

CS 305 Computer Networks

Chapter 3 Transport Layer (2)

Jin Zhang

Department of Computer Science and Engineering
Southern University of Science and Technology

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

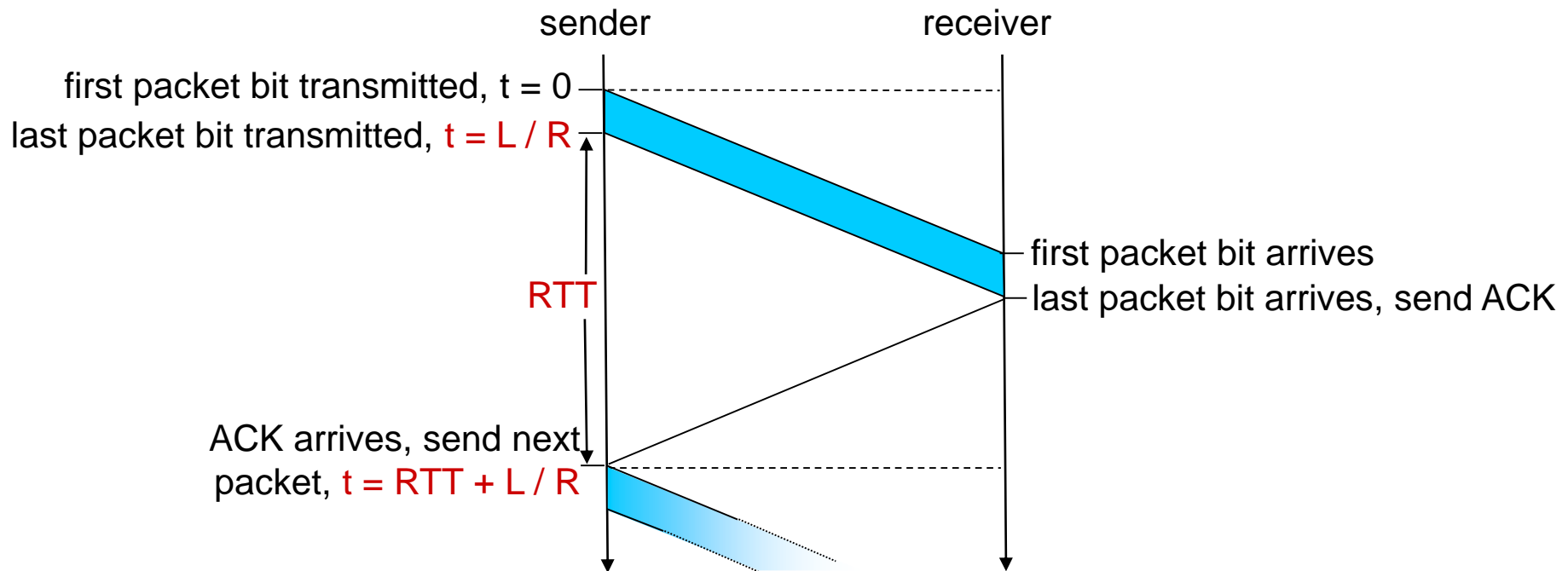
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

Performance of rdt3.0

- ❖ rdt3.0 is correct, but performance is bad
- ❖ e.g.: link rate $R=1$ Gbps, prop. delay $T_{pd}=15$ ms, packet length $L=8000$ bit



- Calculate U_{sender} : **utilization** – fraction of time sender busy sending

Performance of rdt3.0

- ❖ link rate $R=1$ Gbps, prop. delay $T_{pd}=15$ ms, packet length $L=8000$ bit

$$D_{trans} = t = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

- U_{sender} : **utilization** – fraction of time sender busy sending

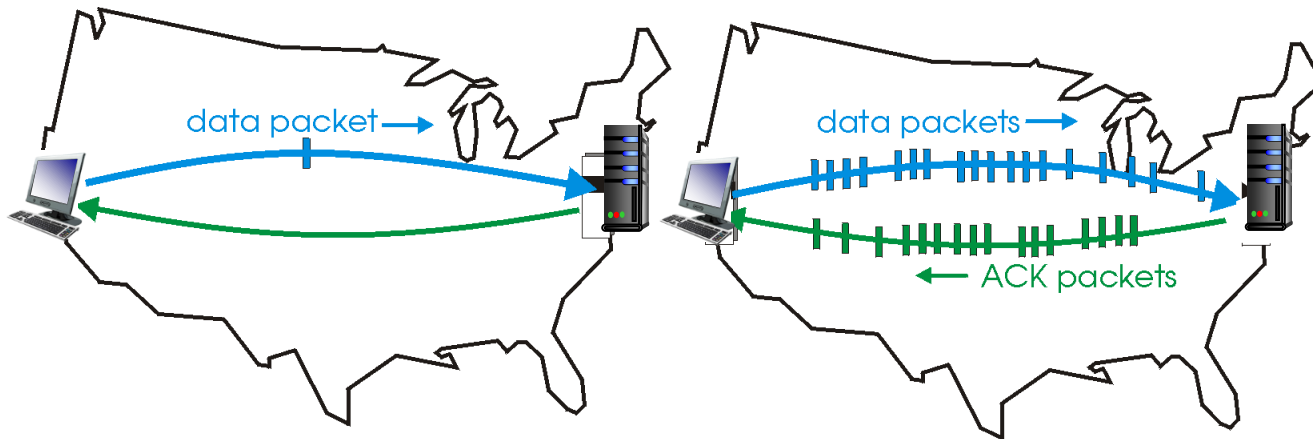
$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

- RTT=30 msec, 1KB pkt every 30 msec:
33kB/sec throughput over 1 Gbps link
- ❖ network protocol limits use of physical resources!

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

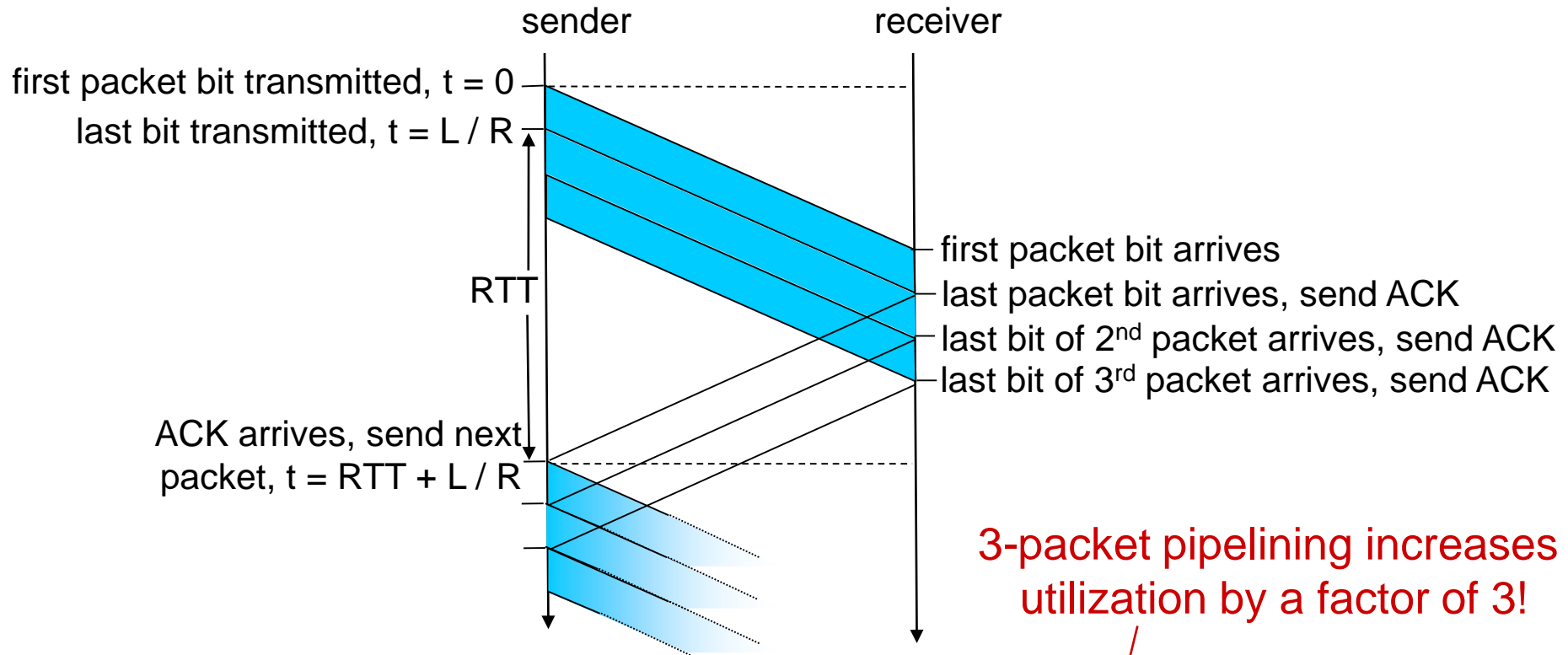


(a) a stop-and-wait protocol in operation

(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!

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

Go-Back-N overview

sender window (N=4)

0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8

sender

send pkt0
send pkt1
send pkt2
send pkt3
(wait)

rcv ack0, send pkt4
rcv ack1, send pkt5

ignore duplicate ACK



pkt 2 timeout

send pkt2
send pkt3
send pkt4
send pkt5

receiver

No buffer,
Cumulative ACK

receive pkt0, send ack0
receive pkt1, send ack1

receive pkt3, discard,
(re)send ack1

receive pkt4, discard,
(re)send ack1

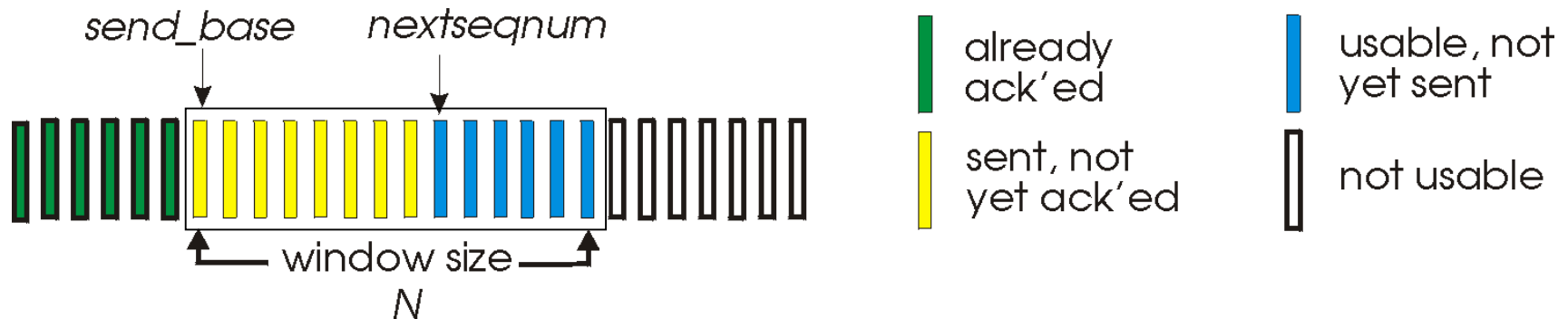
receive pkt5, discard,
(re)send ack1

rcv pkt2, deliver, send ack2
rcv pkt3, deliver, send ack3
rcv pkt4, deliver, send ack4
rcv pkt5, deliver, send ack5

Retransmit all pkts upon
pkt loss or error (GBN)

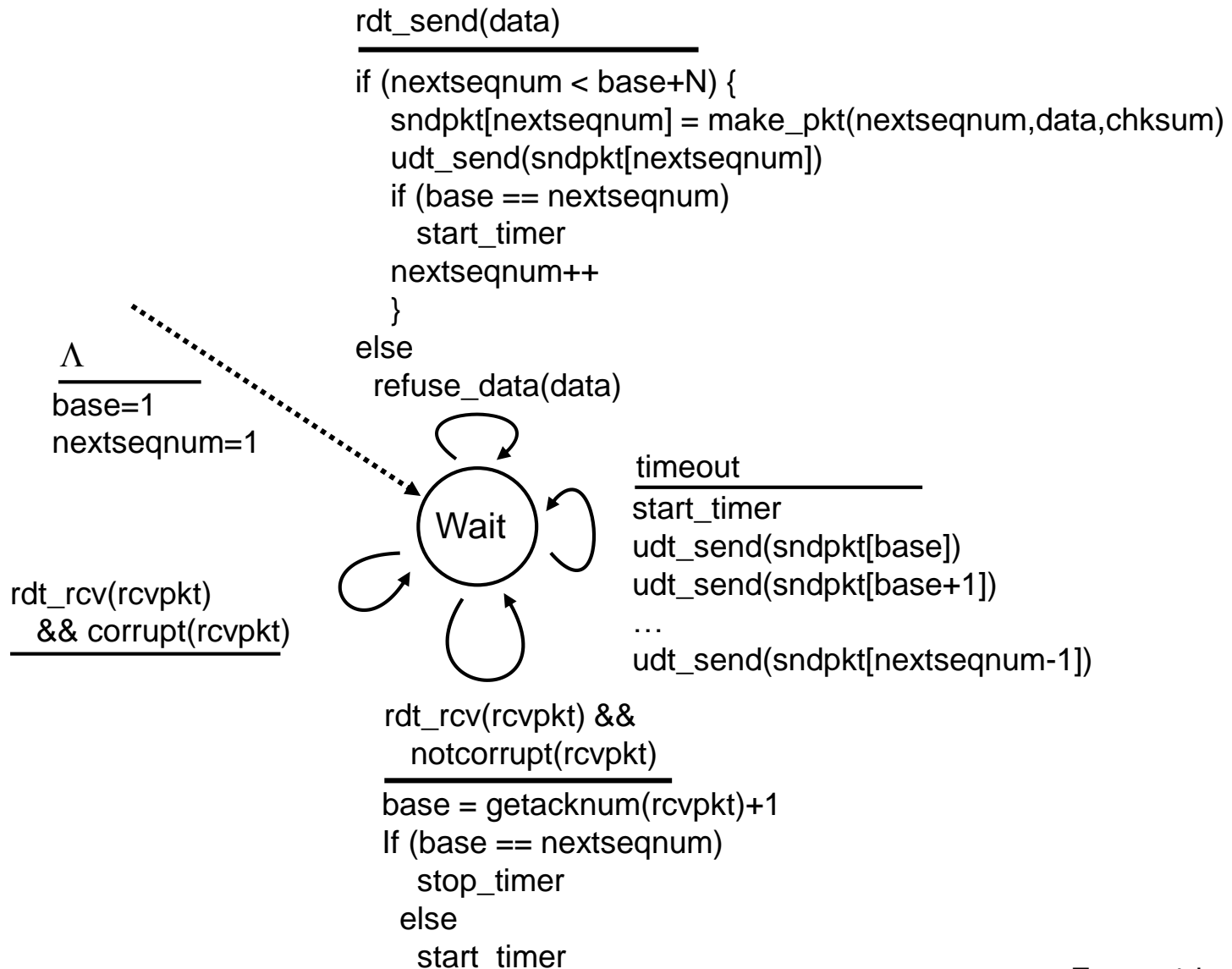
Go-Back-N: sender

- ❖ k-bit seq # in pkt header (not 0 or 1)
- ❖ At most N pkts in flight: window size = N, (N consecutive unacked pkts allowed)

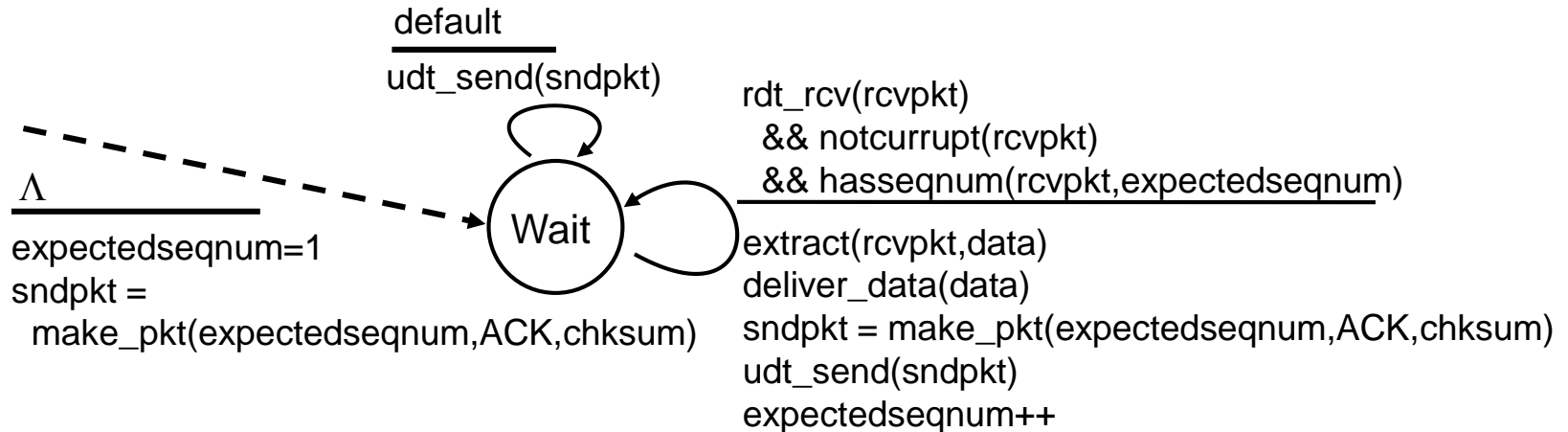


- ❖ ACK(n) means all pkts before pkt n are correctly received -
“cumulative ACK”
 - Sender may receive duplicate ACKs (see receiver)
- ❖ timer for oldest in-flight pkt
- ❖ *timeout(n)*: retransmit packet n and all higher seq # pkts in window

GBN: sender extended FSM



GBN: receiver extended FSM



ACK-only: always send ACK for correctly-received pkt 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 #

Selective repeat

sender window (N=4)

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

sender

send pkt0

send pkt1

send pkt2

send pkt3

(wait)

rcv ack0, send pkt4

rcv ack1, send pkt5

record ack3 arrived



pkt 2 timeout

send pkt2

record ack4 arrived

record ack4 arrived

receiver

have buffer,
individual ACK

receive pkt0, send ack0

receive pkt1, send ack1

receive pkt3, buffer,
send ack3

receive pkt4, buffer,
send ack4

receive pkt5, buffer,
send ack5

rcv pkt2; deliver pkt2,
pkt3, pkt4, pkt5; send ack2

X loss

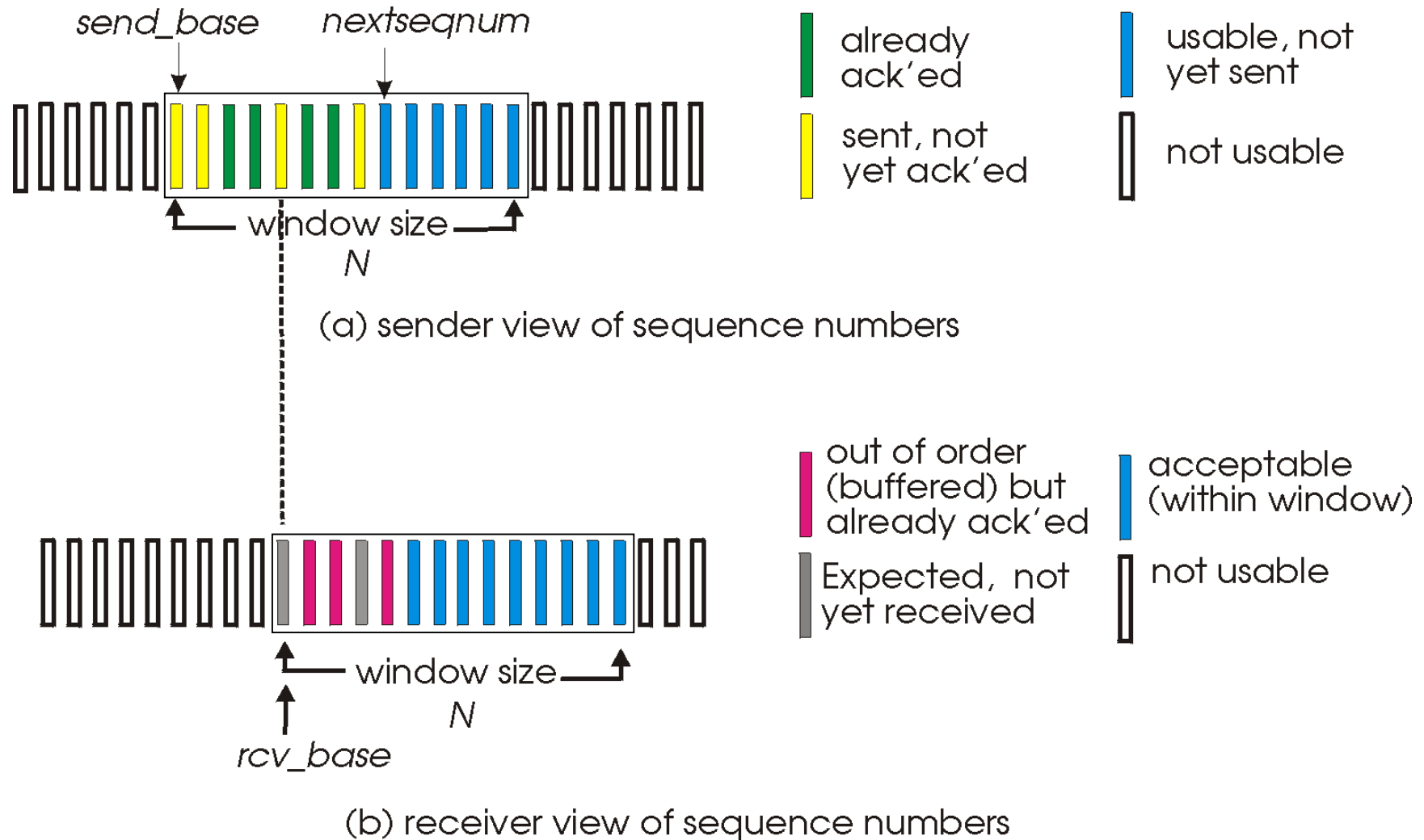
Only retransmit the
unacked pkt (SR)

what happens when ack2 arrives?

Selective repeat

- ❖ receiver *individually* acknowledges all correctly received pkts
 - buffers pkts, as needed, for eventual in-order delivery to upper layer
- ❖ sender only resends pkts for which ACK not received
 - sender timer for each unACKed pkt
- ❖ sender window
 - N consecutive seq #'s
 - limits seq #s of sent, unACKed pkts

Selective repeat: sender, receiver windows



Selective repeat

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

GBN and SR comparison

Go-back-N:

- ❖ sender can have up to N unacked packets in pipeline
- ❖ receiver only sends *cumulative ack*
 - doesn't ack packet if there's a gap
- ❖ sender has timer for oldest unacked packet
 - when timer expires, retransmit *all* unacked packets

Selective Repeat:

- ❖ sender can have up to N unack'ed packets in pipeline
- ❖ rcvr sends *individual ack* for each packet
- ❖ sender maintains timer for each unacked packet
 - when timer expires, retransmit only that unacked packet

Selective repeat: dilemma

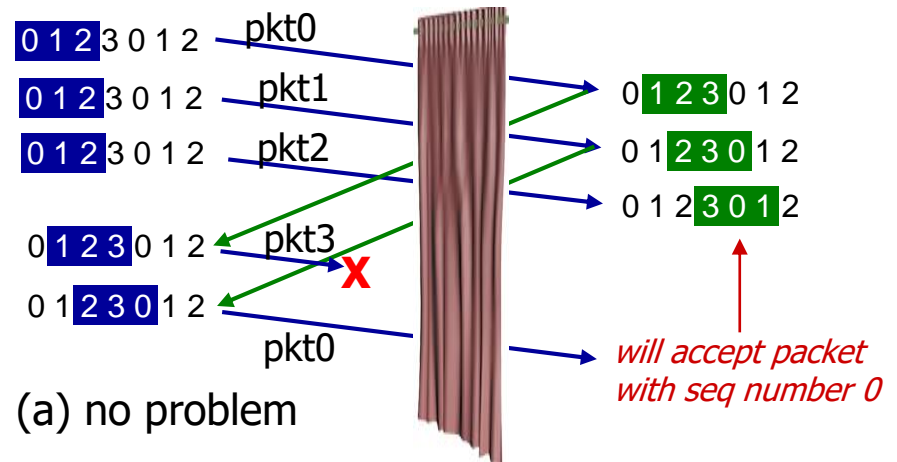
example:

- ❖ seq #'s: 0, 1, 2, 3
- ❖ window size=3
- ❖ receiver sees no difference in two scenarios!
- ❖ duplicate data accepted as new in (b)

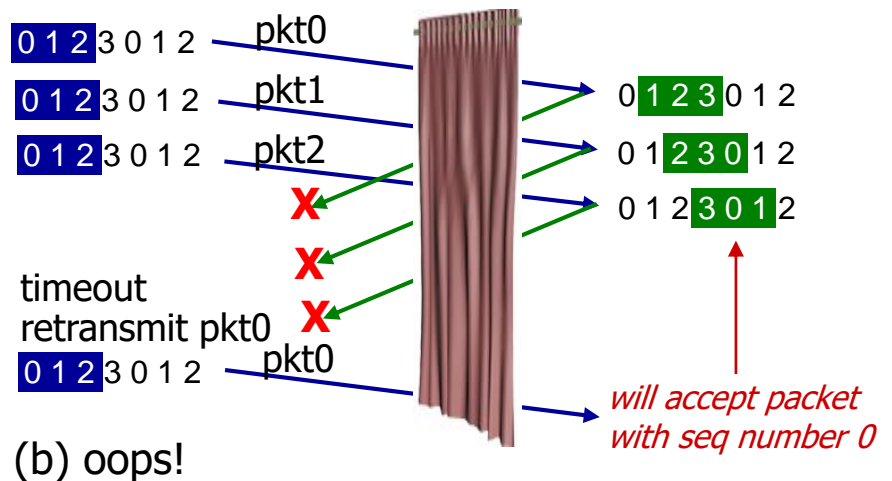
Q: what relationship between seq # size and window size to avoid problem in (b)?

sender window
(after receipt)

receiver window
(after receipt)



*receiver can't see sender side.
receiver behavior identical in both cases!
something's (very) wrong!*



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

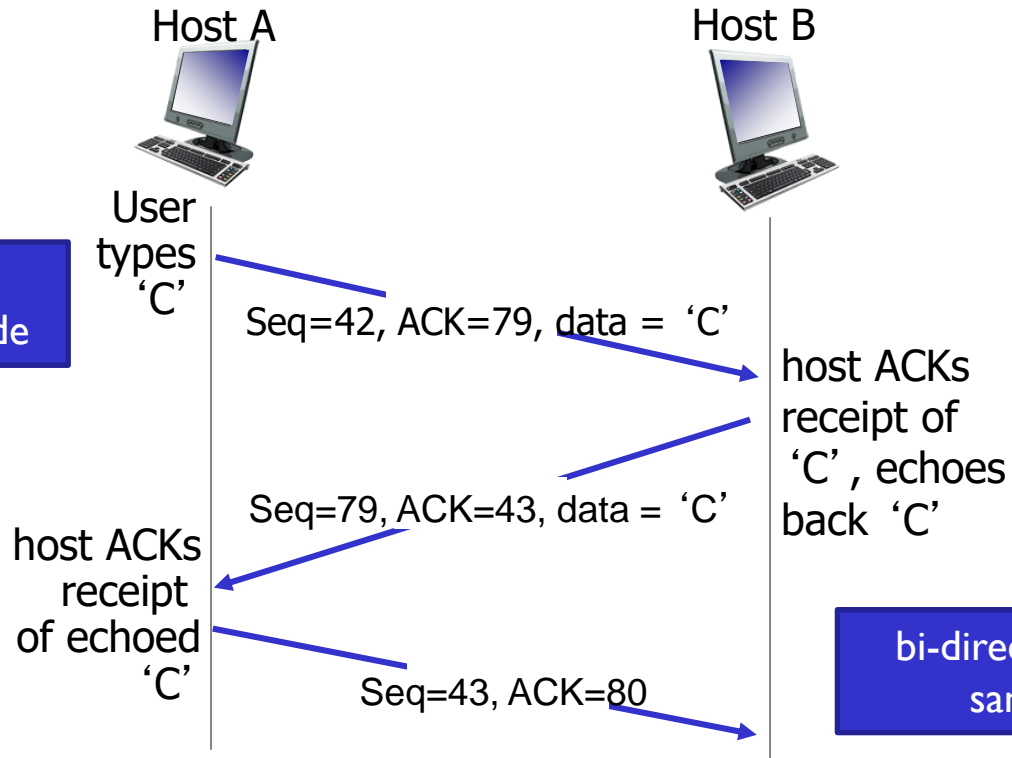
3.7 TCP congestion control

TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

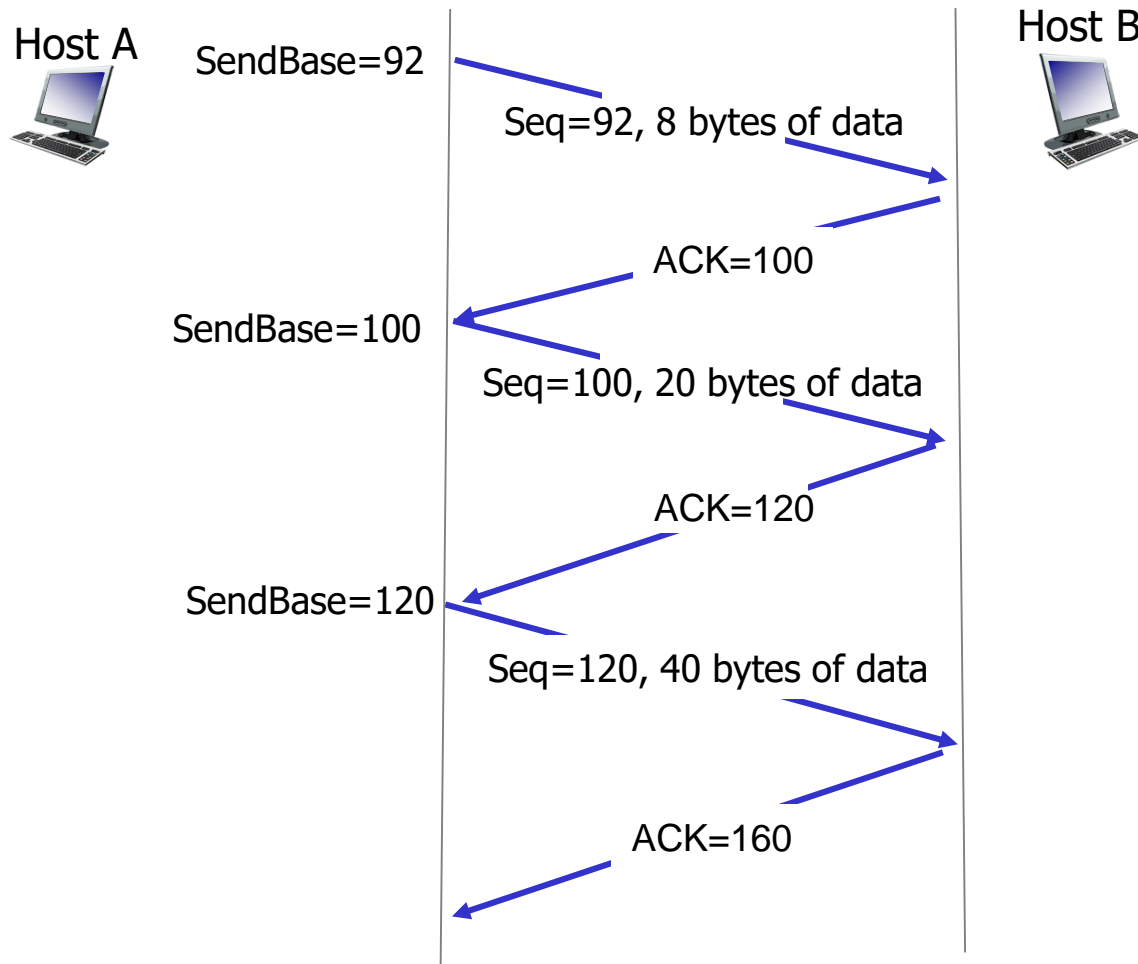
- ❖ **point-to-point:**
 - one sender, one receiver
- ❖ **reliable, in-order *byte stream*:**
 - no “message boundaries”
 - Seq # and Ack # are in unit of byte, or pkt
- ❖ **pipelined:**
 - TCP congestion and flow control set window size
- ❖ **full duplex data:**
 - bi-directional data flow in same connection
 - MSS: maximum segment size
- ❖ **connection-oriented:**
 - handshaking (exchange of control msgs) initializes sender, receiver state before data exchange
- ❖ **flow controlled:**
 - sender will not overwhelm receiver

TCP seq. numbers, ACKs



simple telnet scenario

TCP without retransmission



TCP seq. numbers, ACKs

sequence numbers:

- byte stream “number” of first byte in segment’s data

acknowledgements:

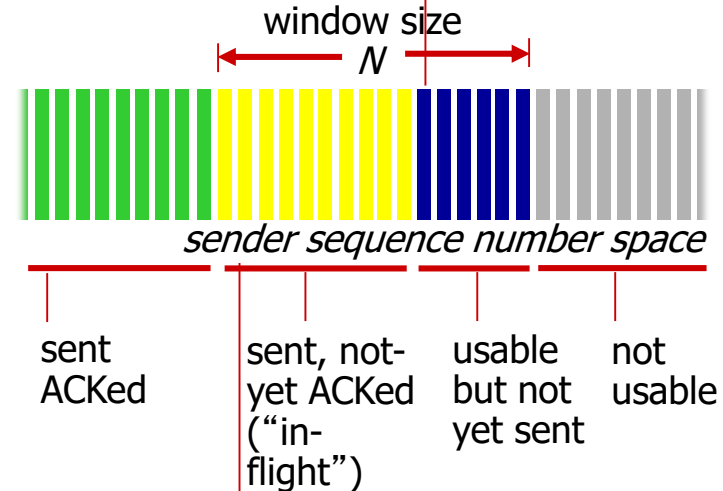
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

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

outgoing segment from sender

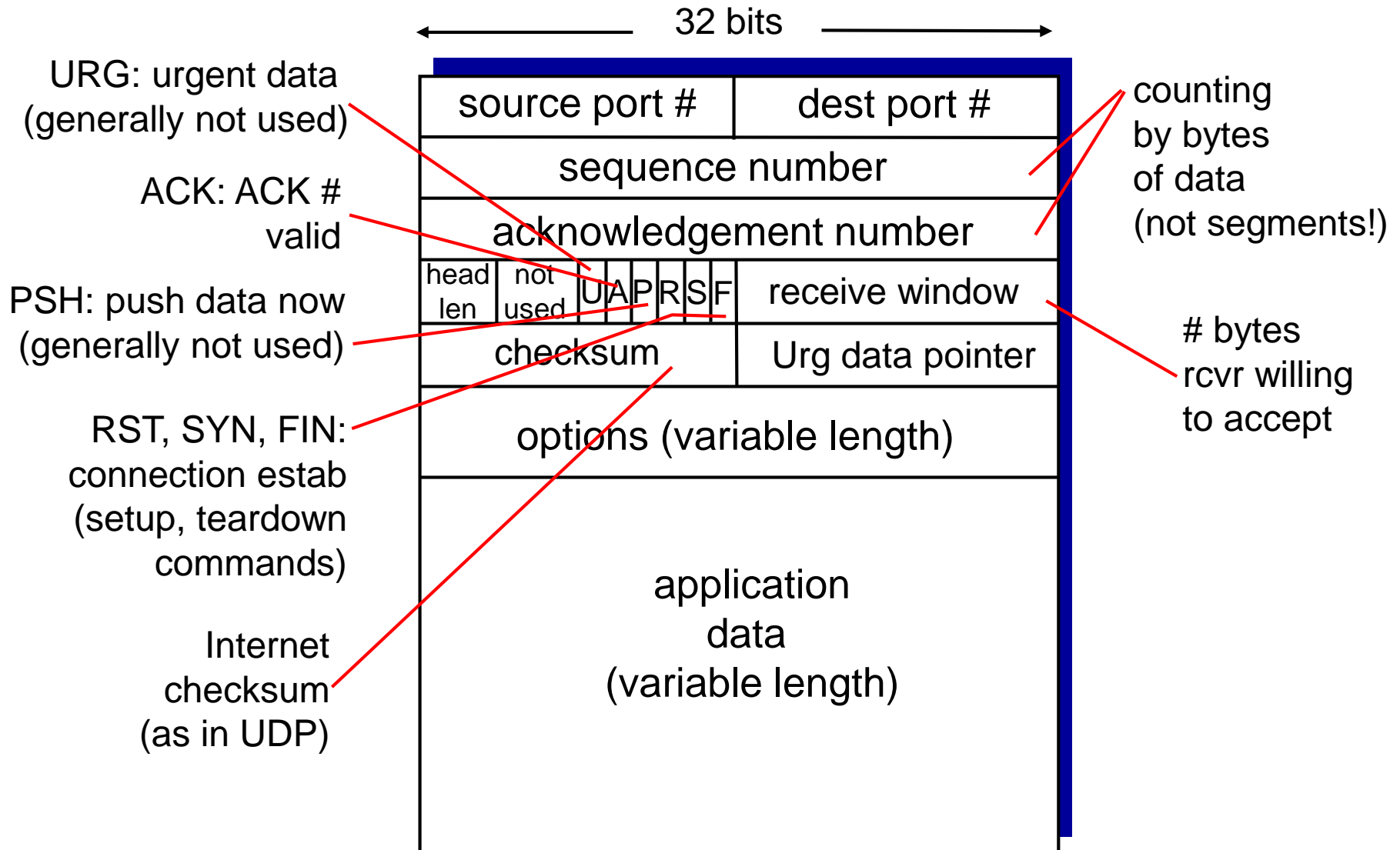
source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer



incoming segment to sender

source port #	dest port #
sequence number	
acknowledgement number	
	A
checksum	urg pointer

TCP segment structure



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

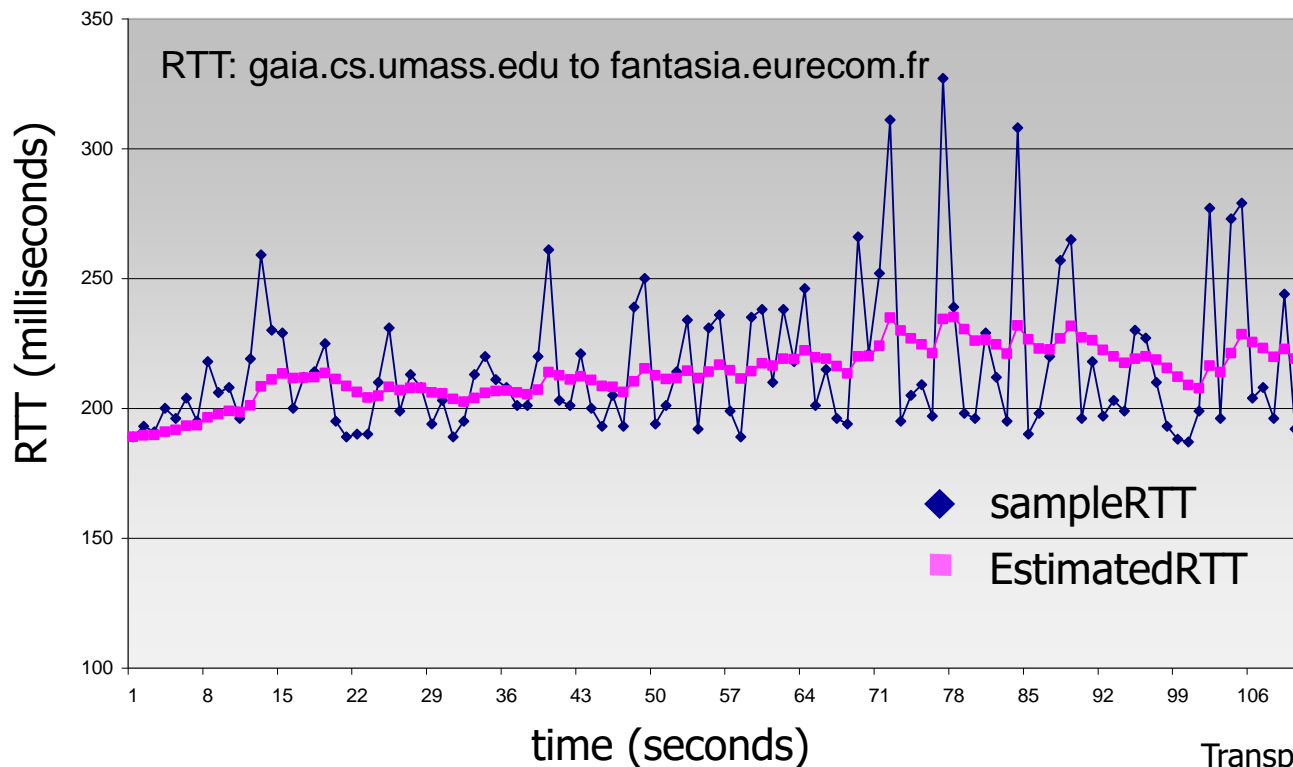
Q: how to estimate RTT?

- ❖ **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

$$\text{EstimatedRTT} = (1 - \alpha) * \text{EstimatedRTT} + \alpha * \text{SampleRTT}$$

- ❖ 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**:

$$\text{DevRTT} = (1-\beta) * \text{DevRTT} + \beta * |\text{SampleRTT} - \text{EstimatedRTT}|$$

(typically, $\beta = 0.25$)

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$



↑
estimated RTT

↑
“safety margin”

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
- Similar with Go-Back-N

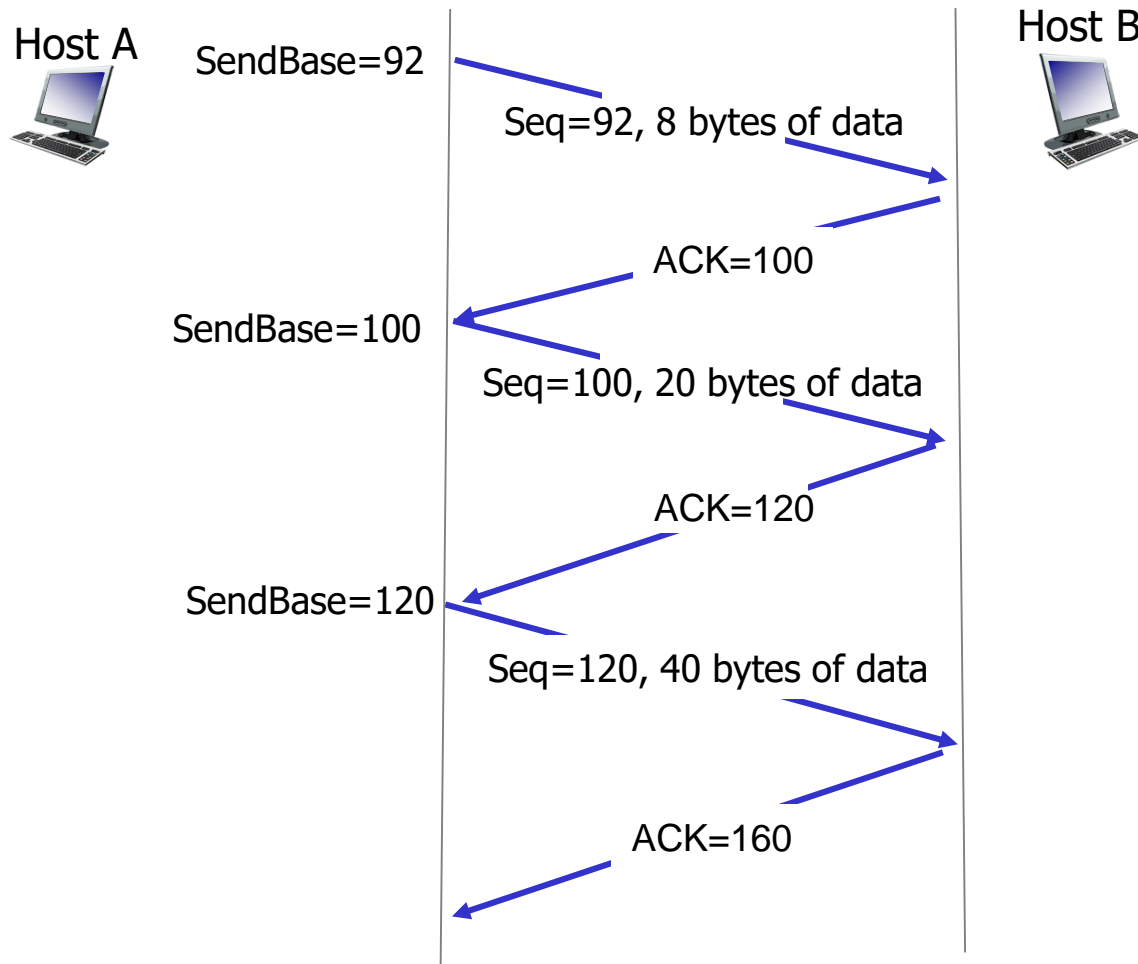
- ❖ retransmissions triggered by:

- timeout events
- duplicate acks

let's initially consider simplified TCP sender:

- ignore duplicate acks
- ignore flow control, congestion control

TCP without retransmission



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`

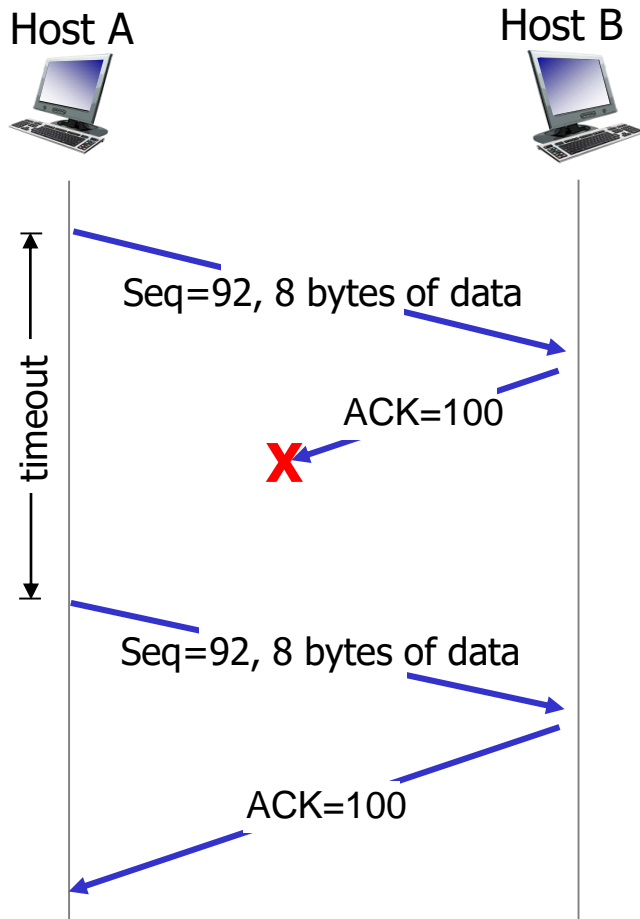
timeout:

- ❖ retransmit segment that caused timeout
- ❖ restart timer

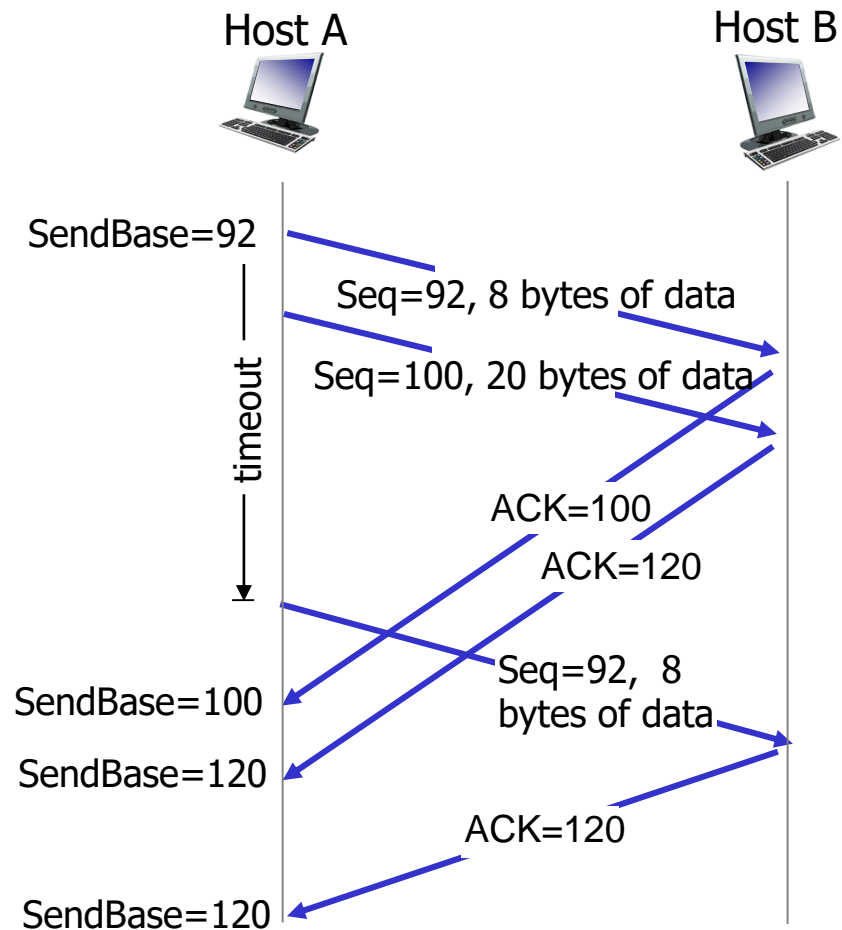
ack rcvd:

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

TCP: retransmission scenarios

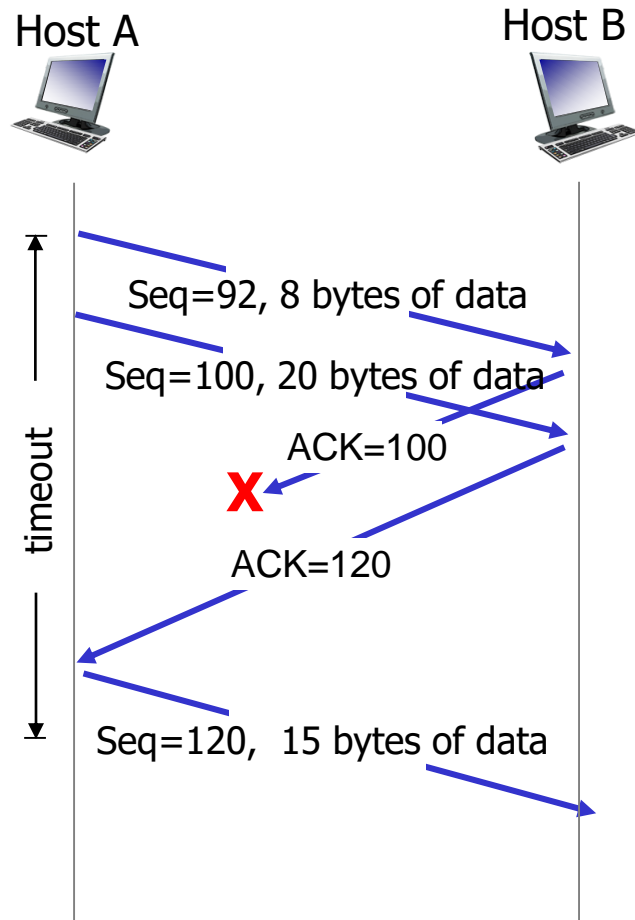


lost ACK scenario



premature timeout

TCP: retransmission scenarios

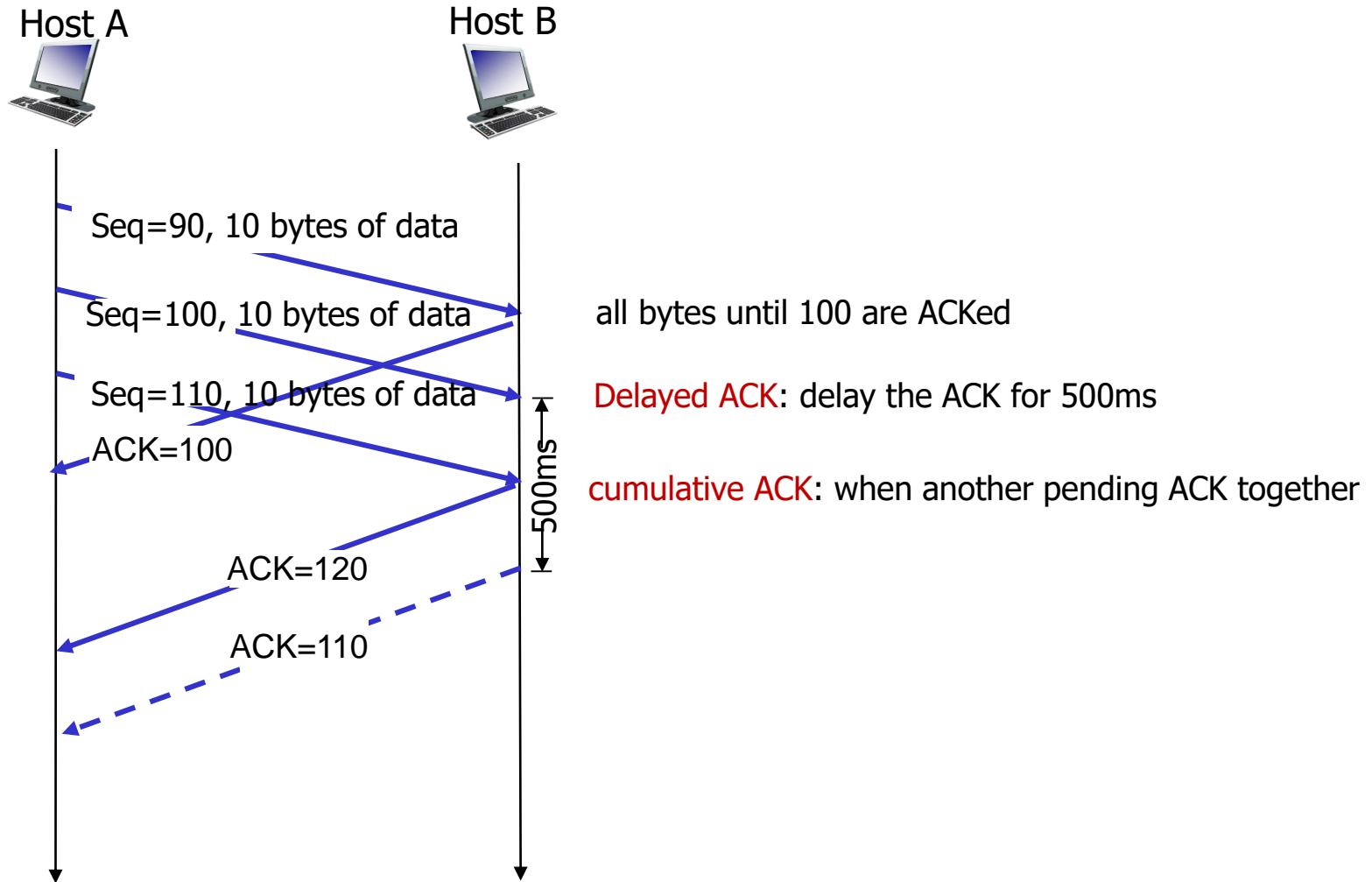


cumulative ACK

TCP receiver [RFC 1122, RFC 2581]

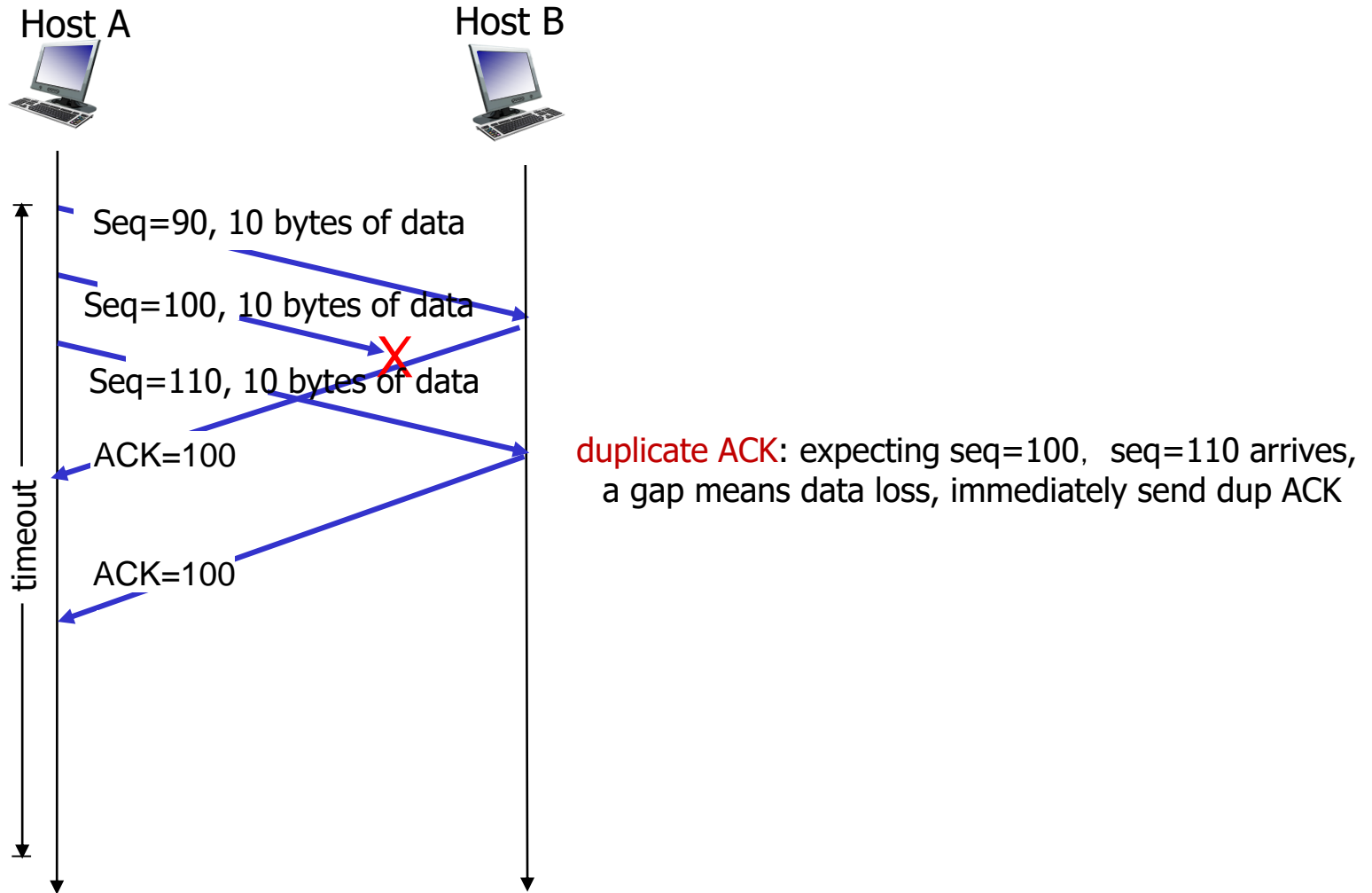
<i>event at receiver</i>	<i>TCP receiver action</i>
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK . Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	cumulative ACK . 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 updated ACK , provided that segment starts at lower end of gap

delayed and cumulative ACK

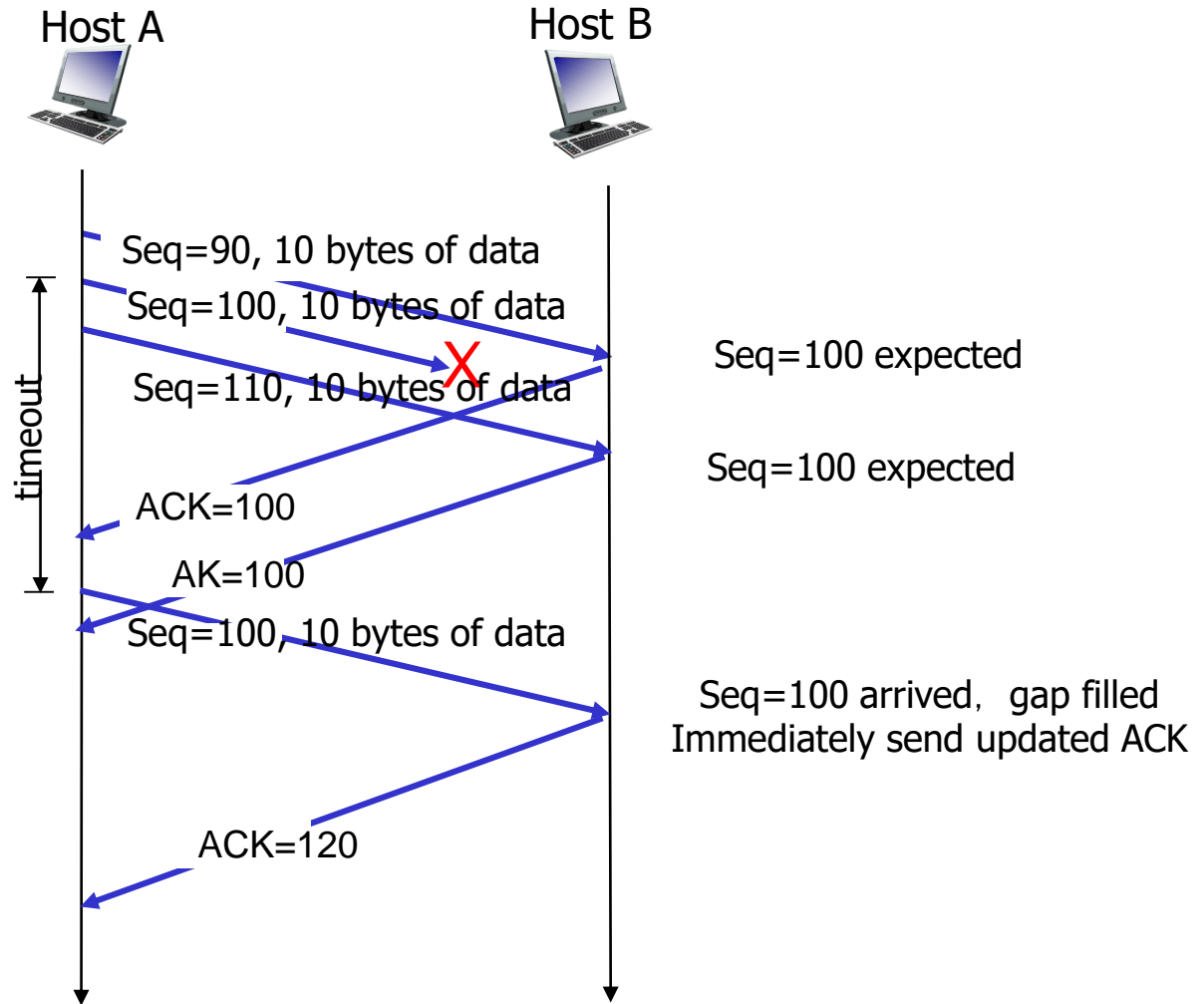


delayed and cumulative ACK

duplicate ACK



Updated 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

