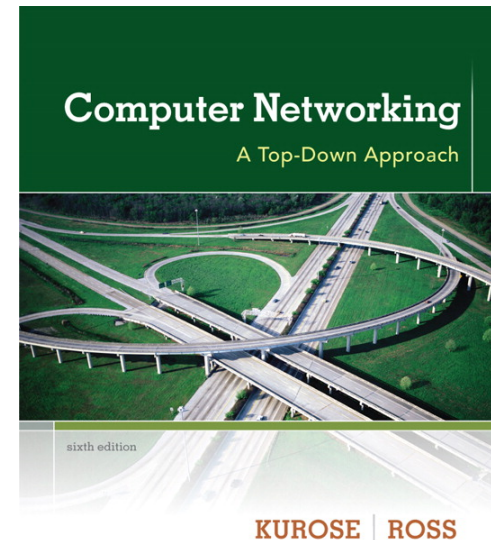


# Chapter 3

## Transport Layer

Reti degli Elaboratori  
Canale AL  
Prof.ssa Chiara Petrioli  
a.a. 2014/2015

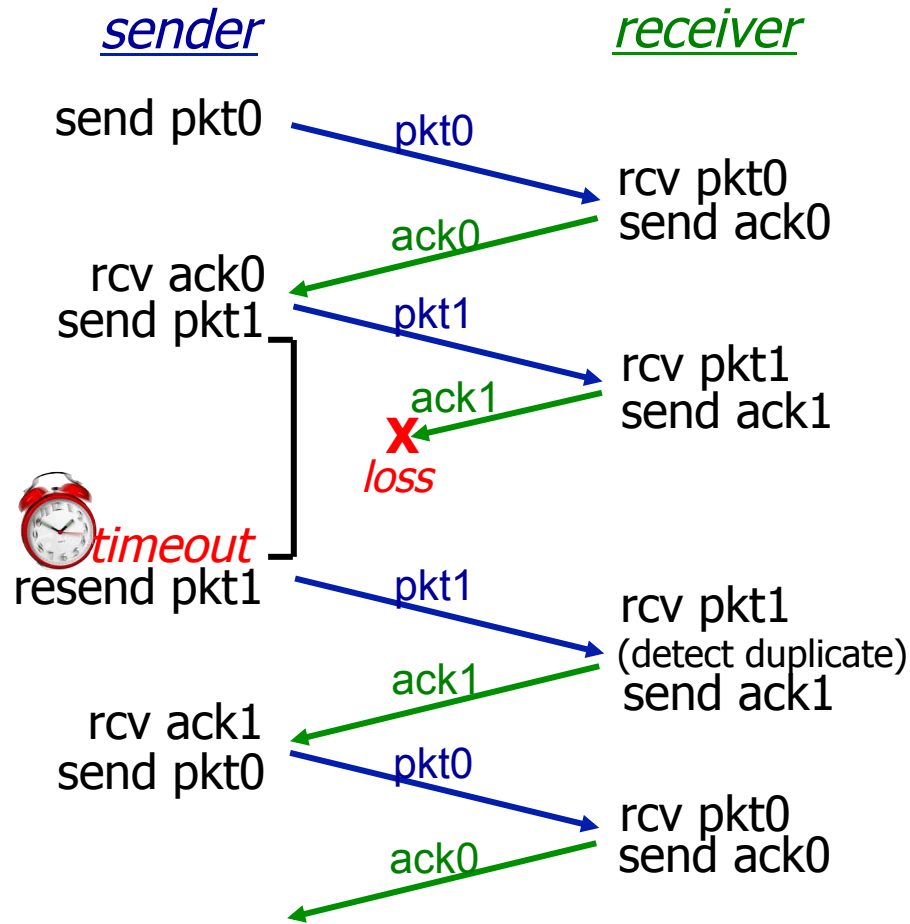
We thank for the support material Prof. Kurose-Ross  
All material copyright 1996-2012  
© J.F Kurose and K.W. Ross, All Rights Reserved



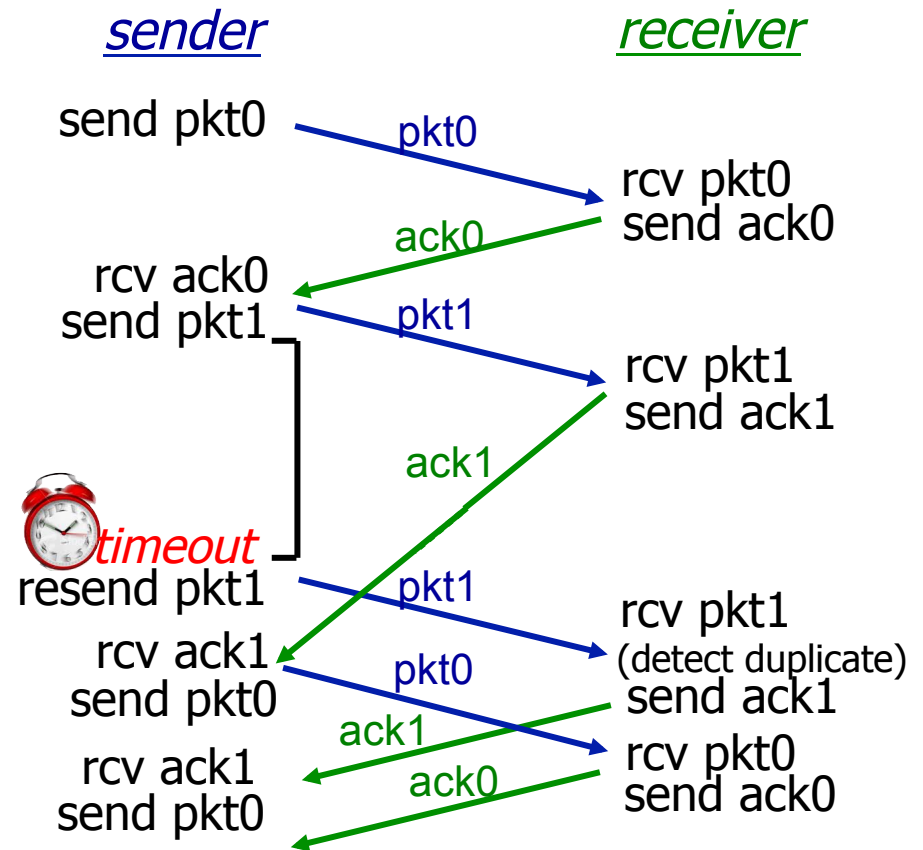
*Computer  
Networking: A Top  
Down Approach*  
6<sup>th</sup> edition  
Jim Kurose, Keith Ross  
Addison-Wesley  
March 2012



# rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

# Performance of rdt3.0

- ❖ rdt3.0 is correct, but performance stinks
- ❖ e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

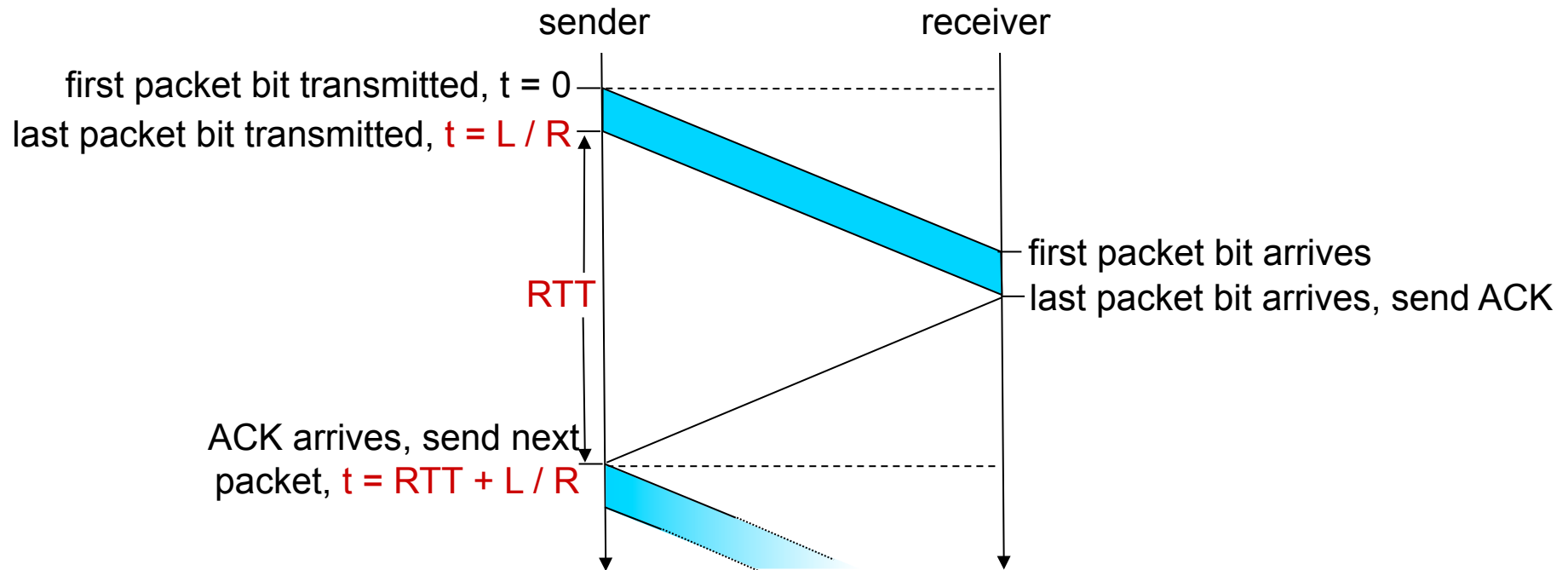
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microseconds}$$

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

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

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

# rdt3.0: stop-and-wait operation

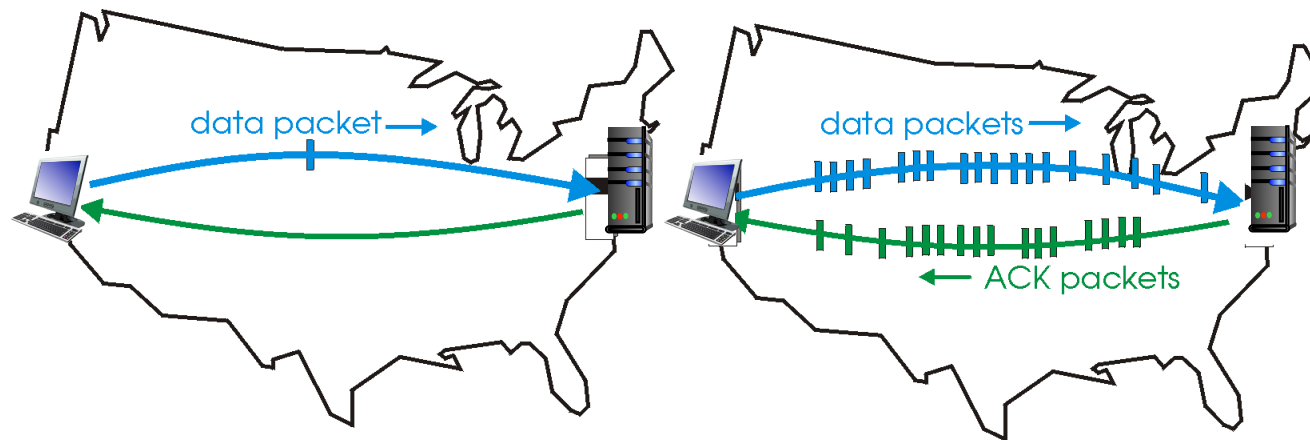


$$U_{\text{sender}} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

# 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

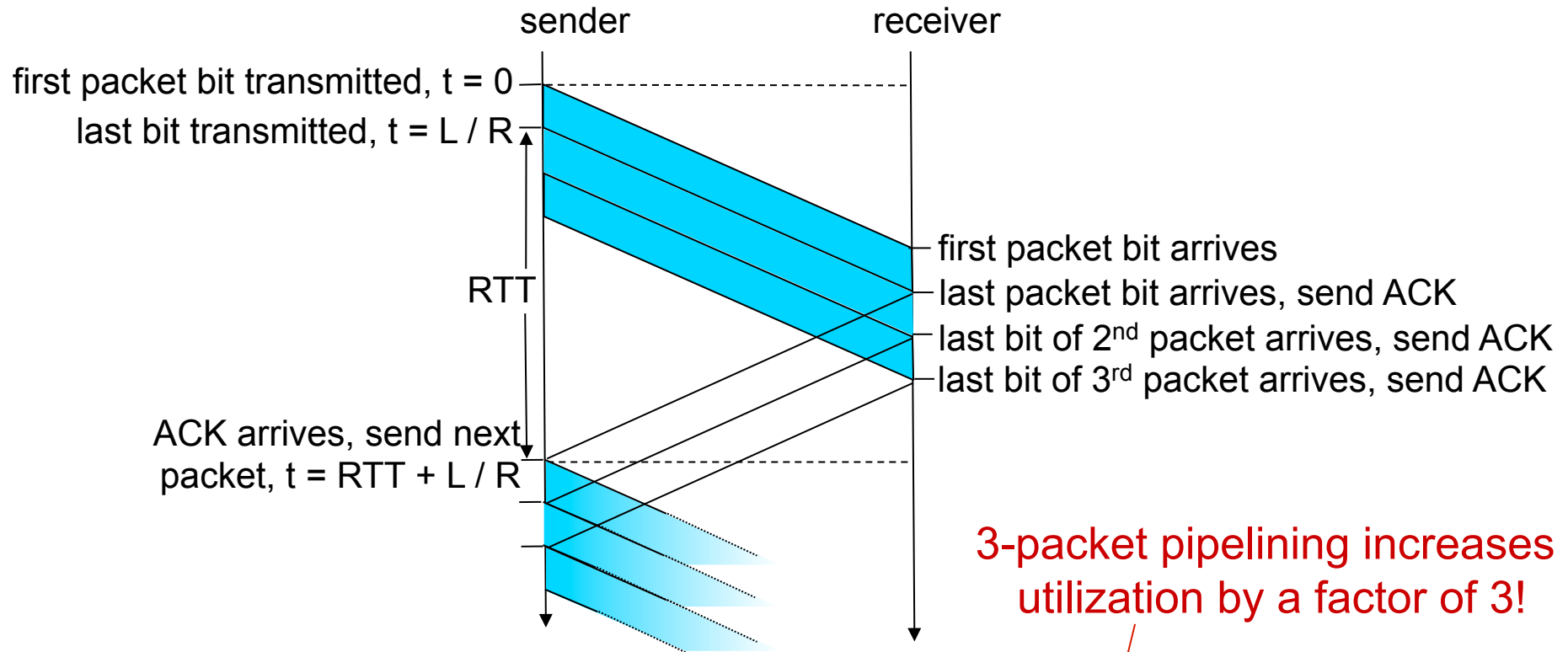


(a) a stop-and-wait protocol in operation

(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

## 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

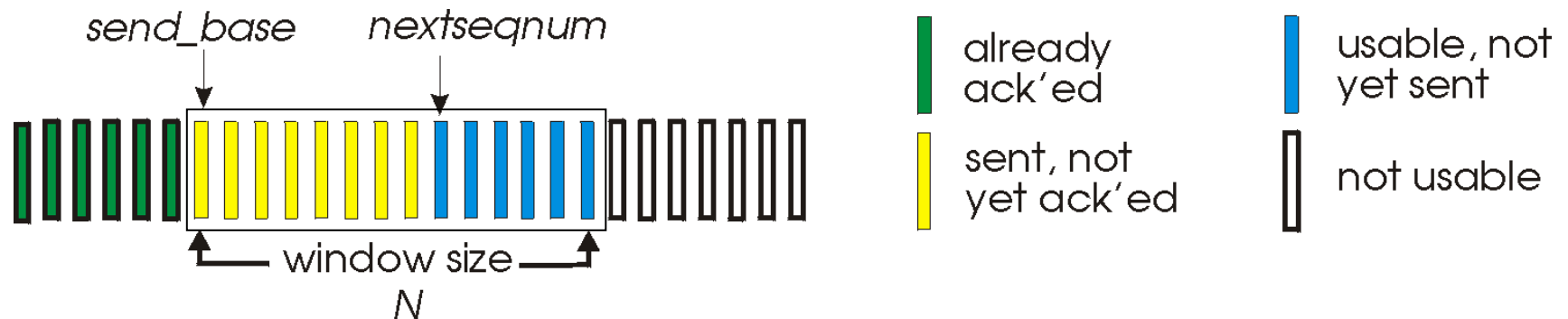
## Selective Repeat:

- ❖ 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



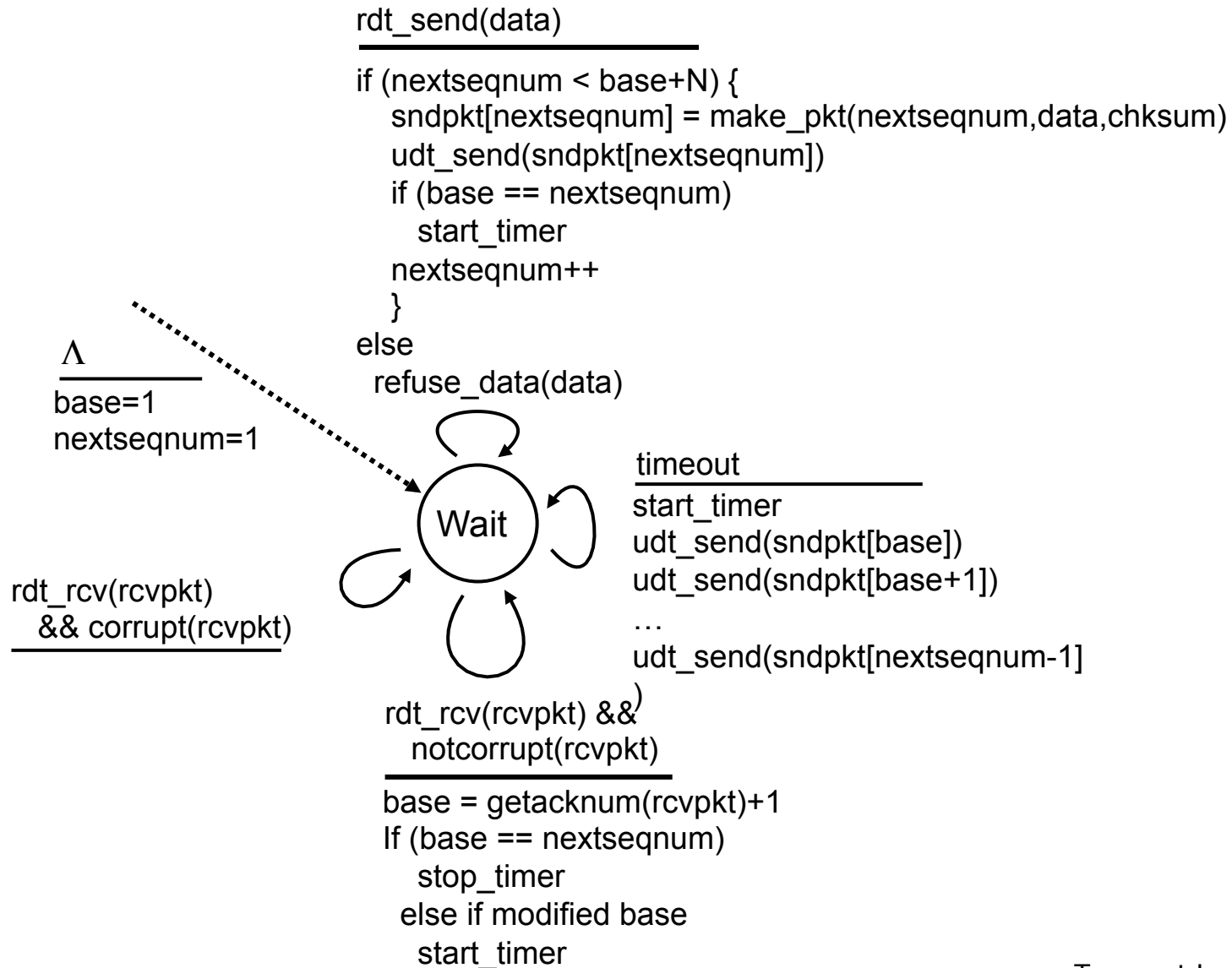
# Go-Back-N: sender

- ❖ k-bit seq # in pkt header
- ❖ “window” of up to N, consecutive unack’ed pkts allowed

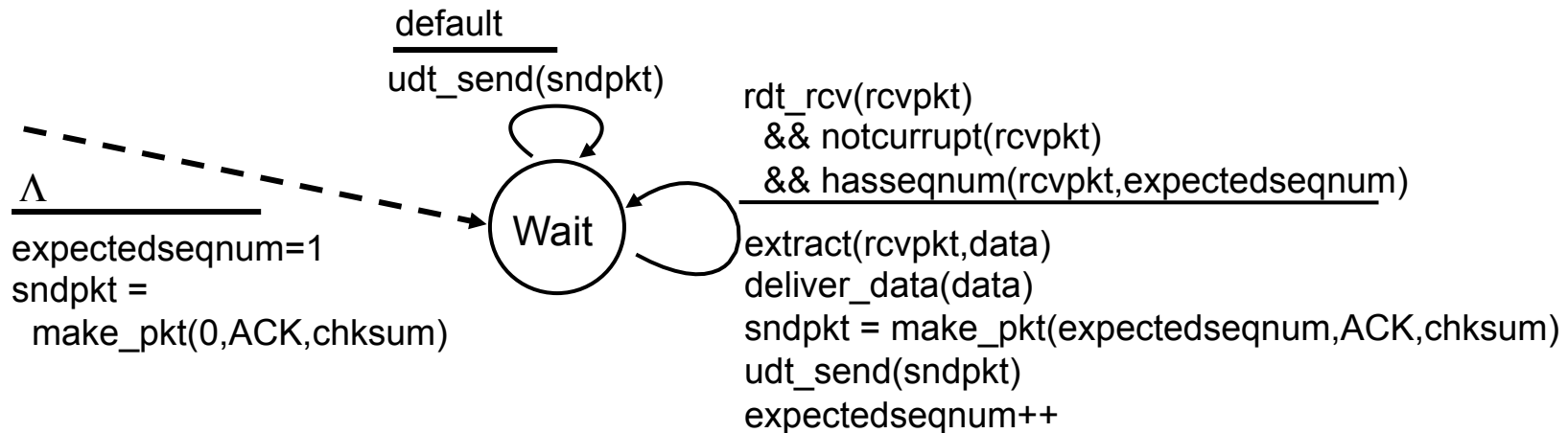


- ❖ ACK(n): ACKs all pkts up to, including seq # n - “cumulative ACK”
  - may receive duplicate ACKs (see receiver)
- ❖ timer for oldest in-flight pkt
- ❖  $timeout(n)$ : retransmit packet n and all higher seq # pkts in window

# GBN: sender extended FSM



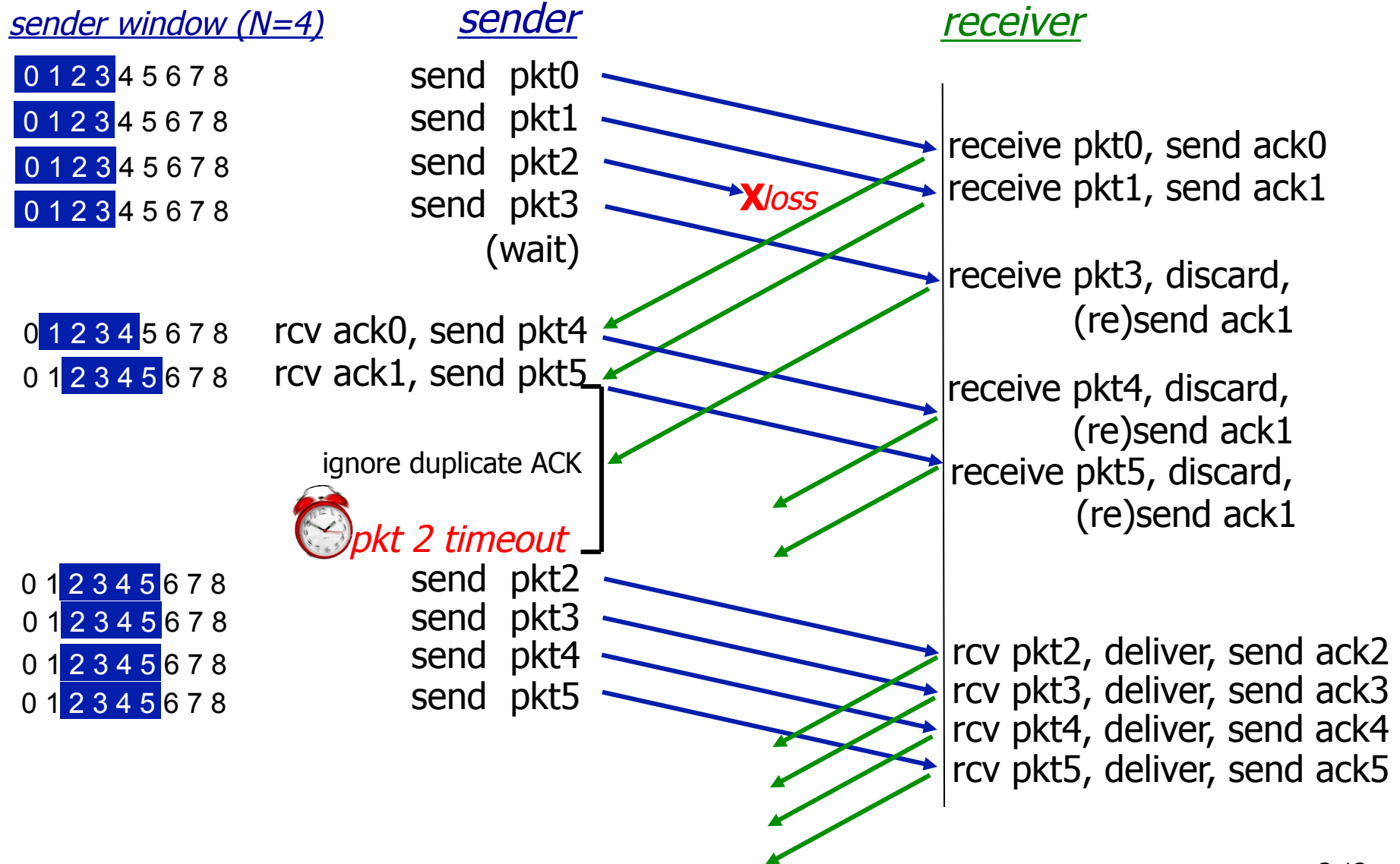
# GBN: receiver extended FSM



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 #

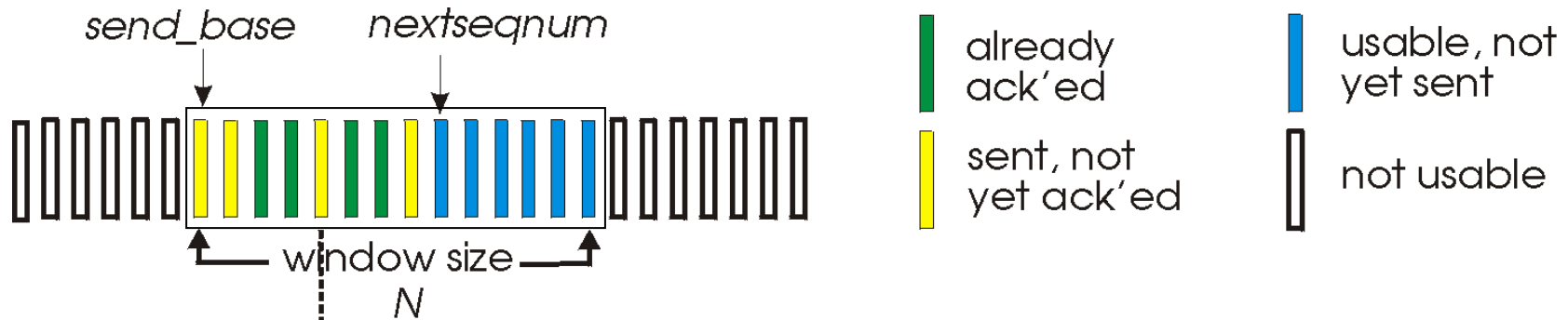
# GBN in action



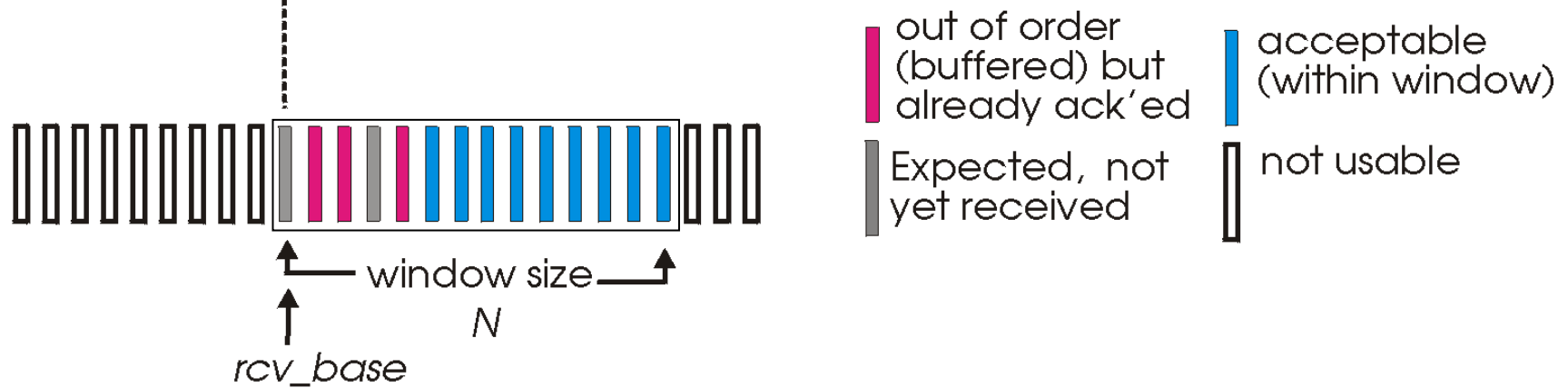
# Selective repeat

- ❖ 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



(a) sender view of sequence numbers



(b) receiver view of sequence numbers

# 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]

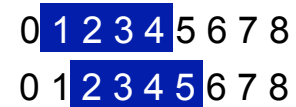
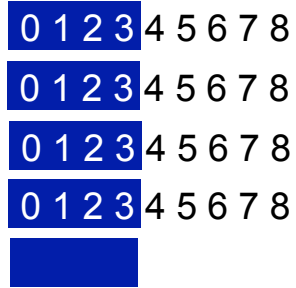
- ❖ ACK(n)

### otherwise:

- ❖ ignore

# Selective repeat in action

sender window (N=4)



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

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

*Q: what happens when ack2 arrives?*

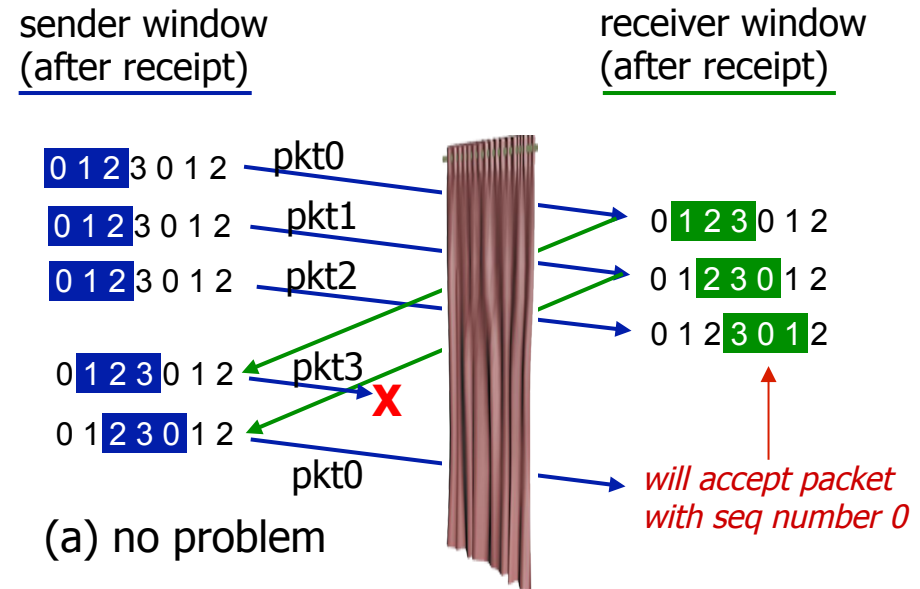


# Selective repeat: dilemma

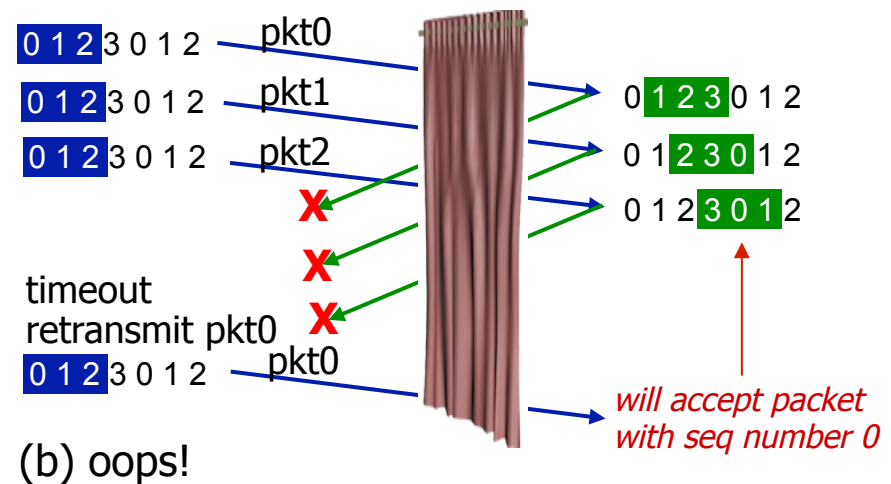
example:

- ❖ 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)?



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

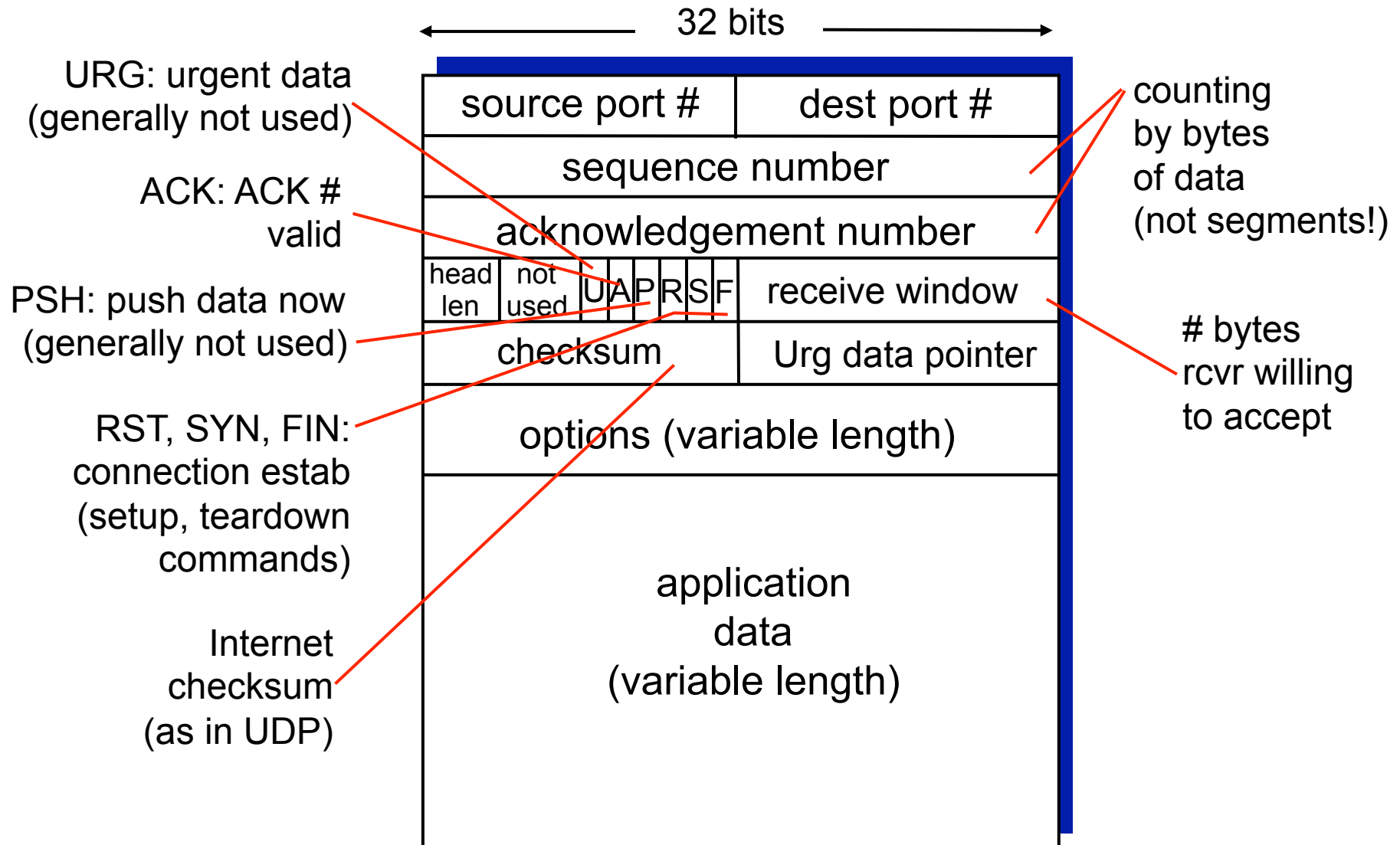
3.7 TCP congestion control

# TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- ❖ **point-to-point:**
  - one sender, one receiver
- ❖ **reliable, in-order *byte stream*:**
  - no “message boundaries”
- ❖ **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 segment structure



# TCP seq. numbers, ACKs

## sequence numbers:

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

## acknowledgements:

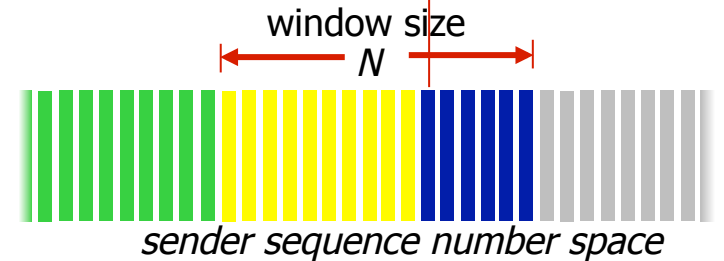
- seq # of next byte expected from other side
- cumulative ACK

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

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

outgoing segment from sender

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



sent  
ACKed

sent, not-  
yet ACKed  
 (“in-  
flight”)

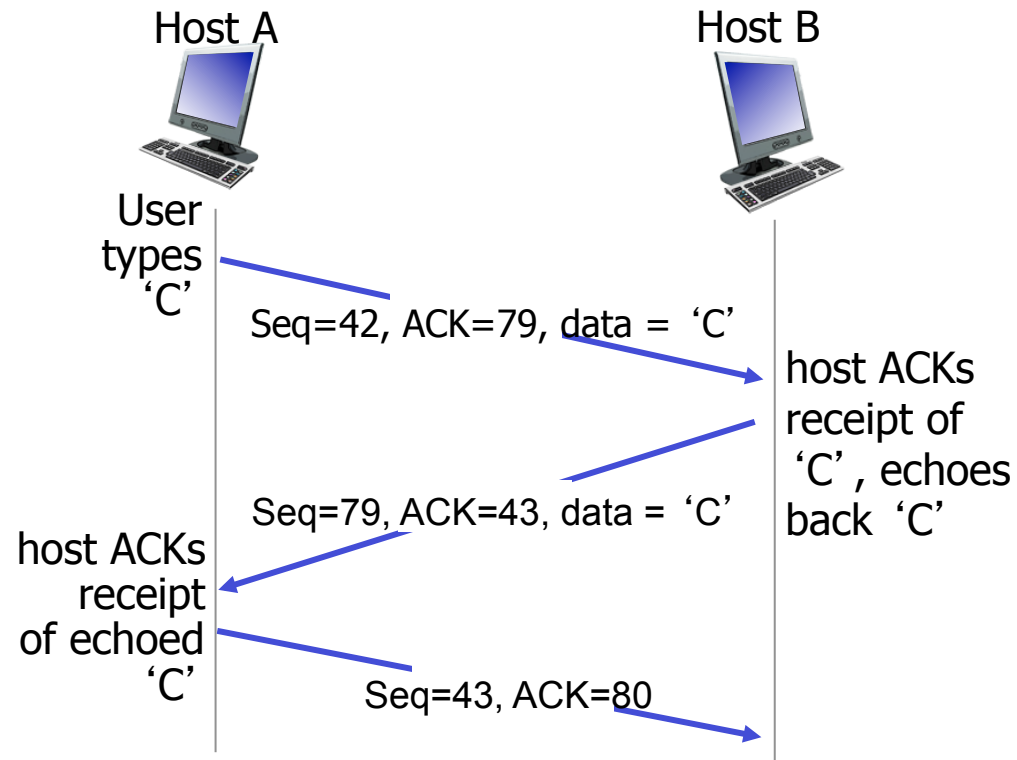
usable  
but not  
yet sent

not  
usable

incoming segment to sender

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

# TCP seq. numbers, ACKs



simple telnet scenario

# 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

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`

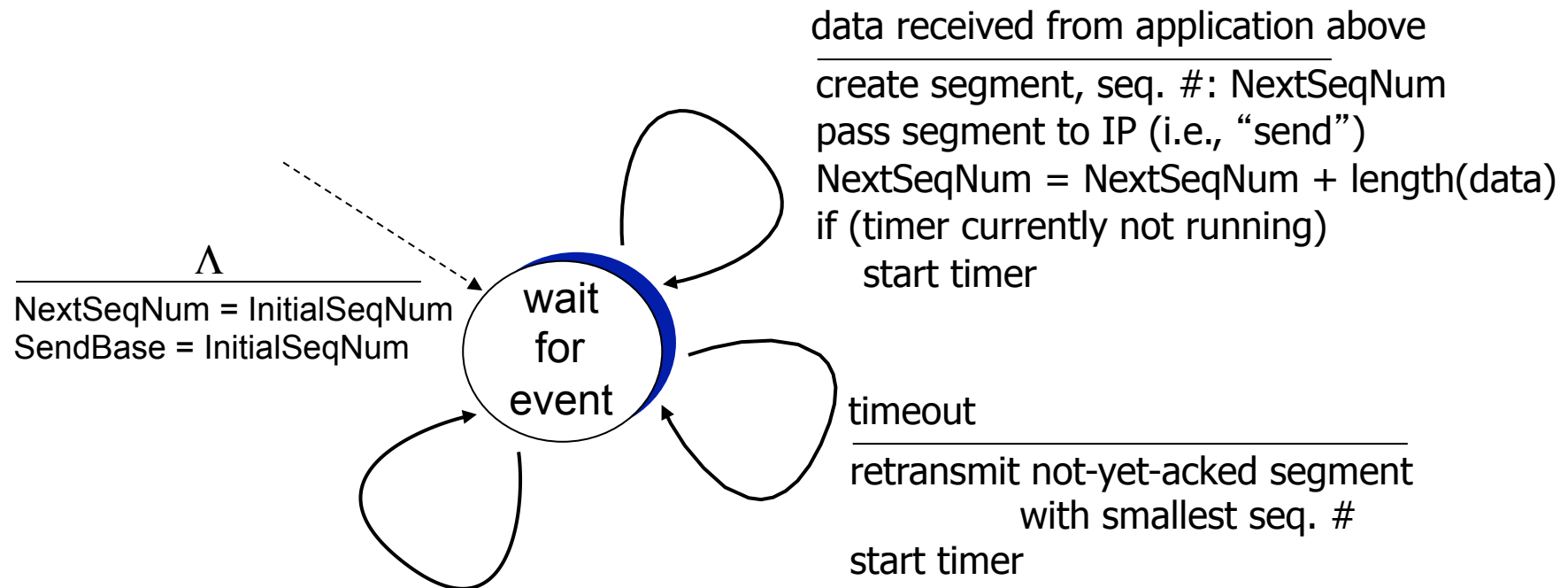
## *timeout:*

- ❖ 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

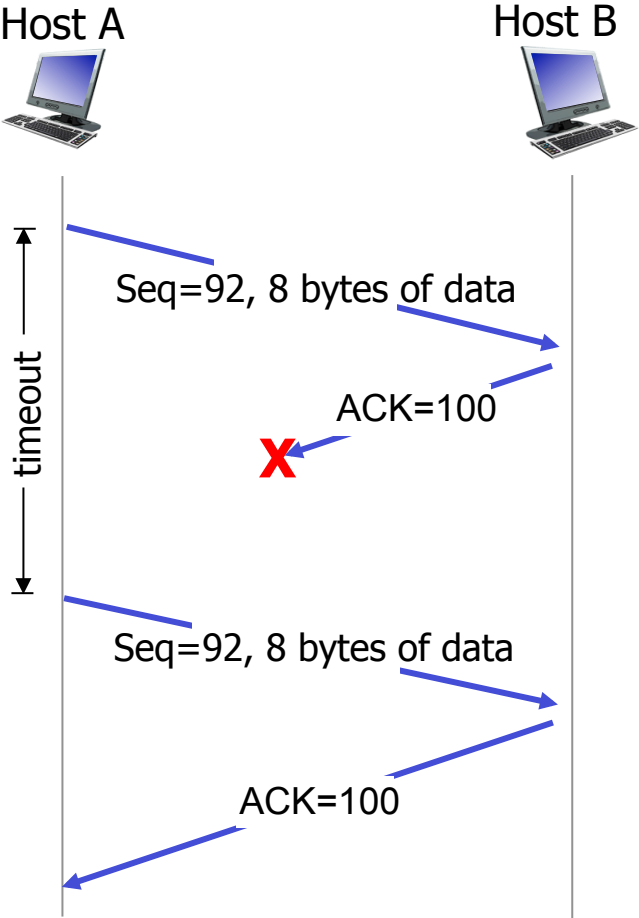
# TCP sender (simplified)



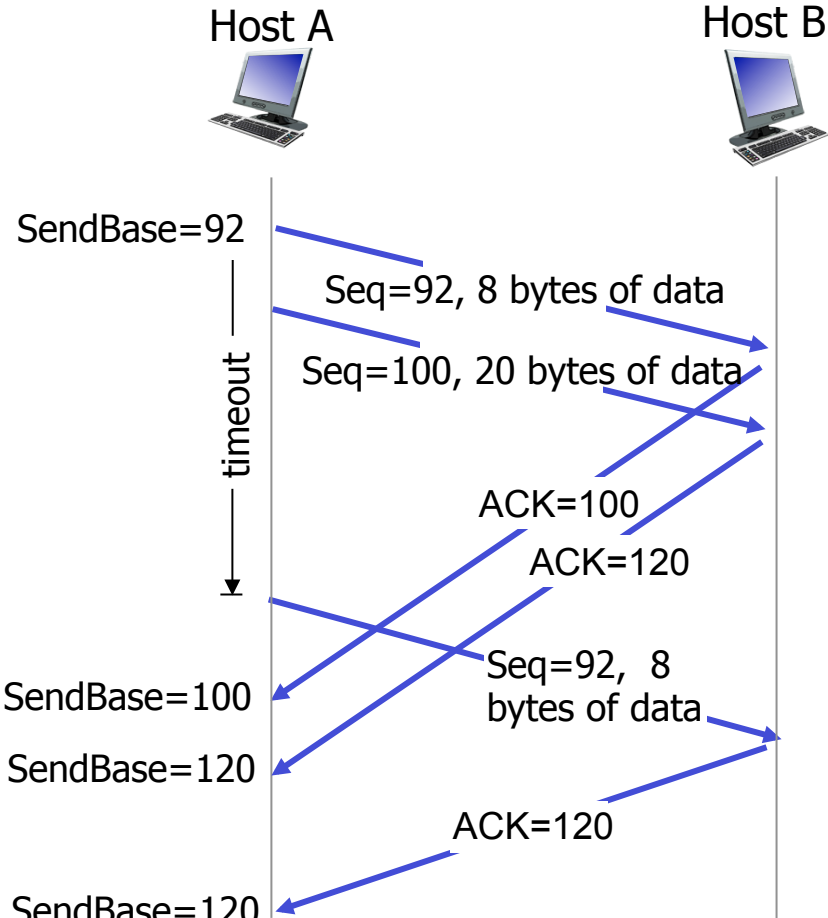
ACK received, with ACK field value y

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

# TCP: retransmission scenarios

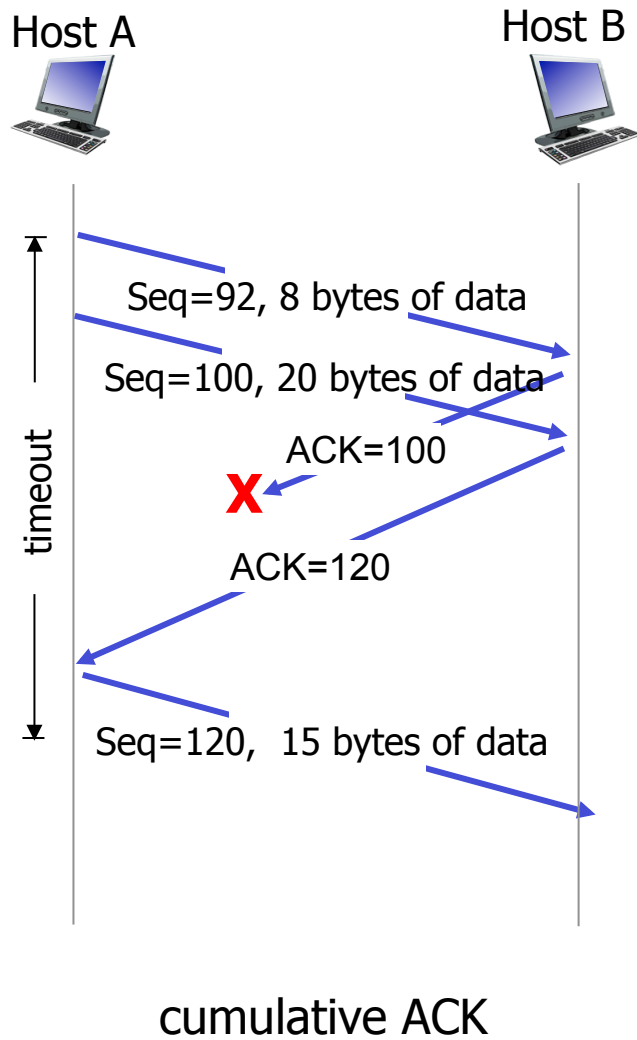


lost ACK scenario



premature timeout

# TCP: retransmission scenarios



# 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

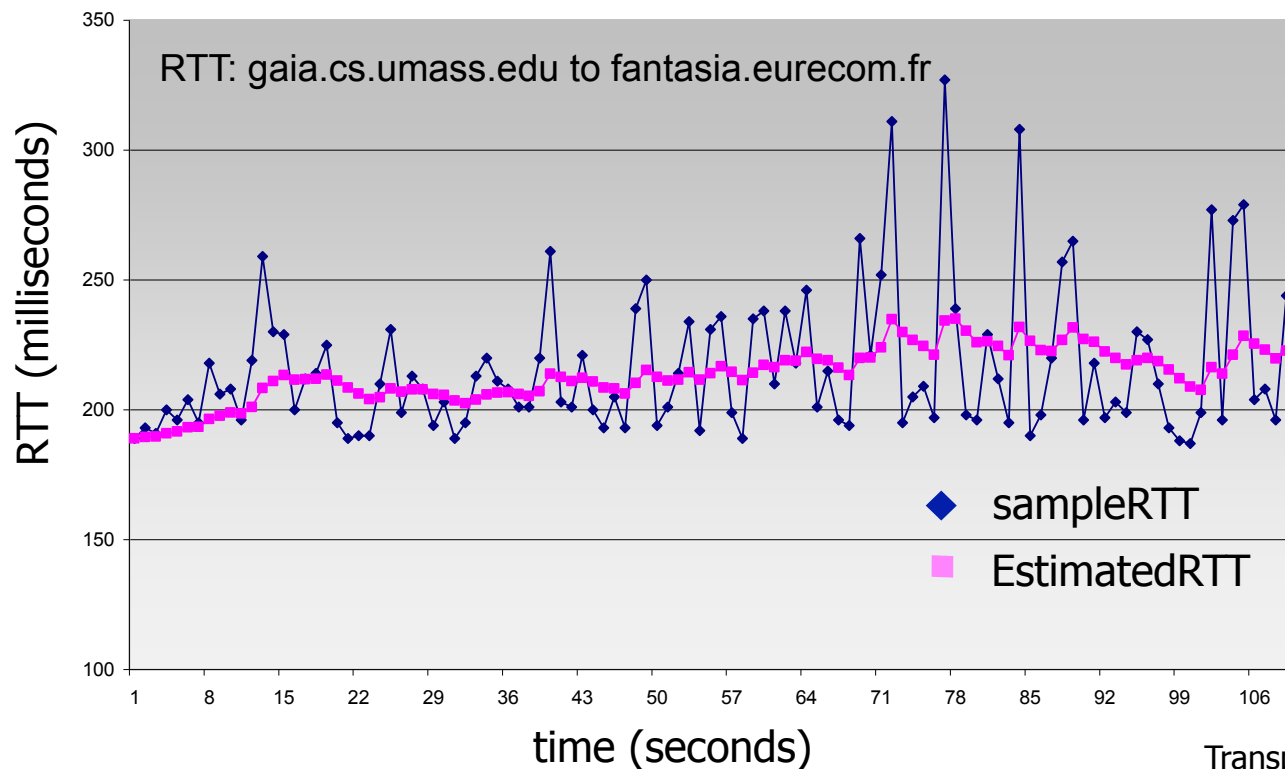
Q: how to estimate RTT?

- ❖ **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 ACK generation [RFC 1122, RFC 2581]

## *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-expected 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

