ECS524
Transport layer:
TCP and congestion control

Prof. Steve Uhlig
steve@eecs.qmul.ac.uk
Office: Eng202

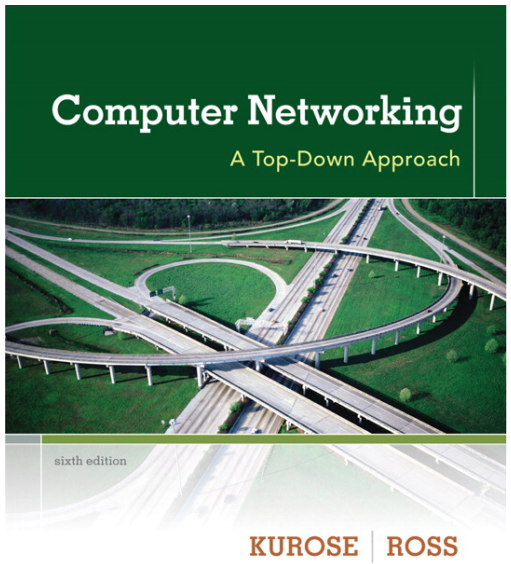
Dr. Felix Cuadrado
felix@eecs.qmul.ac.uk
Office: E205

Slides

Queen Mary University of London

Disclaimer:
Some of the slides' content is borrowed directly from those provided by the authors of the textbook. They are available from

http://www-net.cs.umass.edu/kurose-ross-ppt-6e



Computer Networking: A Top Down Approach
6th edition
Jim Kurose, Keith Ross
Addison-Wesley
March 2012

The Transport Layer

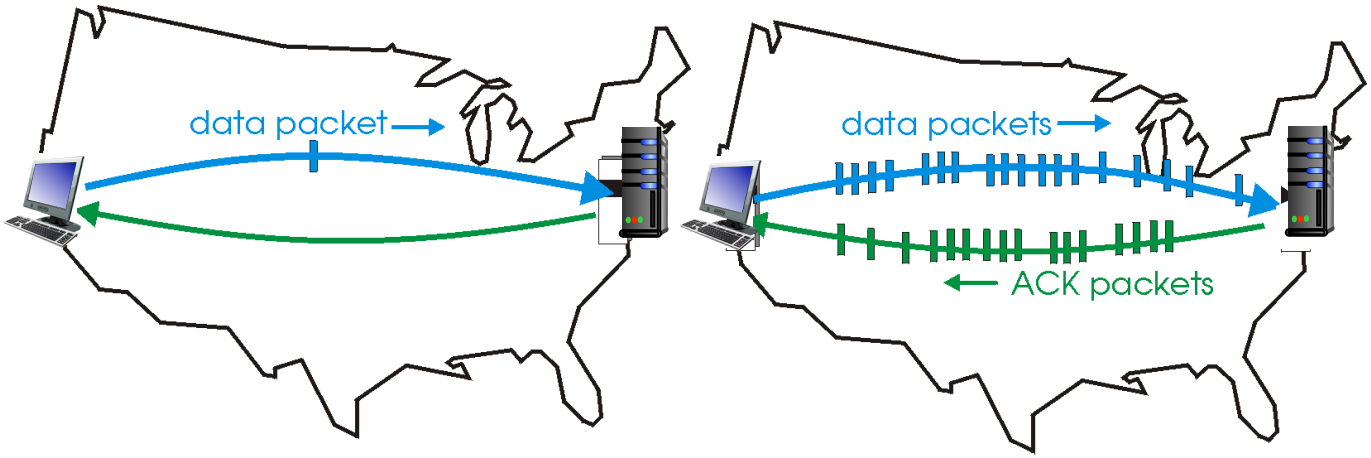
- **Pipelined protocols**
- TCP
- Congestion control

Pipelined protocols

- Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

  range of sequence numbers must be increased

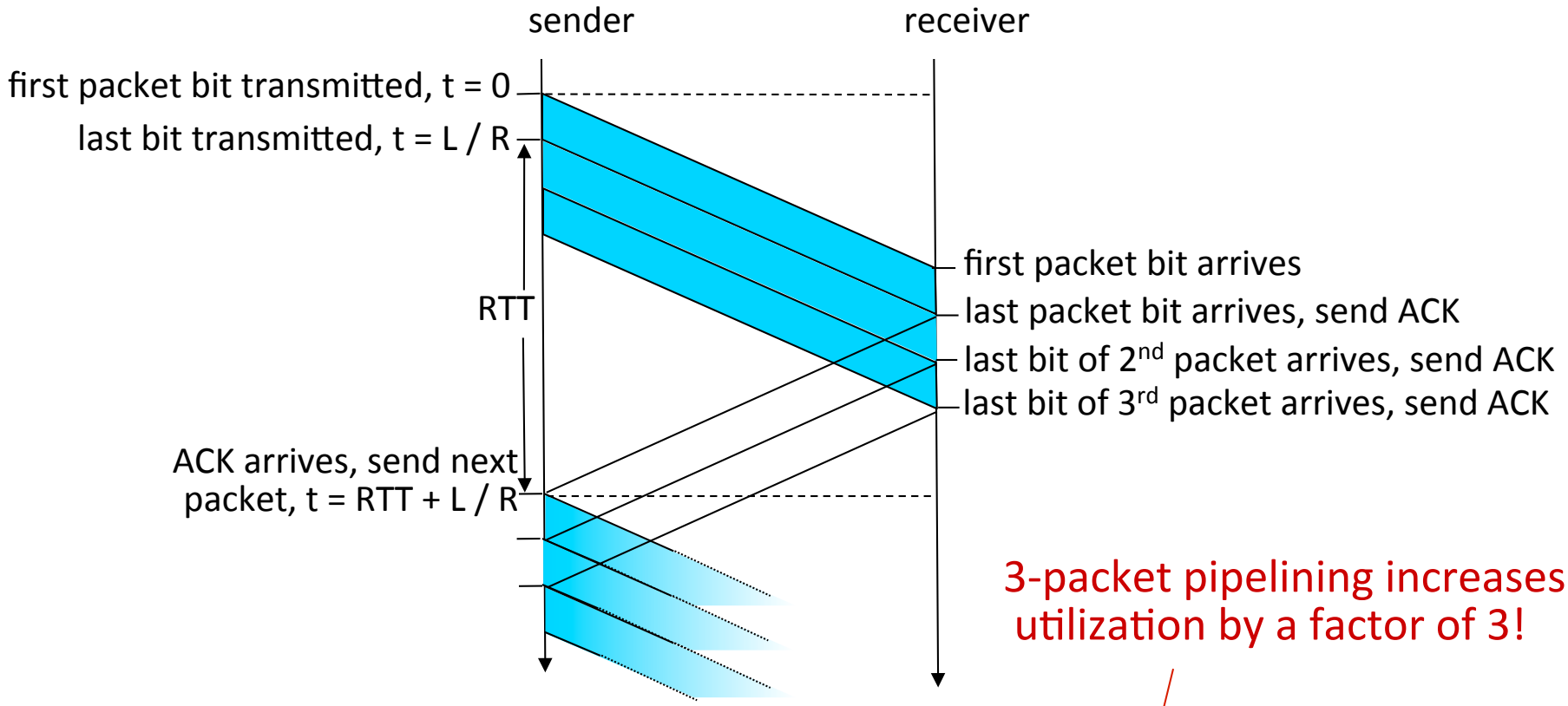
  buffering at sender and/or receiver



(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

- Two generic forms of pipelined protocols: *go-Back-N, selective repeat*

Pipelining: increased utilization



3-packet pipelining increases utilization by a factor of 3!

$$U_{sender} = \frac{3L/R}{RTT + L/R} = \frac{.0024}{30.008} = 0.00081$$

Pipelined protocols: overview

Queen Mary University of London

Go-back-N:

- Sender can have up to N unacked packets in pipeline
- Receiver only sends *cumulative ack*

  Doesn't ack packet if there is a gap
- Sender has timer for oldest unacked packet

  when timer expires, retransmit *all* unacked packets

Selective Repeat:

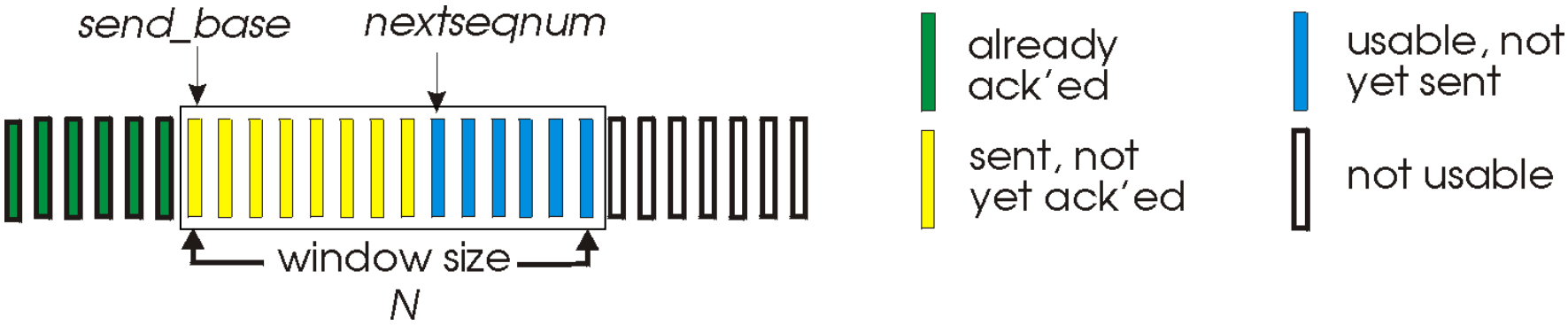
- Sender can have up to N unacked packets in pipeline
- Receiver sends *individual ack* for each packet
- Sender maintains timer for each unacked packet

  when timer expires, retransmit only that unacked packet

Go-Back-N: sender

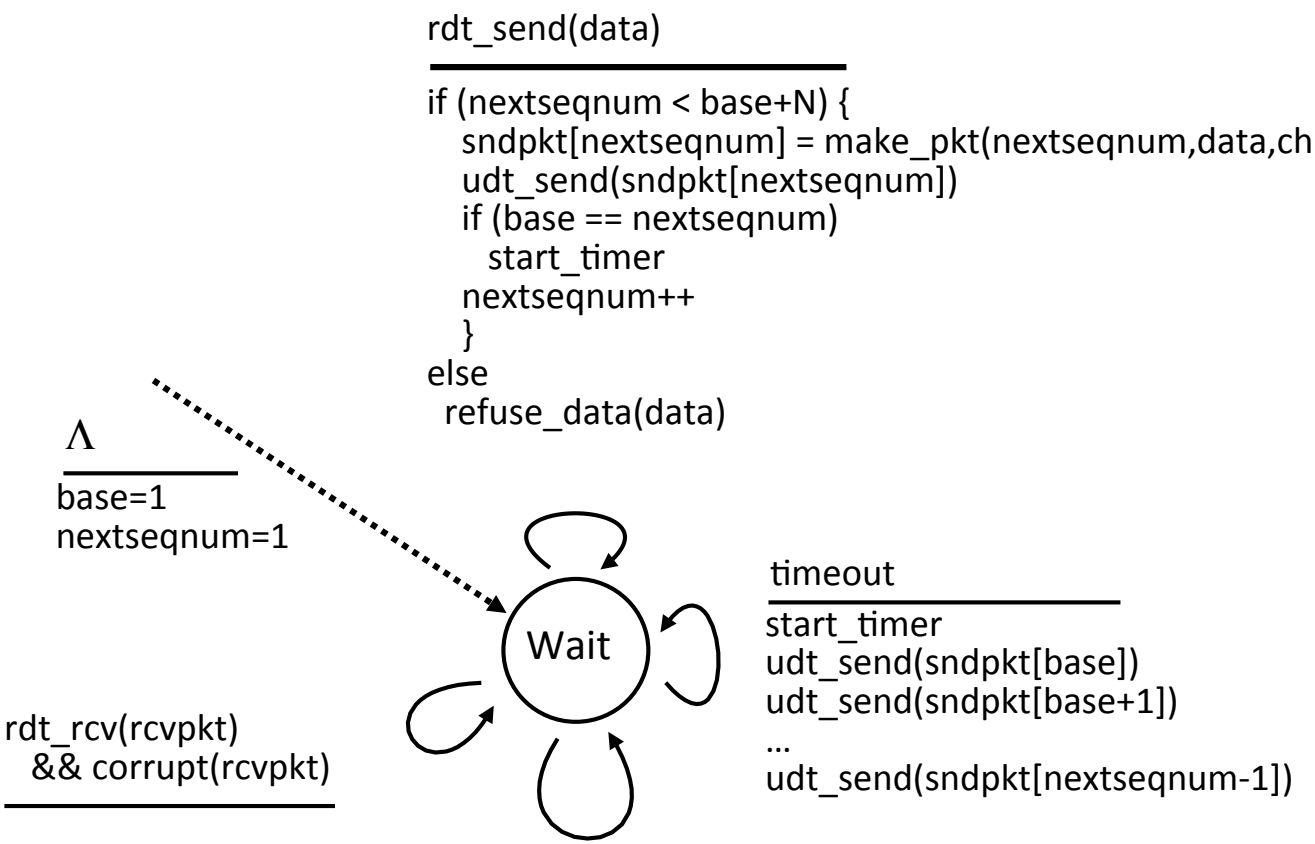
k-bit seq # in packet header

"window" of up to N, consecutive unack-ed packets allowed

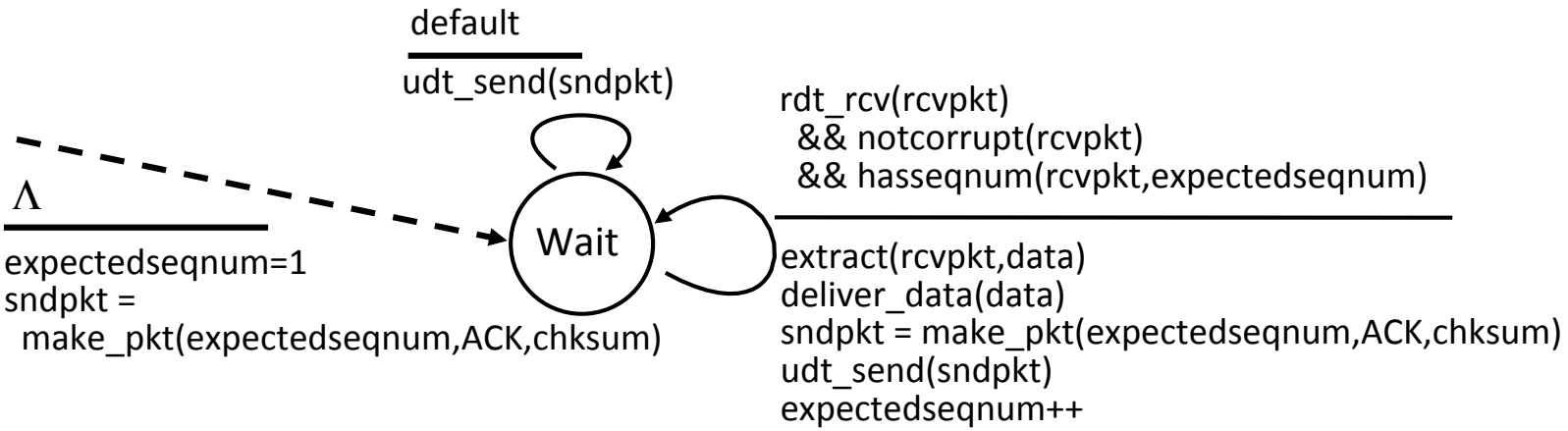


- ACK(n): ACKs all packets up to, including seq # n - *"cumulative ACK"*
  - may receive duplicate ACKs (see receiver)
- Timer for oldest in-flight packet
- *Timeout(n):* retransmit packet n and all higher seq # packets in window

GBN: sender extended FSM



GBN: receiver extended FSM



- ACK-only: always send ACK for correctly-received packet with highest *in-order* seq #

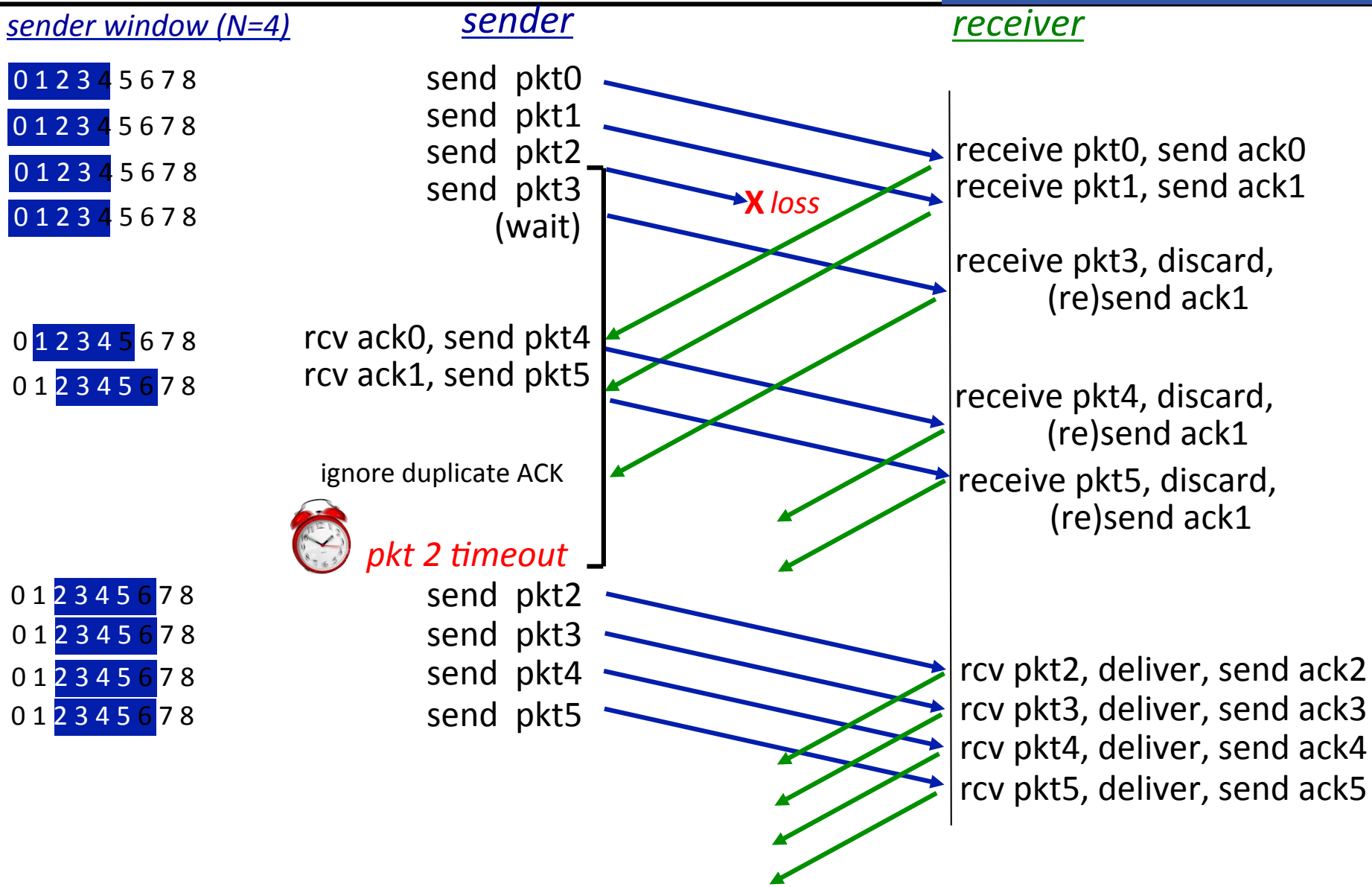
  may generate duplicate ACKs

  need only remember **expectedseqnum**
- Out-of-order pkt:

  discard (don't buffer): *no receiver buffering!*

  re-ACK pkt with highest in-order seq #

GBN in action



Selective repeat

- Receiver *individually* acknowledges all correctly received packets

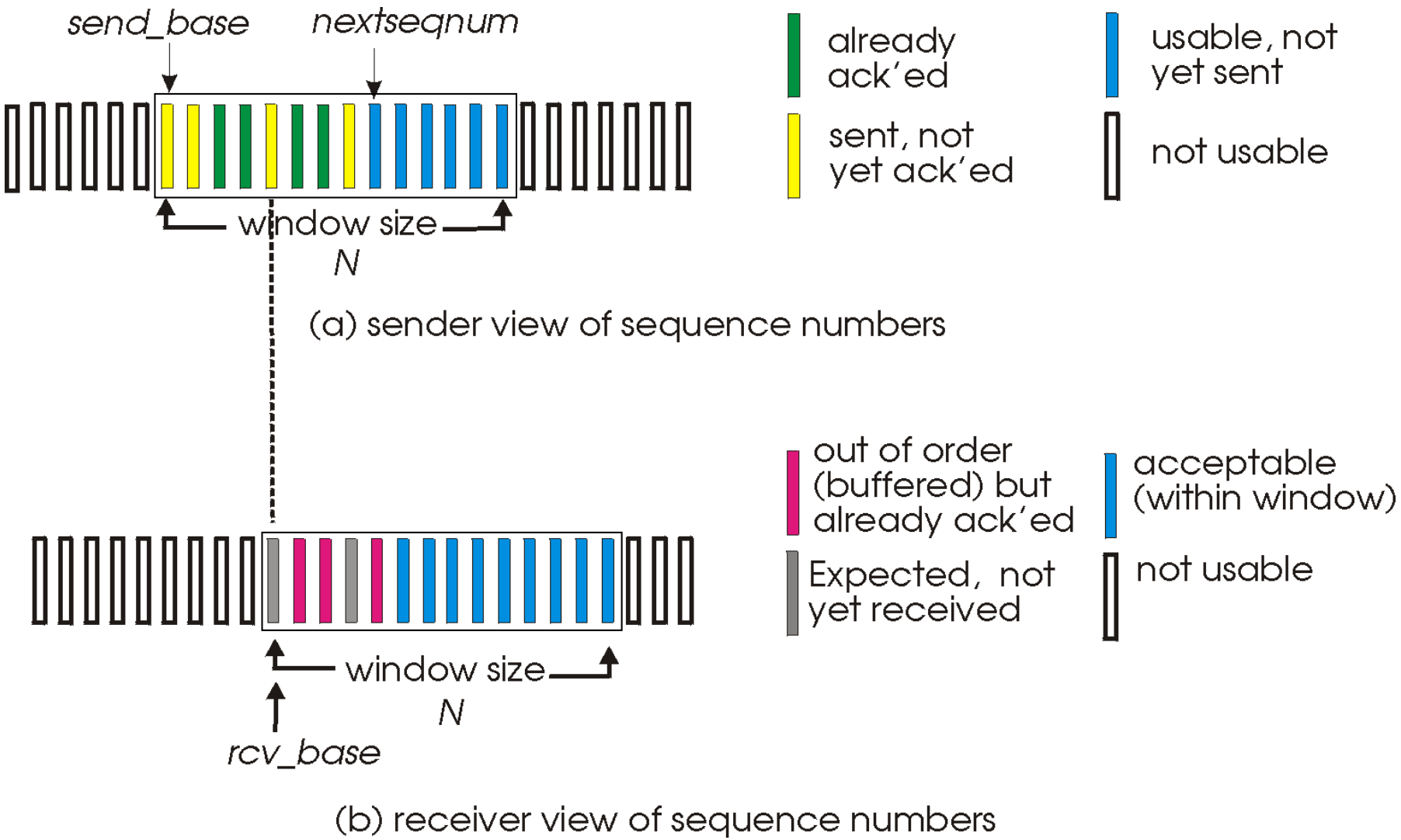
  Buffers packets, as needed, for eventual in-order delivery to upper layer
- Sender only resends packets for which ACK not received

  sender timer for each unACKed packet
- Sender window

  N consecutive seq #'s

  limits seq #'s of sent, unACKed packets

Selective repeat: sender, receiver windows



Selective repeat

Queen Mary University of London

sender

- Data from above:

if next available seq # in window, send pkt

- Timeout(n):

resend pkt n, restart timer

- ACK(n) in [sendbase,sendbase+N]:

mark pkt n as received

if n smallest unACKed pkt, advance window base to next unACKed seq #

receiver

- Pkt n in [rcvbase, rcvbase+N-1]

– send ACK(n)

– out-of-order: buffer

– in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

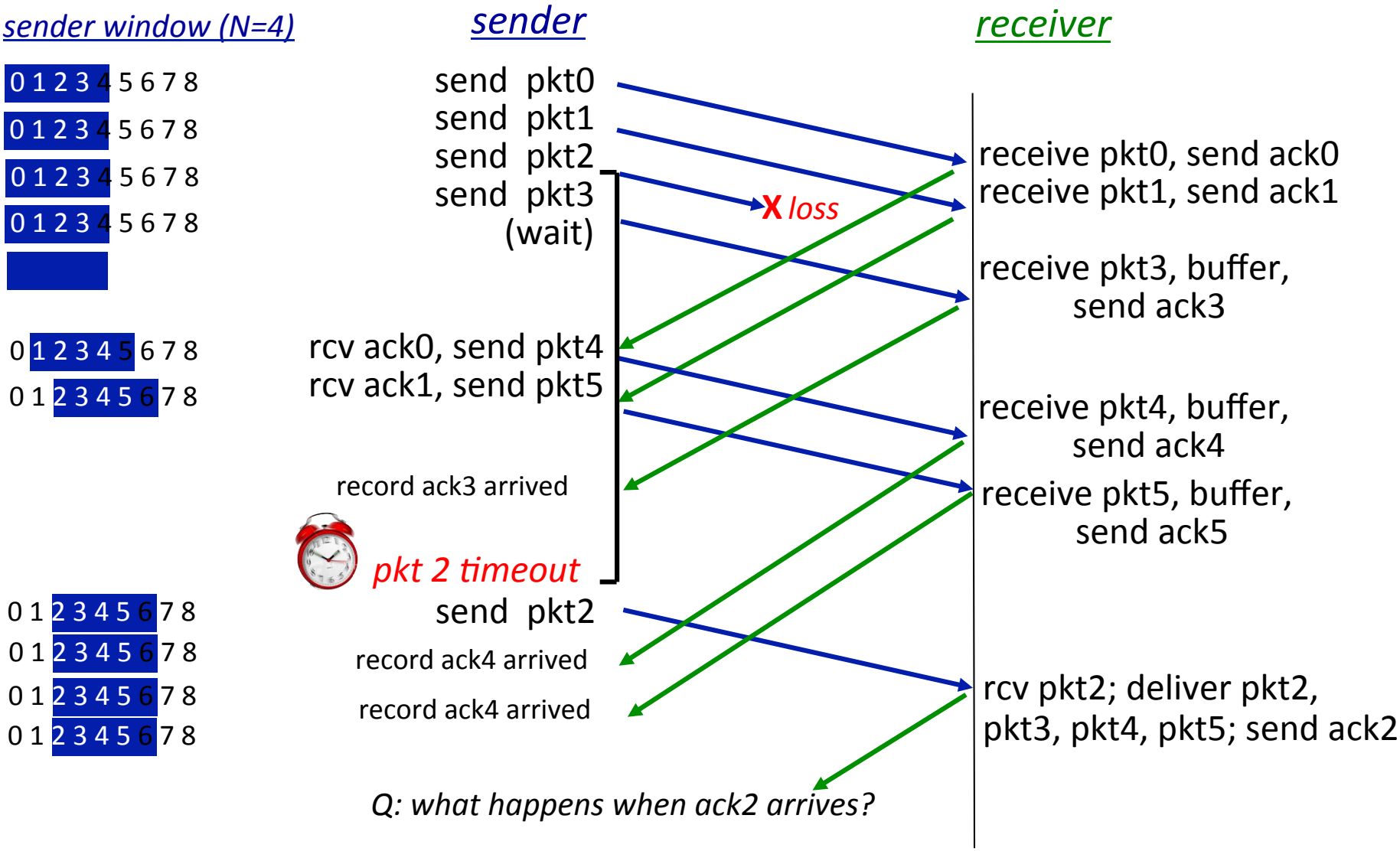
- Pkt n in [rcvbase-N,rcvbase-1]

– ACK(n)

- Otherwise:

– ignore

Selective repeat in action



Selective repeat: dilemma

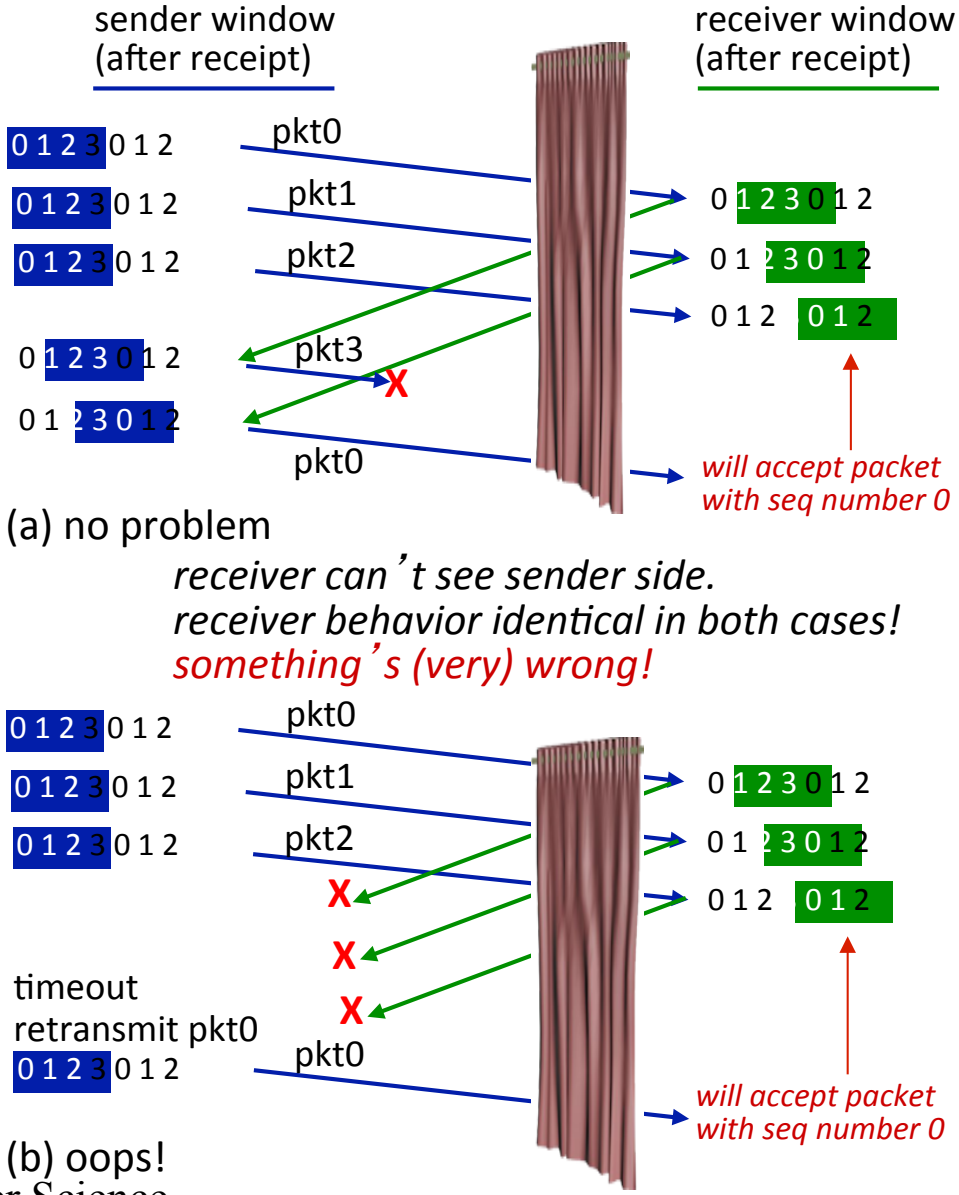
- Example:

seq #'s: 0, 1, 2, 3

window size=3

- receiver sees no difference in two scenarios!
- duplicate data accepted as new in (b)

Q: what relationship between seq # size and window size to avoid problem in (b)?



The Transport layer

- Pipelined protocols
- **TCP**
- Congestion control

TCP: Overview

RFCs: 793,1122,1323, 2018, 2581

Queen Mary University of London

- 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:

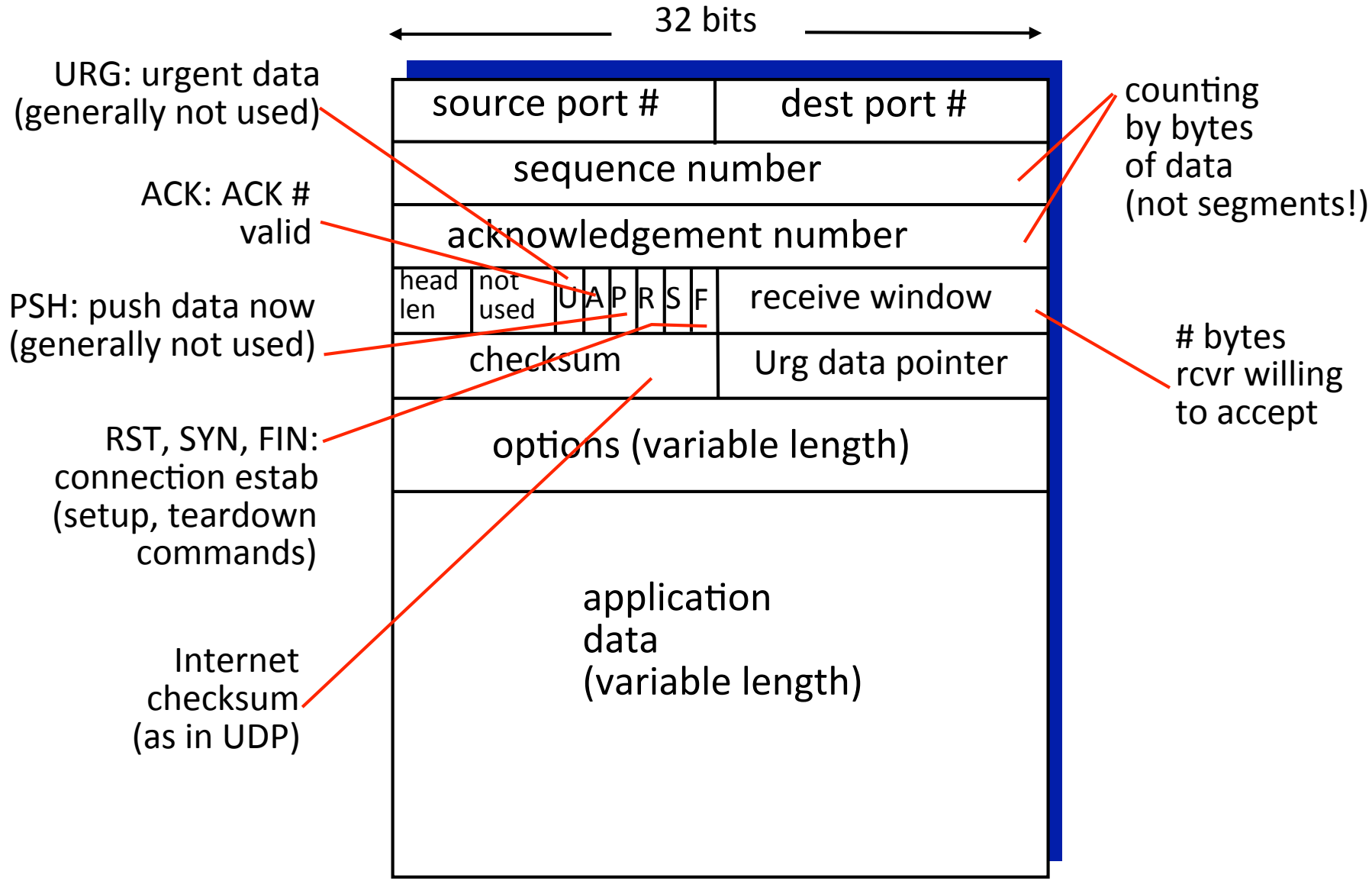
  bi-directional data flow in same connection

  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. numbers, ACKs

- Sequence numbers:

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

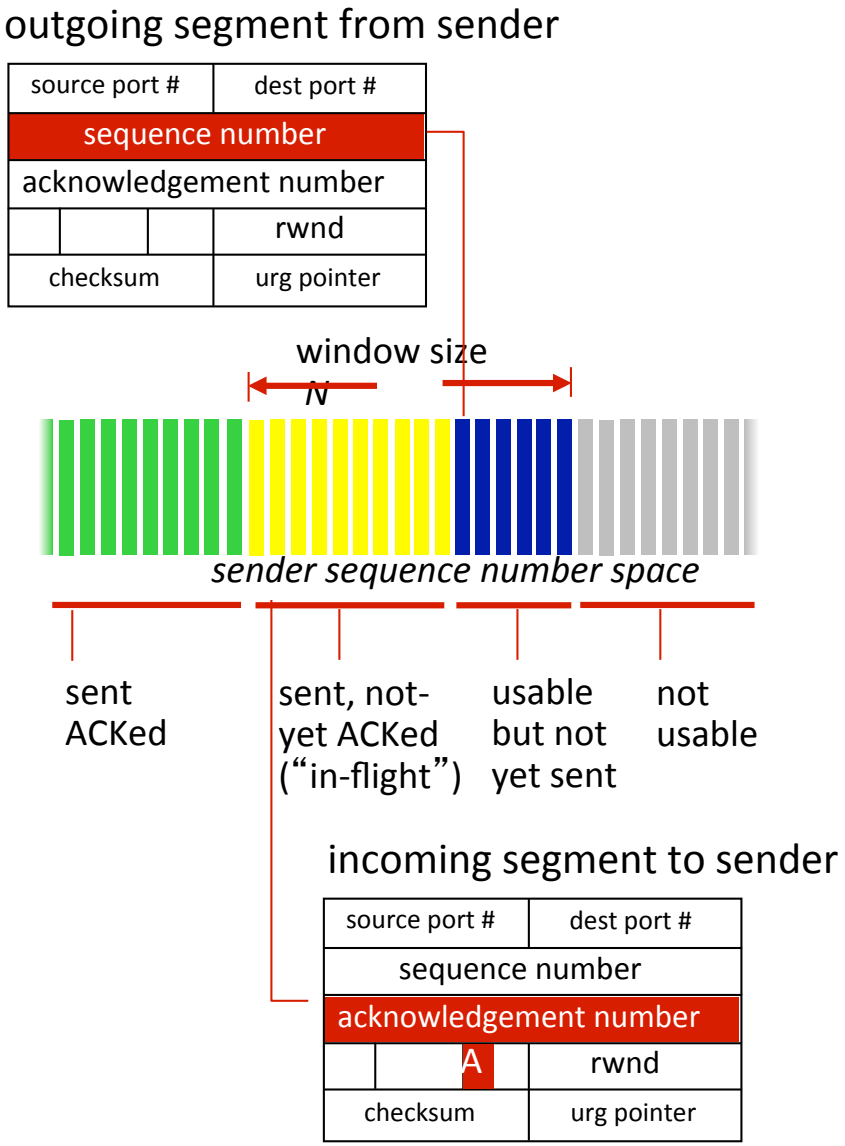
- Acknowledgements:

  seq # of next byte expected from other side

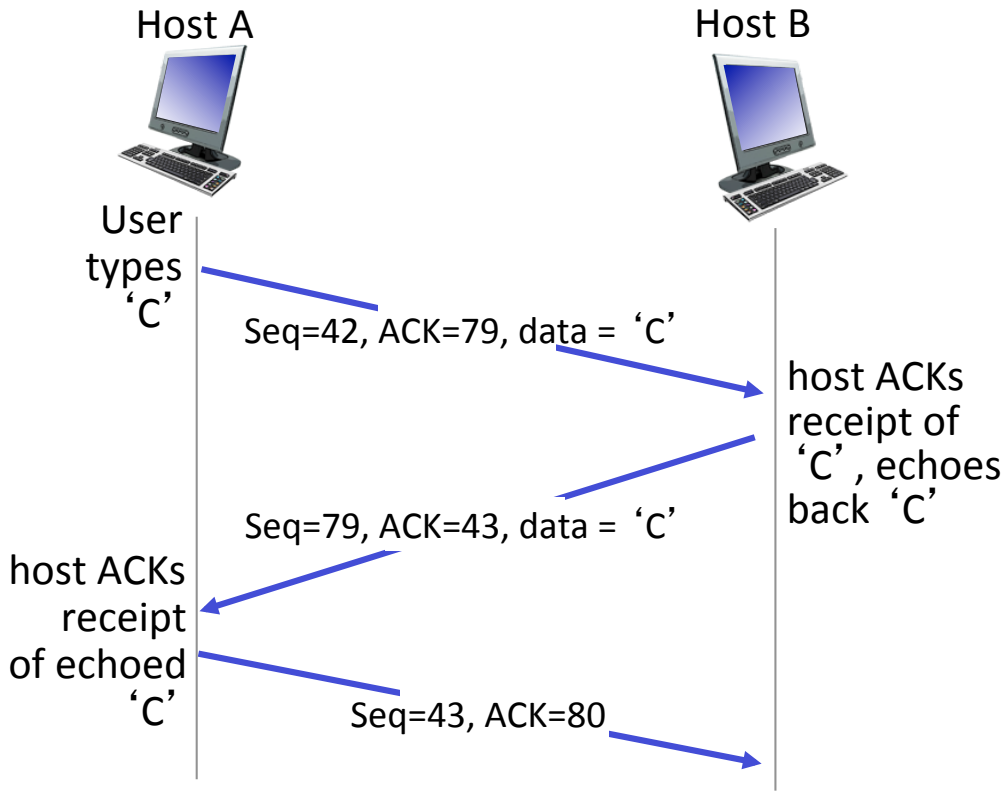
  cumulative ACK

- Q: how receiver handles out-of-order segments

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



TCP seq. numbers, ACKs



simple telnet scenario

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

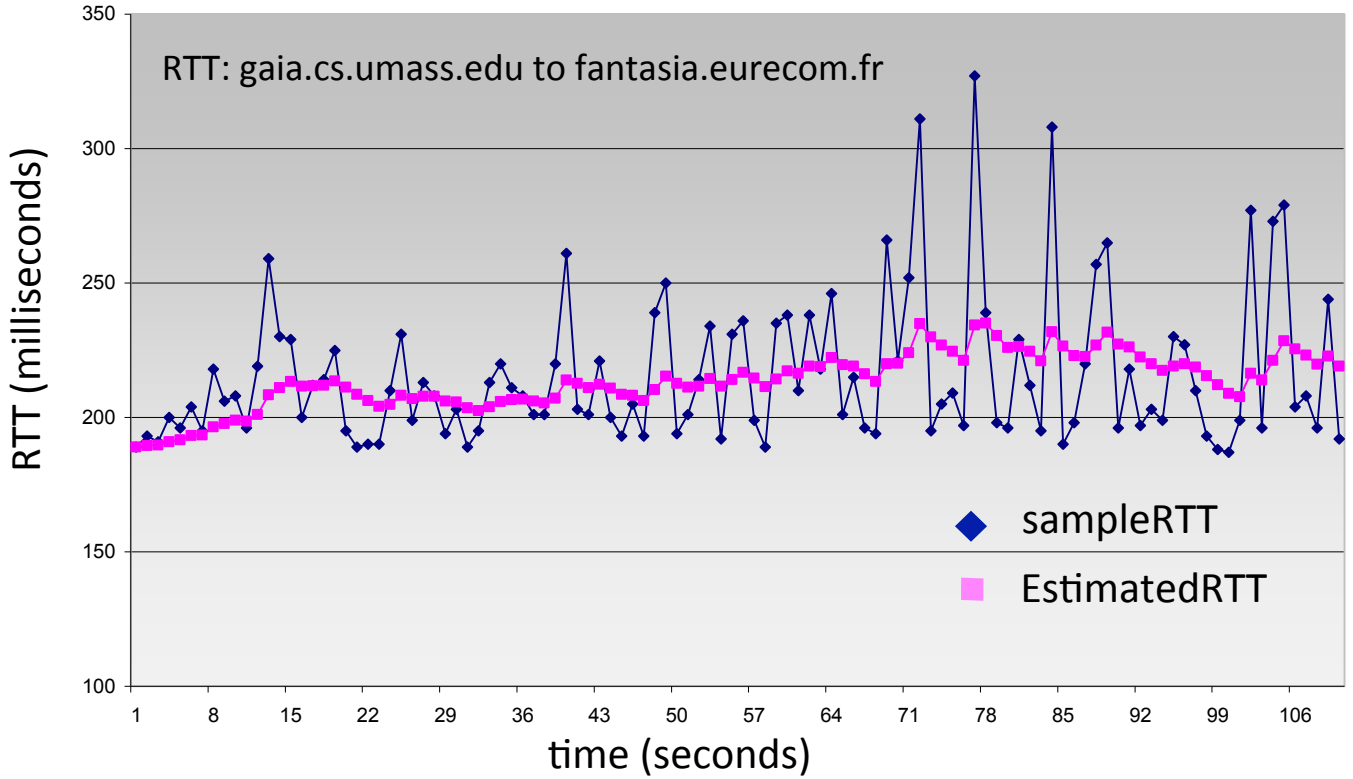
  ignore retransmissions
- **SampleRTT** will vary, want estimated RTT "smoother"

  average several *recent* measurements, not just current **SampleRTT**

TCP round trip time, timeout

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

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



TCP round trip time, timeout

Queen Mary University of London

- 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"

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

- Let's initially consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control

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

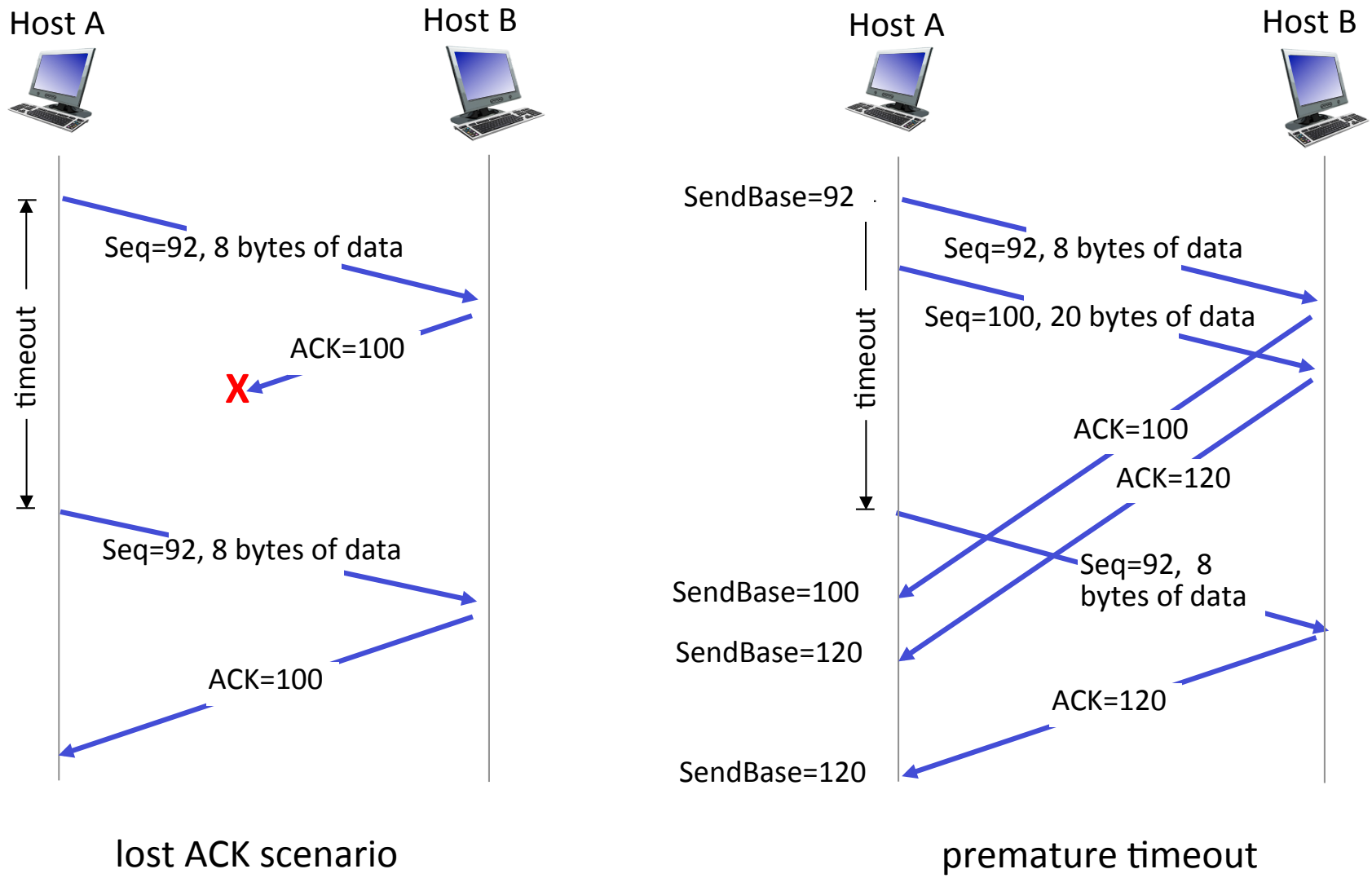
- *Ack rcvd:*

if ack acknowledges previously unacked segments

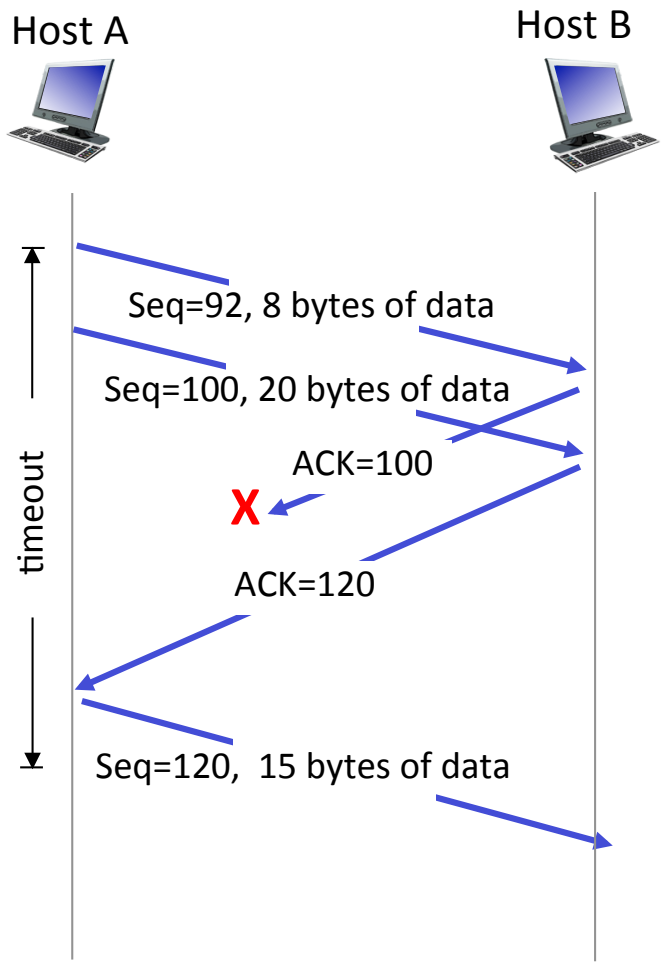
update what is known to be ACKed

start timer if there are still unacked segments

TCP: retransmission scenarios



TCP: retransmission scenarios



cumulative ACK

TCP ACK generation

[RFC 1122, RFC 2581]

Queen Mary University of London

Event at receiver	*TCP receiver action*
Arrival of in-order segment with expected seq #. All data up to expected 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, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send *duplicate ACK*, indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment starts at lower end of gap

TCP fast retransmit

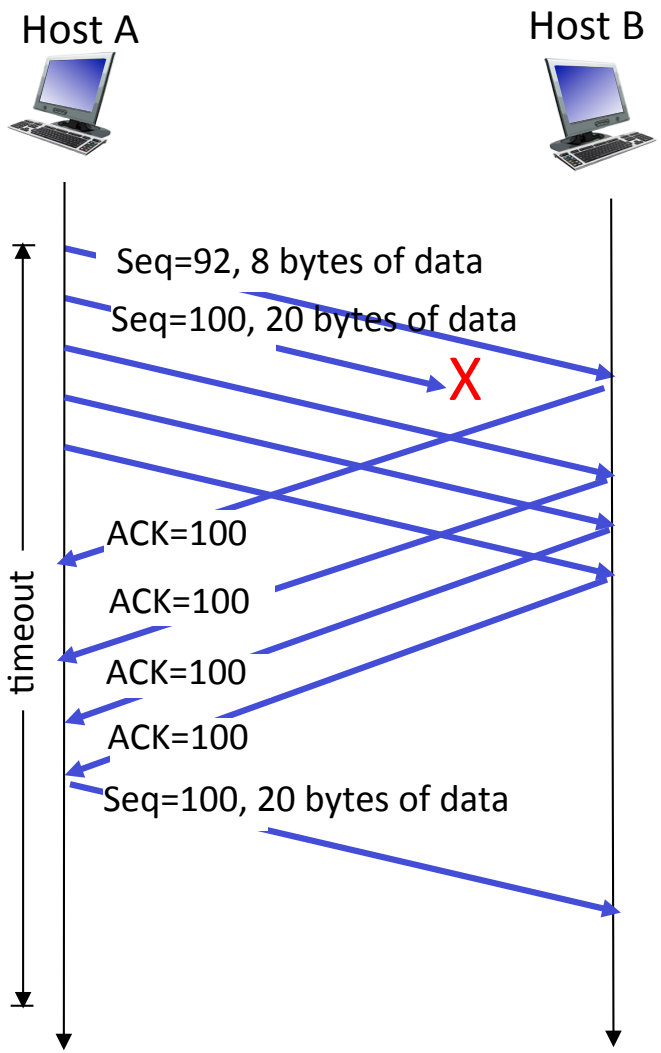
- Time-out period often relatively long:
  - long delay before resending lost packet
- Detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

TCP fast retransmit

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

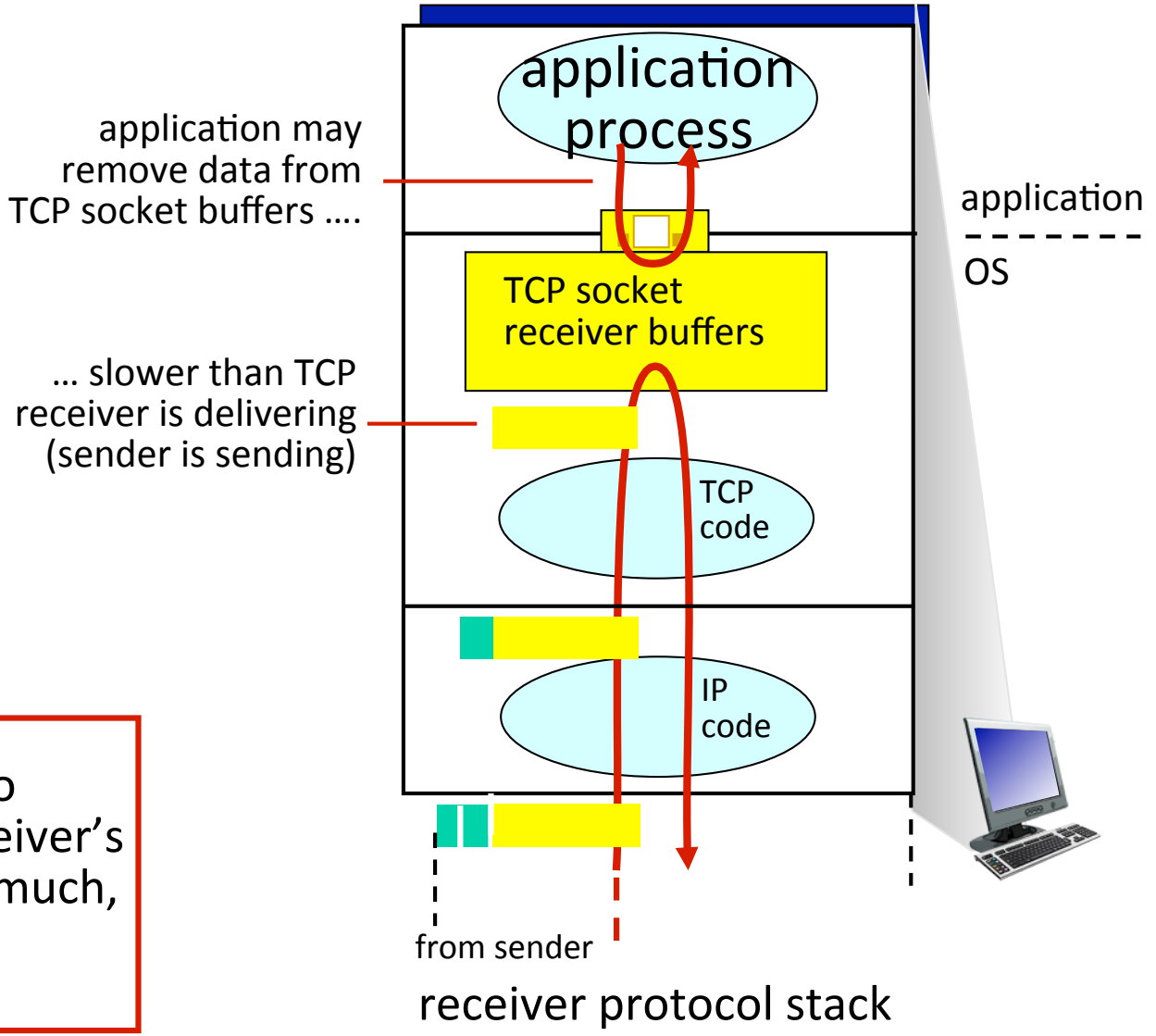
- likely that unacked segment lost, so don't wait for timeout

TCP fast retransmit



fast retransmit after sender receipt of triple duplicate ACK

TCP flow control

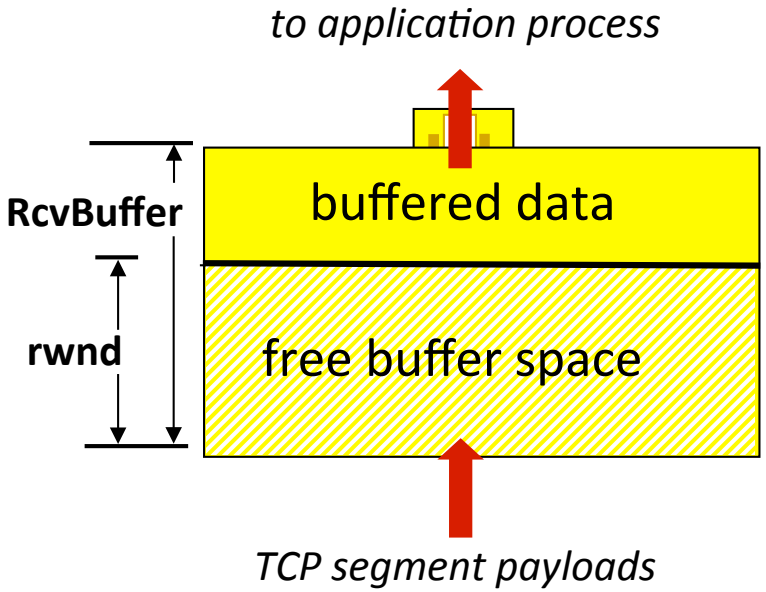


flow control

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



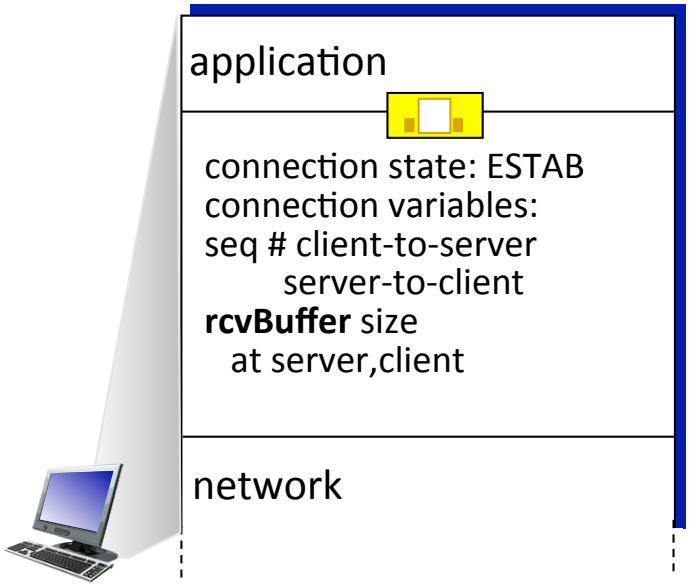
receiver-side buffering

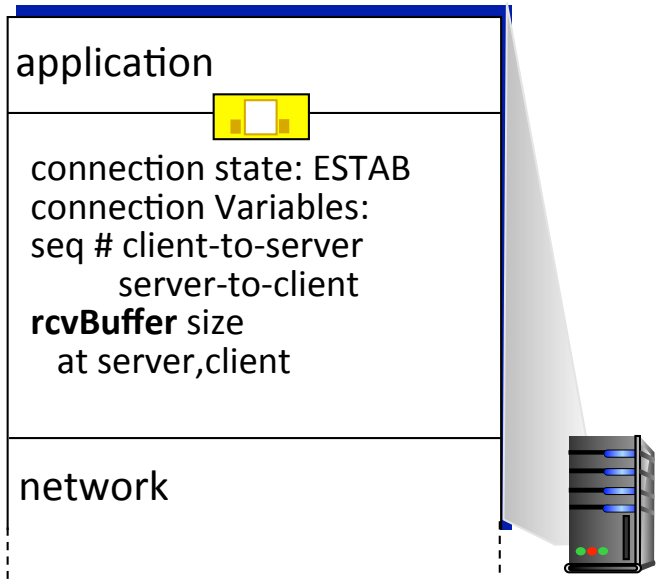
Connection Management

- Before exchanging data, sender/receiver "handshake":

agree to establish connection (each knowing the other willing to establish connection)

agree on connection parameters

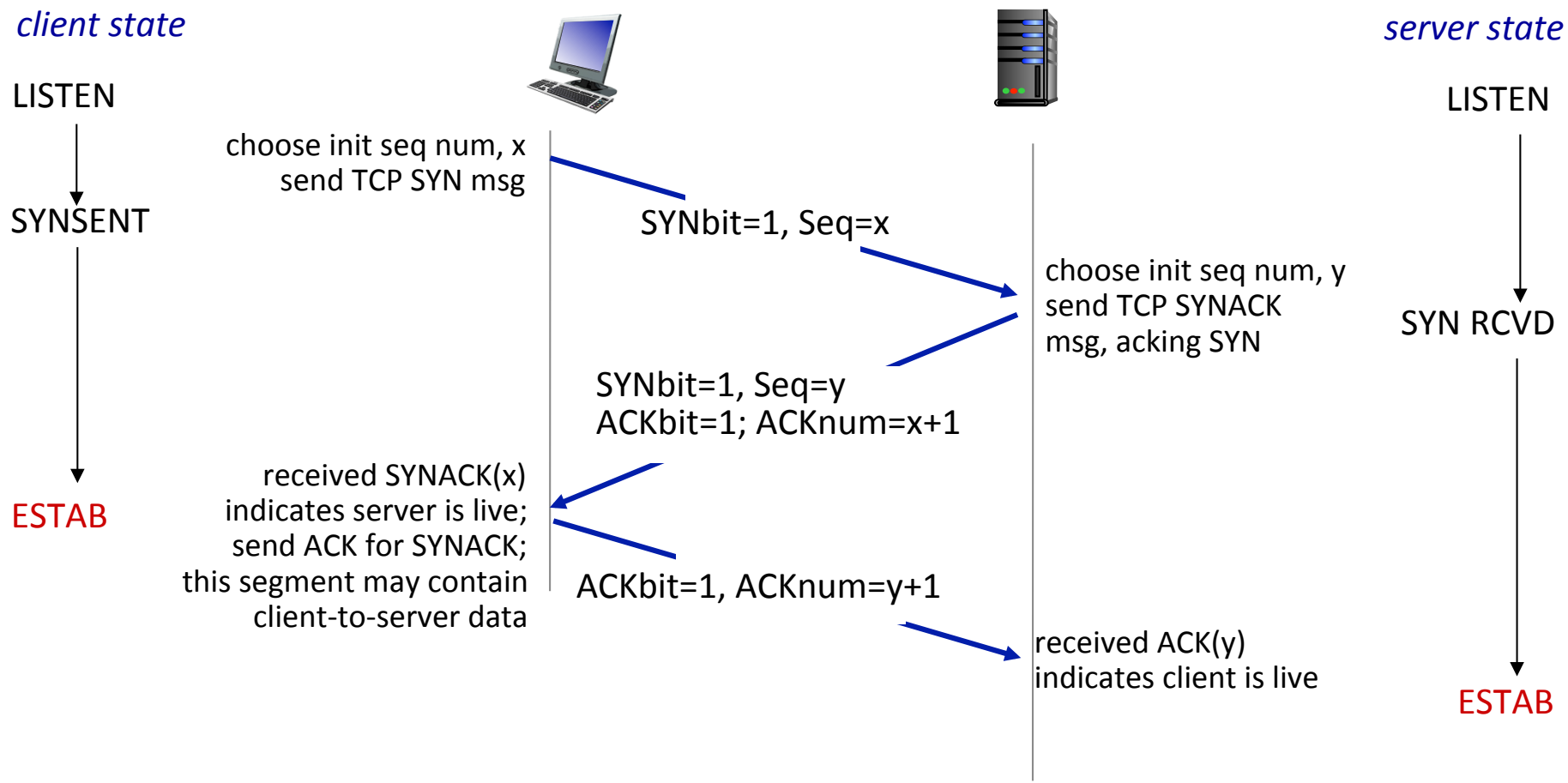




Socket clientSocket =
newSocket("hostname","port number");

Socket connectionSocket =
welcomeSocket.accept();

TCP 3-way handshake



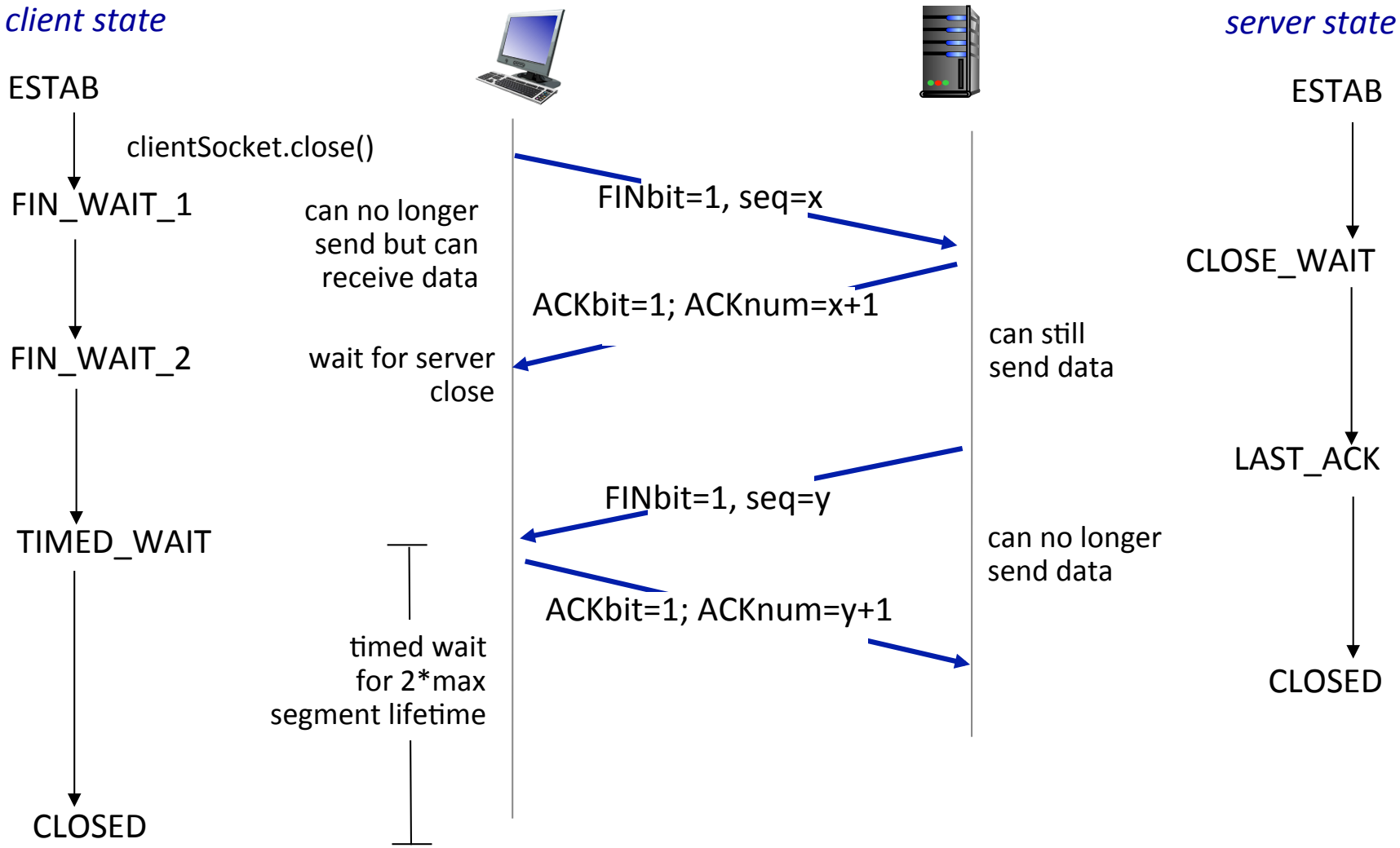
TCP: closing a connection

- 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



The Transport layer

- Reliable transfer
- Pipelined protocols
- 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:

  lost packets (buffer overflow at routers)

  long delays (queuing in router buffers)
- A top-10 problem!

Approaches towards congestion control

Queen Mary University of London

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 for sender to send at

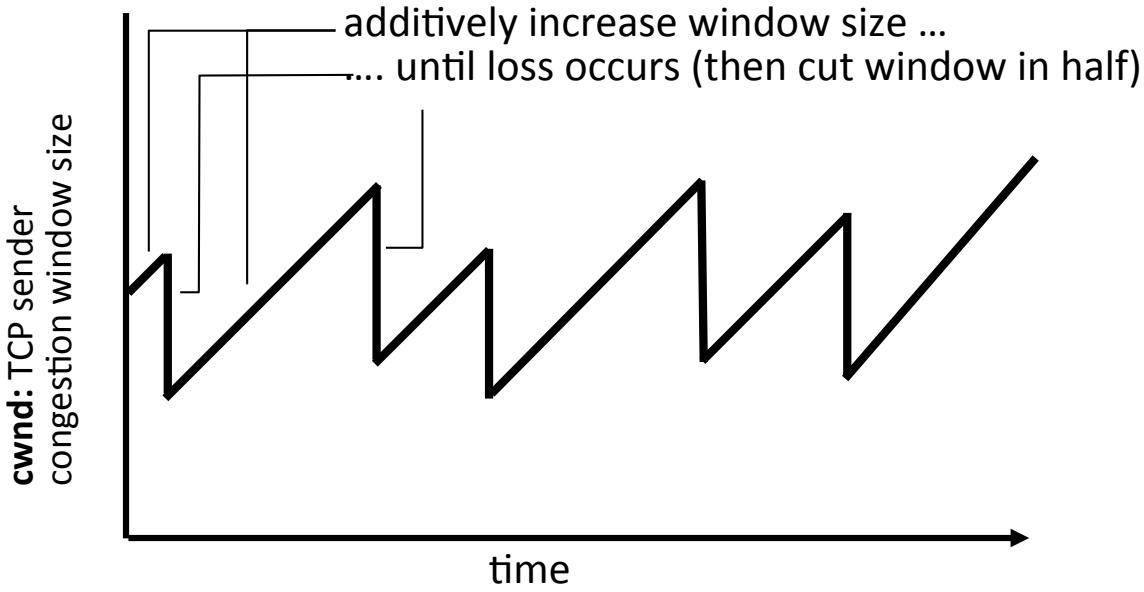
TCP congestion control: additive increase multiplicative decrease

Queen Mary University of London

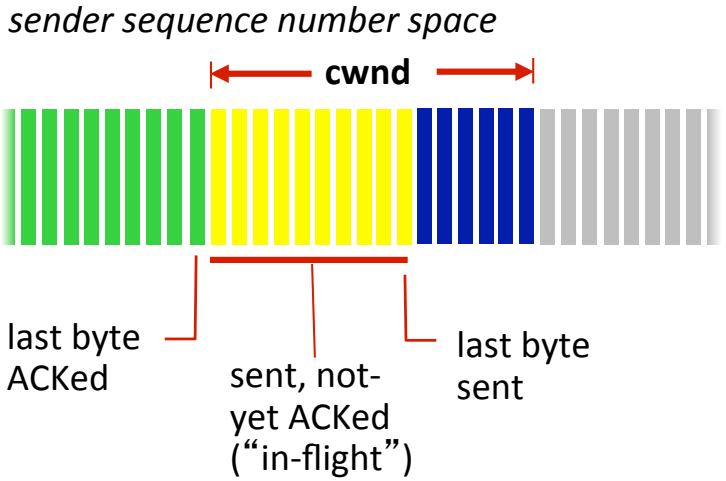
Approach: sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs

- *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:

LastByteSent-LastByteAcked $\leq$ **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 Slow Start

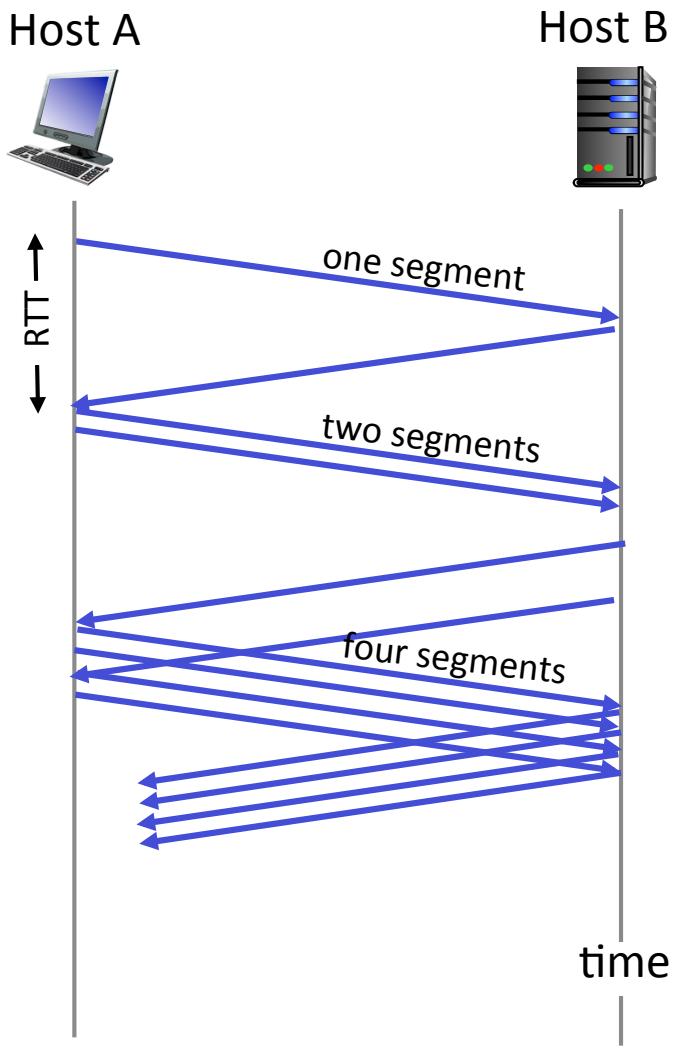
- When connection begins, increase rate exponentially until first loss event:

  initially **cwnd** = 1 MSS

  double **cwnd** every RTT

  done by incrementing **cwnd** for every ACK received

- *Summary:* initial rate is slow but ramps up exponentially fast



TCP: detecting, reacting to loss

- Loss indicated by timeout:

  cwnd set to 1 MSS;

  window then grows exponentially (as in slow start) to threshold, then grows linearly

- Loss indicated by 3 duplicate ACKs: TCP RENO

  dup ACKs indicate network capable of delivering some segments

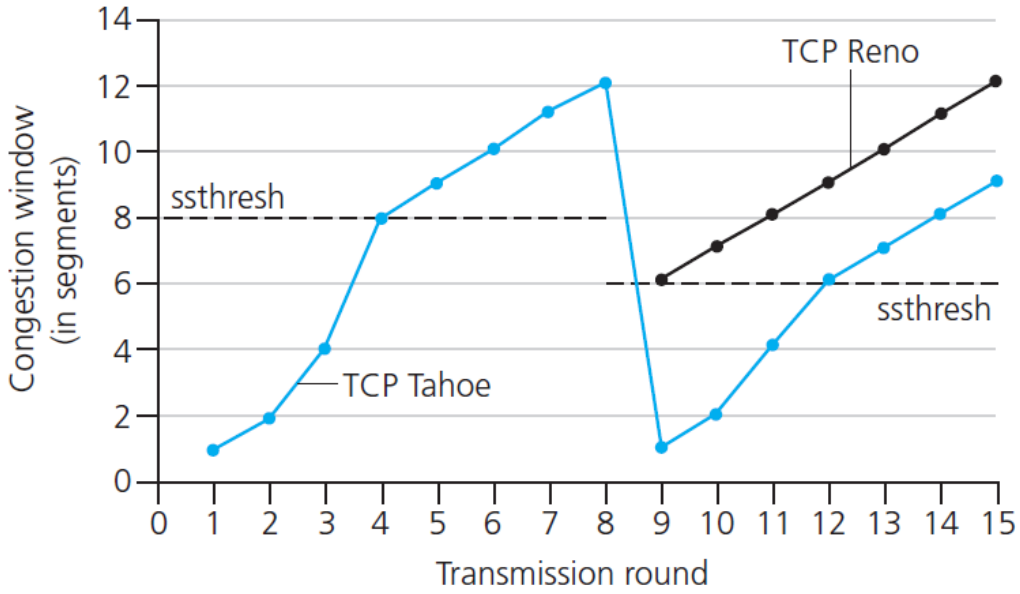
  cwnd is cut in half window then grows linearly

- TCP Tahoe always sets **cwnd** to 1 (timeout or 3 duplicate acks)

TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

A: when **cwnd** gets to 1/2 of its value before timeout.

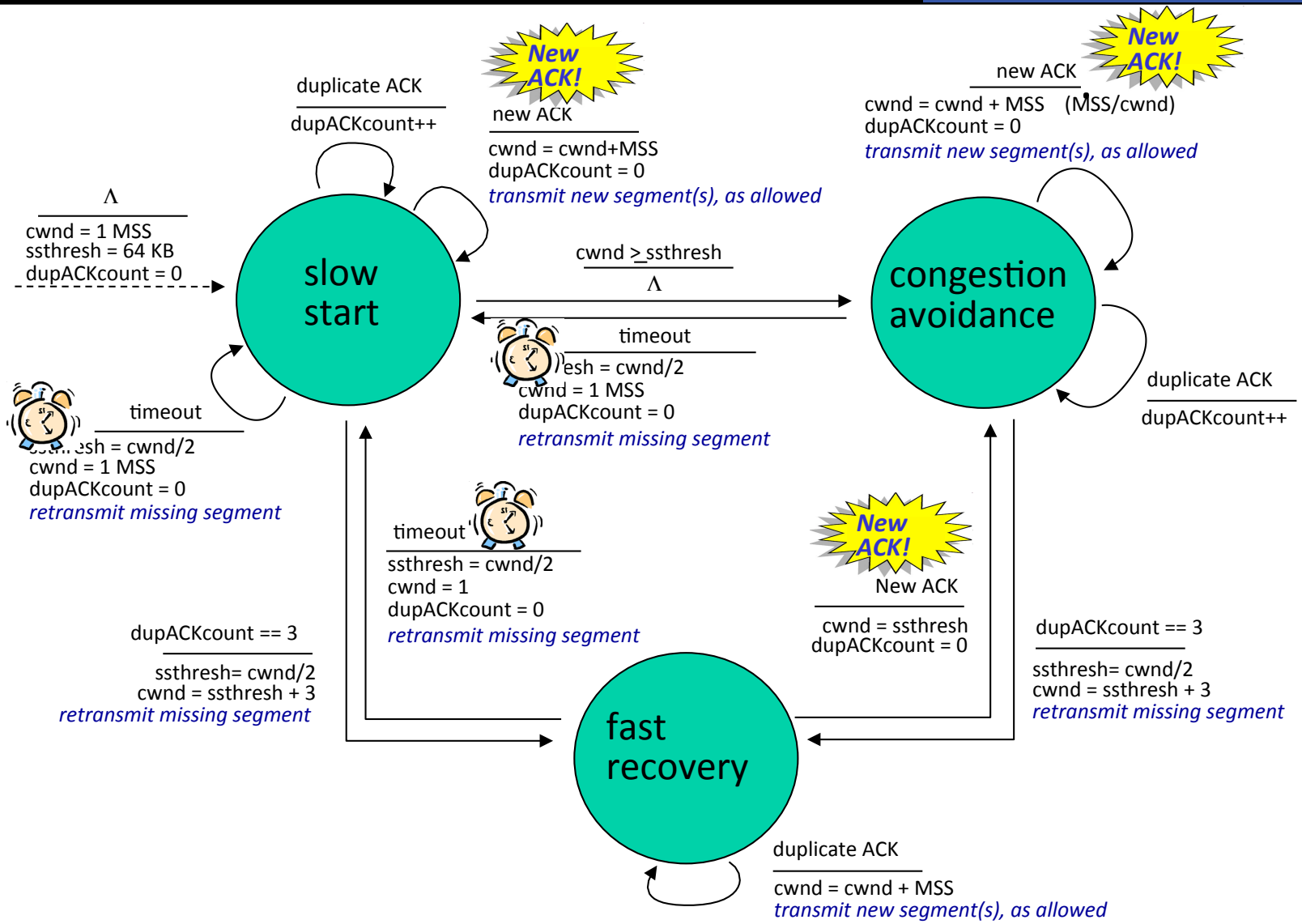


Implementation:

variable **ssthresh**

on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event

Summary: TCP Congestion Control



TCP throughput

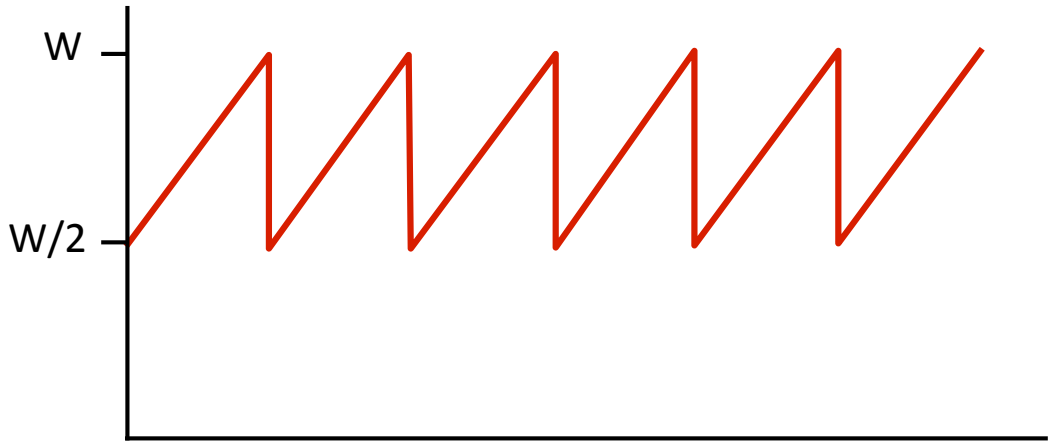
Avg. TCP throughput as function of window size, RTT?

- 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. throughput is 3/4W per RTT

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



TCP Futures: TCP over "long, fat pipes"

- Example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- requires W = 83,333 in-flight segments
- throughput in terms of segment loss probability, L [Mathis 1997]:

$$\text{TCP throughput} = \frac{1.22 \cdot \text{MSS}}{\text{RTT}\sqrt{L}}$$

➜ to achieve 10 Gbps throughput, need a loss rate of $L = 2 \cdot 10^{-10}$ *– a very small loss rate!*

- New versions of TCP for high-speed