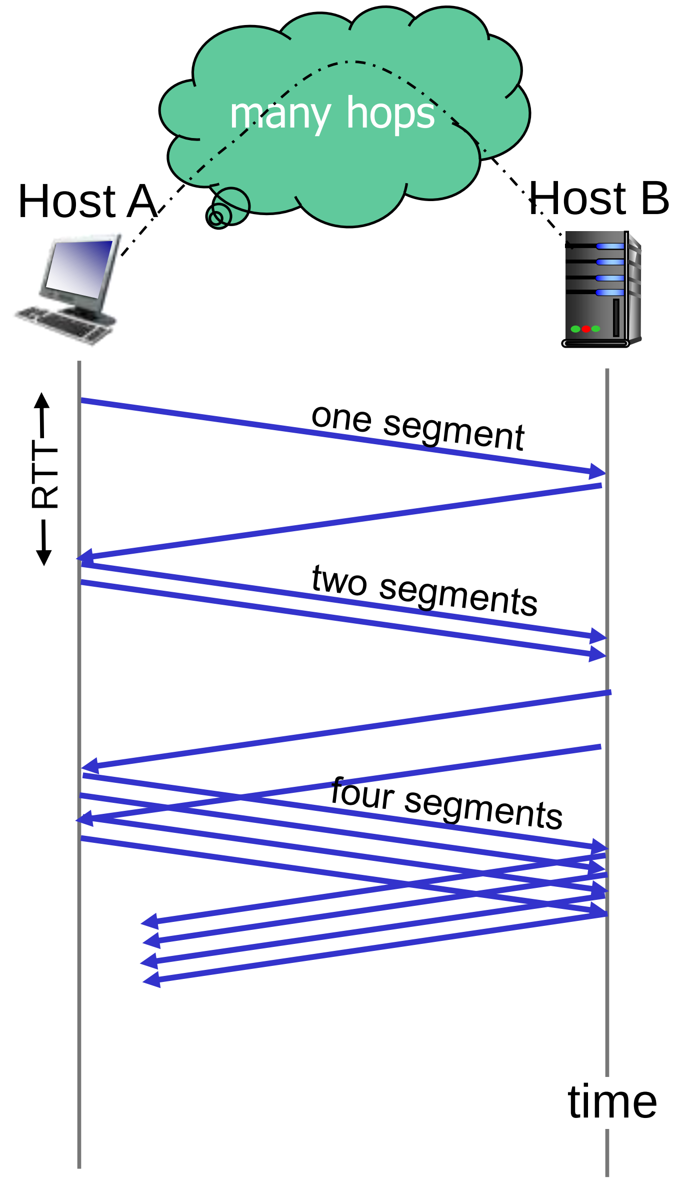



- large delays as arrival rate, $\lambda_{in}$, approaches capacity

# Chapter 3 outline

# TCP Slow Start

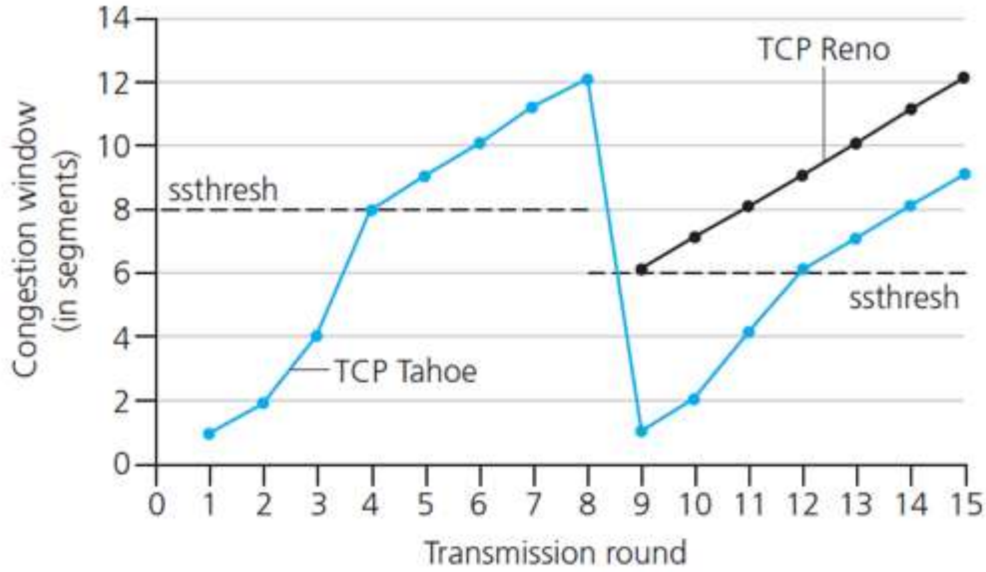
- when connection begins, increase rate exponentially until first loss event:
  - initially **`cwnd`** = 1 MSS
  - double **`cwnd`** every RTT
  - done by incrementing **`cwnd`** by 1 MSS (linear) for every ACK received
- *summary:* initial rate is slow but ramps up exponentially fast



# TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

A: when
cwnd ≥ ssthresh




## Implementation:

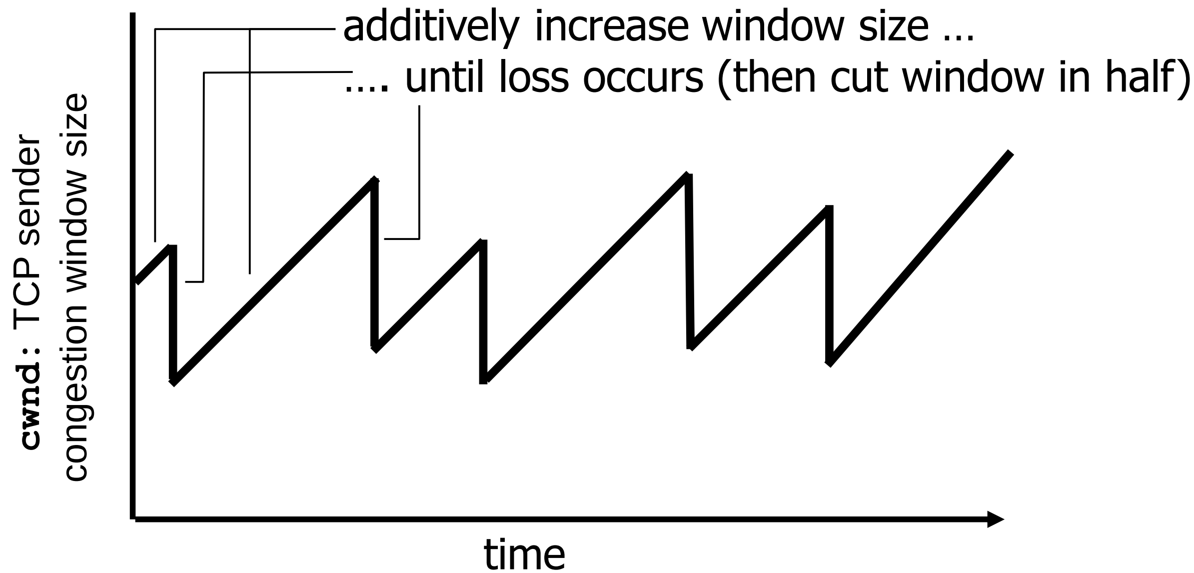
- variable **`ssthresh`**
- on loss event, **`ssthresh`** is set to 1/2 of **`cwnd`**
- CA (Congestion Avoidance):
  - when cwnd ≥ ssthresh, cwnd++ (add 1 MSS every RTT)

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

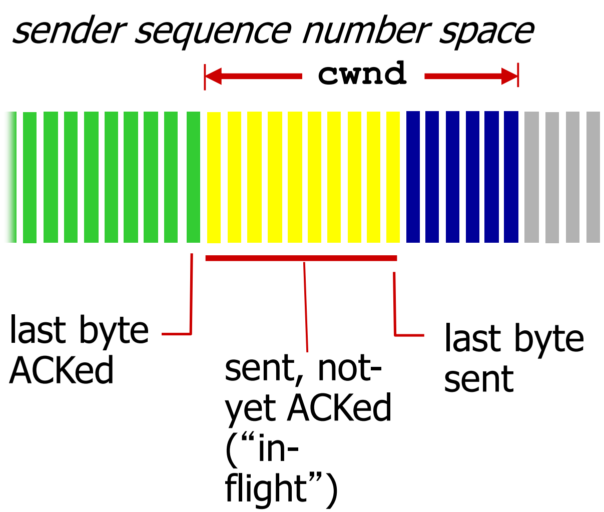
# TCP congestion control: additive increase multiplicative decrease

- *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **`cwnd`** by 1 MSS every RTT until loss detected
  - *multiplicative decrease:* cut **`cwnd`** in half after loss

AIMD saw tooth behavior: probing for bandwidth



# TCP Congestion Control: details




- sender limits transmission:

$$\texttt{LastByteSent-LastByteAcked} \leq \texttt{cwnd}$$

- **cwnd** is dynamic, function of perceived network congestion

*TCP sending rate:*

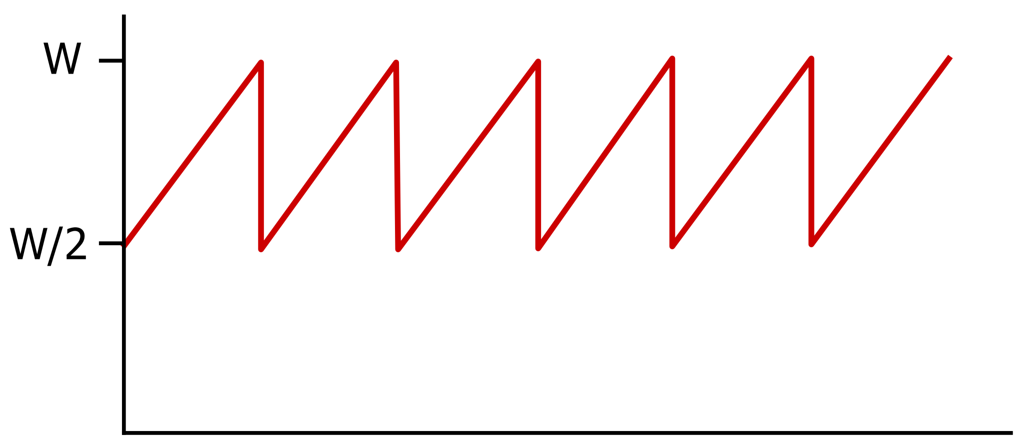
- *roughly:* send cwnd bytes, wait RTT for ACKS, then send more bytes

$$\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

# TCP throughput

- avg. TCP thruput as function of window size, RTT?
  - ignore slow start, assume always data to send
- W: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. thruput is 3/4W per RTT

$$\text{avg TCP thruput} = \frac{3}{4}\frac{W}{RTT} \text{ bytes/sec}$$



# Chapter 3: summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation, implementation in the Internet
  - UDP
  - TCP

next:

- leaving the network "edge" (application, transport layers)
- into the network "core"
- two network layer chapters:
  - data plane
  - control plane