

Chapter 3

Transport Layer

A note on the use of these PowerPoint slides:

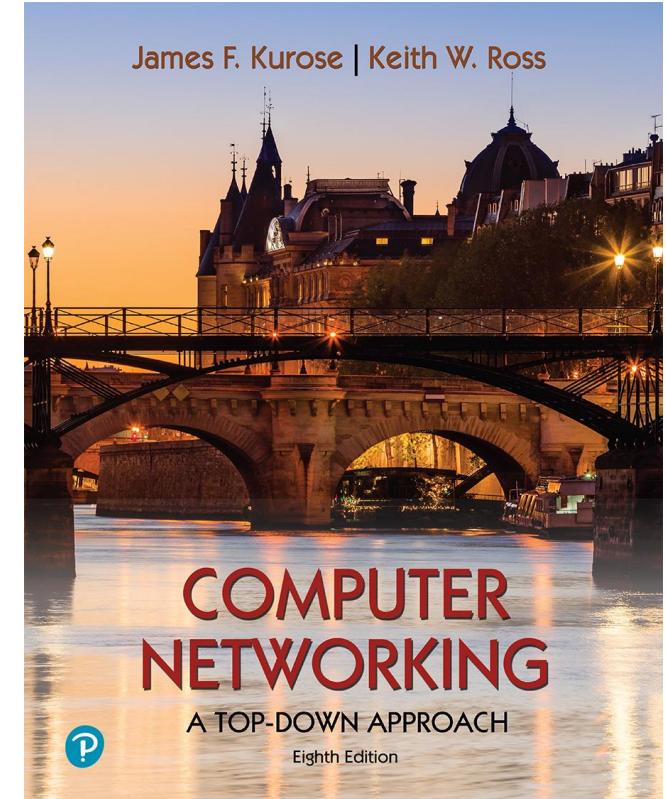
We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use,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

For a revision history, see the slide note for this page.

Thanks and enjoy! JFK/KWR

All material copyright 1996-2020
J.F Kurose and K.W. Ross, All Rights Reserved



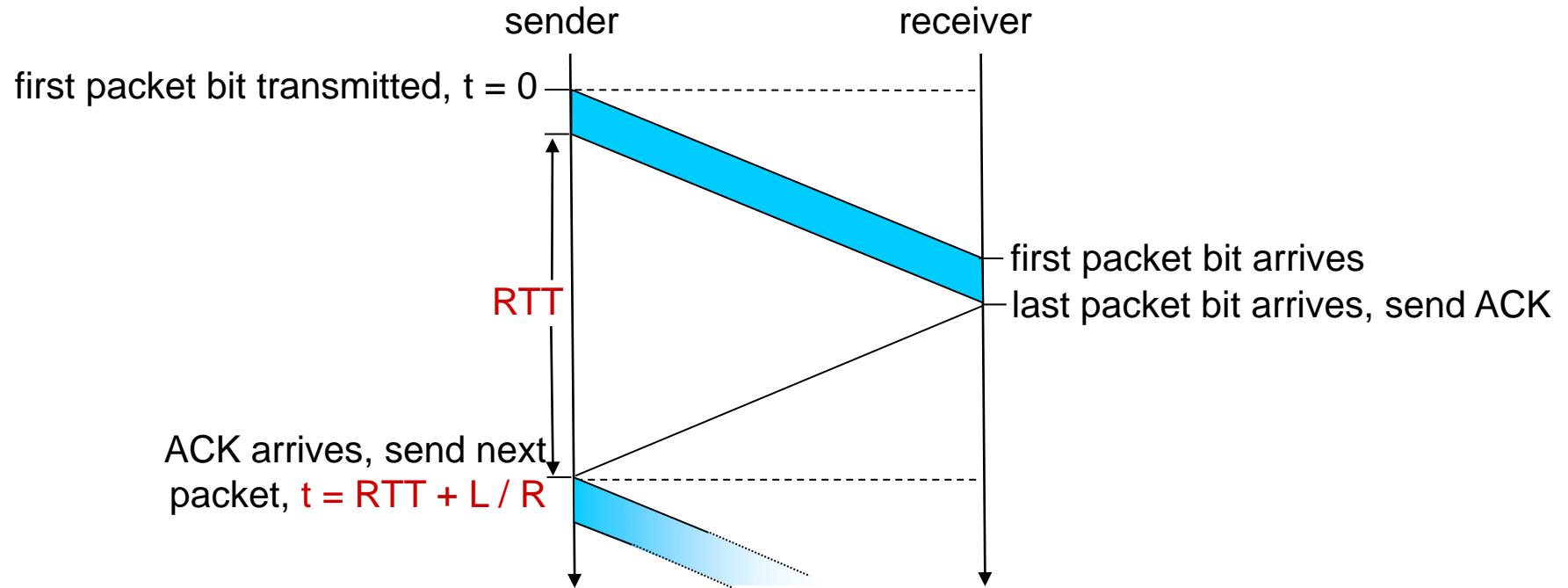
*Computer Networking: A
Top-Down Approach*
8th edition
Jim Kurose, Keith Ross
Pearson, 2020

Performance of rdt3.0 (stop-and-wait)

- U_{sender} : *utilization* – fraction of time sender busy sending
- example: 1 Gbps link, 15 ms prop. delay, 8000 bit packet
 - time to transmit packet into channel:

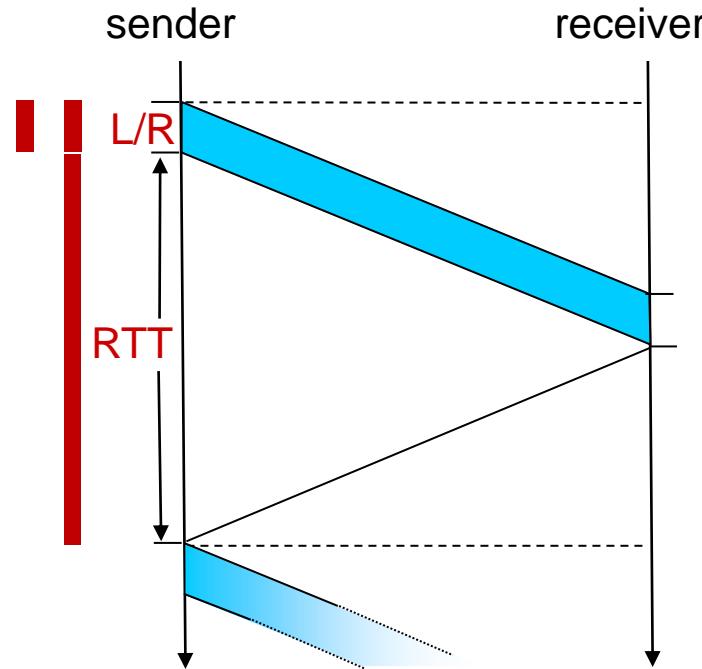
$$D_{\text{trans}} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

rdt3.0: stop-and-wait operation



rdt3.0: stop-and-wait operation

$$\begin{aligned} U_{\text{sender}} &= \frac{L / R}{RTT + L / R} \\ &= \frac{.008}{30.008} \\ &= 0.00027 \end{aligned}$$

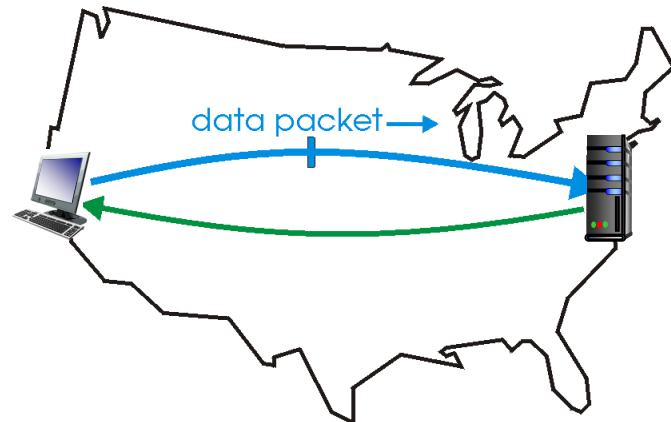


- rdt 3.0 protocol performance stinks!
- Protocol limits performance of underlying infrastructure (channel)

rdt3.0: pipelined protocols operation

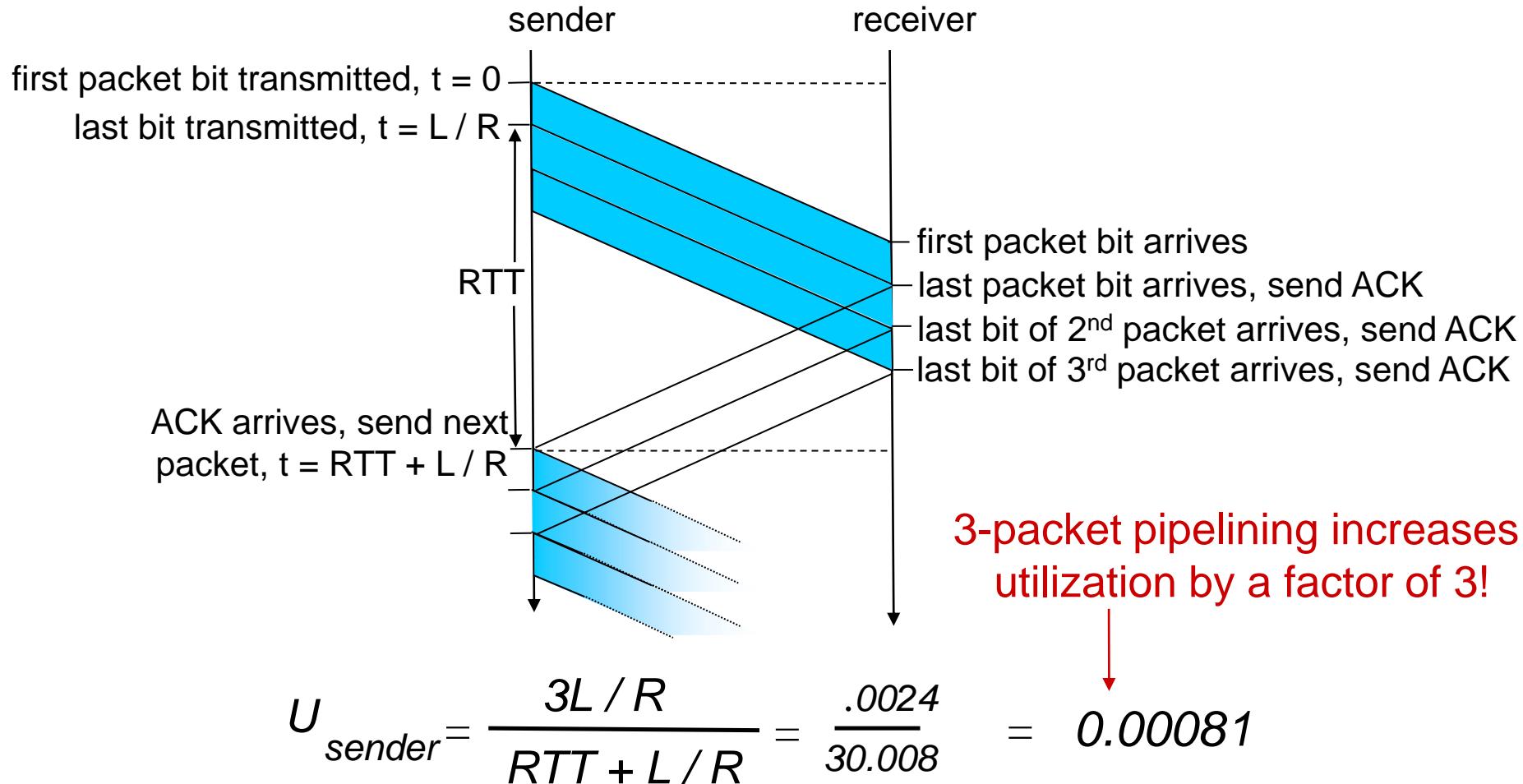
pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

- range of sequence numbers must be increased
- buffering at sender and/or receiver



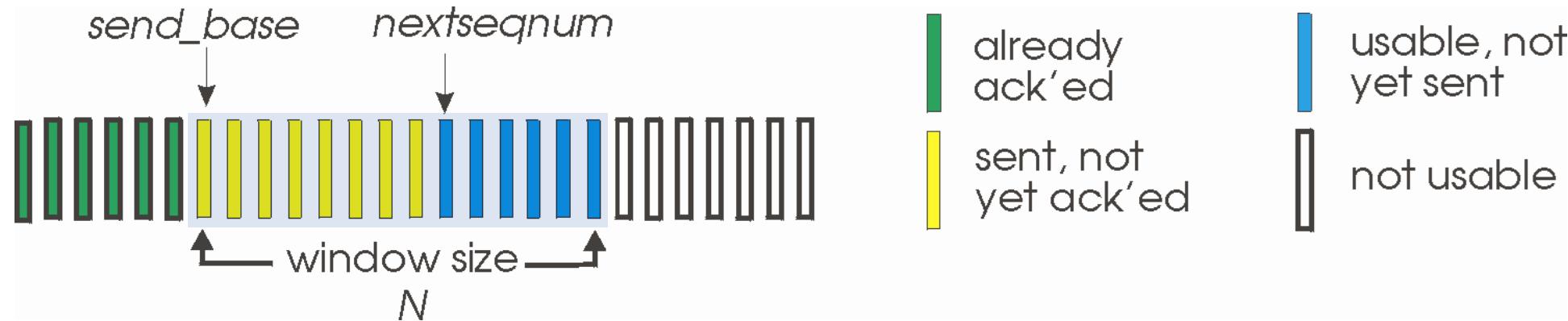
(a) a stop-and-wait protocol in operation

Pipelining: increased utilization



Go-Back-N: sender

- sender: “window” of up to N , consecutive transmitted but unACKed pkts
 - k -bit seq # in pkt header

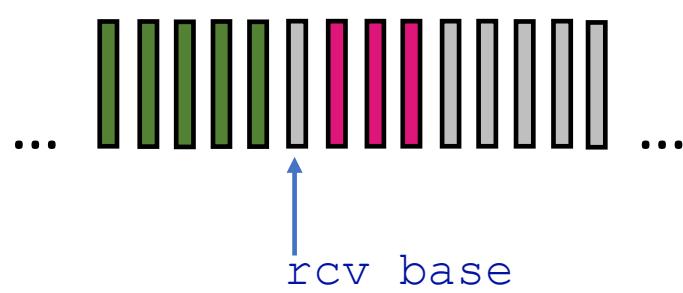


- *cumulative ACK*: $\text{ACK}(n)$: ACKs all packets up to, including seq # n
 - on receiving $\text{ACK}(n)$: move window forward to begin at $n+1$
- timer for oldest in-flight packet
- $\text{timeout}(n)$: retransmit packet n and all higher seq # packets in window

Go-Back-N: receiver

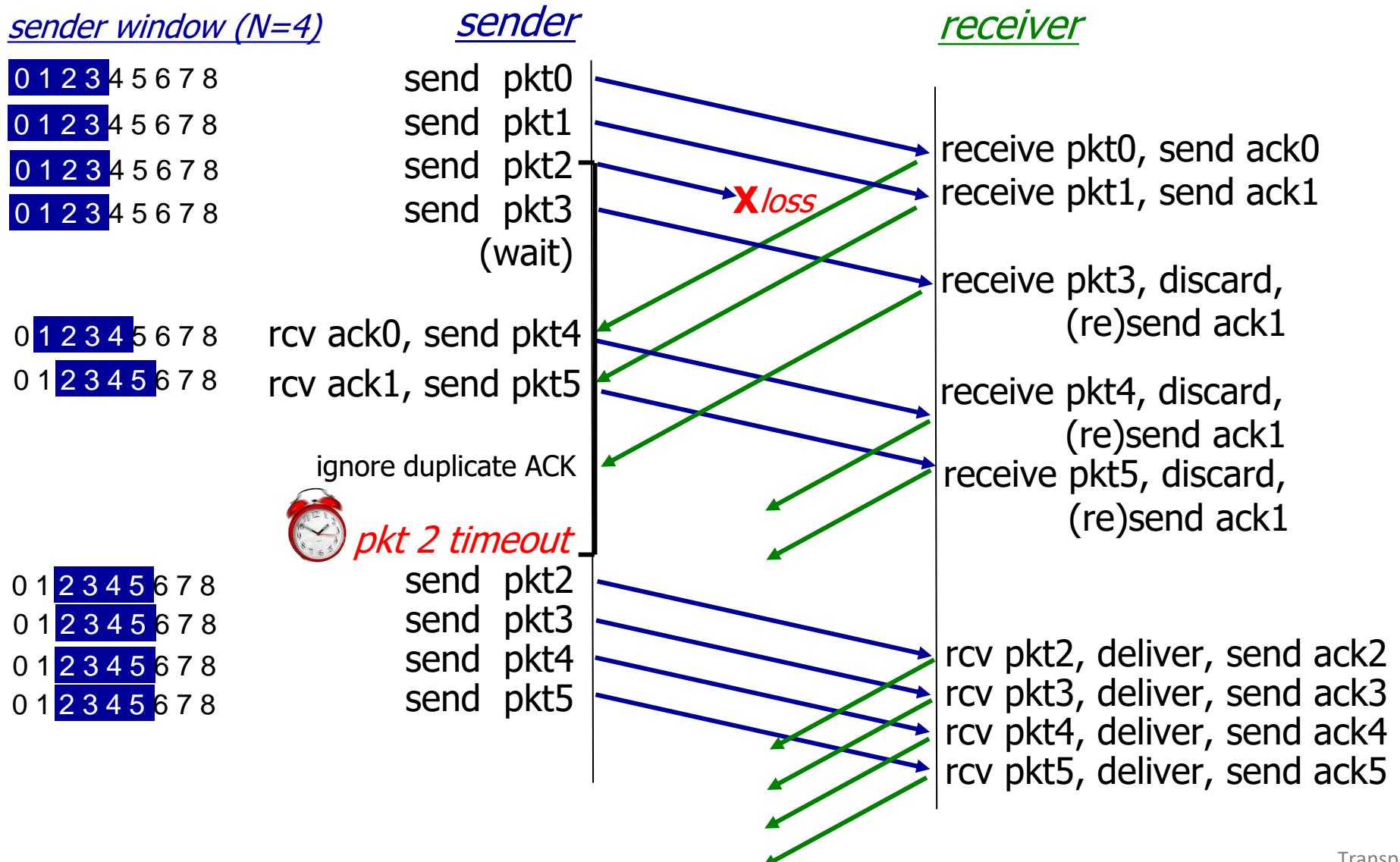
- ACK-only: always send ACK for correctly-received packet so far, with highest *in-order* seq #
 - may generate duplicate ACKs
 - need only remember `rcv_base`
- on receipt of out-of-order packet:
 - can discard (don't buffer) or buffer: an implementation decision
 - re-ACK pkt with highest in-order seq #

Receiver view of sequence number space:



	received and ACKed
	Out-of-order: received but not ACKed
	Not received

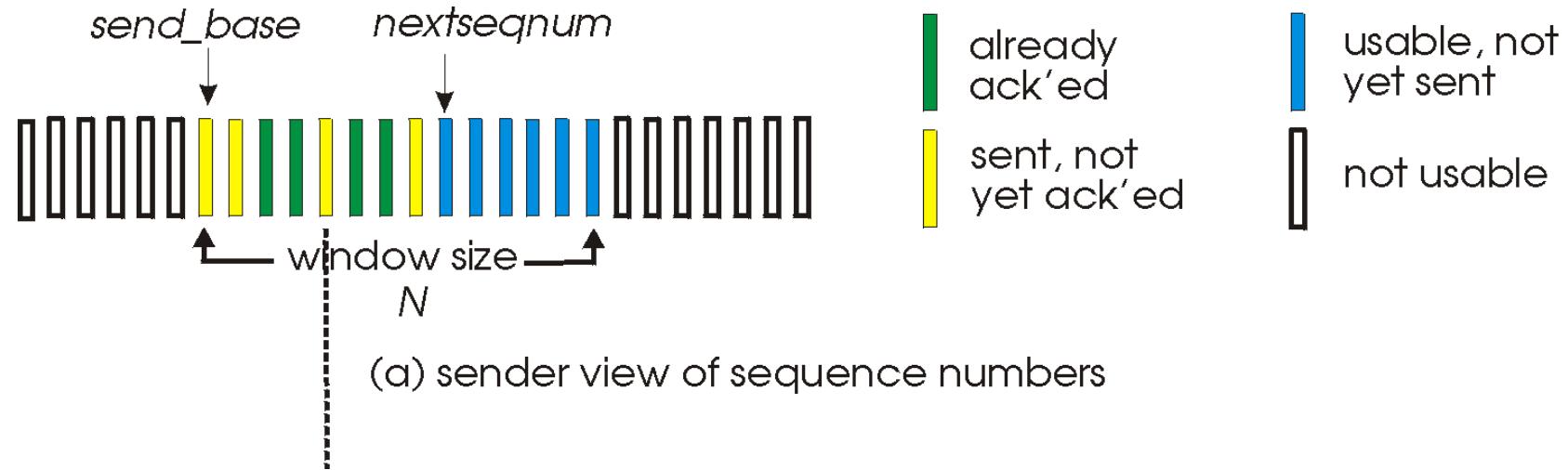
Go-Back-N in action



Selective repeat

- receiver *individually* acknowledges all correctly received packets
 - buffers packets, as needed, for eventual in-order delivery to upper layer
- sender times-out/retransmits individually for unACKed packets
 - sender maintains timer for each unACKed pkt
- sender window
 - N consecutive seq #s
 - limits seq #s of sent, unACKed packets

Selective repeat: sender, receiver windows



Selective repeat: sender and receiver

sender

data from above:

- if next available seq # in window, send packet

timeout(n):

- resend packet n , restart timer

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

- mark packet n as received
- if n smallest unACKed packet, advance window base to next unACKed seq #

receiver

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

- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet

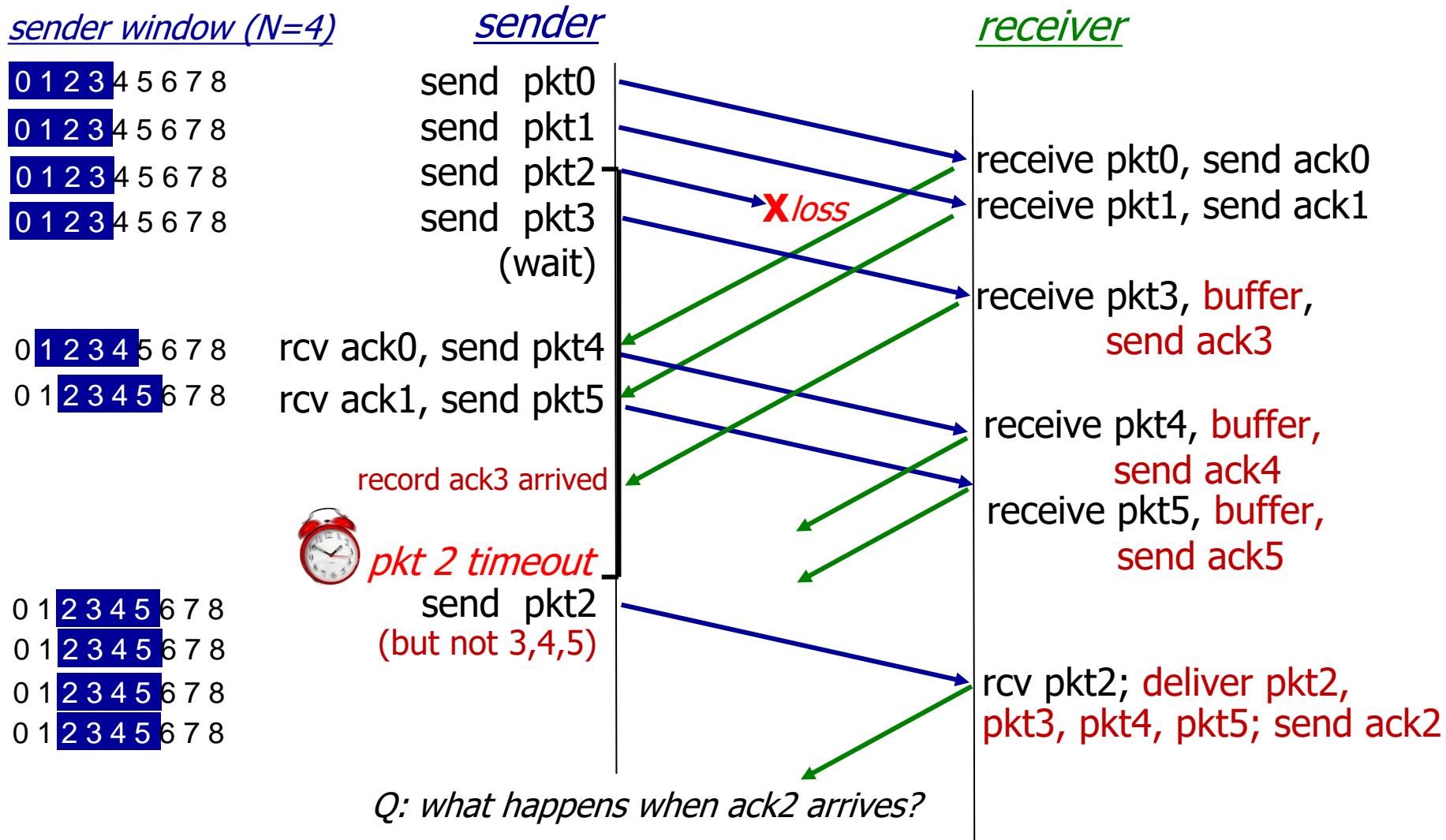
packet n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

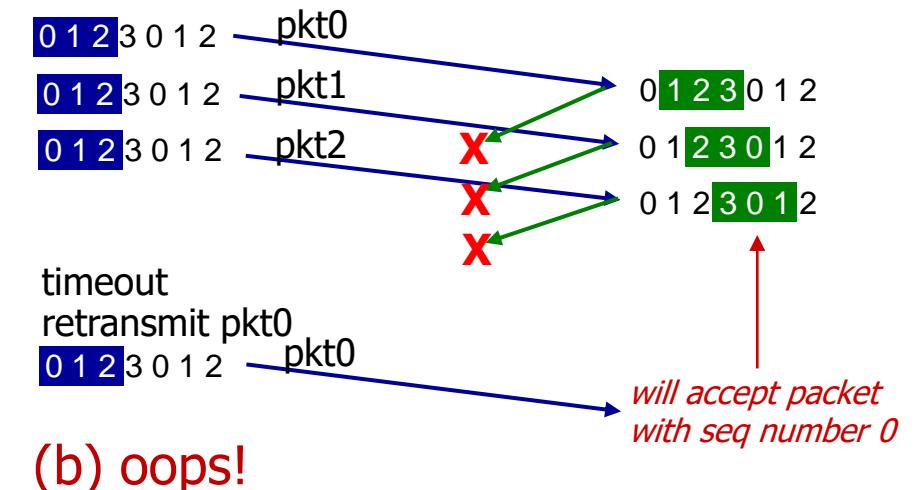
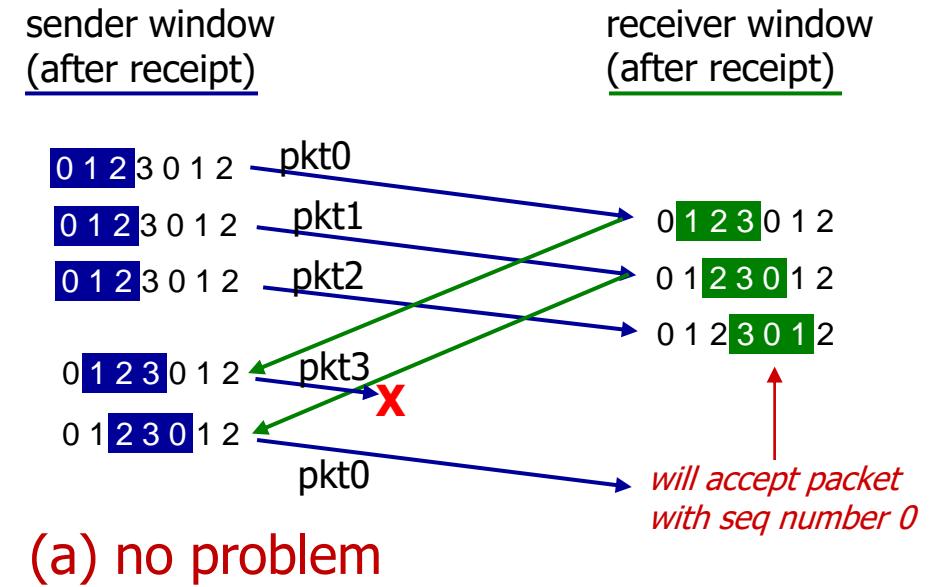
Selective Repeat in action



Selective repeat: a dilemma!

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3



Selective repeat: a dilemma!

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3

Q: what relationship is needed between sequence # size and window size to avoid problem in scenario (b)?

sender window
(after receipt)

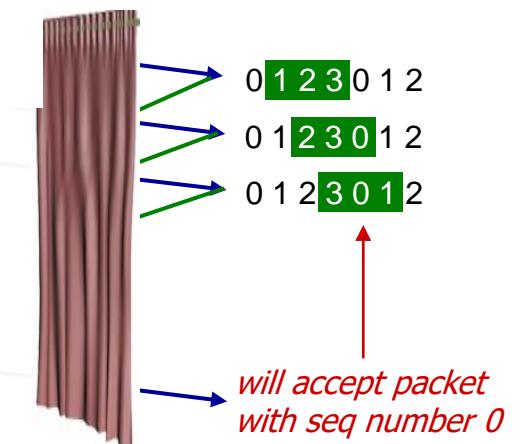
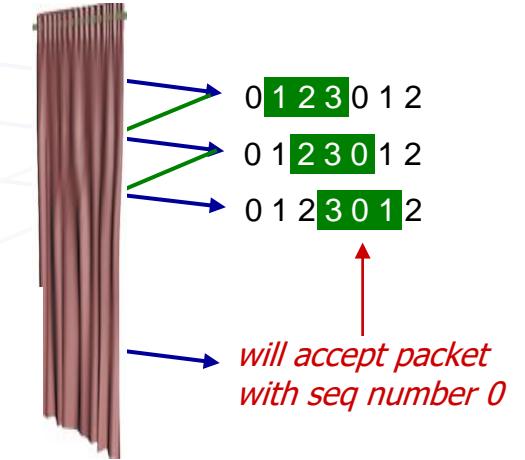
0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2

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

0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2
timeout
retransmit pkt0
0 1 2 3 0 1 2

(b) oops!

receiver window
(after receipt)



Animation Link

https://www2.tkn.tu-berlin.de/teaching/rn/animations/gbn_sr/

Review

- Pipelined Protocol
 - GBN
 - SR

GBN: sender extended FSM

First, check if window is full,
if not send and update variables

N is the window size
base=oldest unack'd
nextseqnum=seqno of next pckt

Why wait?

rdt_rcv(rcvpkt)
&& corrupt(rcvpkt)

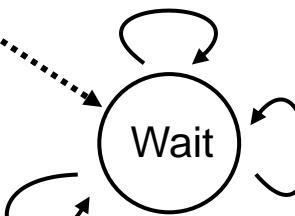
Cumulative ACKs, receiver only
ACKs when packets delivered
in order.

Here we need to update base
and stop timer if nothing to send
or restart if some pkts left

rdt_send(data)

```
if (nextseqnum < base+N) {  
    sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)  
    udt_send(sndpkt[nextseqnum])  
    if (base == nextseqnum)  
        start_timer  
    nextseqnum++  
}  
else  
    refuse_data(data)
```

Timer associated with oldest unacked
base == nextseqnum, when?

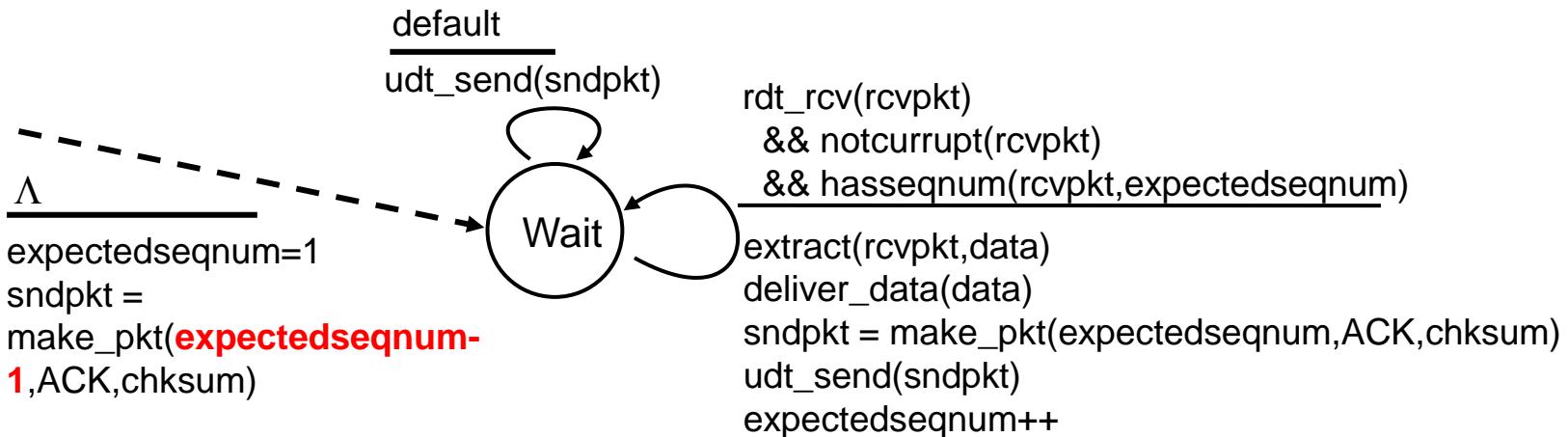


timeout

```
start_timer  
udt_send(sndpkt[base])  
udt_send(sndpkt[base+1])  
...  
udt_send(sndpkt[nextseqnum-1])
```

On timeout resend all
unack'd packets

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 #

Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- **Connection-oriented transport: TCP**
 - segment structure
 - reliable data transfer
 - flow control
 - connection management
- Principles of congestion control
- TCP congestion control



TCP: overview

RFCs: 793, 1122, 2018, 5681, 7323

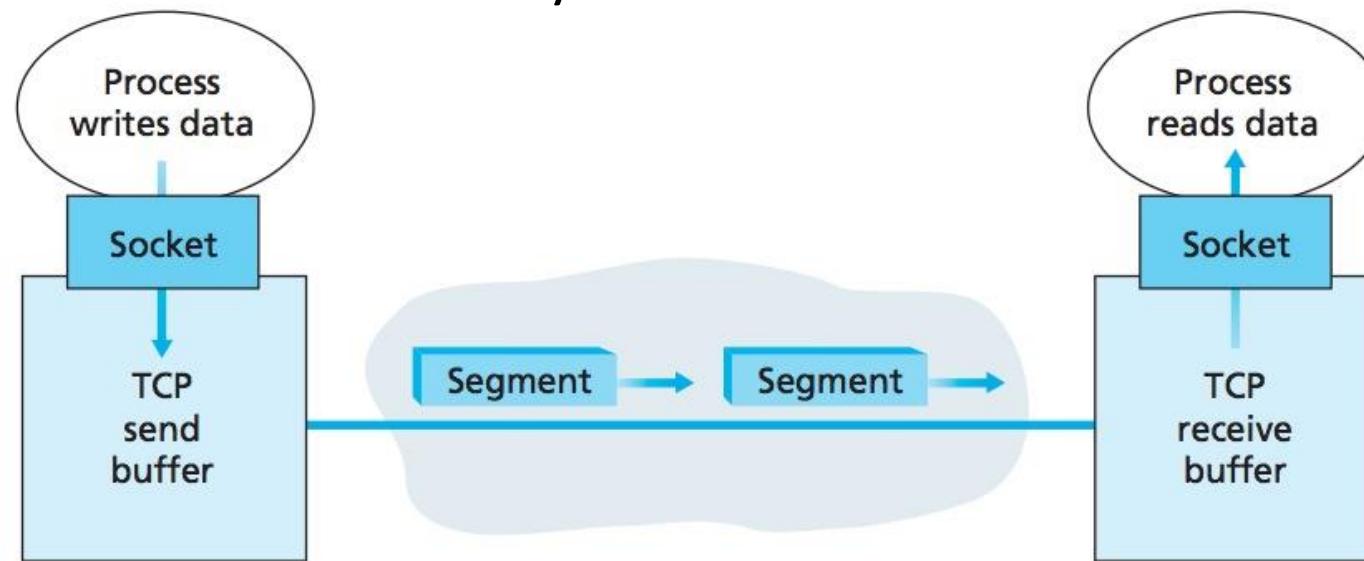
- point-to-point:
 - one sender, one receiver
- reliable, in-order *byte steam*:
 - no “message boundaries”
- full duplex data:
 - bi-directional data flow in same connection
 - MSS: maximum segment size
- cumulative ACKs
- pipelining:
 - TCP congestion and flow control set window size
- connection-oriented: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

TCP connection

- TCP connection consists of
 - Buffers and variables
 - a socket connection to a process in one host, and another set of buffers, variables, and a socket connection to a process in another host.
- Nothing is stored in the network elements (routers, switches, and repeaters) between the hosts.

TCP Send/Receive buffers

- The client process passes a stream of data through the socket.
- Once the data passes through the Socket, the data is in the hands of TCP running in the client.
- TCP directs this data to the connection's send buffer, which is one of the buffers that is set aside during the initial three-way handshake.
- From time to time, TCP will grab chunks of data from the send buffer and pass the data to the network layer.



TCP segment structure

ACK: seq # of next expected byte; A bit: this is an ACK

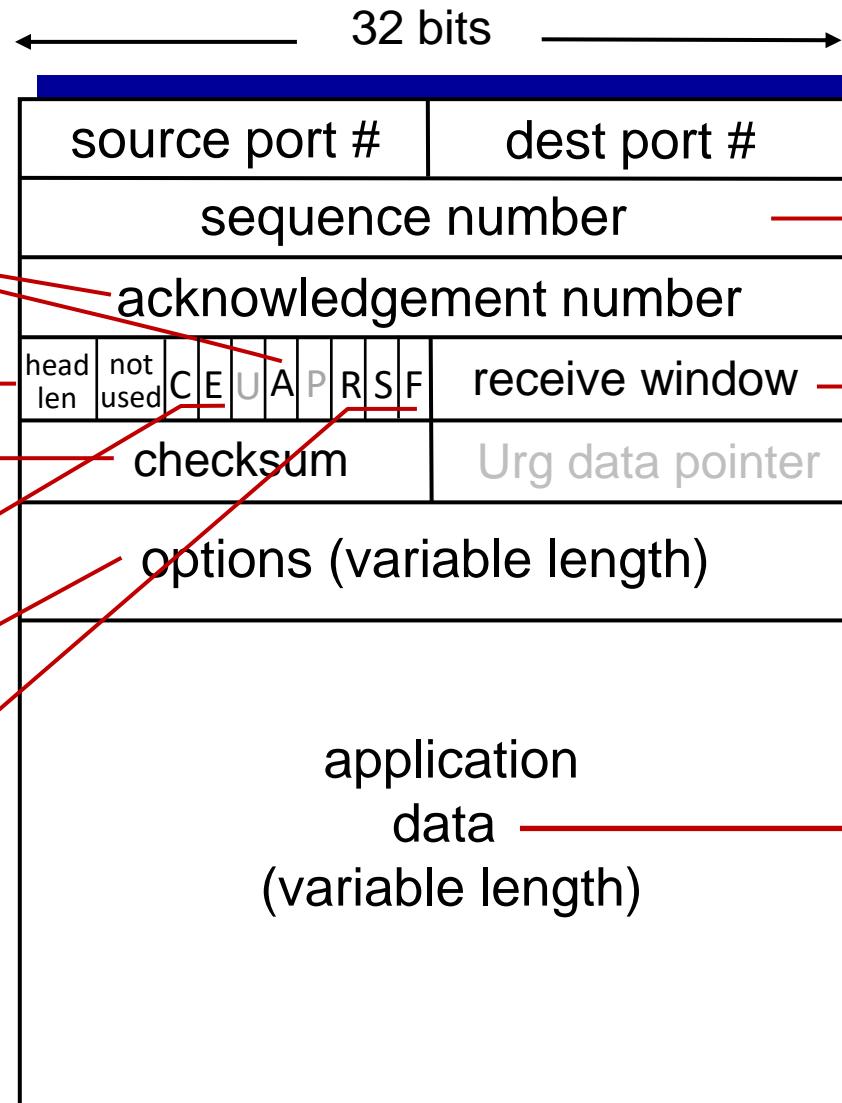
length (of TCP header)

Internet checksum

C, E: congestion notification

TCP options

RST, SYN, FIN: connection management



segment seq #: counting bytes of data into bytestream (not segments!)

flow control: # bytes receiver willing to accept

data sent by application into TCP socket

TCP sequence numbers, ACKs

Sequence numbers:

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

Acknowledgements:

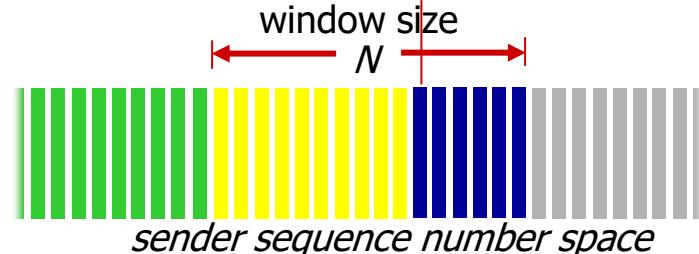
- seq # of next byte expected from other side
- cumulative ACK

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



sent
ACKed

sent, not-
yet ACKed
(“in-flight”)

usable
but not
yet sent

not
usable

outgoing segment from receiver

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

A

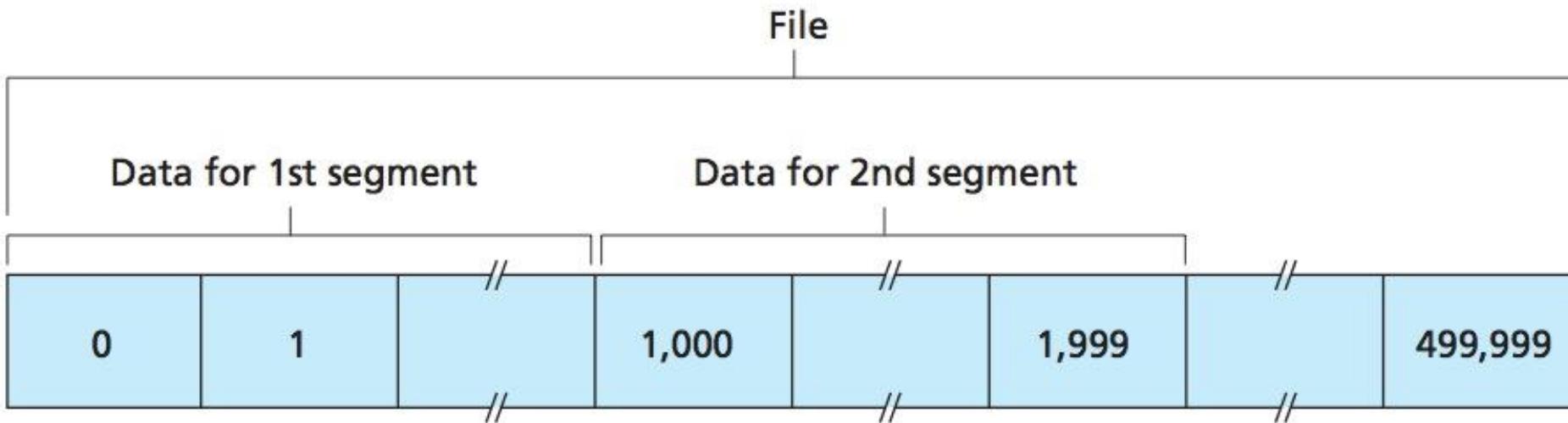
Sequence Numbers

- TCP views data as an unstructured, but ordered, stream of bytes.
- Sequence numbers are over the stream of transmitted bytes and *not* over the series of transmitted segments.
- The **sequence number for a segment** is therefore the byte-stream number of the first byte in the segment.

An example

- A process in Host A wants to send a stream of data to a process in Host B.
- The TCP in Host A will implicitly number each byte in the data stream.
 - For a file consisting of 500,000 bytes,
 - MSS being 1,000 bytes, and that the first byte of the data stream is numbered 0.
 - TCP constructs 500 segments out of the data stream.

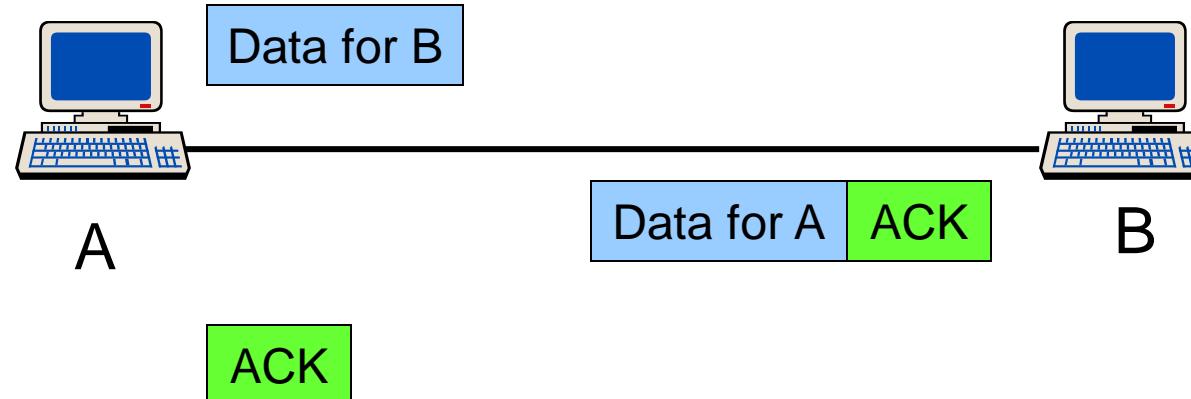
TCP Sequence Numbers



The first segment gets assigned sequence number 0, the second segment gets assigned sequence number 1,000, the third segment gets assigned sequence number 2,000, and so on.

Full duplex

- TCP is full-duplex, so that Host A may be receiving data from Host B while it sends data to Host B (as part of the same TCP connection).



Acknowledgment numbers

- A sends a segment to B, what would B put in the Ack sequence number in the next segment it sends to A?
 - *next byte Host B is expecting from Host A.*

- A sends a segment to B, what would A put in the Ack sequence number in the next segment it sends to B?
 - *next byte Host A is expecting from Host B.*

An example

- Host A has received all bytes numbered 0 through 535 from B.
- Host A is waiting for byte 536 and all the subsequent bytes in Host B's data stream.
- What does Host A put in the acknowledgment number field of the next segment it sends to B?

Host A puts 536 in the acknowledgment number field of the segment it sends to B

Another example

- Host A has received all bytes numbered 0 through 535 from B.
- Host A has also received another segment containing bytes 900 through 1,000
 - For some reason Host A has not yet received bytes 536 through 899.
- What does Host A put in the acknowledgment number field of the next segment it sends to B?

A's next segment to B will contain 536 in the acknowledgment number field

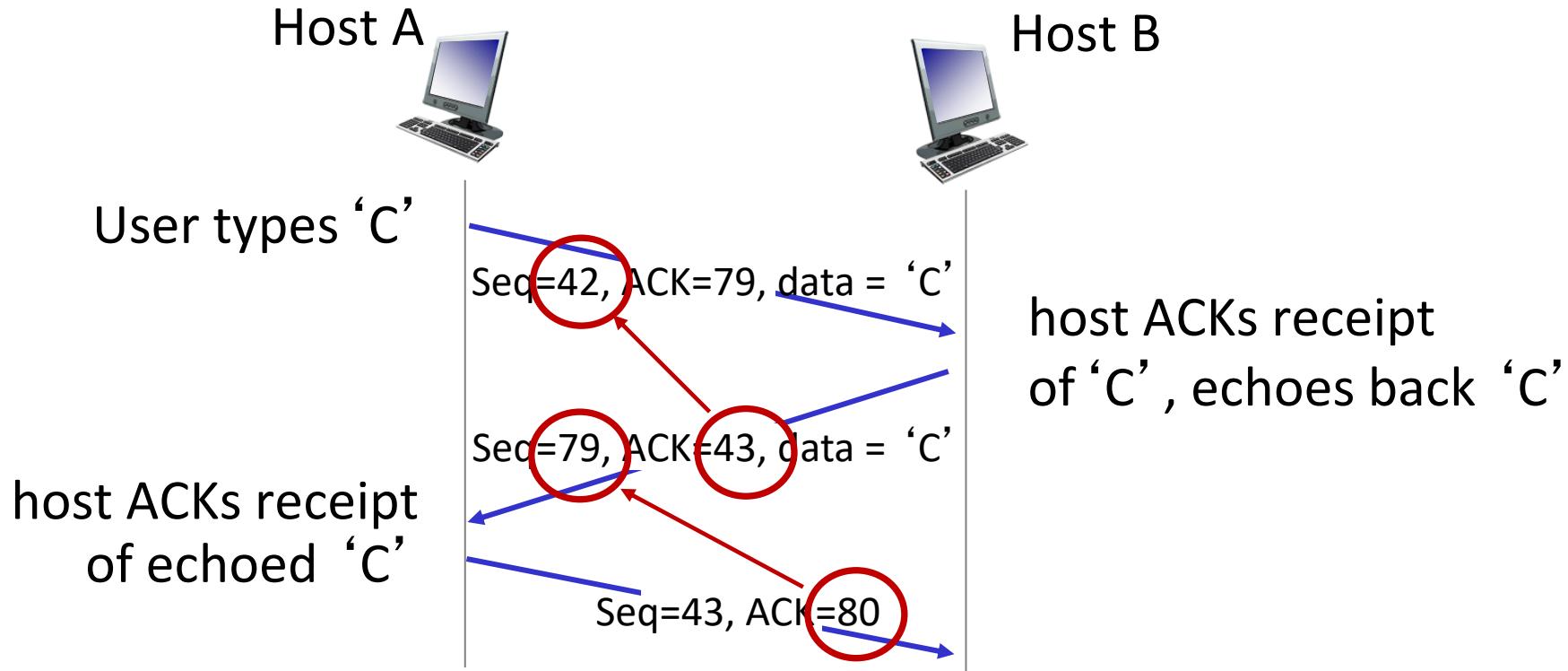
Out of order segments

- TCP only acknowledges bytes up to the first missing byte in the stream, TCP is said to provide **cumulative acknowledgments**.
- For out of order segments it can either discard them or buffer them (the approach actually taken in practice)

The seg/ack numbers base

- we assumed that the initial sequence number was zero.
- In truth, both sides of a TCP connection randomly choose an initial sequence number.
- Why?

TCP sequence 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

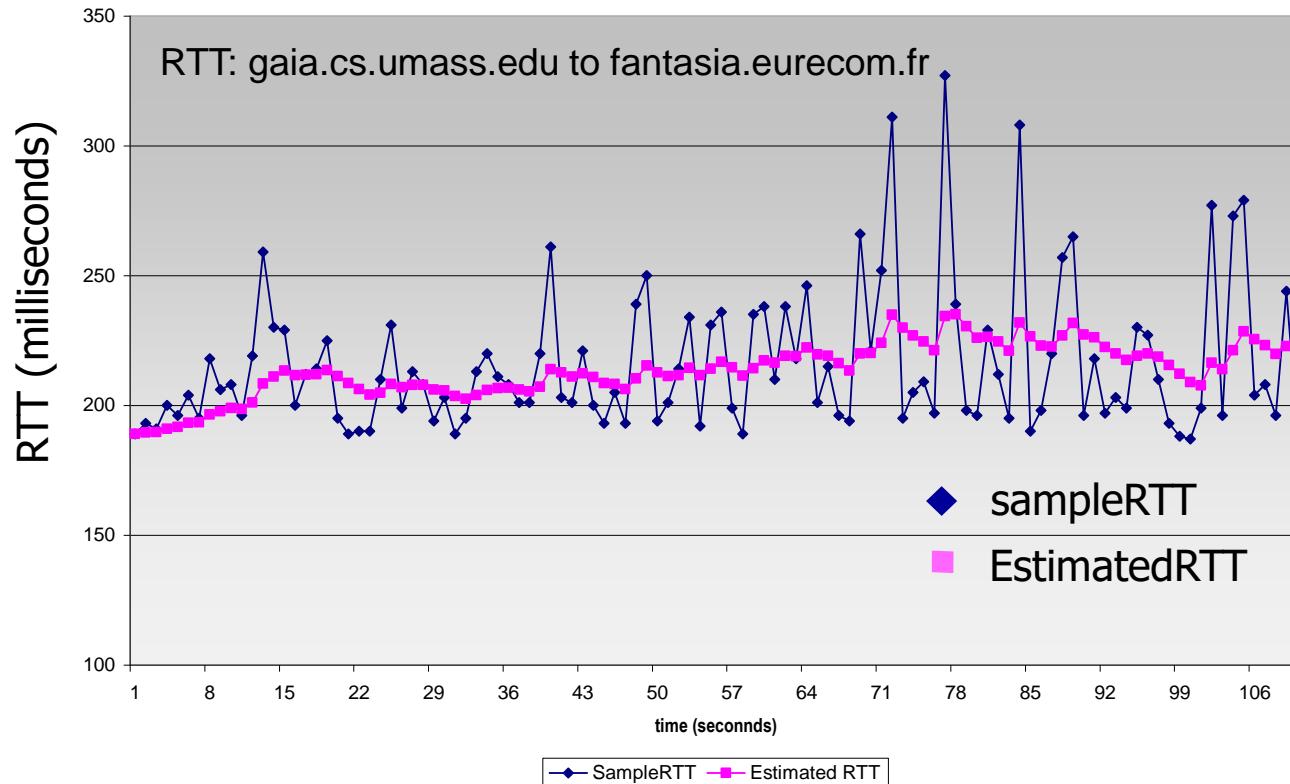
Q: how to estimate RTT?

- **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

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

- ❖ 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”

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

TCP Sender (simplified)

event: data received from application

- 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

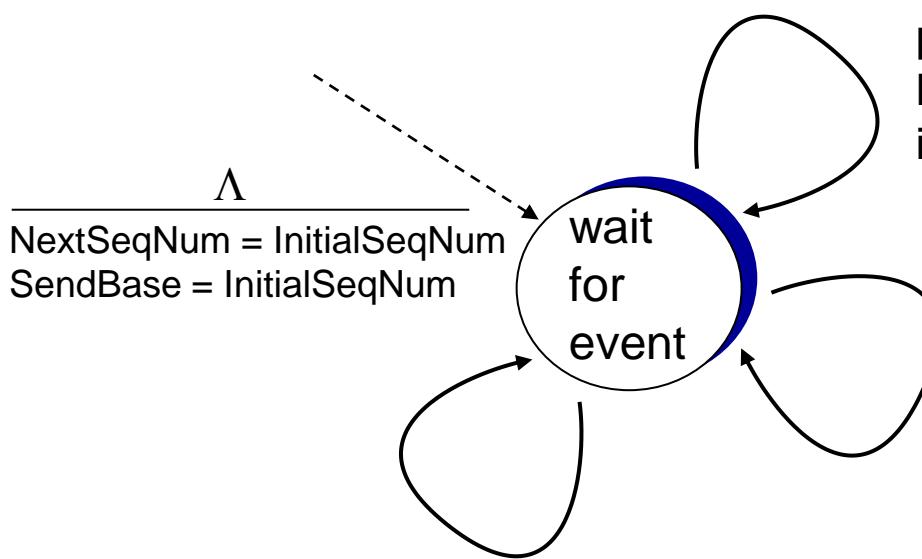
event: timeout

- retransmit segment that caused timeout
- restart timer

event: ACK received

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

TCP sender (simplified)



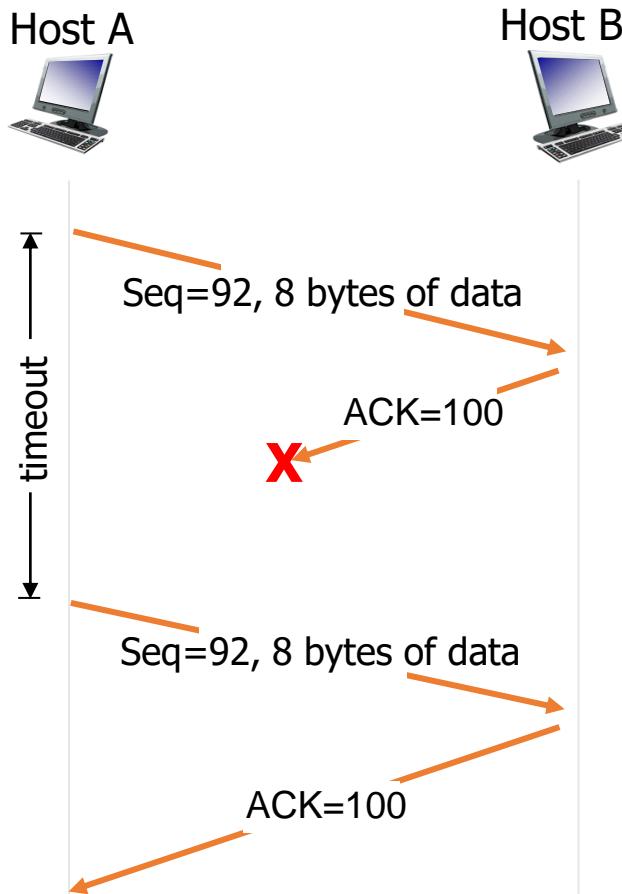
data received from application above
create segment, seq. #: NextSeqNum
pass segment to IP (i.e., “send”)
 $\text{NextSeqNum} = \text{NextSeqNum} + \text{length(data)}$
if (timer currently not running)
start timer

timeout
retransmit not-yet-acked segment
with smallest seq. #
start timer

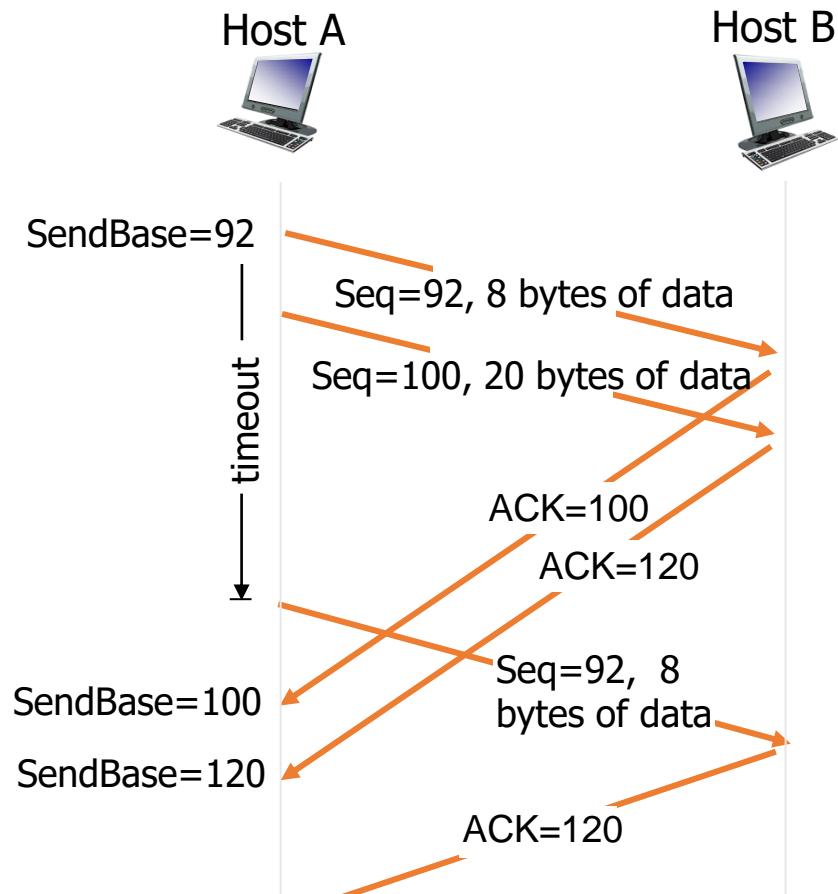
ACK received, with ACK field value y

```
if (y > SendBase) {
    SendBase = y
    /* SendBase-1: last cumulatively ACKed byte */
    if (there are currently not-yet-acked segments)
        start timer
}
```

TCP: retransmission scenarios

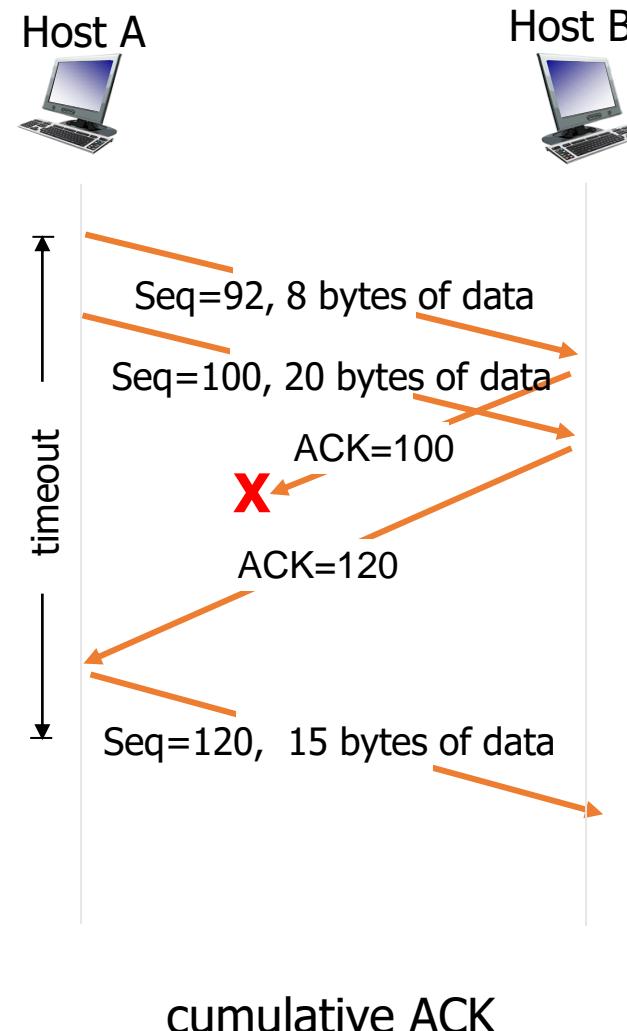


lost ACK scenario



premature timeout

TCP: retransmission scenarios



Modified TCP – Doubling Timeout Interval

- Whenever the timeout event occurs it sets the next timeout interval to twice the previous value,
 - rather than deriving it from the last EstimatedRTT and DevRTT
 - However, in case of other two events (app data received or ACK received), it is derived.
- It also provides a limited form of congestion control.
 - Timer expiration due to congestion
 - It is better to wait rather than keep sending

Modified TCP – Fast Retransmit

- Timeout-triggered retransmissions can take long
- Duplicate ACKs can help sender to detect packet loss
- A **duplicate ACK** is an ACK that reacknowledges a segment for which the sender has already received an earlier acknowledgment.
- When it is sent/received?

TCP fast retransmit

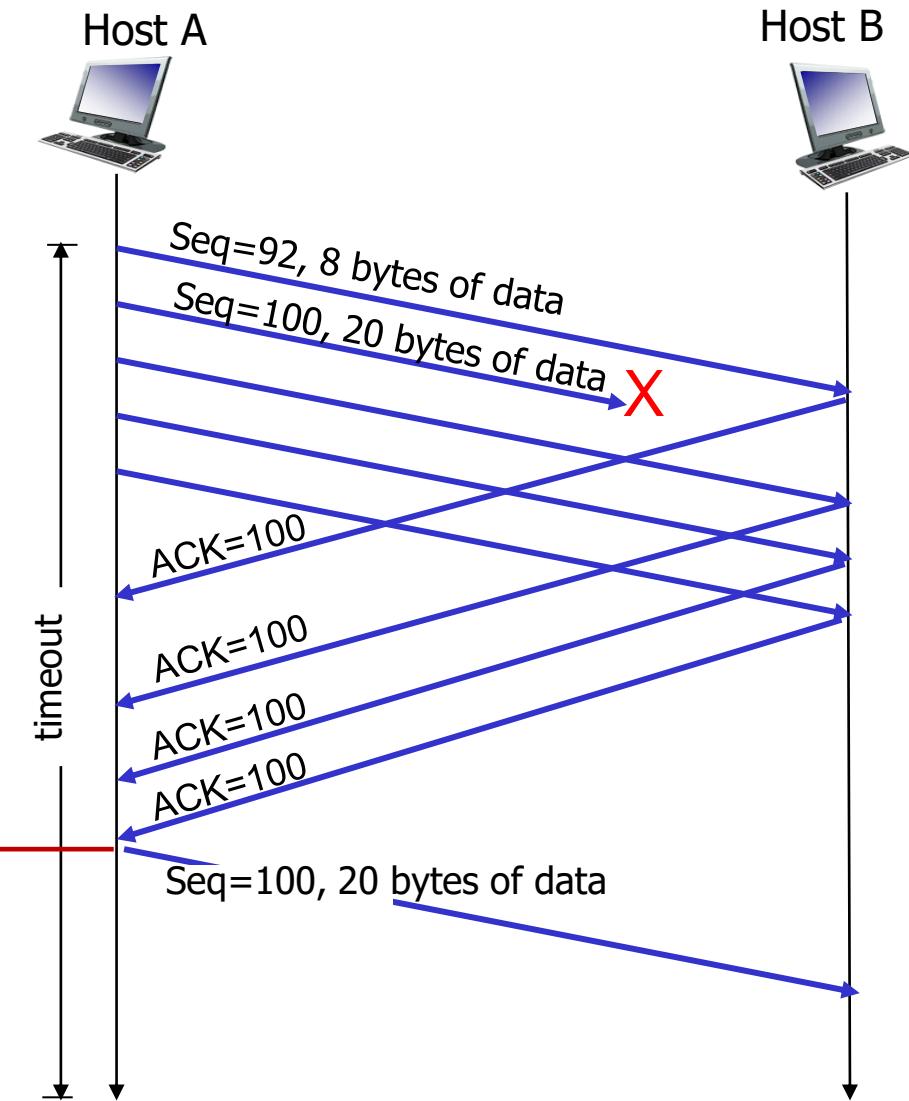
TCP fast retransmit

if sender receives 3 additional ACKs for same data (“triple duplicate ACKs”), resend unACKed segment with smallest seq #

- likely that unACKed segment lost, so don’t wait for timeout



Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!



TCP Receiver: ACK generation [RFC 5681]

