

# Lesson 05-03: TCP

CS 356 Computer Networks

Mikyung Han

[mhan@cs.utexas.edu](mailto:mhan@cs.utexas.edu)

## Example Protocols

FTP, HTTP, SMTP

Application

TCP, UDP

Transport

IP

Network

Ethernet, WiFi

Link

802.3 PHY

Physical

## Responsible for

application specific needs

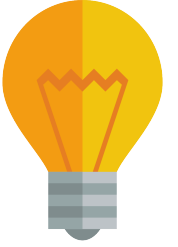
process to process data transfer

host to host data transfer across different network

data transfer between physically adjacent nodes

bit-by-bit or symbol-by-symbol delivery

## Internet Reference Model



# Outline

## I. Recap

# Selective repeat: a dilemma!

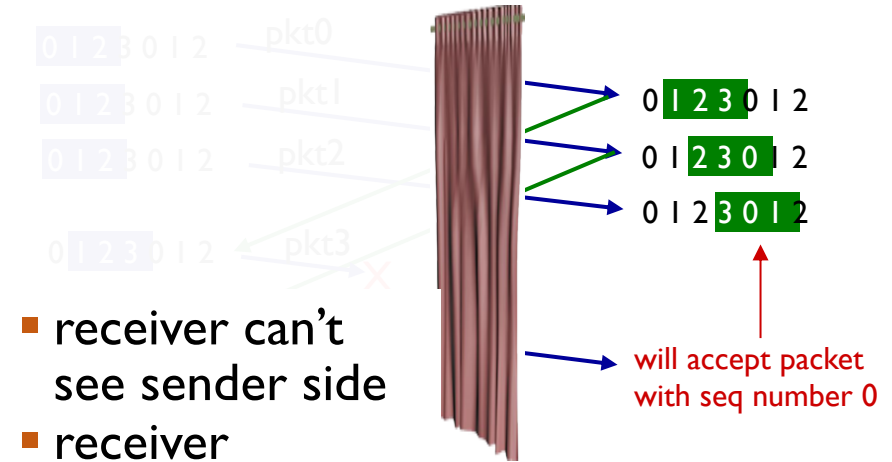
example:

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

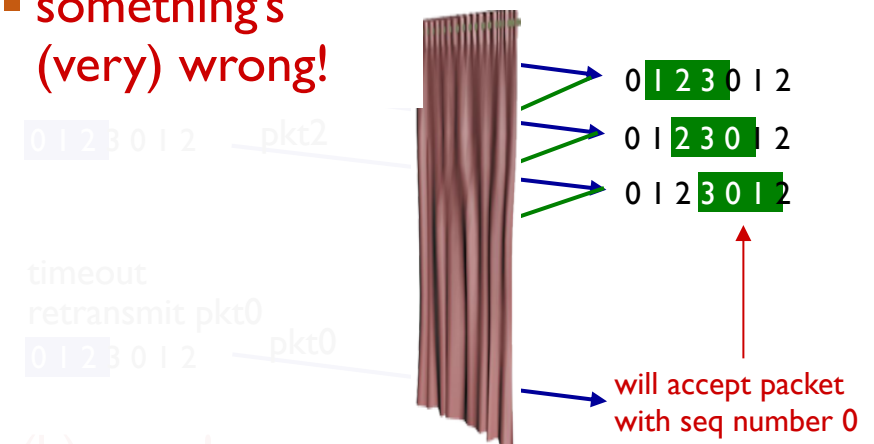
What should be the  
relationship btw seq #  
size and window size?

sender window  
(after receipt)

receiver window  
(after receipt)



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



(b) oops!

# Outline

1. Recap

 2. TCP overview

# TCP vs rdt

- What are the similarities?
- What are the differences?

# TCP: overview

RFCs: 793, 1122, 2018, 5681, 7323

- point-to-point:
  - one sender, one receiver
- reliable, in-order byte stream:
  - no “message boundaries”
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- cumulative ACKs
- timeouts
- pipelining:
  - TCP congestion and flow control set window size
- connection-oriented (handshake)
- flow controlled:
  - sender will not overwhelm receiver

# TCP sequence numbers, ACKs

## Sequence numbers:

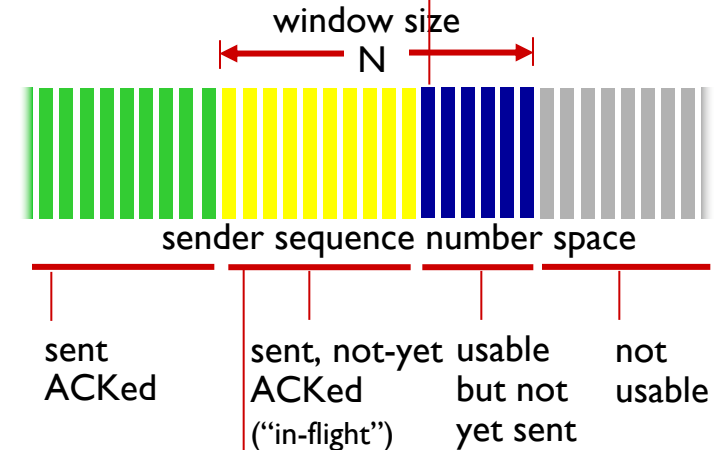
- byte stream “number” of first byte in segment’s data

## Acknowledgements:

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

outgoing segment from sender

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



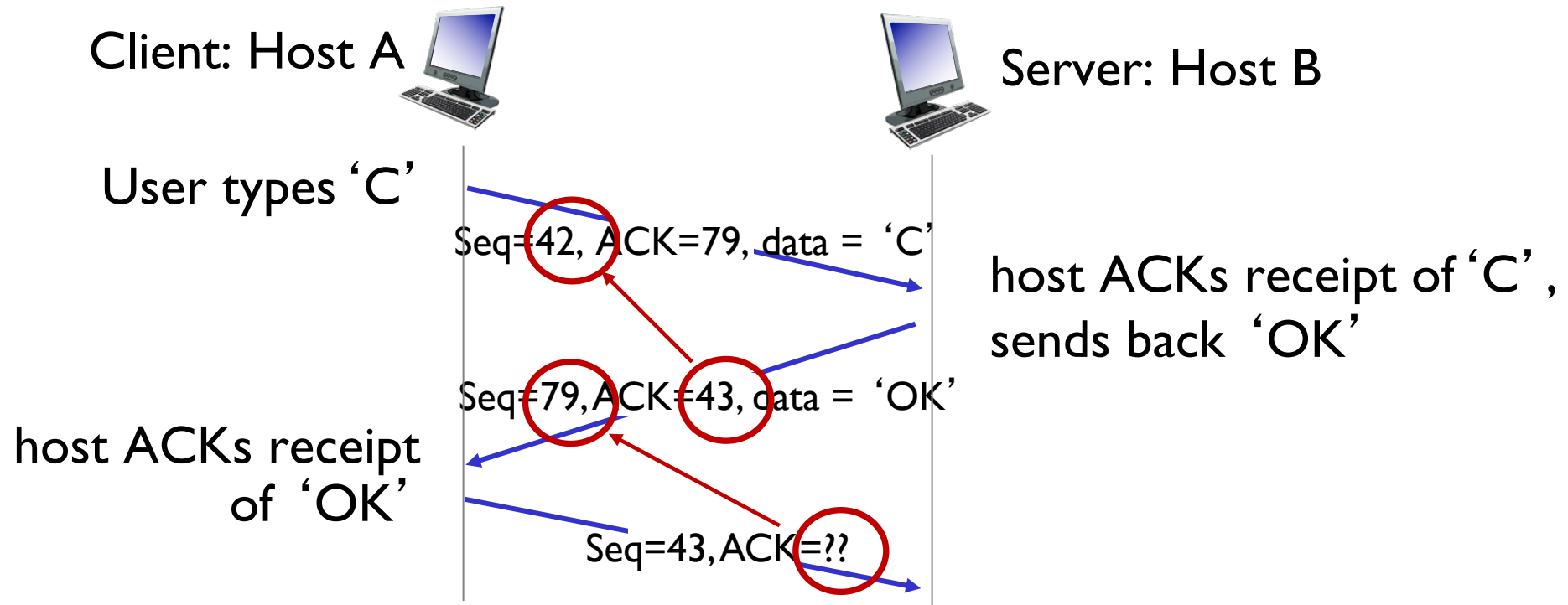
outgoing segment from receiver

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



# TCP ACKs can piggyback to DATA

## simple telnet scenario



Does the last segment have DATA? Why then seq no?

# Outline

1. Recap
2. TCP overview
-  3. TCP timeout

# How to set TCP timeout value?

- What happens if timeout value is too short?
- What happens if timeout value is too long?
- We know it should be at least longer than... what?

# How to set TCP timeout value?

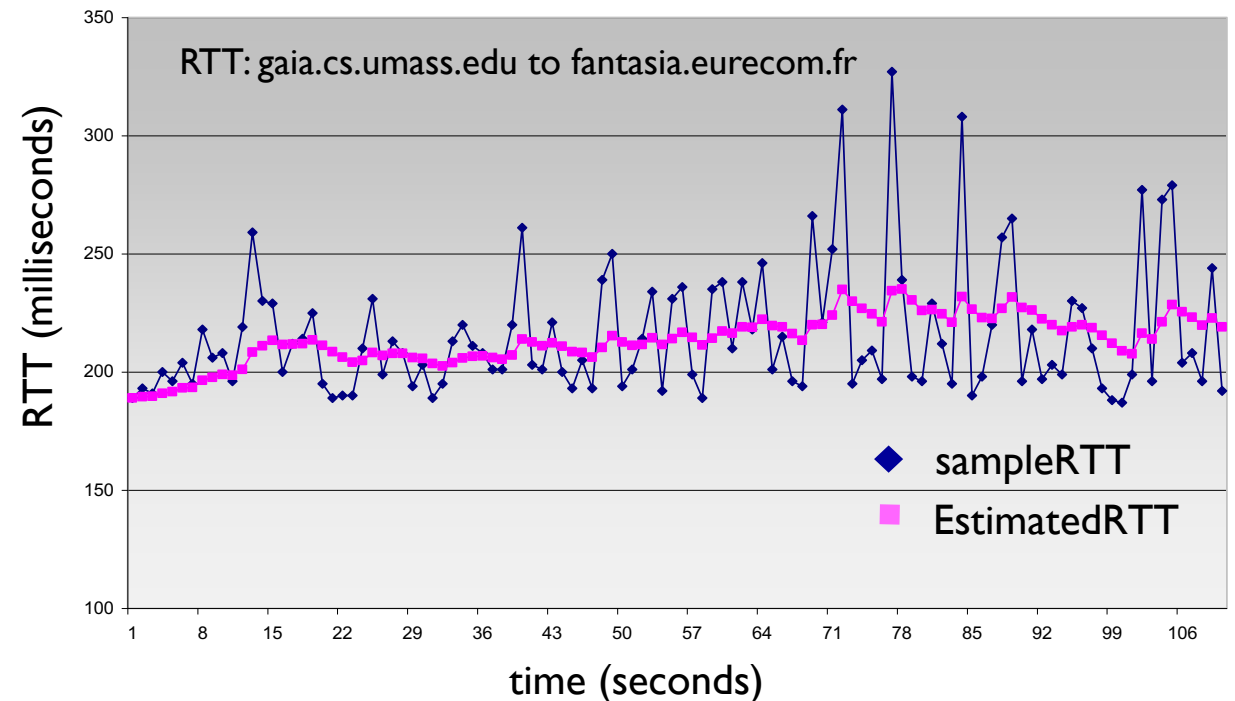
- **too short:** premature timeout, unnecessary retransmissions
- **too long:** slow reaction to segment loss
- It should be at least longer than RTT but RTT varies!
- TCP maintains timer for its **oldest unACKed segment**

TCP uses EWMA of Sample RTT plus safety margin

# Estimate RTT uses EWMA to smooth out

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

- exponential weighted moving average (EWMA)
- SampleRTT: measured time from segment transmission until ACK receipt
- influence of past sample decreases exponentially fast
- typical value:  $\alpha = 0.125$



# In addition, safety margin is added

- timeout interval: EstimatedRTT plus “safety margin”
  - large variation in EstimatedRTT: want a larger safety margin

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



↑  
estimated RTT


↑  
“safety margin”

- DevRTT: EWMA of SampleRTT deviation from EstimatedRTT:

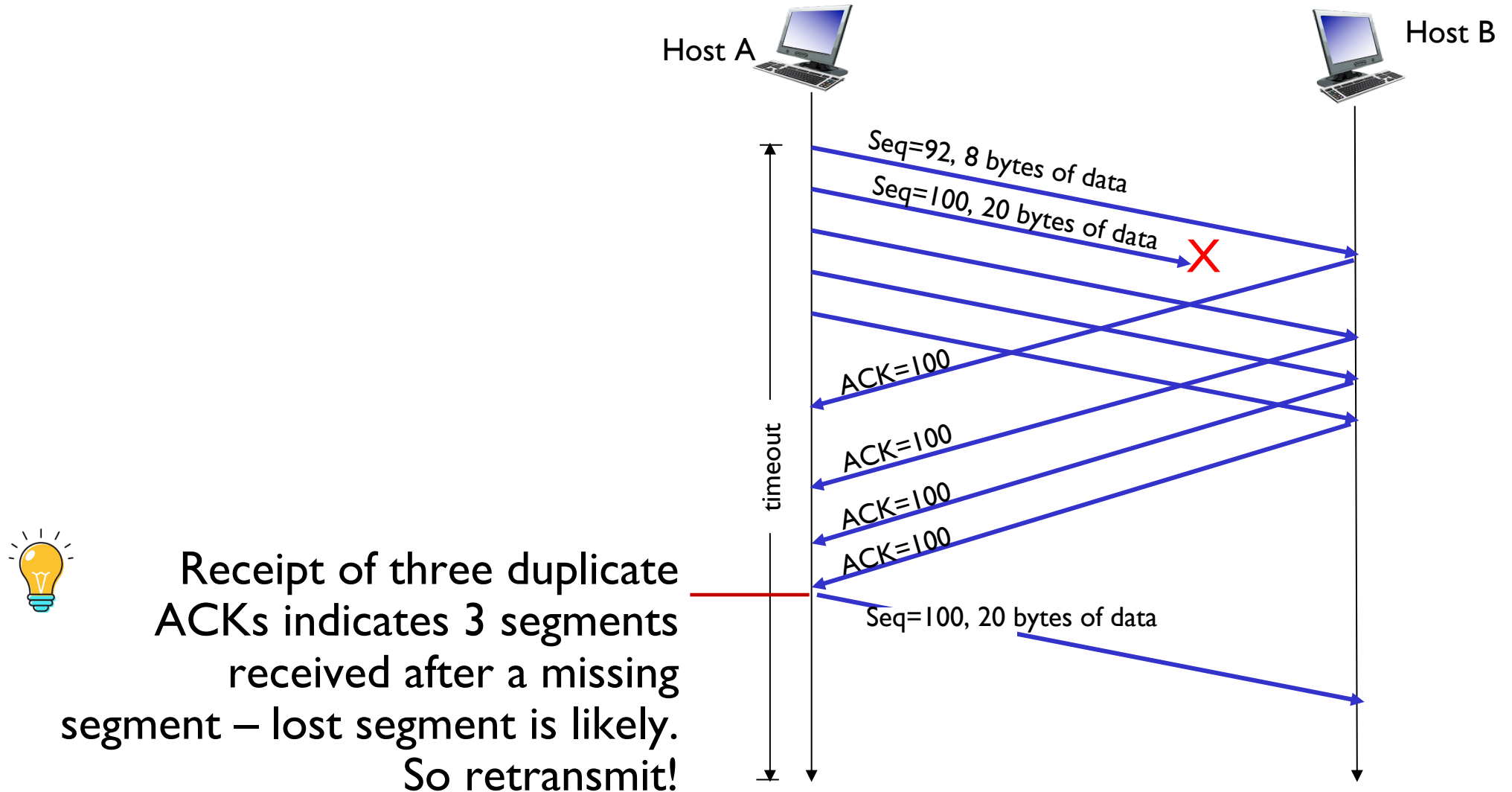
$$\text{DevRTT}_n = (1 - \beta) * \text{DevRTT}_{n-1} + \beta * |\text{SampleRTT}_n - \text{EstimatedRTT}_n|$$

(typically,  $\beta = 0.25$ )

# Outline

1. Recap
2. TCP overview
3. TCP timeout
-  4. **TCP retransmissions**

# TCP fast retransmit: upon receiving triple dup ACKs immediately retransmit without timeout





# T/F? Timeout interval for retransmission is derived from EstimatedRTT and DevRTT

- TCP assumes packet is lost upon timeout
- TCP assumes the packet is lost due to congestion
- With these assumptions, is it a good idea to retransmit as soon as possible?

Doubles the timeout interval each time TCP retransmits!

# Outline

1. Recap
2. TCP overview
3. TCP timeout
4. TCP interesting scenarios
-  5. **TCP flow control**

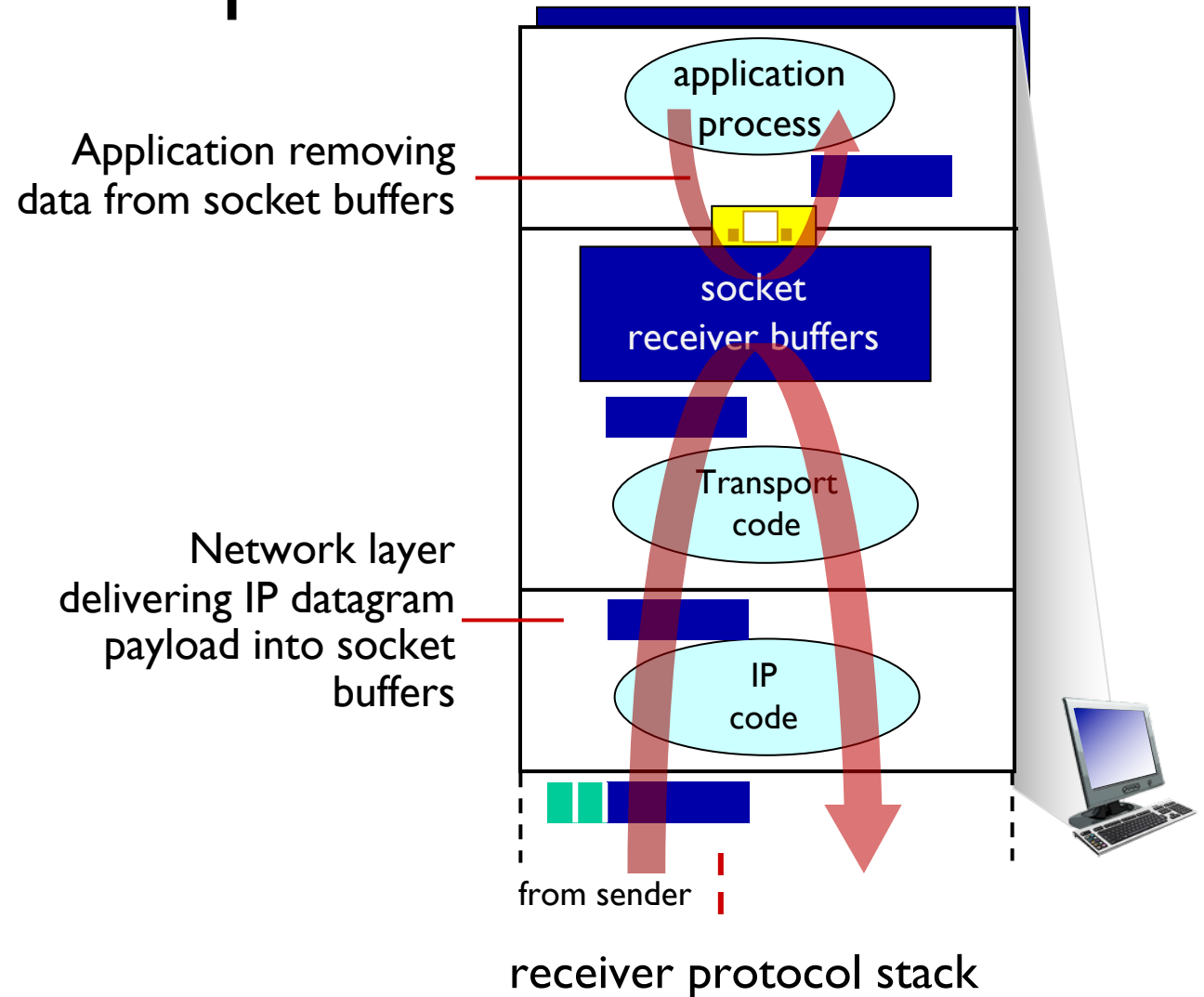
# Why in POP Dostoyevsky caused packet losses?

- Even when client/server is within the same host
- No network between client and server thus no network loss!

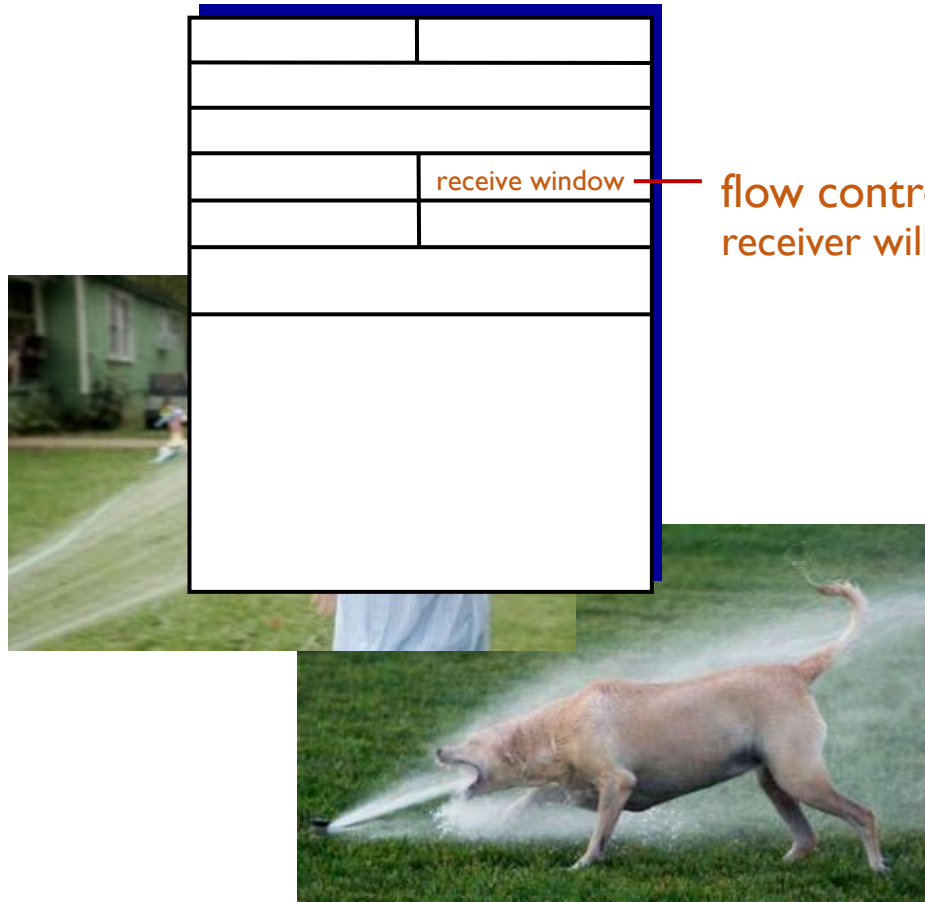
Where were 20,000+ loss happening then?

# Loss happens if network delivers faster than what application layer can process

Loss was happening in the socket buffer of the receiver!



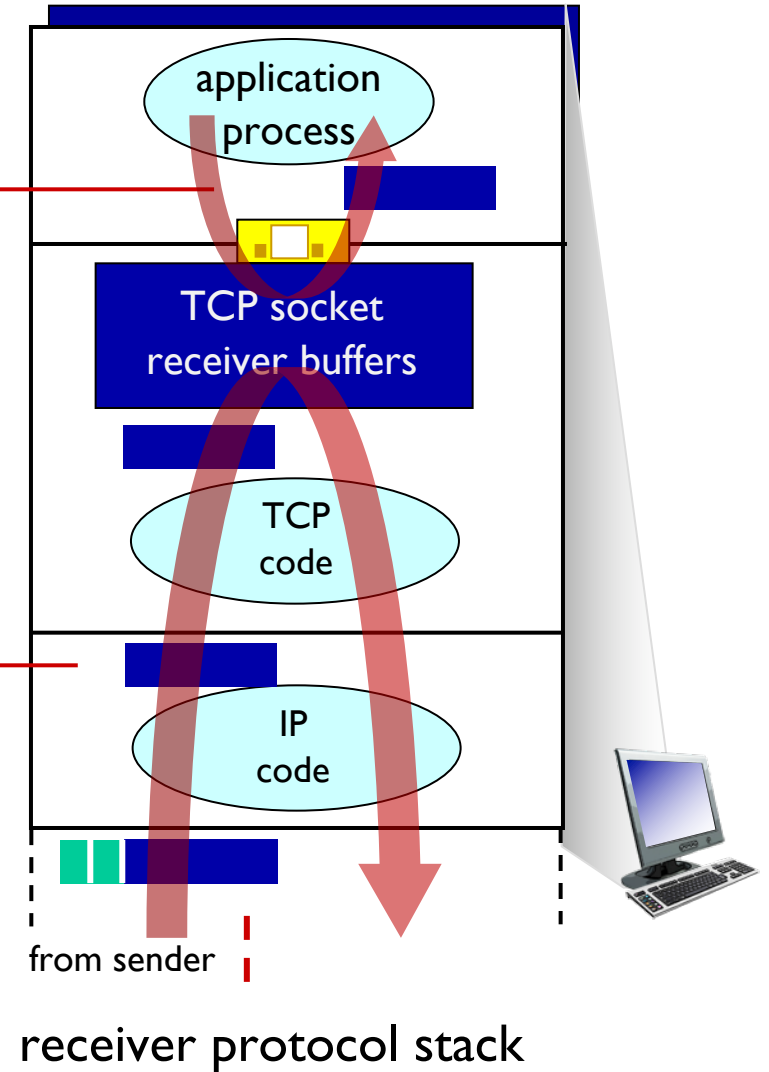
# TCP flow control ensures NOT to overflow receiver socket buffer



Application removing  
data from TCP socket  
buffers

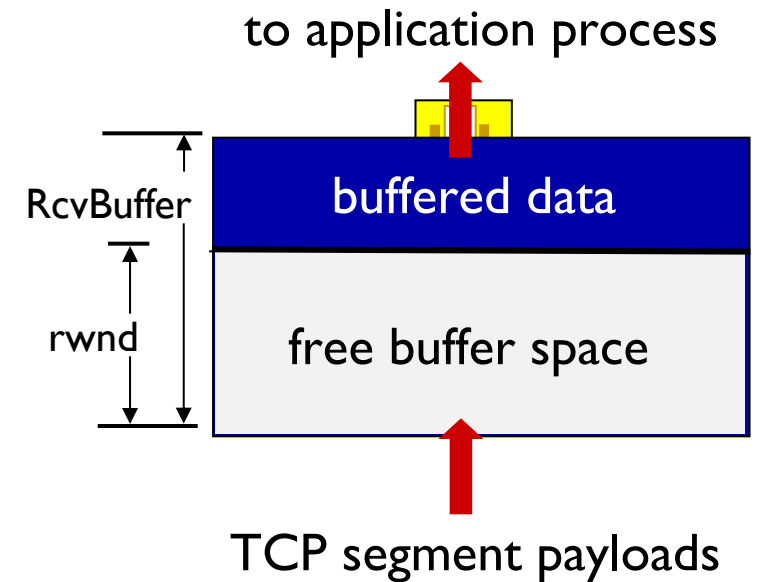
flow control: # bytes  
receiver willing to accept

Network layer  
delivering IP datagram  
payload into TCP  
socket buffers



# TCP sender limits in-flight packets smaller than rwnd


- TCP receiver “advertises” free buffer space in rwnd field in TCP header
  - RcvBuffer size set via socket options (default 4096 bytes)



TCP receiver-side buffering

Guarantees receiver buffer will not overflow!

# Outline

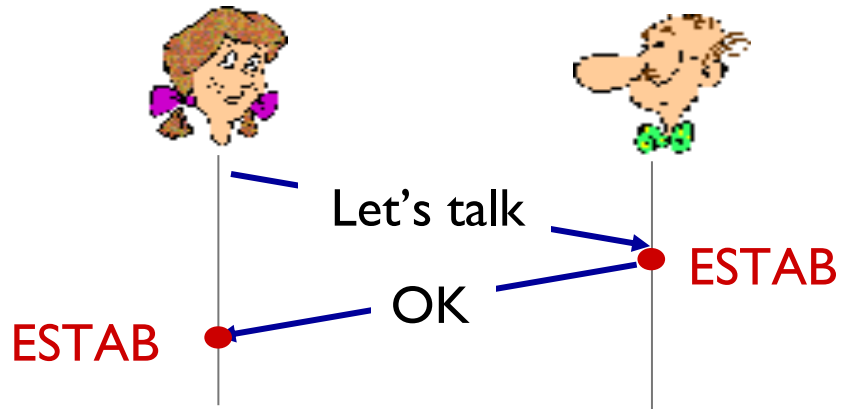
1. Recap
2. TCP overview
3. TCP timeout
4. TCP interesting scenarios
5. TCP flow control
-  6. TCP connection management

# TCP has “handshake” prior to actual data exchange

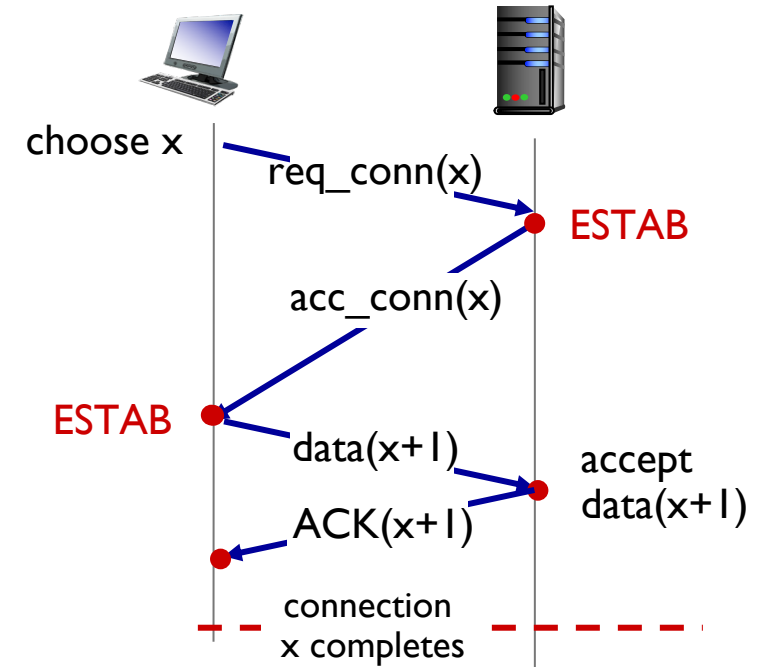
- agree to establish connection
- agree on connection parameters (e.g., starting seq #s, rwnds)



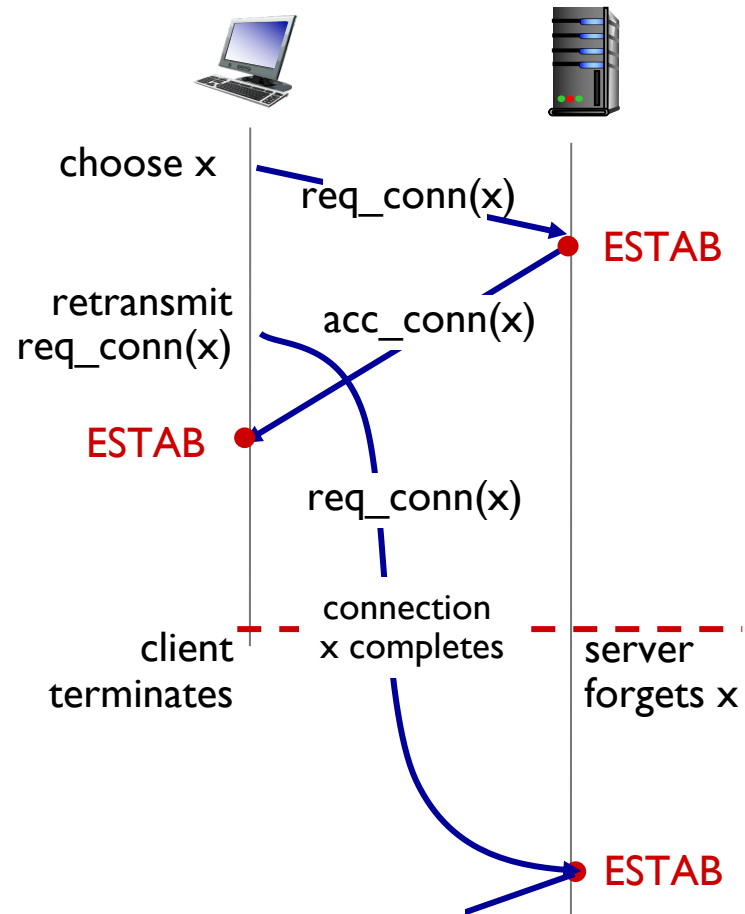
# We could use 2-way handshake




No problem!

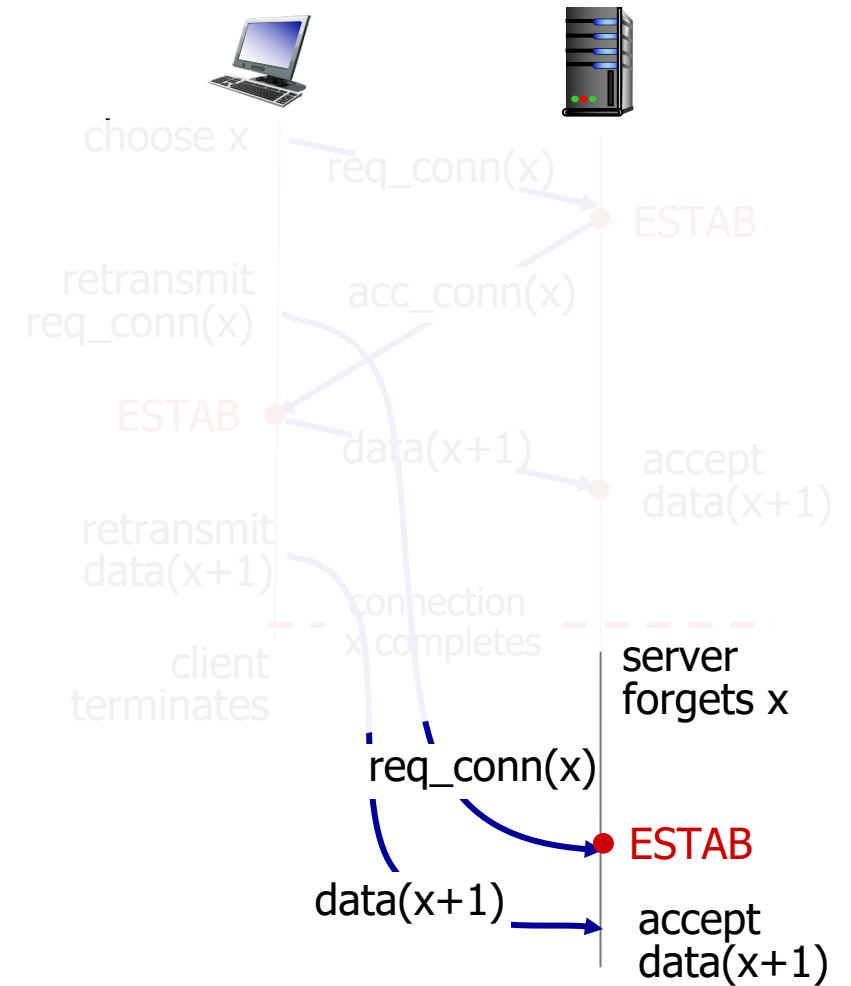



# 2-way handshake is not enough!



 Problem: half open connection! (no client)

# 2-way handshake is not enough!



 Problem: dup data accepted!

# TCP 3-way handshake

## Client state

```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

LISTEN

```
clientSocket.connect((serverName, serverPort))
```

SYNSENT

ESTAB

choose init seq num, x  
send TCP SYN msg

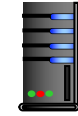
received SYNACK(x)  
indicates server is live;  
send ACK for SYNACK;  
this segment may contain  
client-to-server data



SYNbit=1, Seq=x

SYNbit=1, Seq=y  
ACKbit=1; ACKnum=x+1

ACKbit=1, ACKnum=y+1



choose init seq num, y  
send TCP SYNACK  
msg, acking SYN

received ACK(y)  
indicates client is live

## Server state

```
serverSocket = socket(AF_INET, SOCK_STREAM)  
serverSocket.bind(("", serverPort))  
serverSocket.listen(1)  
connectionSocket, addr = serverSocket.accept()
```

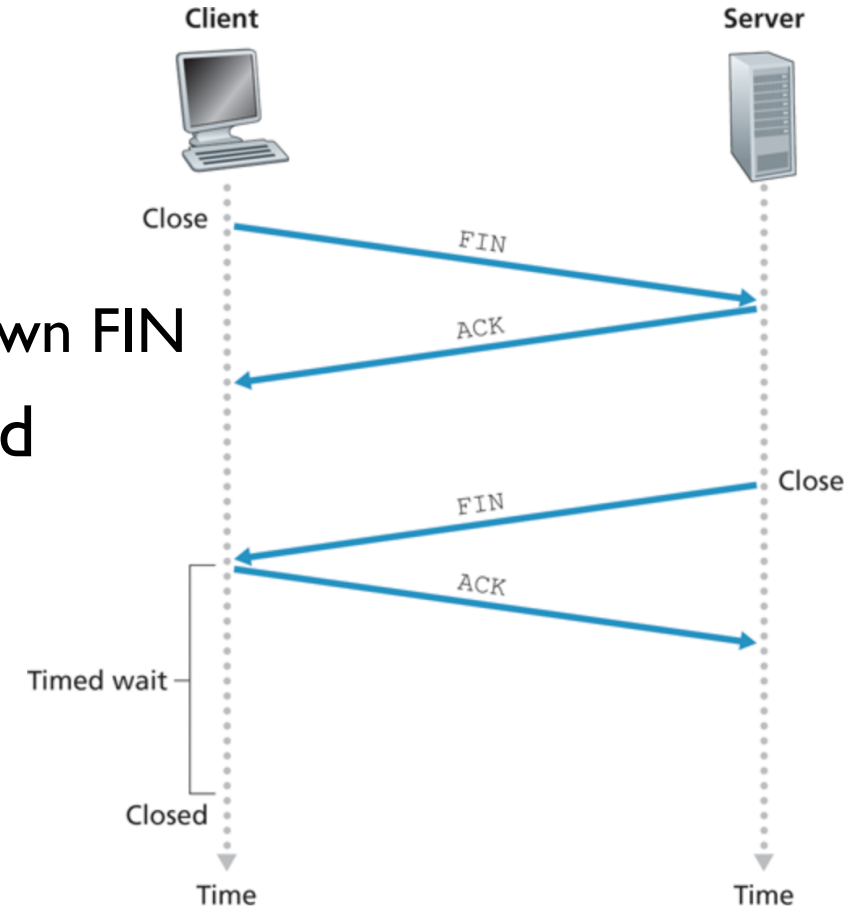
LISTEN

SYN RCVD


ESTAB

# Closing a TCP 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



# Outline

1. Recap
2. TCP overview
3. TCP timeout
4. TCP interesting scenarios
5. TCP flow control
6. TCP connection management
-  7. TCP seq num wrap around

# Sequence number with 2 bits

0, 1, 2, 3, 0, 1, 2, 3, ...

- Can we allow window size 5?

- 0, 1, 2, 3, 0, 1, 2, 3, ...

- How about window size 3?

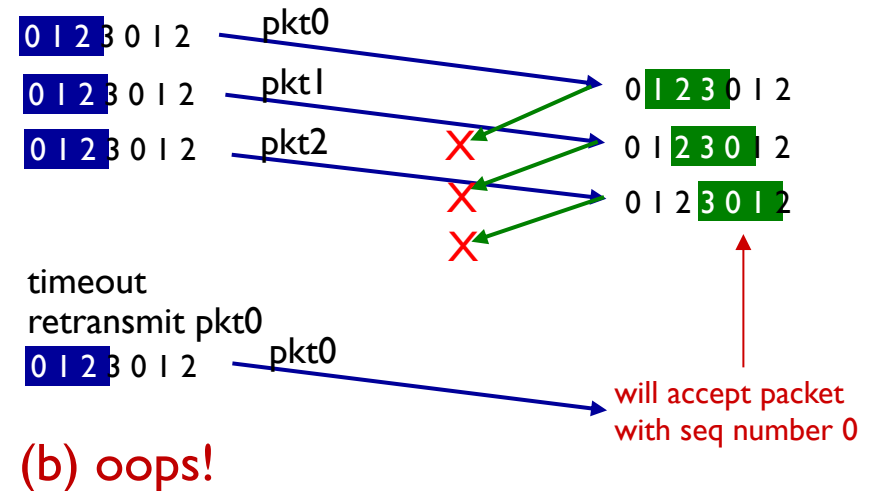
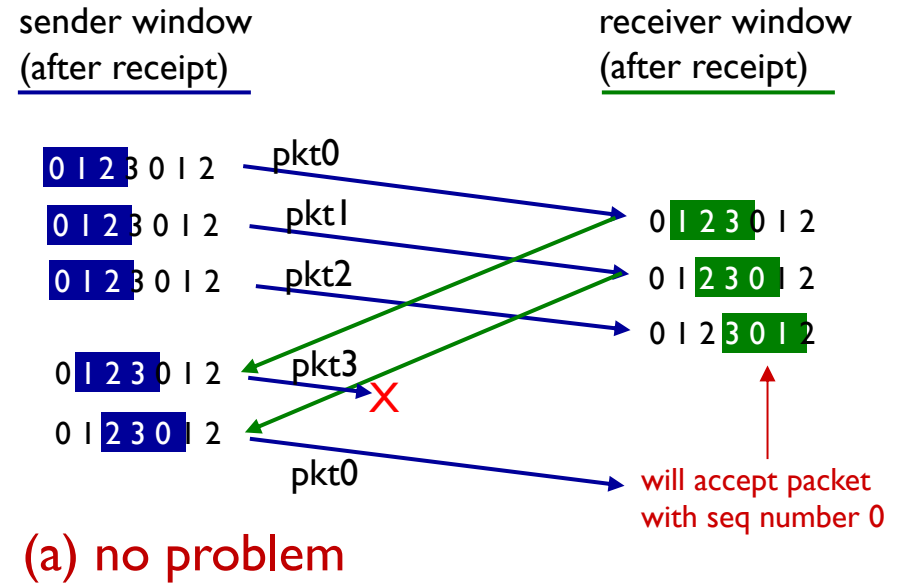
Receiver cannot distinguish 1<sup>st</sup> and 5<sup>th</sup> segment  
because they have the same seq no of 0

# Seq no and window size

example:

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

Why is this happening?





# Seq no and window size

example:

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

WHY is this happening?

sender window  
(after receipt)

0 1 2 3 0 1 2  
pkt0  
0 1 2 3 0 1 2  
pkt1  
0 1 2 3 0 1 2  
pkt2  
0 1 2 3 0 1 2  
pkt3

receiver window  
(after receipt)

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

will accept packet  
with seq number 0

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

0 1 2 3 0 1 2  
pkt2  
0 1 2 3 0 1 2  
pkt0  
0 1 2 3 0 1 2  
pkt0

timeout

retransmit pkt0

will accept packet  
with seq number 0

(b) oops!

# Sequence number with 2 bits

0, 1, 2, 3, 0, 1, 2, 3, ...

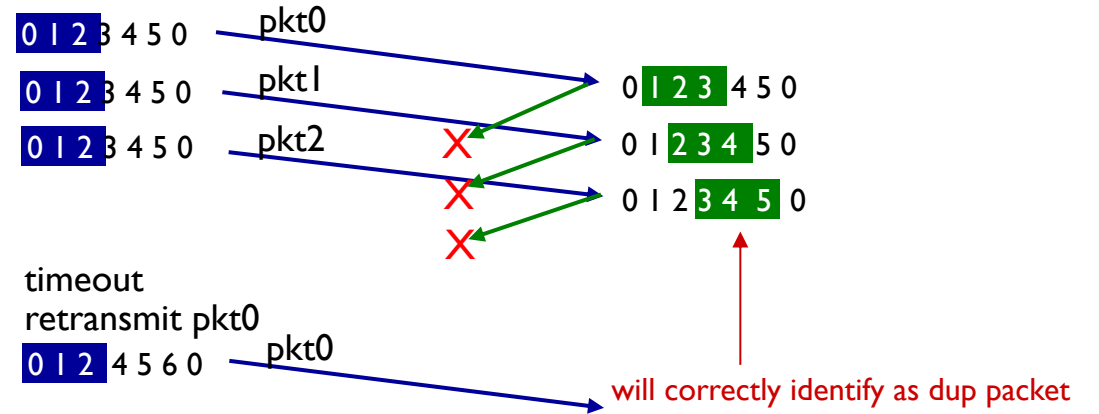
- Sender's retransmission of 1<sup>st</sup> segment falls into receiver's window of 5<sup>th</sup> segment
  - If seq no space is infinite would this ever happen?
- The "highest" seq no in receiver window should NOT overlap with the "lowest" seq no in sender window

Sequence no space should fit entire sender window  
and receiver window WITHOUT overlap!

# Seq no $\geq 2 \times$ window size

example:

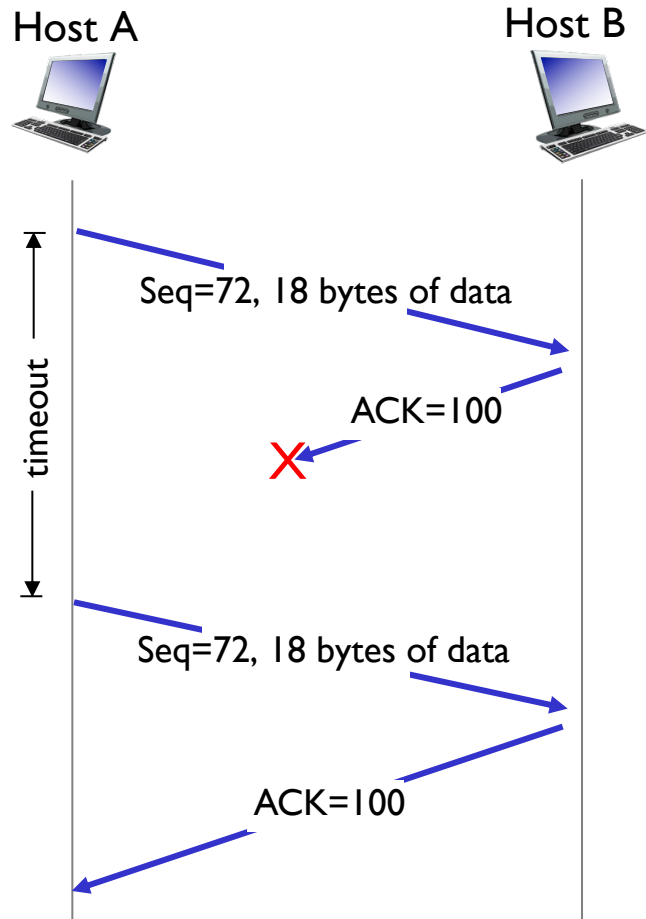
- seq #s: 0, 1, 2, 3, 4, 5
- window size=3



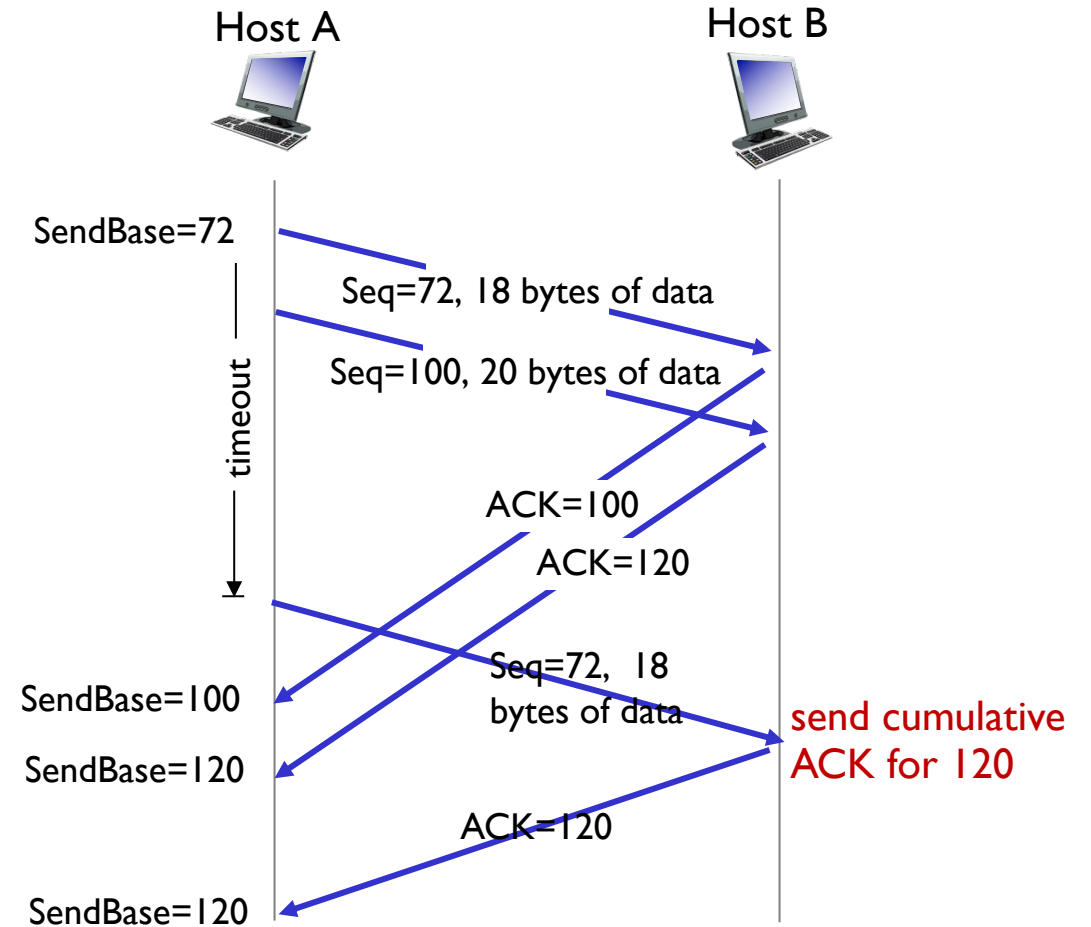
With sufficiently large seq number space,  
sender's window does NOT overlap with receiver's window

# Backup slides

# TCP: retransmission scenarios

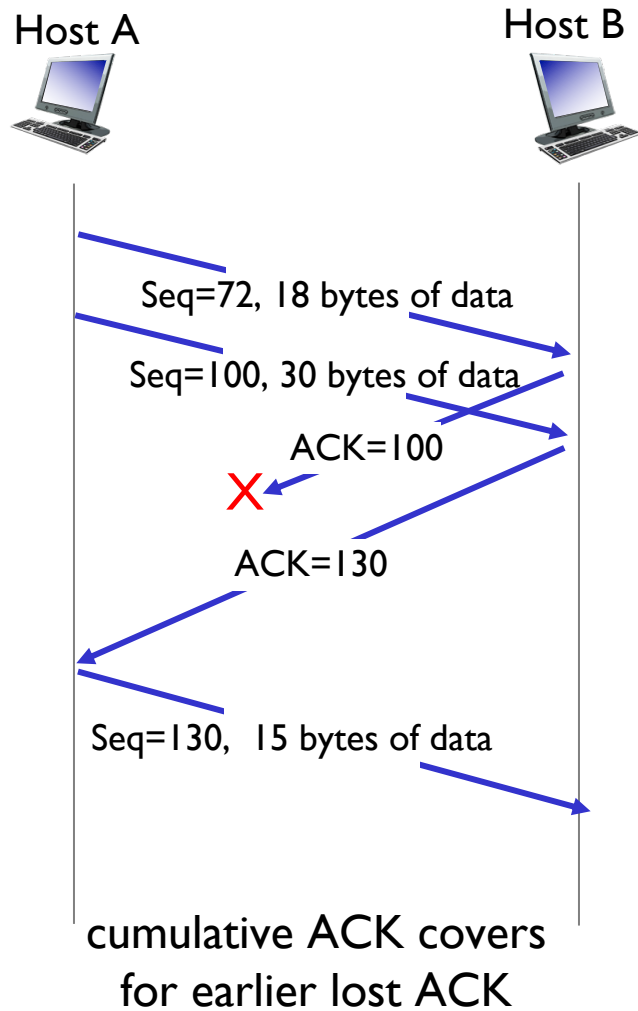


lost ACK scenario



premature timeout

# TCP: retransmission scenarios



# Acknowledgements

Slides are adopted from Kurose' Computer Networking Slides