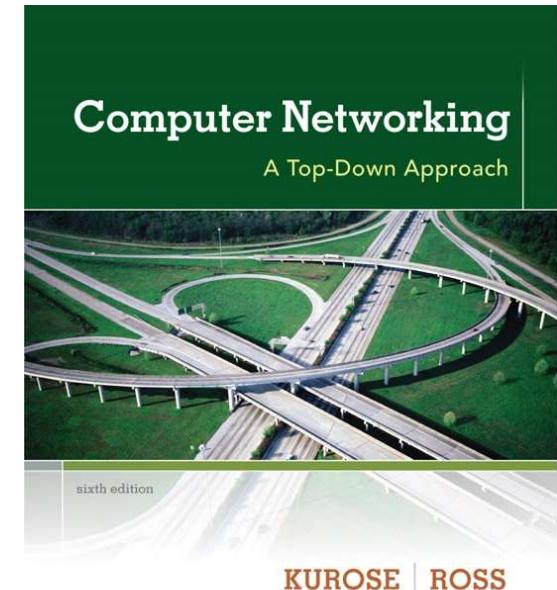


# Chapter 3

## Transport Layer



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

© All material copyright 1996-2013  
© J.F Kurose and K.W. Ross, All Rights Reserved

# Chapter 3: Transport Layer

## our goals:

- ❖ understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- ❖ learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion control

# 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

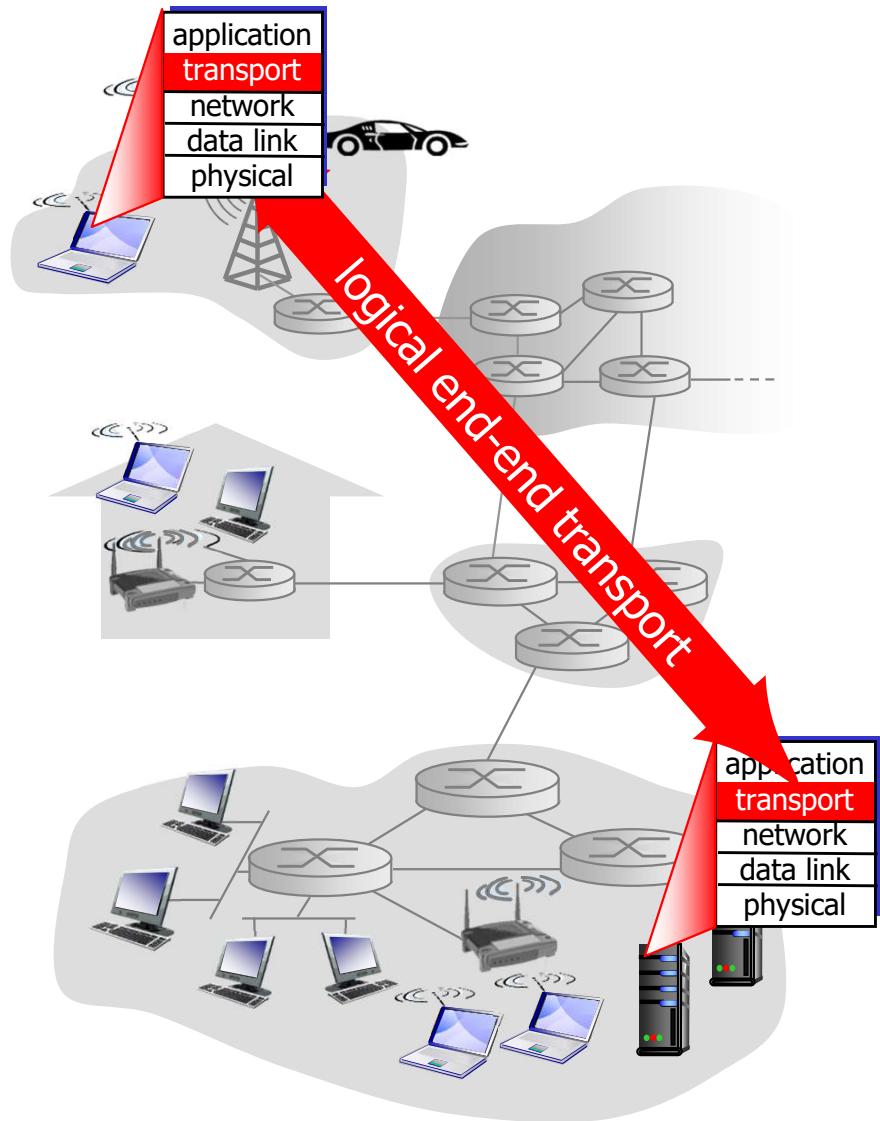
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

# Transport services and protocols

- ❖ provide *logical communication* between app processes running on different hosts
- ❖ transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- ❖ more than one transport protocol available to apps
  - Internet: TCP and UDP



# Transport vs. network layer

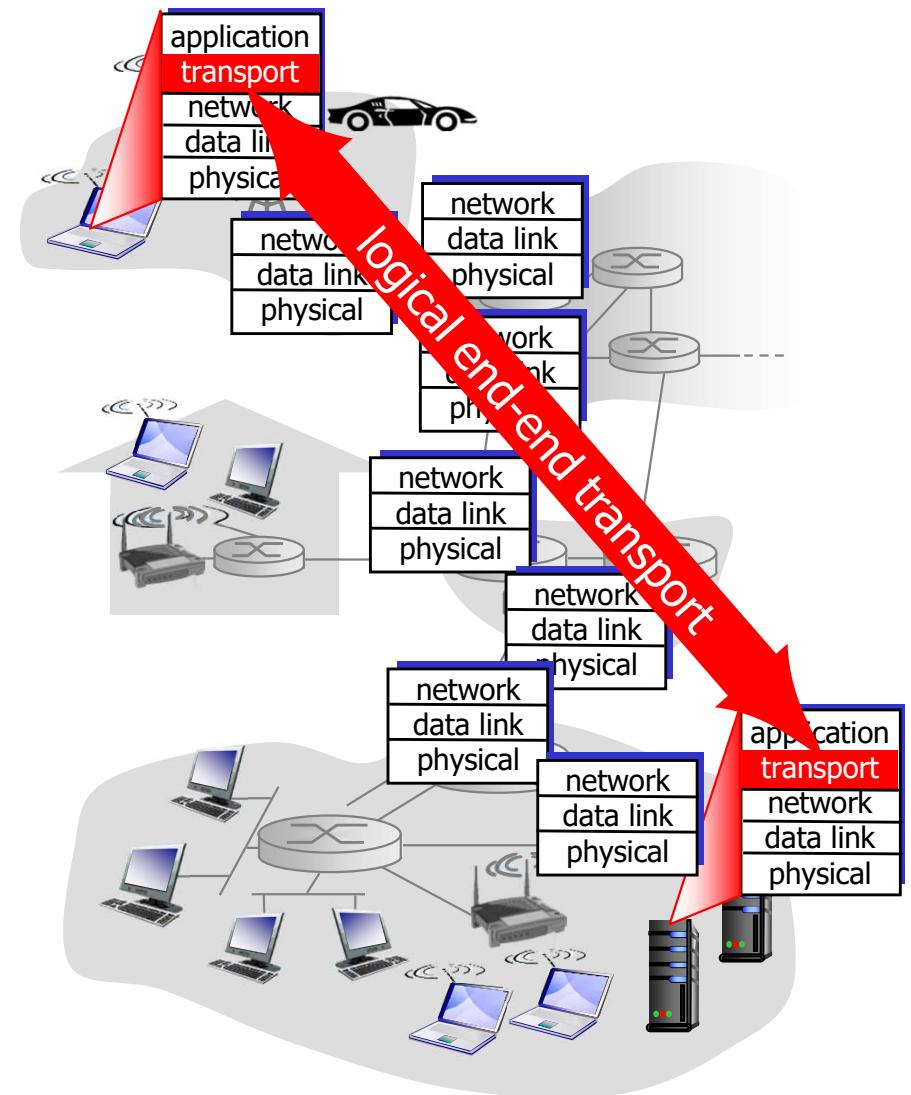
- ❖ *network layer*: logical communication between hosts
- ❖ *transport layer*: logical communication between processes
  - relies on, enhances, network layer services

*household analogy:*

- 12 kids in Ann's house sending letters to 12 kids in Bill's house:*
- ❖ hosts = houses
  - ❖ processes = kids
  - ❖ app messages = letters in envelopes
  - ❖ transport protocol = Ann and Bill who demux to in-house siblings
  - ❖ network-layer protocol = postal service

# Internet transport-layer protocols

- ❖ reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- ❖ unreliable, unordered delivery: UDP
  - no-frills extension of “best-effort” IP
- ❖ services not available:
  - delay guarantees
  - bandwidth guarantees



# 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

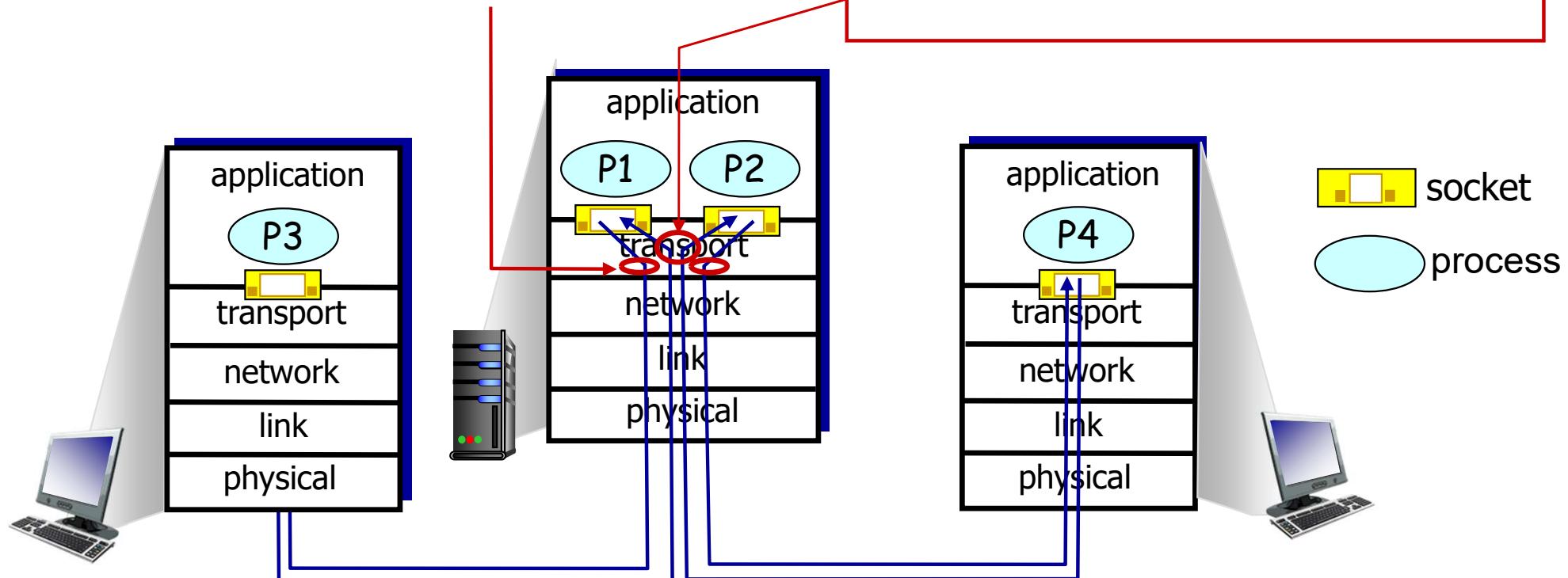
# Multiplexing/demultiplexing

*multiplexing at sender:*

handle data from multiple sockets, add transport header (later used for demultiplexing)

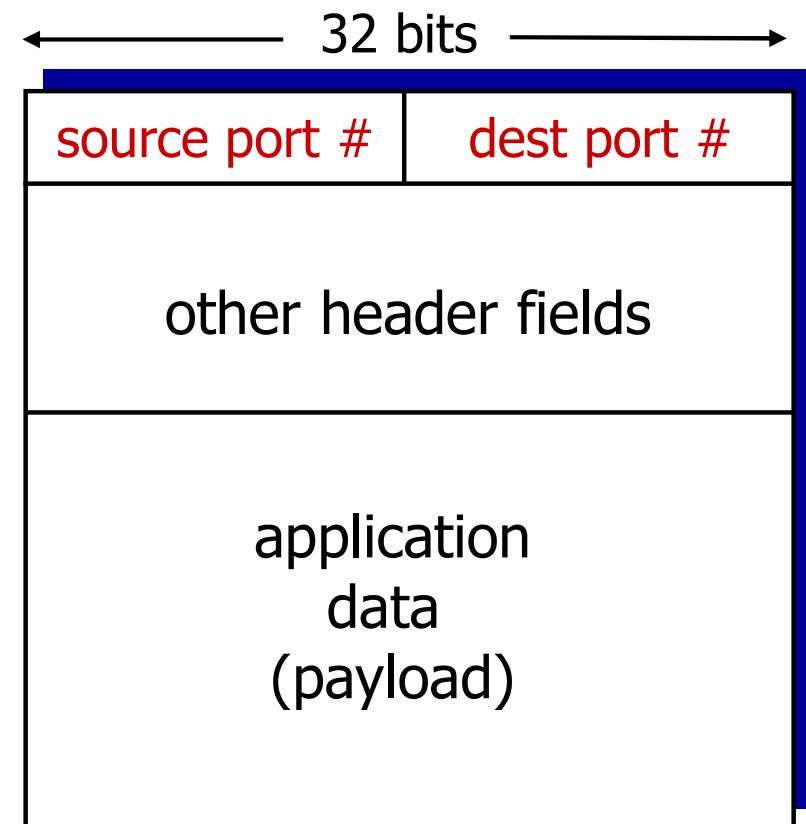
*demultiplexing at receiver:*

use header info to deliver received segments to correct socket



# How demultiplexing works

- ❖ host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries one transport-layer segment
  - each segment has source, destination port number
- ❖ host uses *IP addresses & port numbers* to direct segment to appropriate socket



TCP/UDP segment format

# Connectionless demultiplexing

- ❖ *recall:* created socket has host-local port #:

```
DatagramSocket mySocket1  
= new DatagramSocket(12534);
```

- ❖ *recall:* when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

- ❖ when host receives UDP segment:

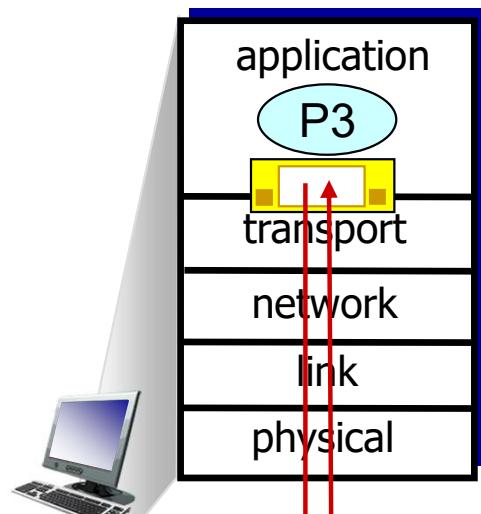
- checks destination port # in segment
- directs UDP segment to socket with that port #



IP datagrams with *same dest. port #*, but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

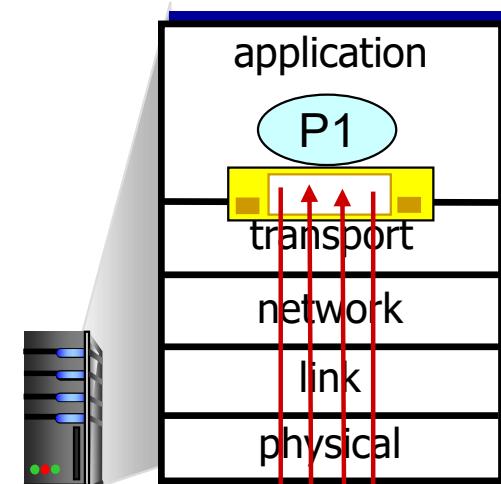
# Connectionless demux: example

```
DatagramSocket  
mySocket2 = new  
DatagramSocket  
(9157);
```



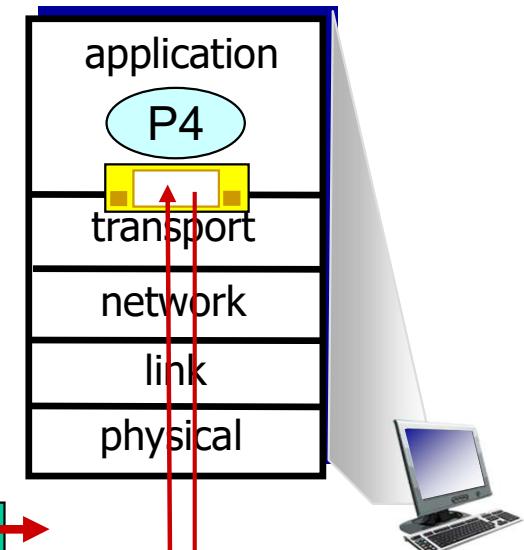
source port: 9157  
dest port: 6428

```
DatagramSocket  
serverSocket = new  
DatagramSocket  
(6428);
```

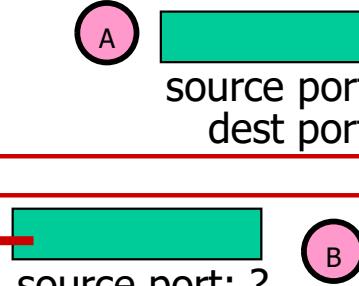


source port: 6428  
dest port: 9157

```
DatagramSocket  
mySocket1 = new  
DatagramSocket  
(5775);
```



source port: ?  
dest port: ?



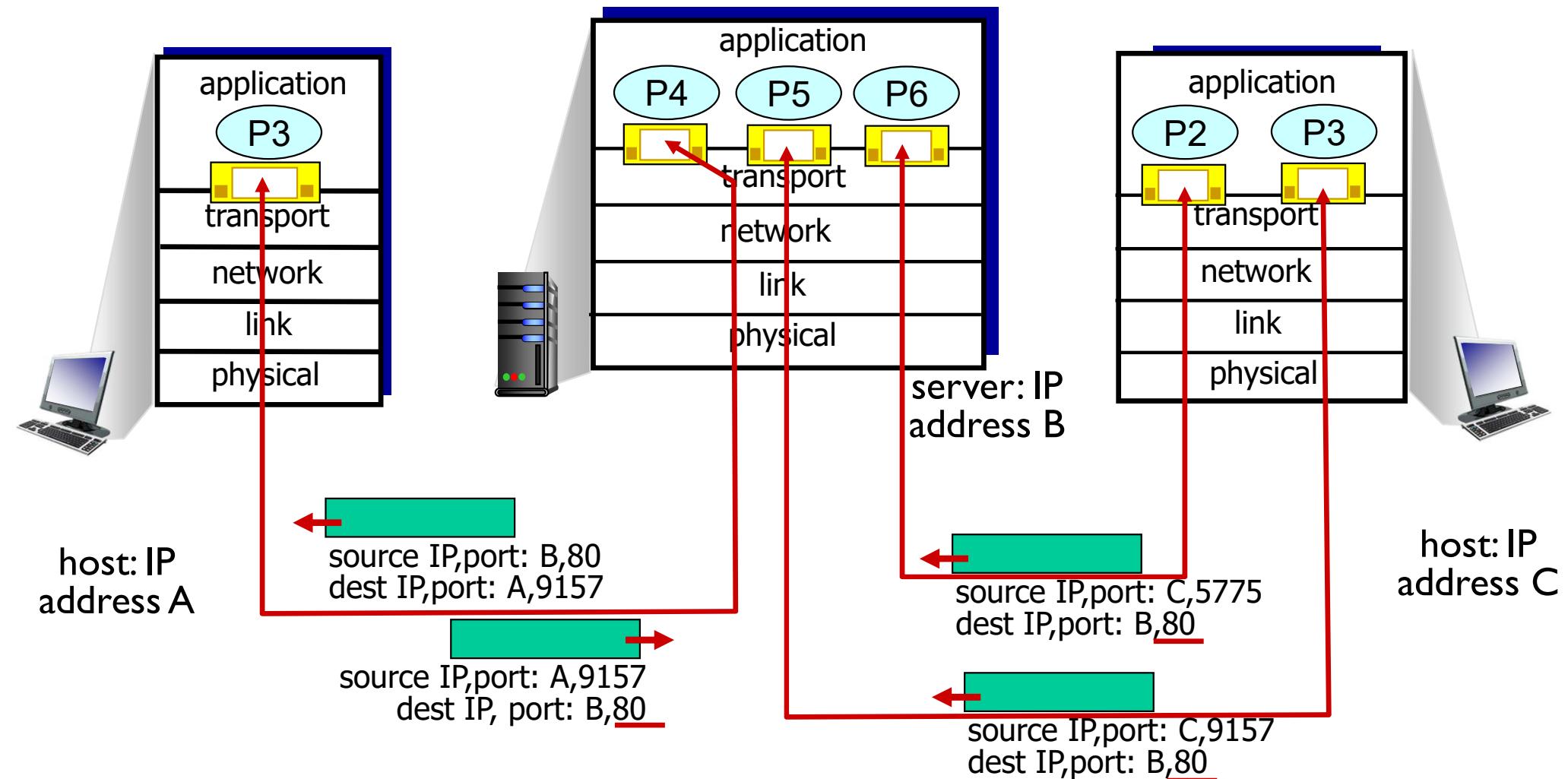
# Exercise : Transport Layer Design

- ❖ จากสไลด์ที่ผ่านมา กำหนดให้ Application ใช้พอร์ตดังนี้
  - P1:8113, P4:2021
- ❖ ให้บอกว่า Source Port และ Destination Port ของ datagram a และ b มีค่าเป็นอะไร

# Connection-oriented demux

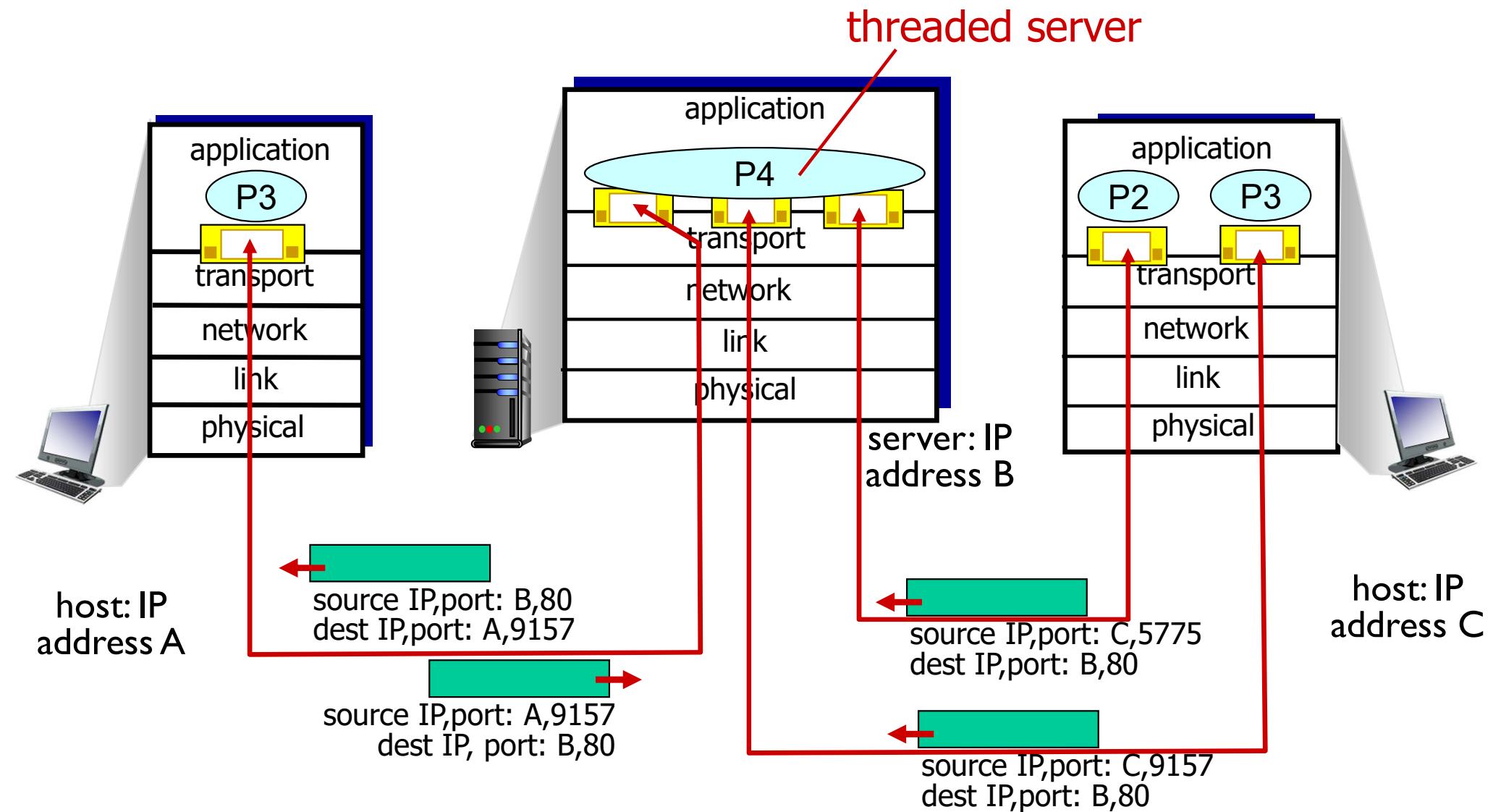
- ❖ TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- ❖ demux: receiver uses all four values to direct segment to appropriate socket
- ❖ server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- ❖ web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

# Connection-oriented demux: example



three segments, all destined to IP address: B,  
dest port: 80 are demultiplexed to *different* sockets

# Connection-oriented demux: example



# Exercise : Planet Transport

- ❖ สมมติว่ามีดาวดวงหนึ่ง ซึ่งทุกคนจะอยู่ในครอบครัวที่มี 6 คน ทุกครอบครัวจะมีบ้าน 1 หลัง แต่ละบ้านจะมีบ้านเลขที่ซึ่งไม่ซ้ำกัน และคนในบ้านจะมีชื่อซึ่งไม่ซ้ำ
- ❖ กำหนดให้ดาวนี้มีบริการส่งจดหมาย โดยจดหมายต้องใส่ช่อง โดยหน้าช่องจะใส่บ้านเลขที่ โดยห้ามเขียนอะไรมากกว่านี้
- ❖ กำหนดให้ 1 บ้านจะมีสมาชิก 1 คนทำหน้าที่ส่งจดหมายไปถึงสมาชิก
- ❖ จะทำอย่างไร ที่จะสามารถส่งจดหมายไปถึงบุคคลที่ต้องการได้ โดยไม่ระบุชื่อคนบนช่อง
- ❖ ตอบใน chat กลุ่มละ 1 โพสต์

# 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

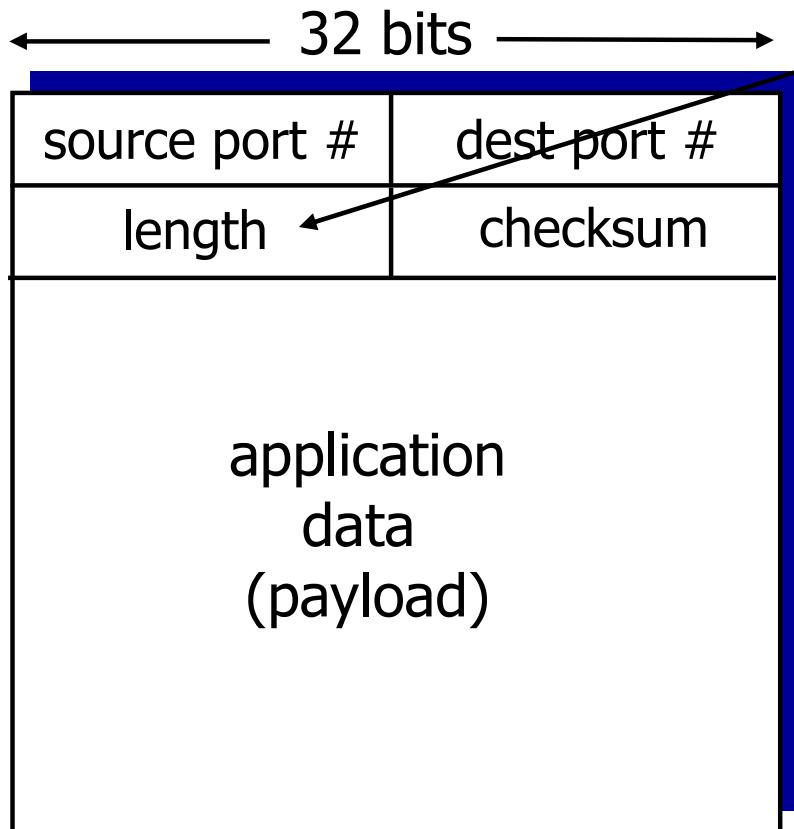
3.6 principles of congestion control

3.7 TCP congestion control

# UDP: User Datagram Protocol [RFC 768]

- ❖ “no frills,” “bare bones” Internet transport protocol
- ❖ “best effort” service, UDP segments may be:
  - lost
  - delivered **out-of-order** to app
- ❖ ***connectionless***:
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others
- ❖ UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- ❖ **reliable transfer over UDP**:
  - add reliability at application layer
  - application-specific error recovery!

# UDP: segment header



UDP segment format

length, in bytes of  
UDP segment,  
including header

## why is there a UDP?

- ❖ no connection establishment (which can add delay)
- ❖ simple: no connection state at sender, receiver
- ❖ small header size
- ❖ no congestion control: UDP can blast away as fast as desired

## Exercise : UDP

- ❖ เป็นไปได้หรือไม่ ที่จะให้ผู้ที่ใช้ UDP สามารถส่งข้อมูลที่ reliable ให้ กันกว่าได้หรือไม่ และต้องทำอะไรเพิ่มเติมหรือไม่
- ❖ ตอบใน chat กลุ่มละ 1 โพสต์

## Exercise : UDP

- ❖ กำหนดให้ Process บน Host C ใช้ ใช้ UDP พอร์ต 6789 สมมติว่า Host A และ B ส่ง UDP ไปยัง Host C พอร์ต 6789 ทั้ง Host A และ B จะส่งไปยัง Socket เดียวกันใน Host C หรือไม่ และ Host C จะรู้ได้อย่างไรว่าข้อมูลไดมาจากการ Host ไหน อธิบาย
- ❖ ตอบใน chat กลุ่มละ 1 โพสต์

# UDP checksum

**Goal:** detect “errors” (e.g., flipped bits) in transmitted segment

## sender:

- ❖ treat segment contents, including header fields, as sequence of 16-bit integers
- ❖ checksum: addition (one's complement sum) of segment contents
- ❖ sender puts checksum value into UDP checksum field

## receiver:

- ❖ compute checksum of received segment
- ❖ check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected.  
*But maybe errors nonetheless? More later*  
....

# Internet checksum: example

example: add two 16-bit integers

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
<hr/>																
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
	<hr/>															
sum	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	1	1	

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

# 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

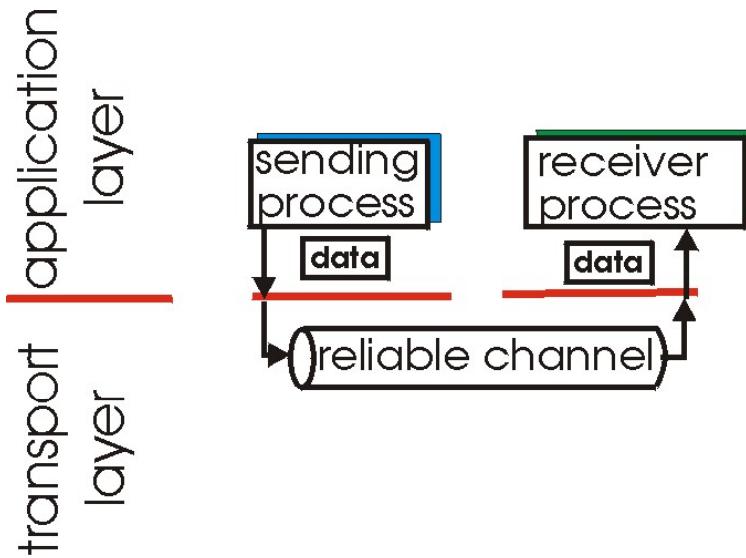
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

# Principles of reliable data transfer

- ❖ important in application, transport, link layers
  - top-10 list of important networking topics!

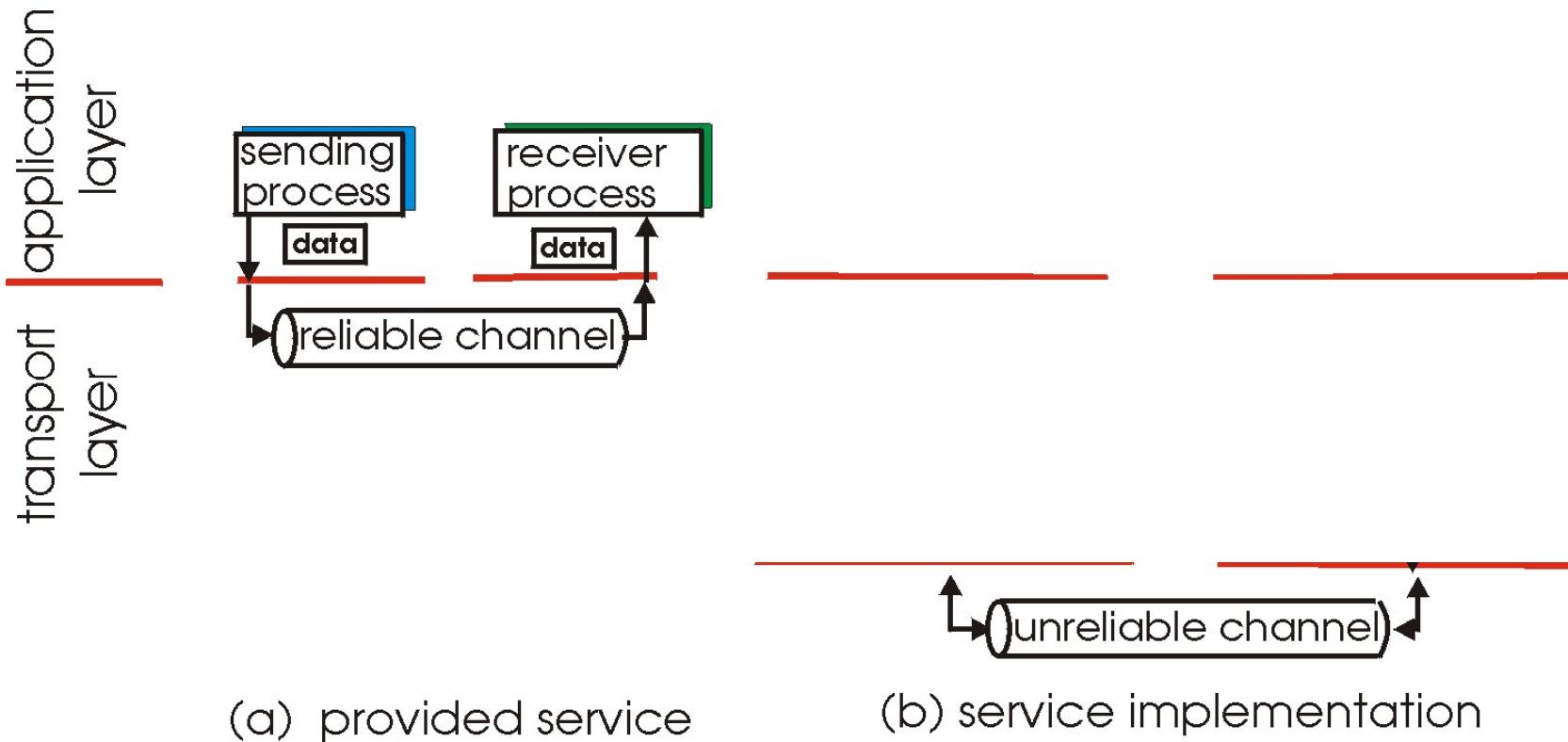


(a) provided service

- ❖ characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

# Principles of reliable data transfer

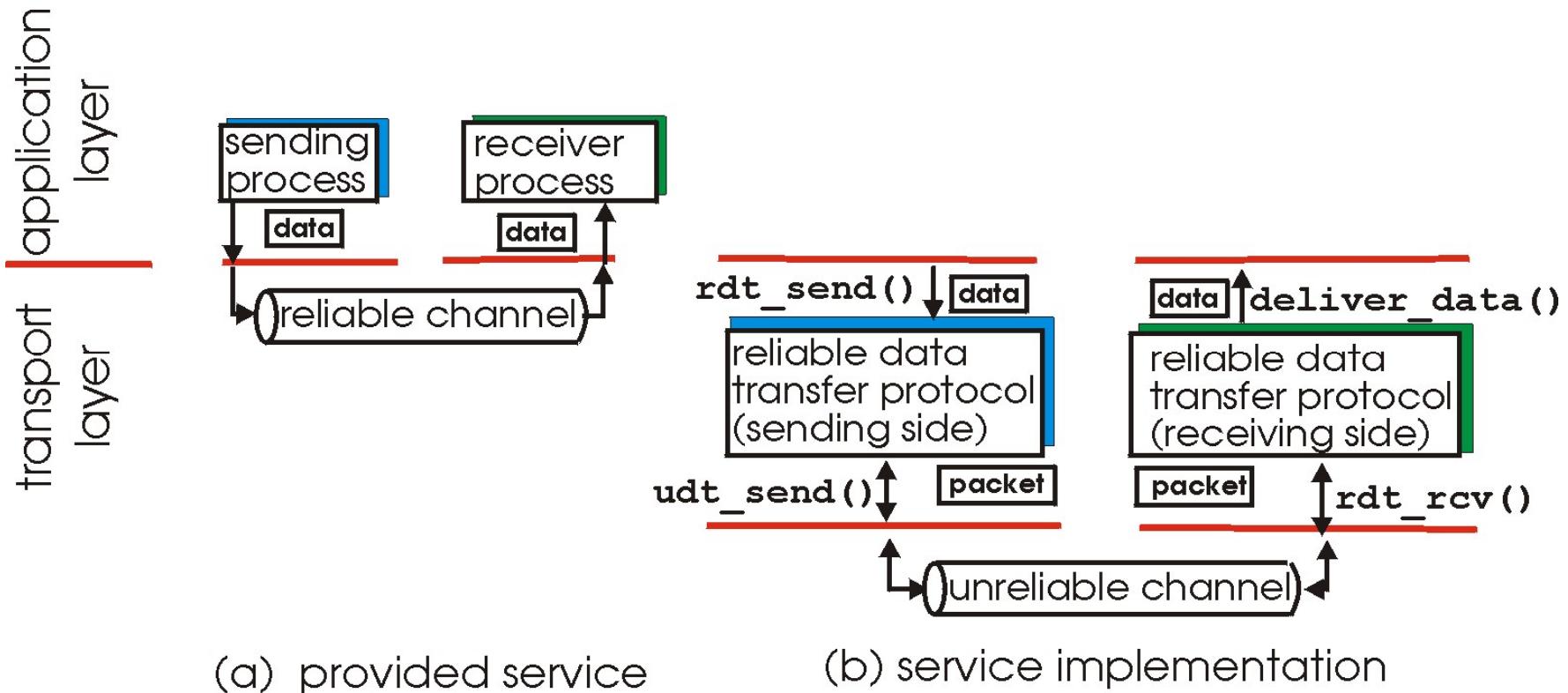
- ❖ important in application, transport, link layers
  - top-10 list of important networking topics!



- ❖ characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

# Principles of reliable data transfer

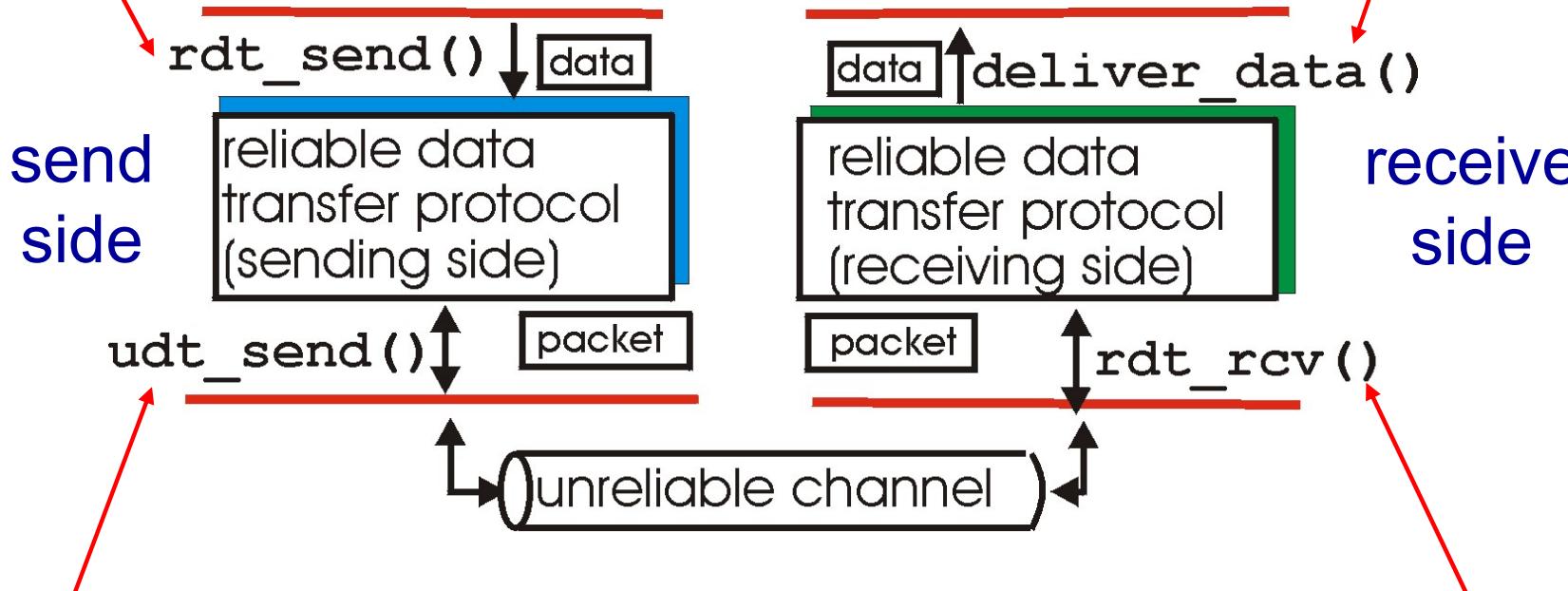
- ❖ important in application, transport, link layers
  - top-10 list of important networking topics!



- ❖ characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

# Reliable data transfer: getting started

**rdt\_send()** : called from above, (e.g., by app.). Passed data to deliver to receiver upper layer



**udt\_send()** : called by rdt, to transfer packet over unreliable channel to receiver

**deliver\_data()** : called by **rdt** to deliver data to upper

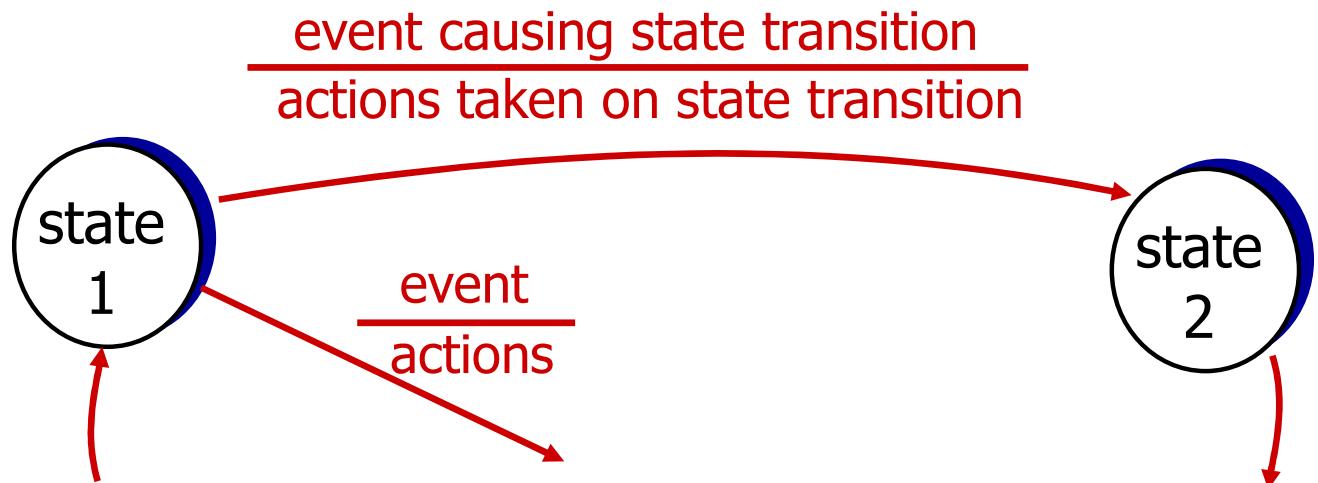
**rdt\_rcv()** : called when packet arrives on rcv-side of channel

# Reliable data transfer: getting started

we'll:

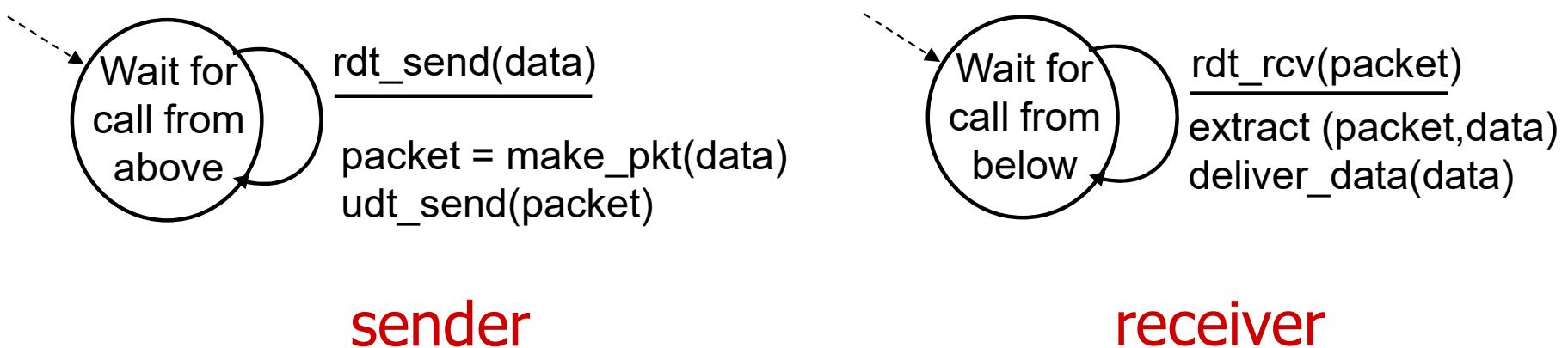
- ❖ incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- ❖ consider only unidirectional data transfer
  - but control info will flow on both directions!
- ❖ use finite state machines (FSM) to specify sender, receiver

**state:** when in this “state” next state uniquely determined by next event



# rdt1.0: reliable transfer over a reliable channel

- ❖ underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- ❖ separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel



# rdt2.0: channel with bit errors

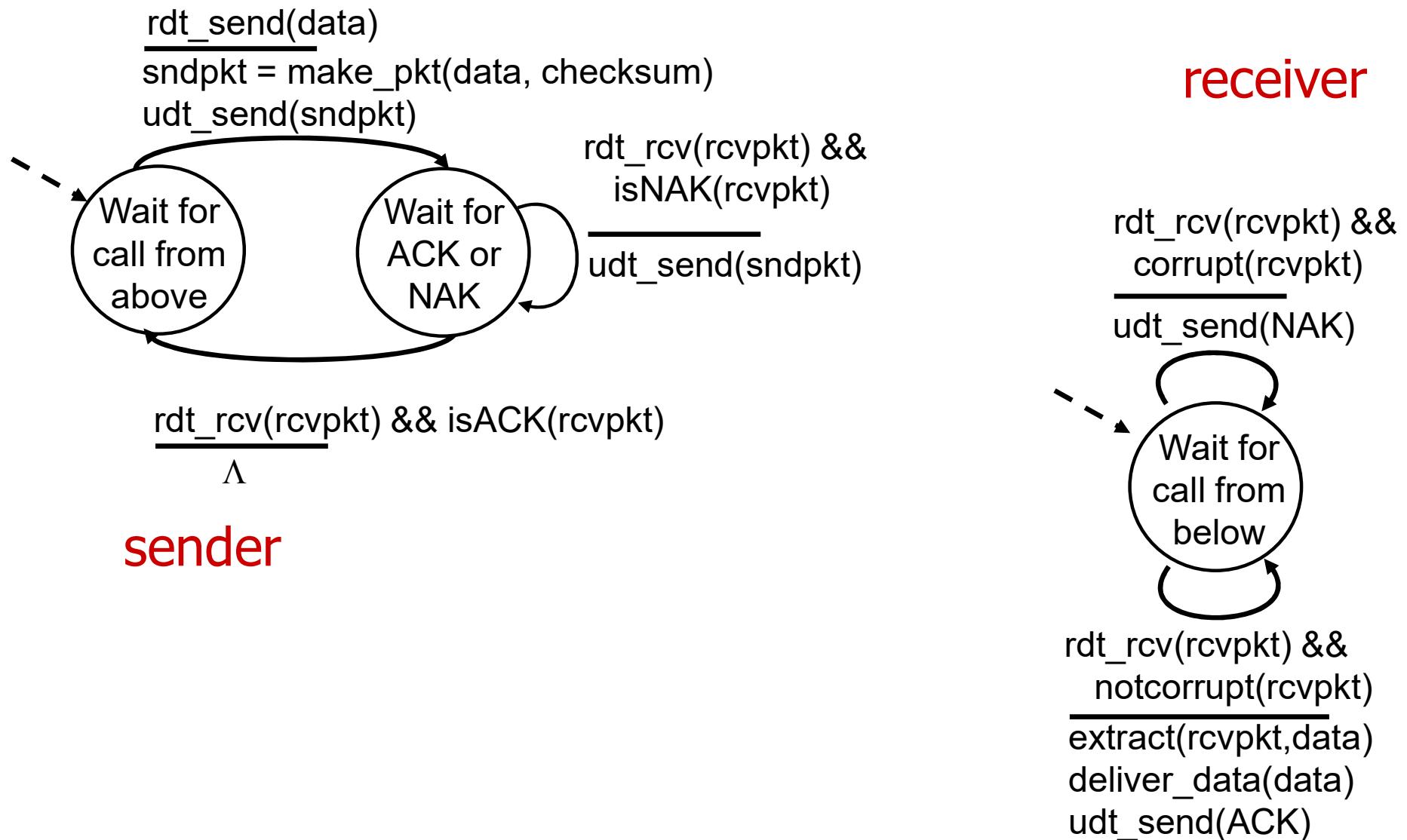
- ❖ underlying channel may flip bits in packet
  - checksum to detect bit errors
- ❖ *the question: how to recover from errors:*

*How do humans recover from “errors”  
during conversation?*

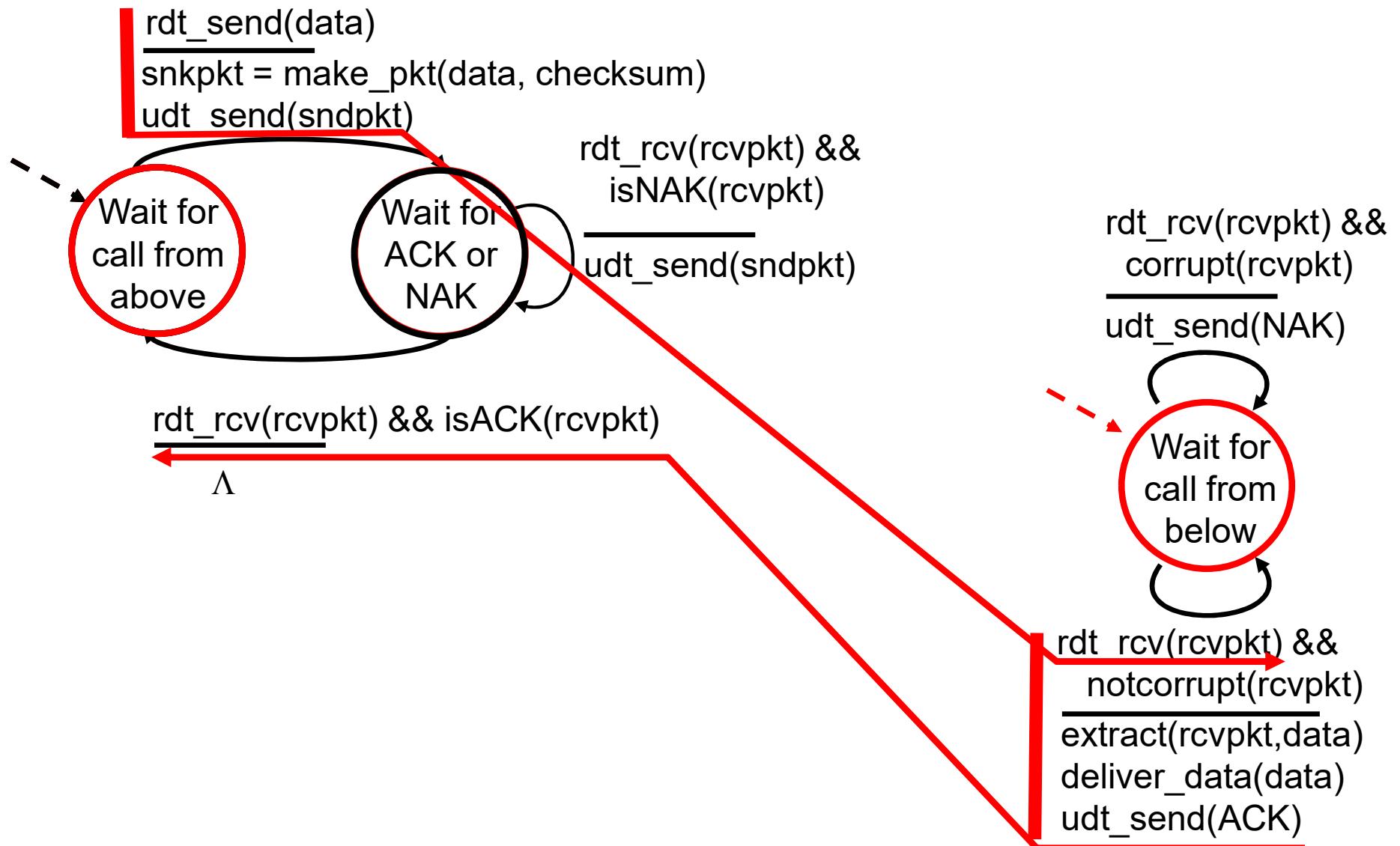
# rdt2.0: channel with bit errors

- ❖ underlying channel may flip bits in packet
  - checksum to detect bit errors
- ❖ *the question: how to recover from errors:*
  - **acknowledgements (ACKs):** receiver explicitly tells sender that pkt received OK
  - **negative acknowledgements (NAKs):** receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
- ❖ new mechanisms in `rdt2.0` (beyond `rdt1.0`):
  - error detection
  - feedback: control msgs (ACK,NAK) from receiver to sender

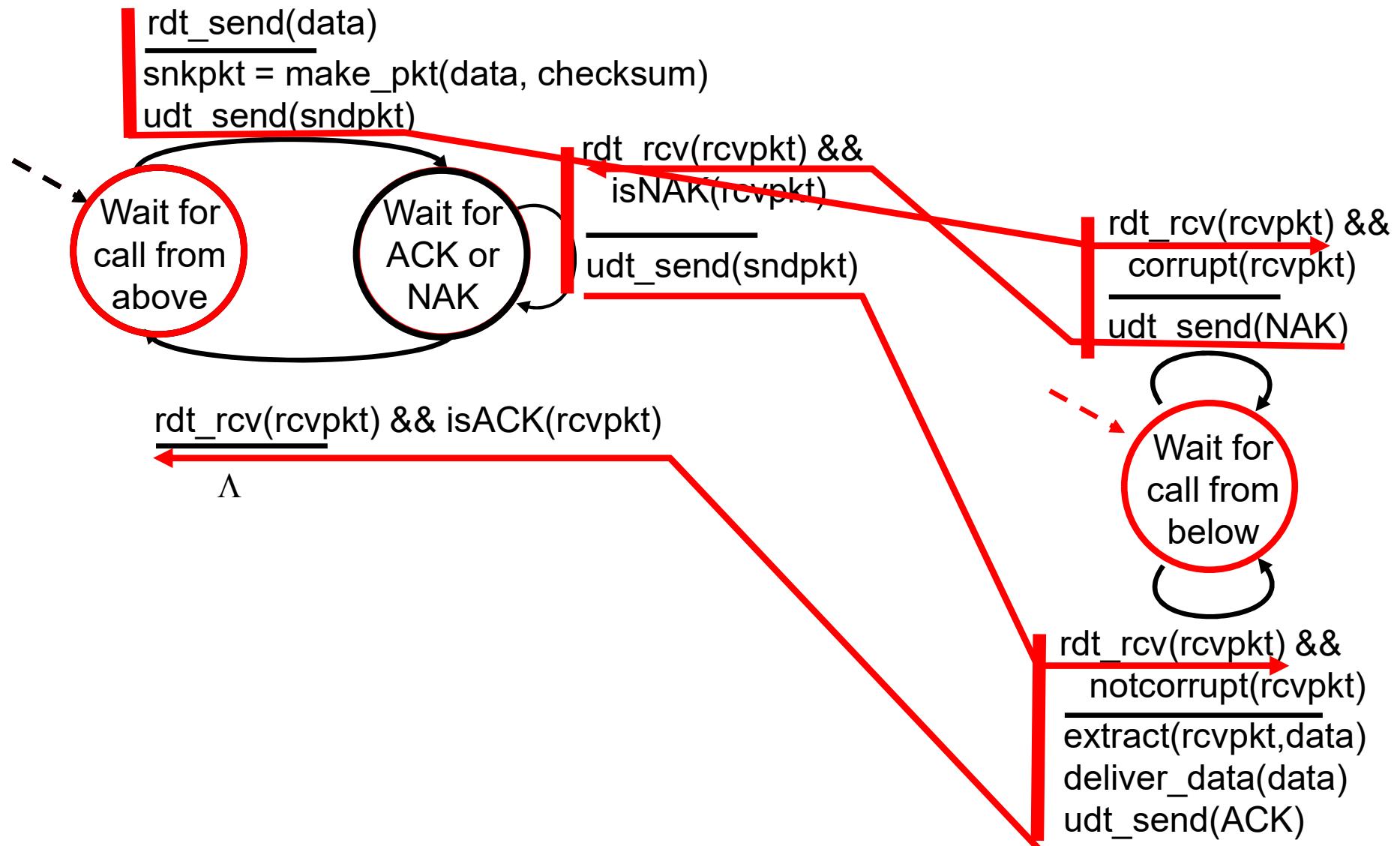
# rdt2.0: FSM specification



# rdt2.0: operation with no errors



# rdt2.0: error scenario



# rdt2.0 has a fatal flaw!

## what happens if ACK/NAK corrupted?

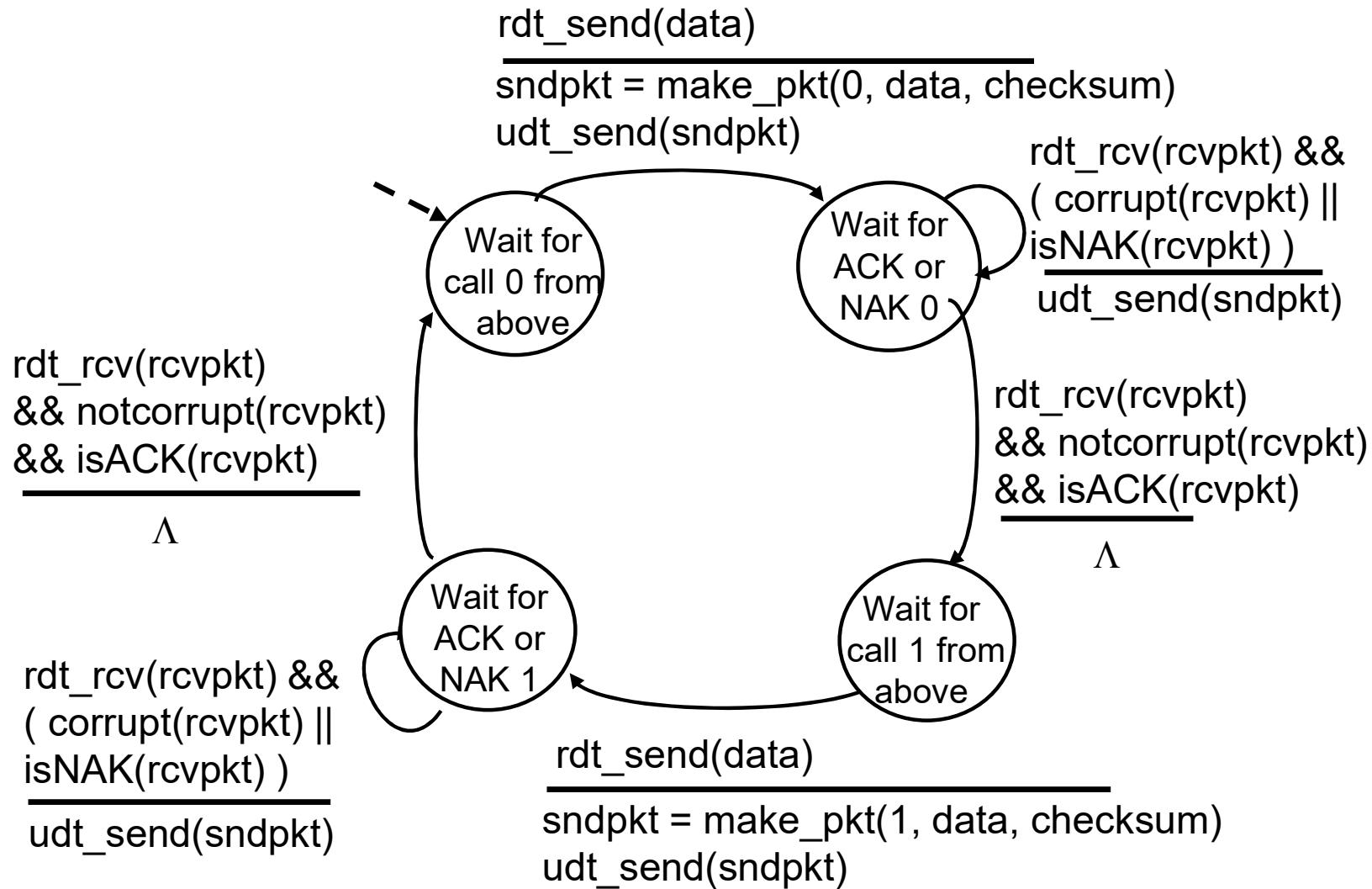
- ❖ sender doesn't know what happened at receiver!
- ❖ can't just retransmit: possible duplicate

## handling duplicates:

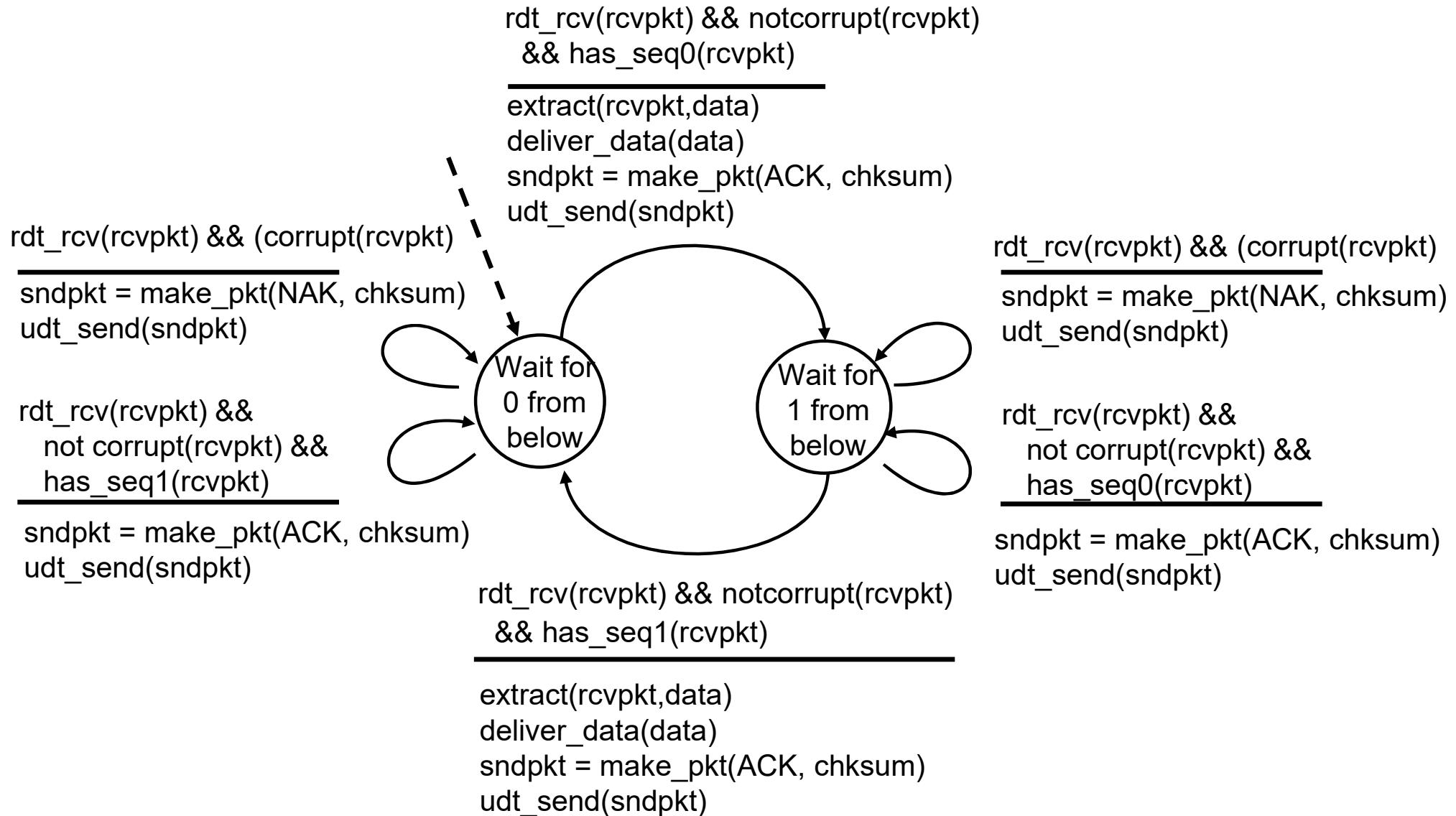
- ❖ sender retransmits current pkt if ACK/NAK corrupted
- ❖ sender adds *sequence number* to each pkt
- ❖ receiver discards (doesn't deliver up) duplicate pkt

stop and wait  
sender sends one packet,  
then waits for receiver  
response

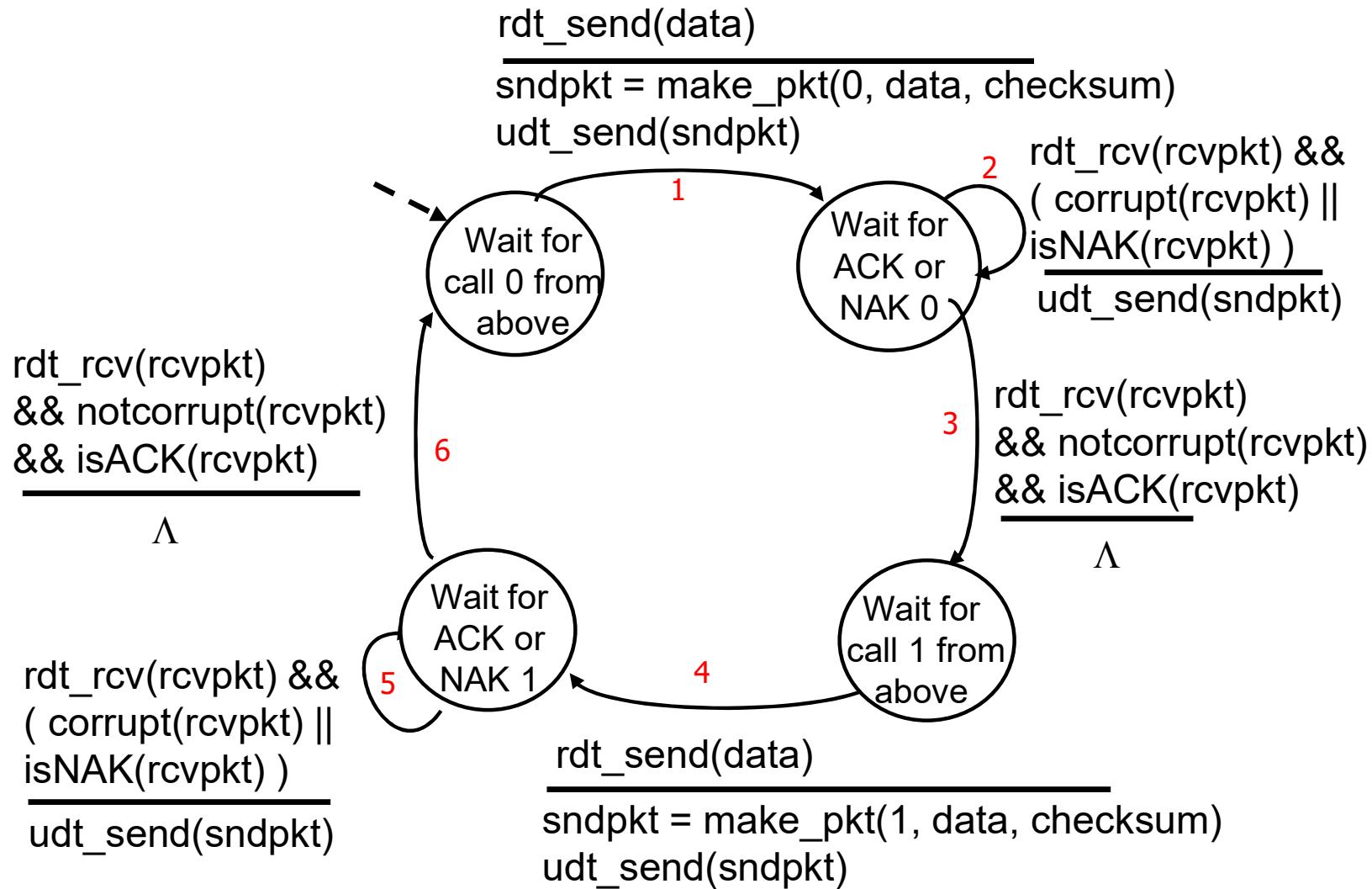
# rdt2.1: sender, handles garbled ACK/NAKs



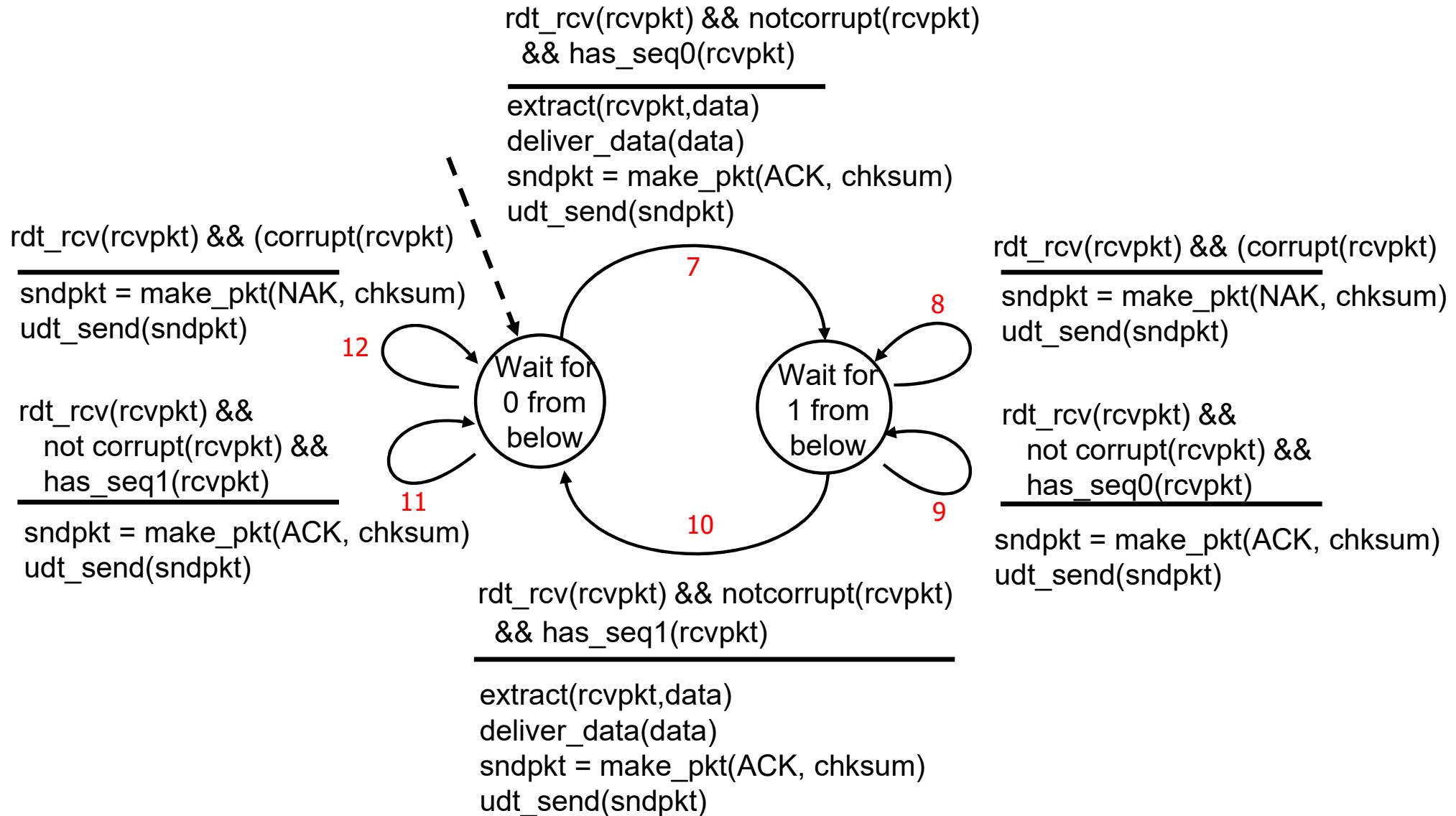
# rdt2.1: receiver, handles garbled ACK/NAKs



# rdt2.1: sender, handles garbled ACK/NAKs



# rdt2.1: receiver, handles garbled ACK/NAKs



## Exercise : rdt2.1

- ❖ จาก Slide ที่ 39 และ 40 ให้เขียนลำดับ event ที่เกิดขึ้น ทั้งส่งและรับ
  - กรณี ส่งสำเร็จ แต่ข้อมูลเสียหาย
  - กรณี ส่งสำเร็จ ข้อมูลไม่เสียหาย แต่ Ack เสียหาย
  - กรณี ส่งสำเร็จ ข้อมูลไม่เสียหาย และ Ack ไม่เสียหาย
  - กรณี ส่ง 2 ครั้ง ครั้งแรกข้อมูลครบ Ack ครบ ครั้งที่ 2 ข้อมูลเสียหาย
  - กรณี ส่ง 2 ครั้ง ครั้งแรกข้อมูลครบ Ack ครบ ครั้งที่ 2 ข้อมูลครบ Ack เสียหาย
- ❖ ให้ทำทุกคน แล้วเปรียบเทียบกัน ถ้าตรงกันจึงส่งในชื่อกลุ่ม กลุ่มละ 1 โพสต์ ใน chat

# rdt2.1: discussion

## sender:

- ❖ seq # added to pkt
- ❖ two seq. #'s (0,1) will suffice. Why?
- ❖ must check if received ACK/NAK corrupted
- ❖ twice as many states
  - state must “remember” whether “expected” pkt should have seq # of 0 or 1

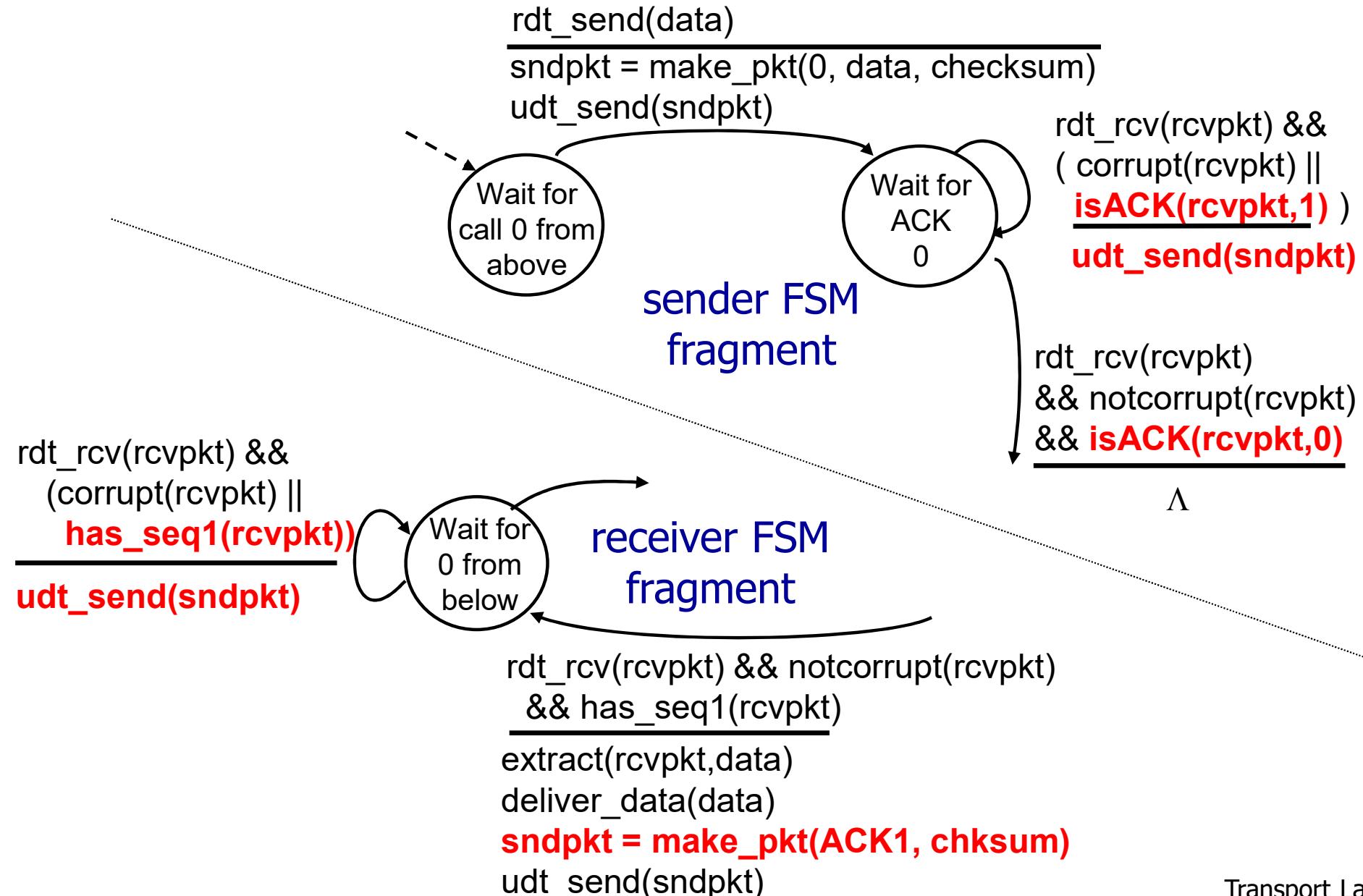
## receiver:

- ❖ must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- ❖ note: receiver can *not* know if its last ACK/NAK received OK at sender

## rdt2.2: a NAK-free protocol

- ❖ same functionality as rdt2.1, using ACKs only
- ❖ instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed
- ❖ duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

# rdt2.2: sender, receiver fragments



# rdt3.0: channels with errors and loss

## new assumption:

underlying channel can  
also lose packets  
(data, ACKs)

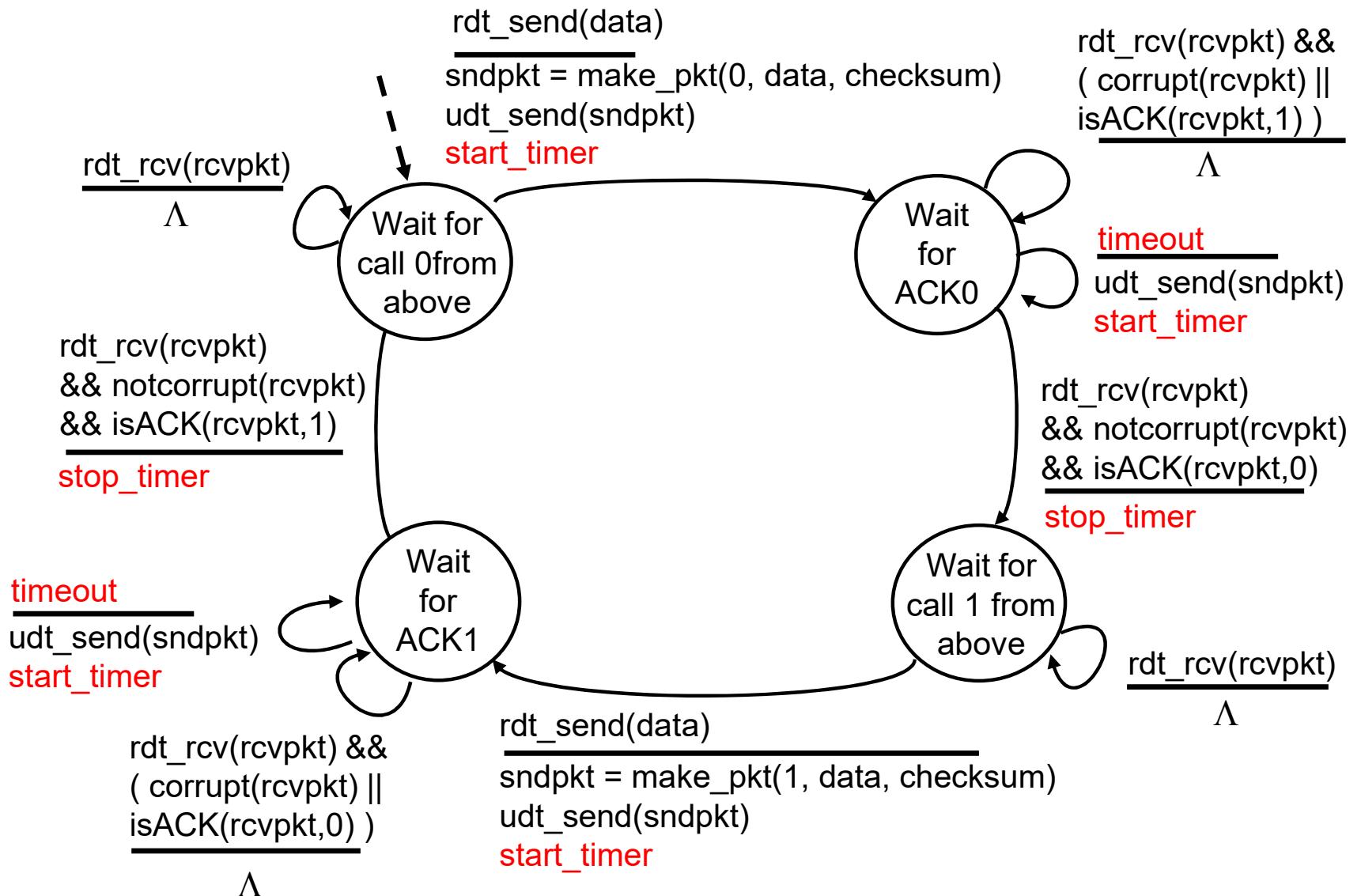
- checksum, seq. #,  
ACKs, retransmissions  
will be of help ... but  
not enough

## approach: sender waits

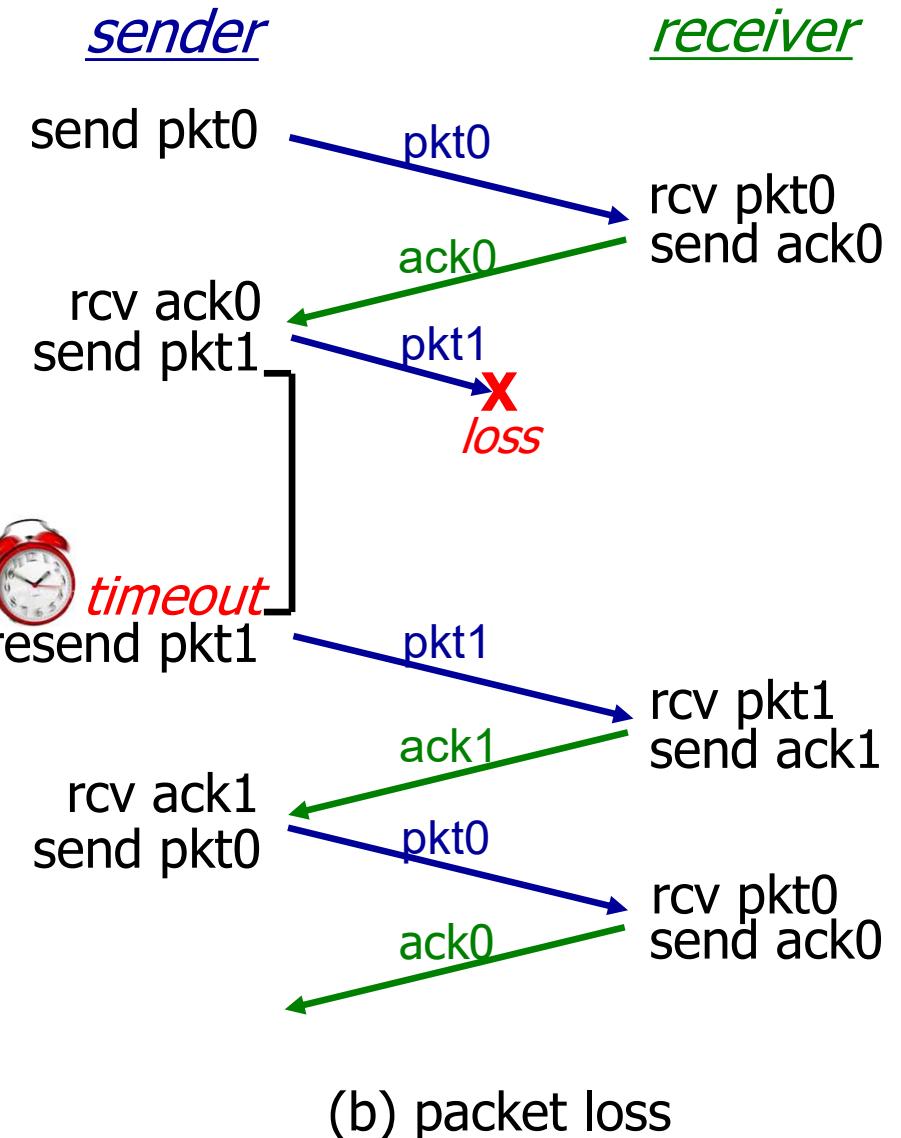
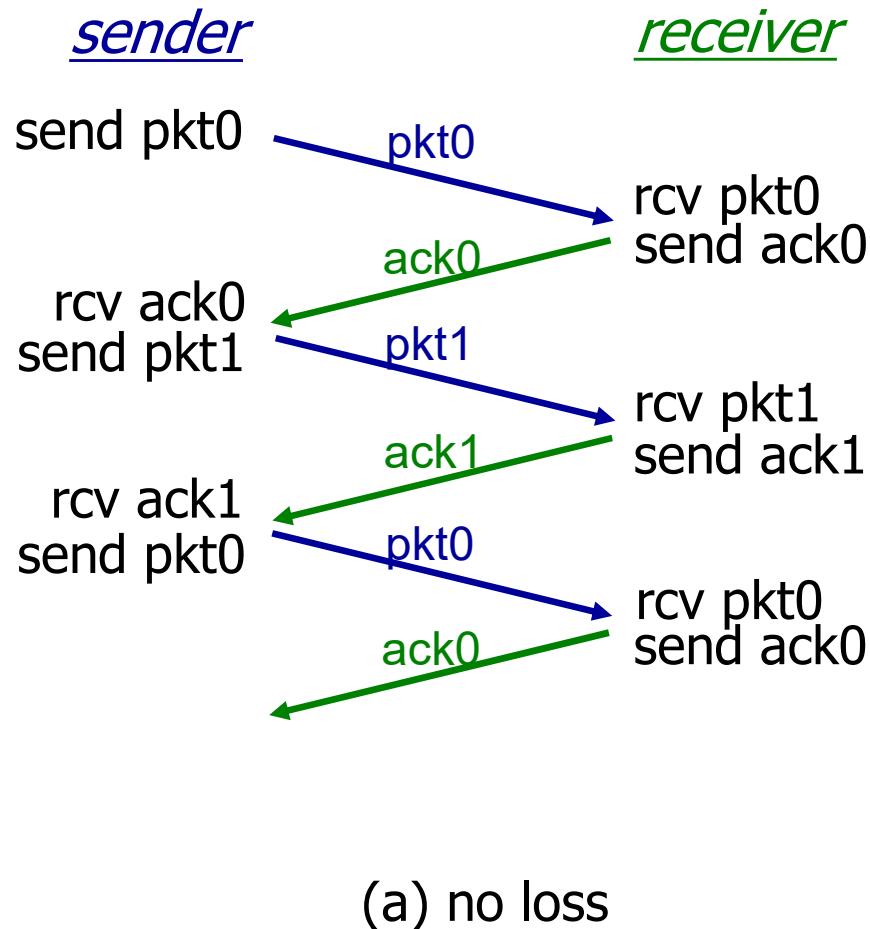
“reasonable” amount of  
time for ACK

- ❖ retransmits if no ACK  
received in this time
- ❖ if pkt (or ACK) just delayed  
(not lost):
  - retransmission will be  
duplicate, but seq. #'s  
already handles this
  - receiver must specify seq  
# of pkt being ACKed
- ❖ requires countdown timer

# rdt3.0 sender

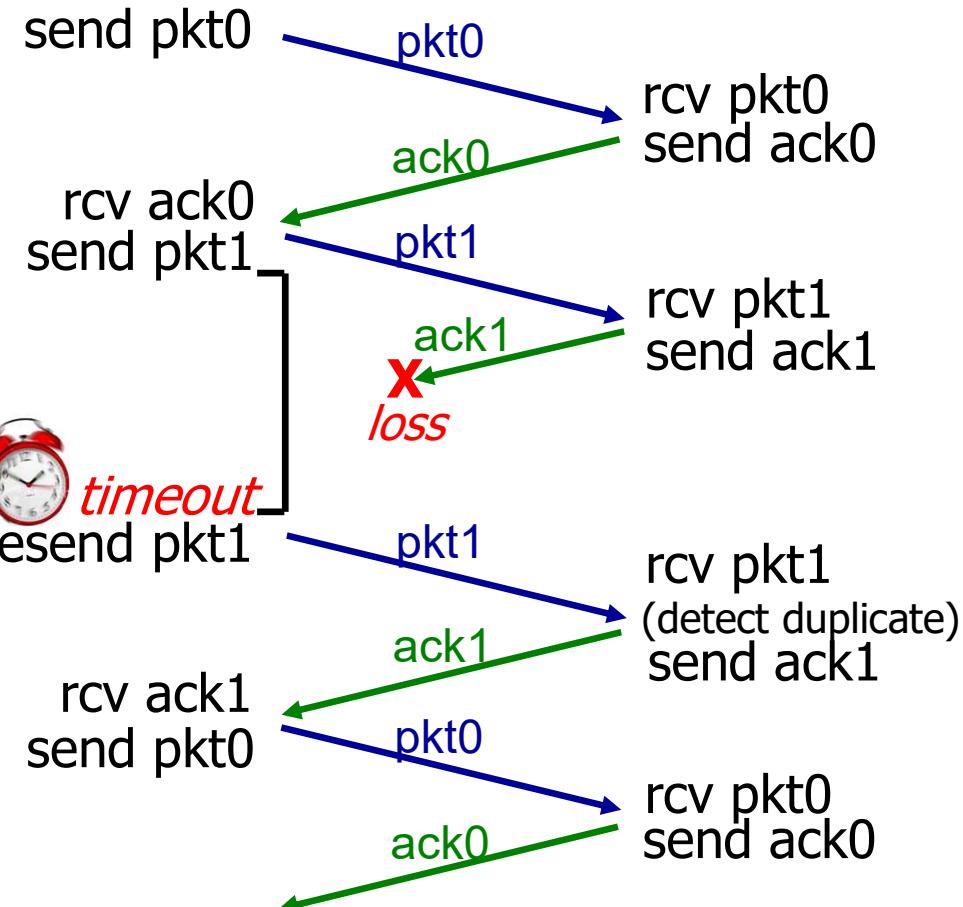


# rdt3.0 in action



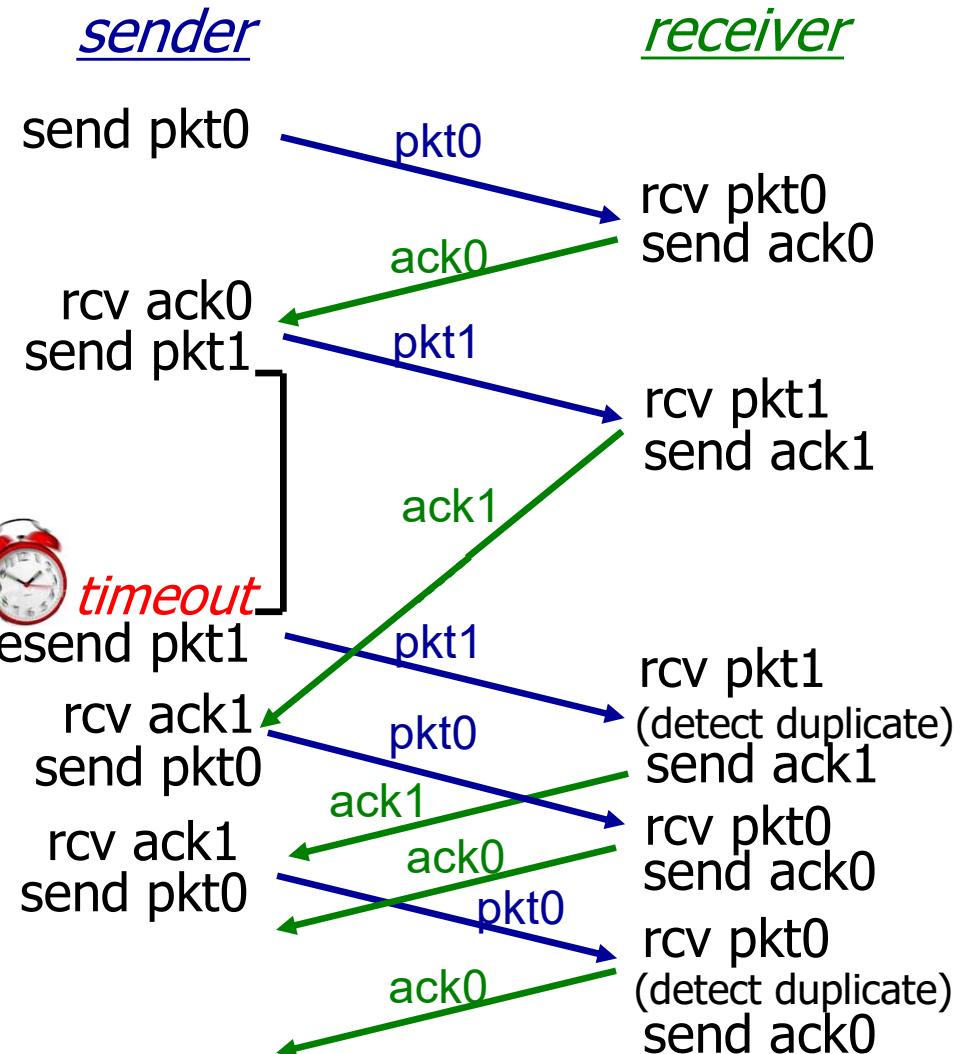
# rdt3.0 in action

sender



(c) ACK loss

receiver



(d) premature timeout/ delayed ACK

## Exercise : rdt 3.0

- ❖ กำหนดให้ roundtrip delay ระหว่าง sender กับ receiver มีค่าคงที่และฟัง sender รู้ค่านี้ ใน rdt 3.0 ยังจำเป็นต้องมี timer หรือไม่ โดย Packet สามารถ lost ได้ จงอธิบาย

# Performance of rdt3.0

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

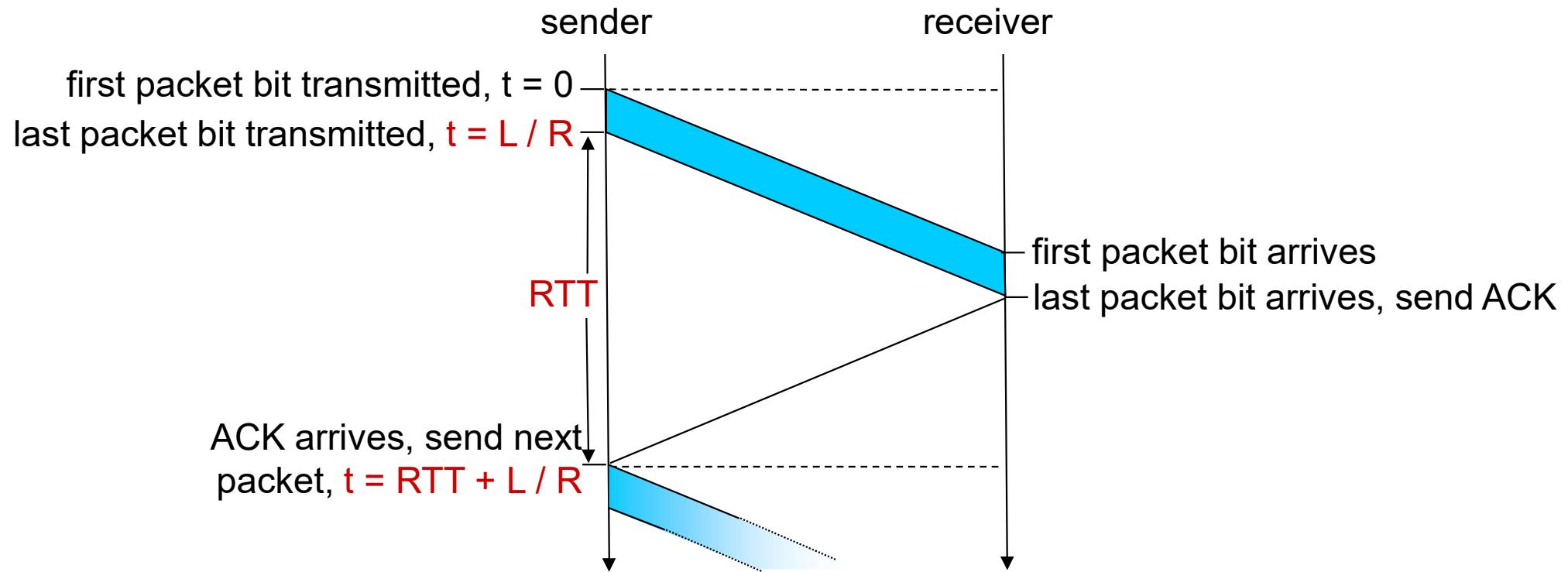
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

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

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

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

# rdt3.0: stop-and-wait operation

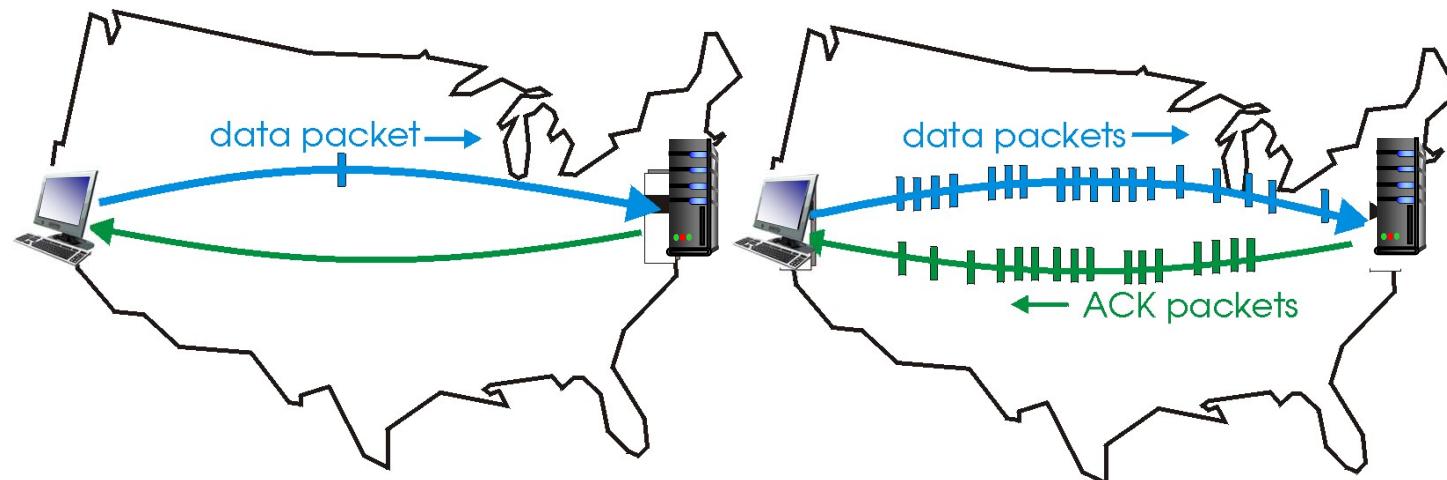


$$U_{\text{sender}} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

# 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

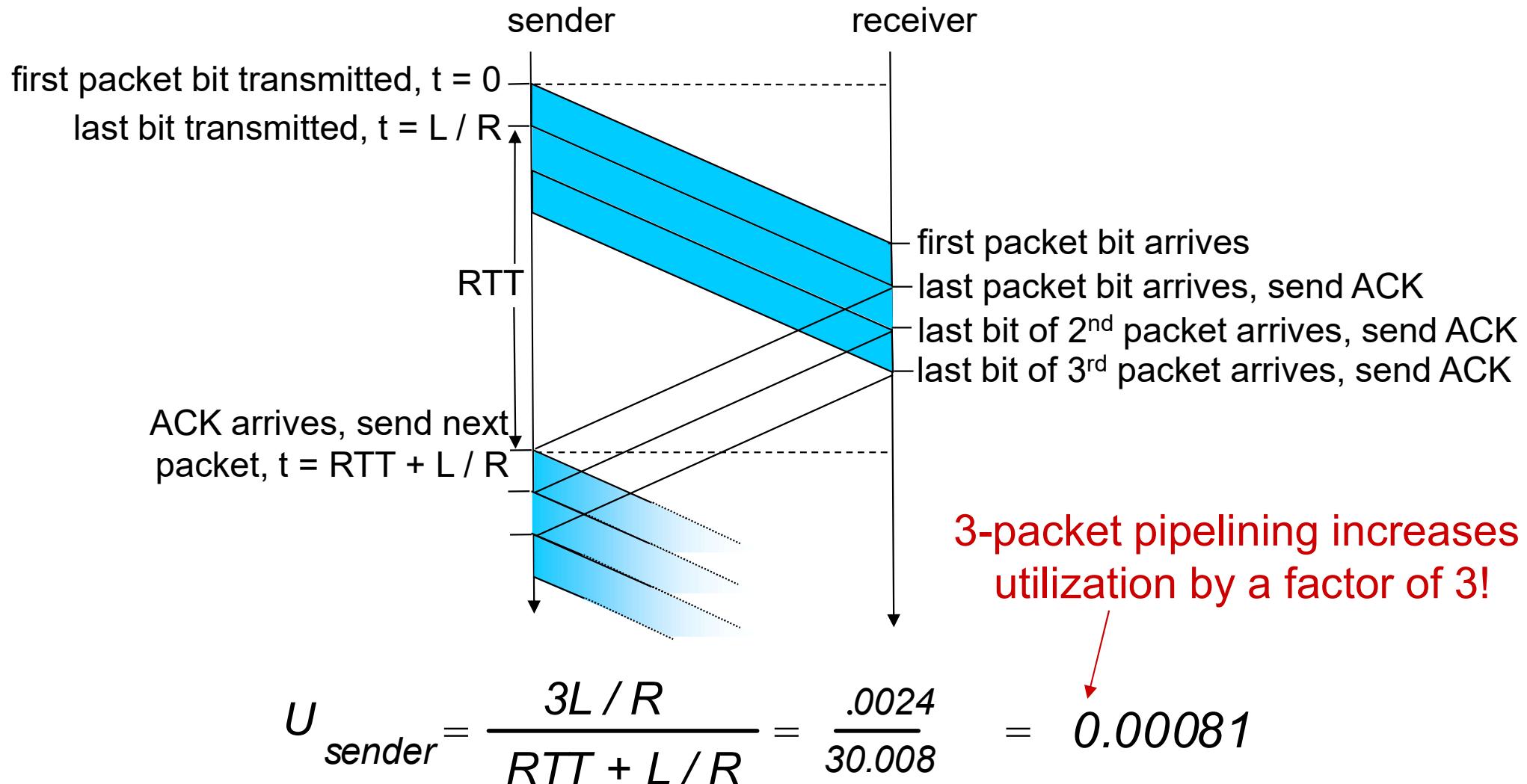


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

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

# Pipelining: increased utilization



# Pipelined protocols: overview

## Go-back-N:

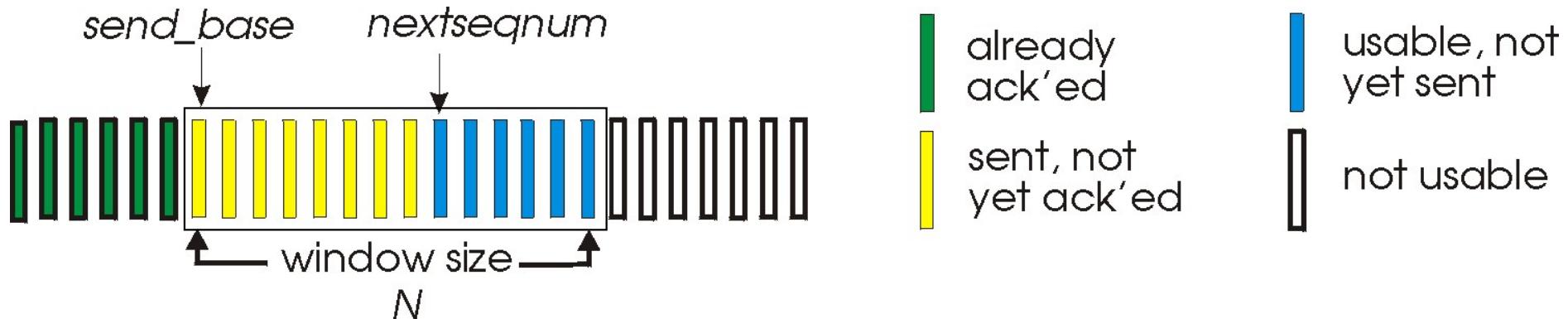
- ❖ sender can have up to N unacked packets in pipeline
- ❖ receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap
- ❖ sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

## Selective Repeat:

- ❖ sender can have up to N unacked 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

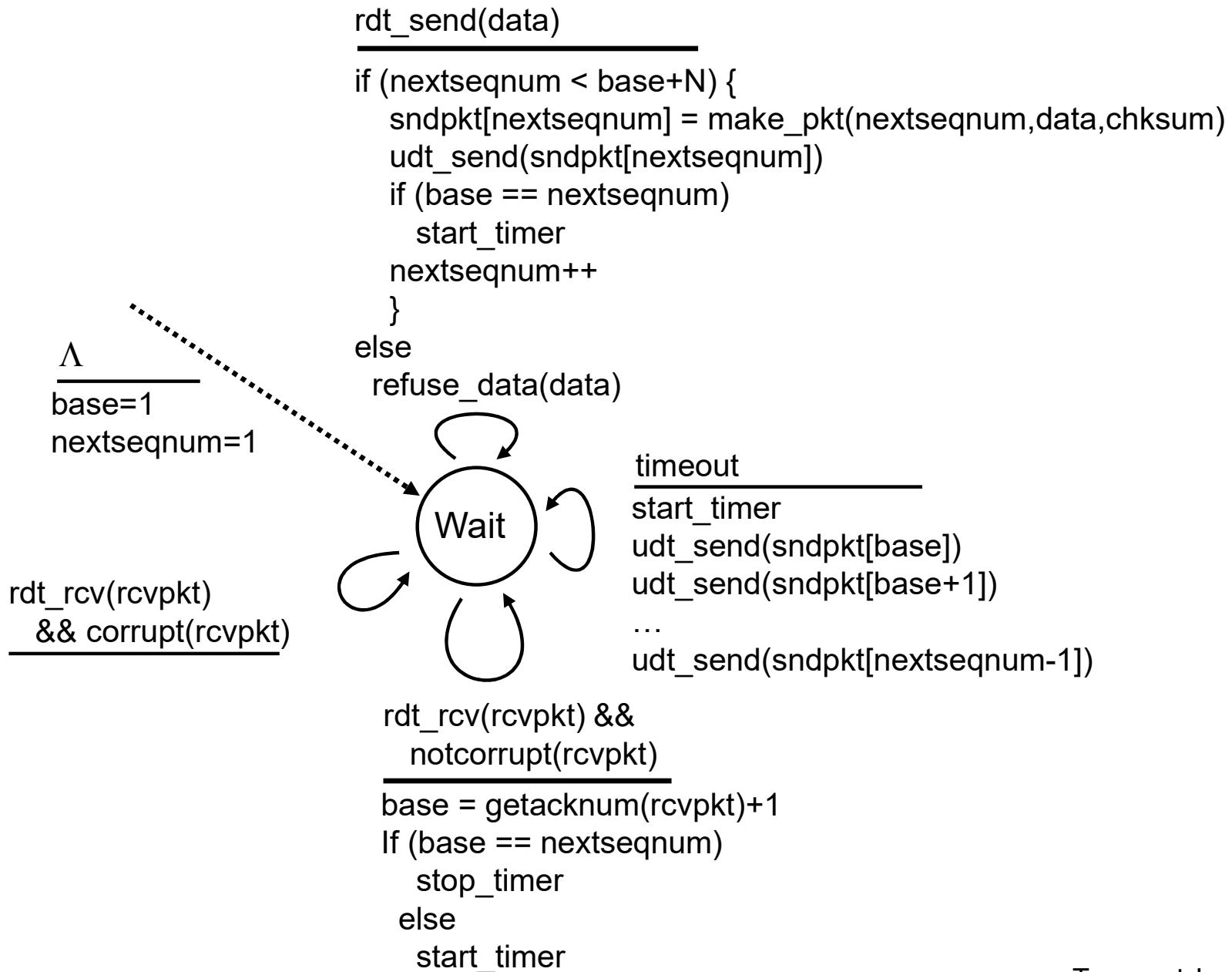
# Go-Back-N: sender

- ❖ 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
- ❖  $\text{timeout}(n)$ : retransmit packet n and all higher seq # pkts in window

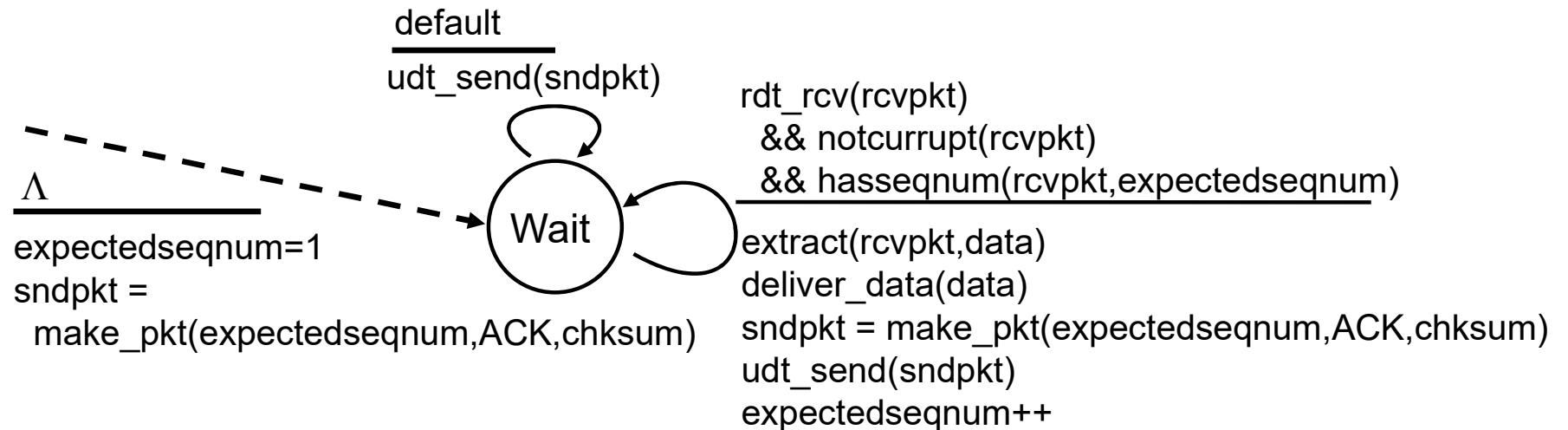
# GBN: sender extended FSM



## Exercise : GBN

- ❖ จาก state diagram ใน slide 57
- ❖ ให้ตอบว่า ทำไมต้องมีเงื่อนไข  $\text{nextseqnum} < \text{base} + N$
- ❖ ให้ตอบว่าเงื่อนไข  $\text{base} == \text{nextseqnum}$  จะเกิดขึ้นเมื่อใด ตอบให้ครบถ้วน
- ❖ ทำไม  $\text{rdt\_rcv}(\text{rcvpkt}) \&\& \text{corrupt}(\text{rcvpkt})$  จึงไม่มี Action

# 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 #

## Exercise : GBN

- ❖ จาก state diagram ใน slide 59
- ❖ ให้ตอบว่า ถ้าได้รับ packet ที่ 3 ก่อน packet ที่ 2 จะเกิดอะไรขึ้น

# GBN in action

*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*

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

## Exercise : GBN

- ❖ ไปที่

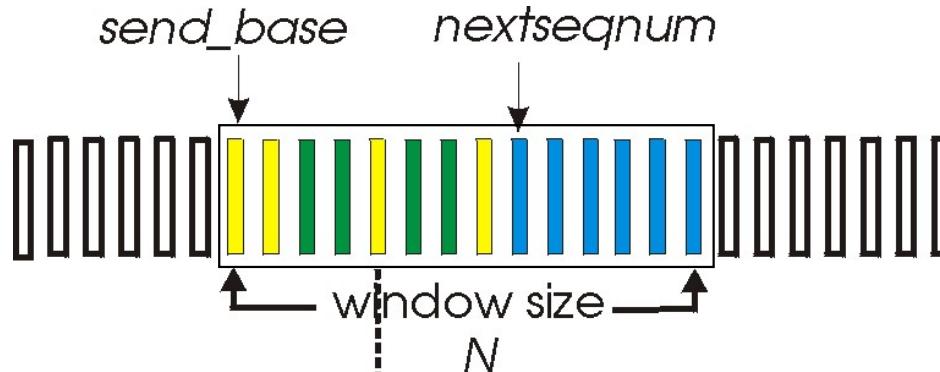
[https://media.pearsoncmg.com/aw/ecs\\_kurose\\_computeretwork\\_7/cw/content/interactiveanimations/go-back-n-protocol/index.html](https://media.pearsoncmg.com/aw/ecs_kurose_computeretwork_7/cw/content/interactiveanimations/go-back-n-protocol/index.html)

- ❖ ทดลองใช้เพื่อทำความเข้าใจ
- ❖ จากนั้นให้ทดลองส่ง 5 Packet และ pause เอาไว้ จากนั้นให้ kill packet แรก และ resume ให้อธิบายว่าเกิดอะไรขึ้น
- ❖ ให้ทำซ้ำแต่ให้ kill ack ของ packet แรกแทน อธิบายสิ่งที่เกิดขึ้น

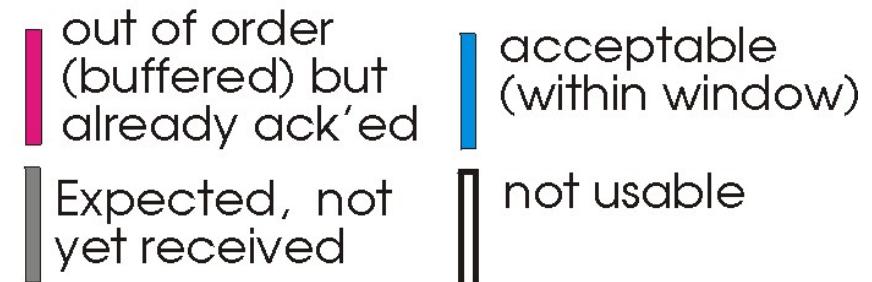
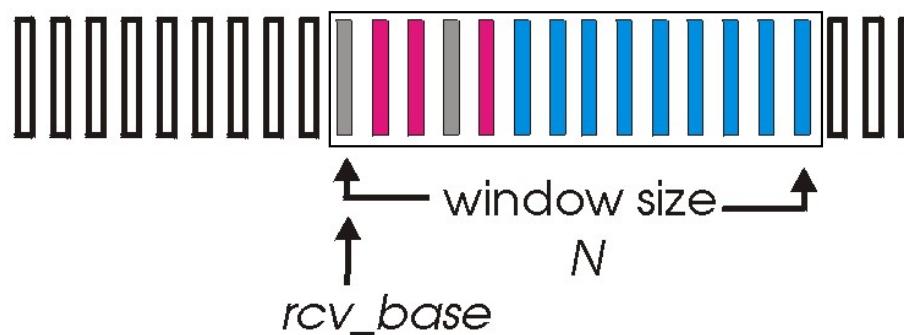
# Selective repeat

- ❖ 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]**

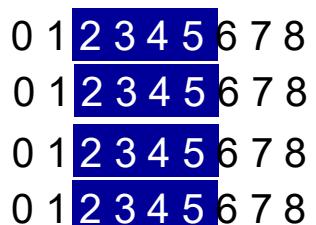
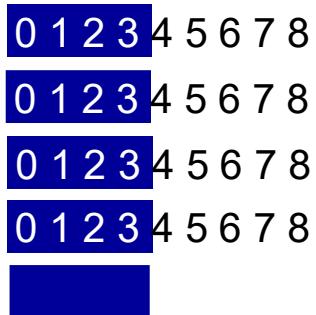
- ❖ ACK(n)

### **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 ack5 arrived

*Q: what happens when ack2 arrives?*

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

## Exercise : Selective

- ❖ ไปที่

[https://media.pearsoncmg.com/aw/ecs\\_kurose\\_computer\\_network\\_7/cw/content/interactiveanimations/selective-repeat-protocol/index.html](https://media.pearsoncmg.com/aw/ecs_kurose_computer_network_7/cw/content/interactiveanimations/selective-repeat-protocol/index.html)

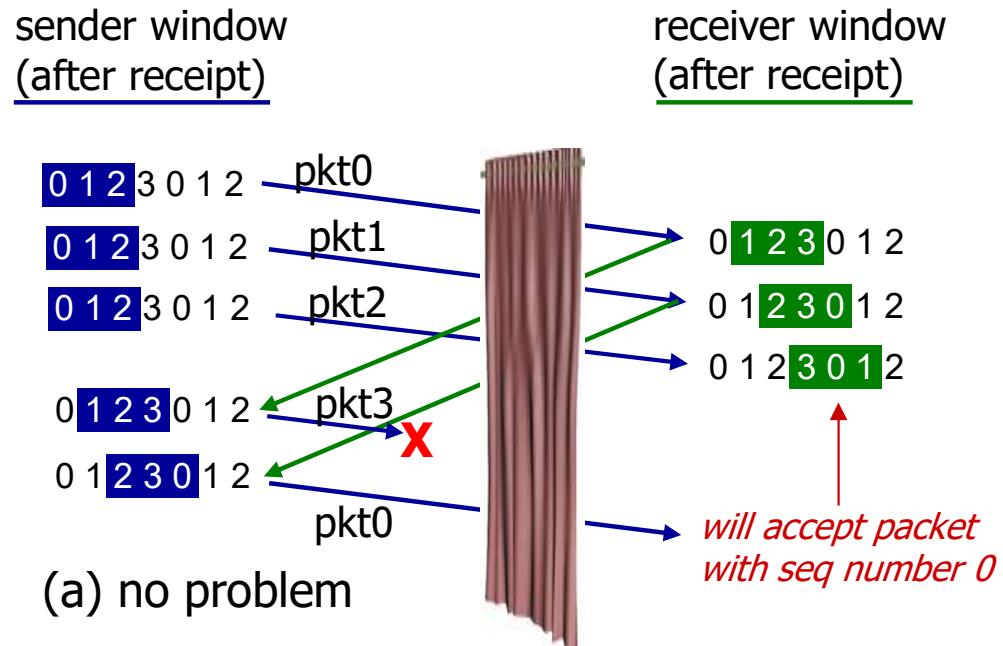
- ❖ ทดลองใช้เพื่อทำความเข้าใจ
- ❖ จากนั้นให้ทดลองส่ง 5 Packet และ pause เอาไว้ จากนั้นให้ kill packet แรก และ resume ให้อธิบายว่าเกิดอะไรขึ้น
- ❖ ให้ทำซ้ำแต่ให้ kill ack ของ packet แรกแทน อธิบายสิ่งที่เกิดขึ้น

# Selective repeat: dilemma

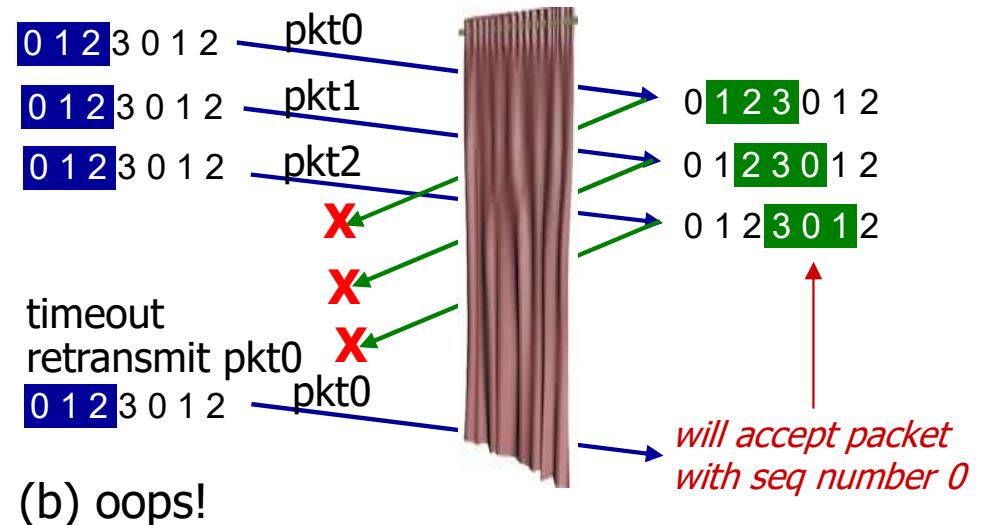
## example:

- ❖ 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

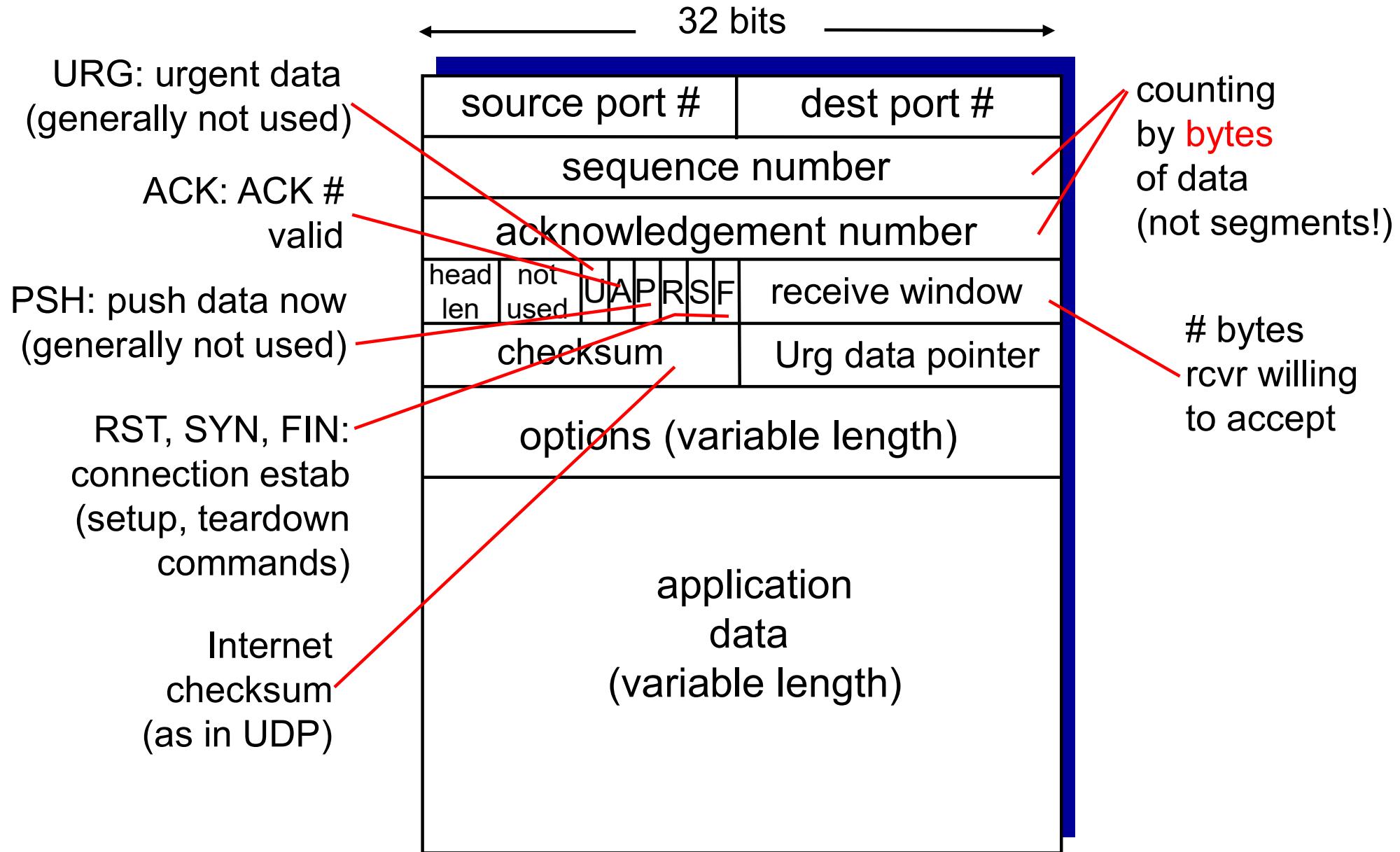
3.7 TCP congestion control

# TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- ❖ **point-to-point:**
  - one sender, one receiver
- ❖ **reliable, in-order byte 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) init's 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

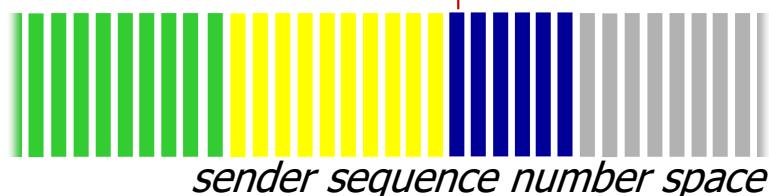
- A: TCP spec doesn’t say,  
- up to implementor

outgoing segment from sender

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

window size

$N$



sent  
ACKed

sent, not-  
yet ACKed  
("in-  
flight")

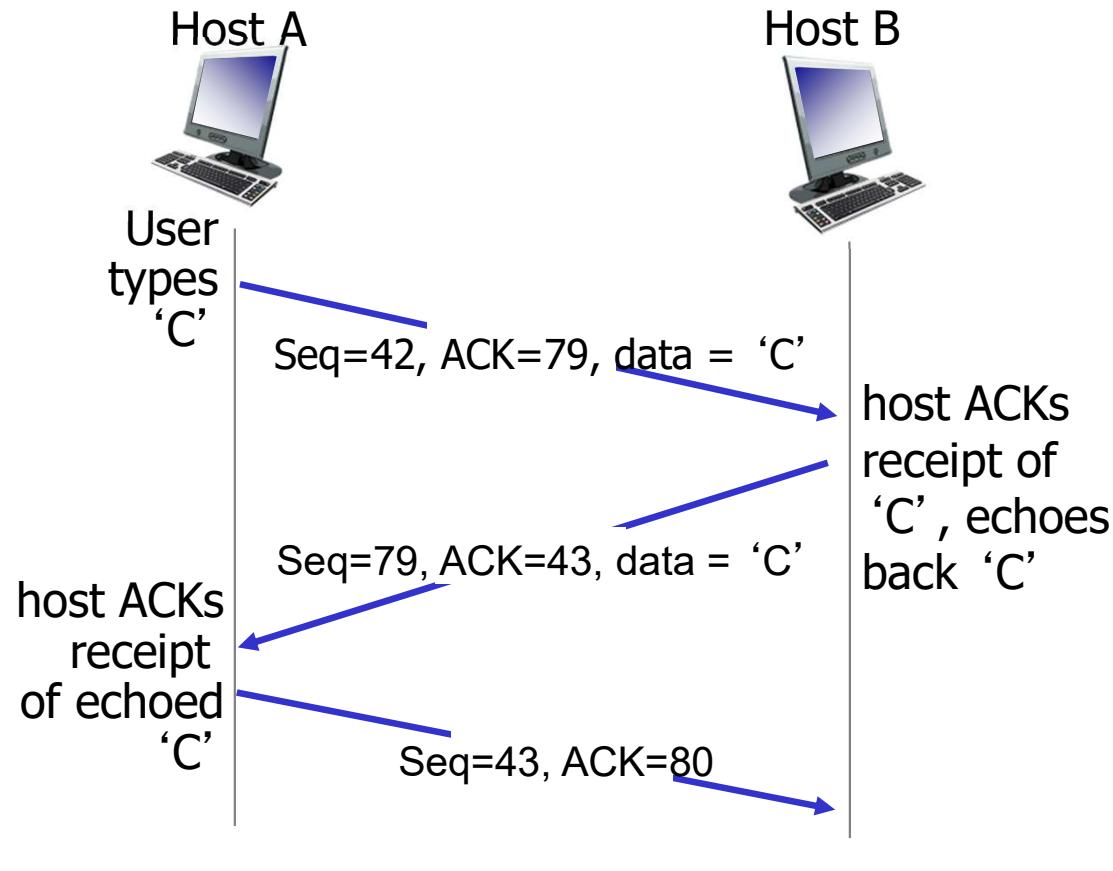
usable  
but not  
yet sent

not  
usable

incoming segment to sender

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

# TCP seq. numbers, ACKs

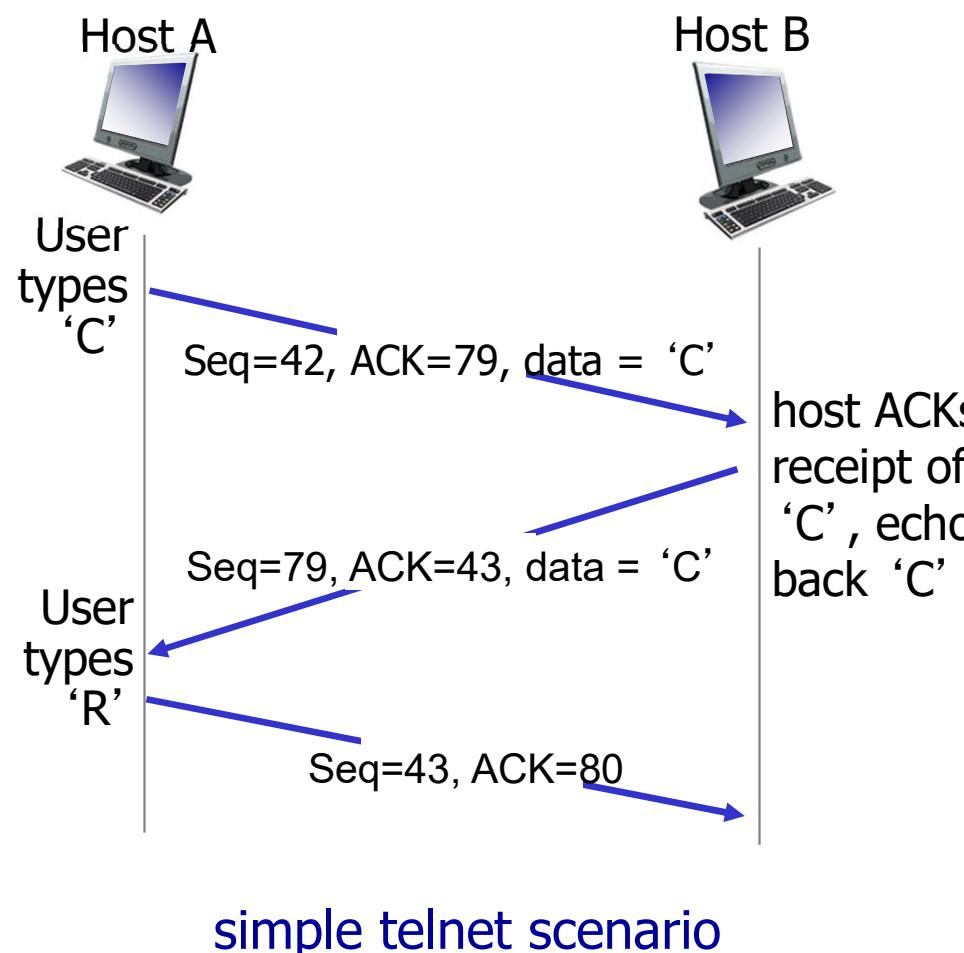


## Exercise : TCP

- ❖ สมมติว่า Host A ส่ง TCP Segment 2 Segment ต่อเนื่องกัน ไปยัง Host B ถ้า Segment แรก มี Sequence Number 90 และ Segment ที่ 2 มี Sequence Number 110
  - ใน Segment แรกมีข้อมูลเท่าไร
  - สมมติว่า Segment แรก เกิด Lost แต่ Segment ที่ 2 ส่งถึง Host B ใน Acknowledgment Number ที่ส่งกลับจะเป็นค่าเท่าไร

## Exercise : TCP

- ❖ จาก Slide Telnet ก่อนหน้านี้ หลังจากที่ผู้ใช้กด 'C' ก็จะ 'R' จะมีการส่งข้อมูลอย่างไร จะมีการส่งกี่ segment ให้ใส่ Seq# และ Ack# ให้ถูกต้อง



## Exercise : TCP

- ❖ ถ้าจะส่งไฟล์ขนาดใหญ่มาก (L Bytes) จาก Host A ไปยัง Host B กำหนดให้ MSS = 536 ไบต์
  - จงหาว่า L มีค่ามากที่สุดได้เท่าไร ที่จะทำให้ TCP ยังสามารถทำงานได้ กำหนดให้ TCP Seq# มีขนาด 4 ไบต์
  - สมมติว่าจะส่งไฟล์ดังกล่าวผ่าน Link ที่มีความเร็ว 155 Mbps โดยกำหนดให้ Header ของ Transport, Network และ Data Link รวมกันมีขนาด 66 ไบต์ โดยสมมติว่าสามารถส่งข้อมูลต่อเนื่องกันได้เลย โดยไม่ต้องมี Ack จะใช้เวลาในการส่งเท่าไร (ไม่คิด Propagation Delay)

# 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

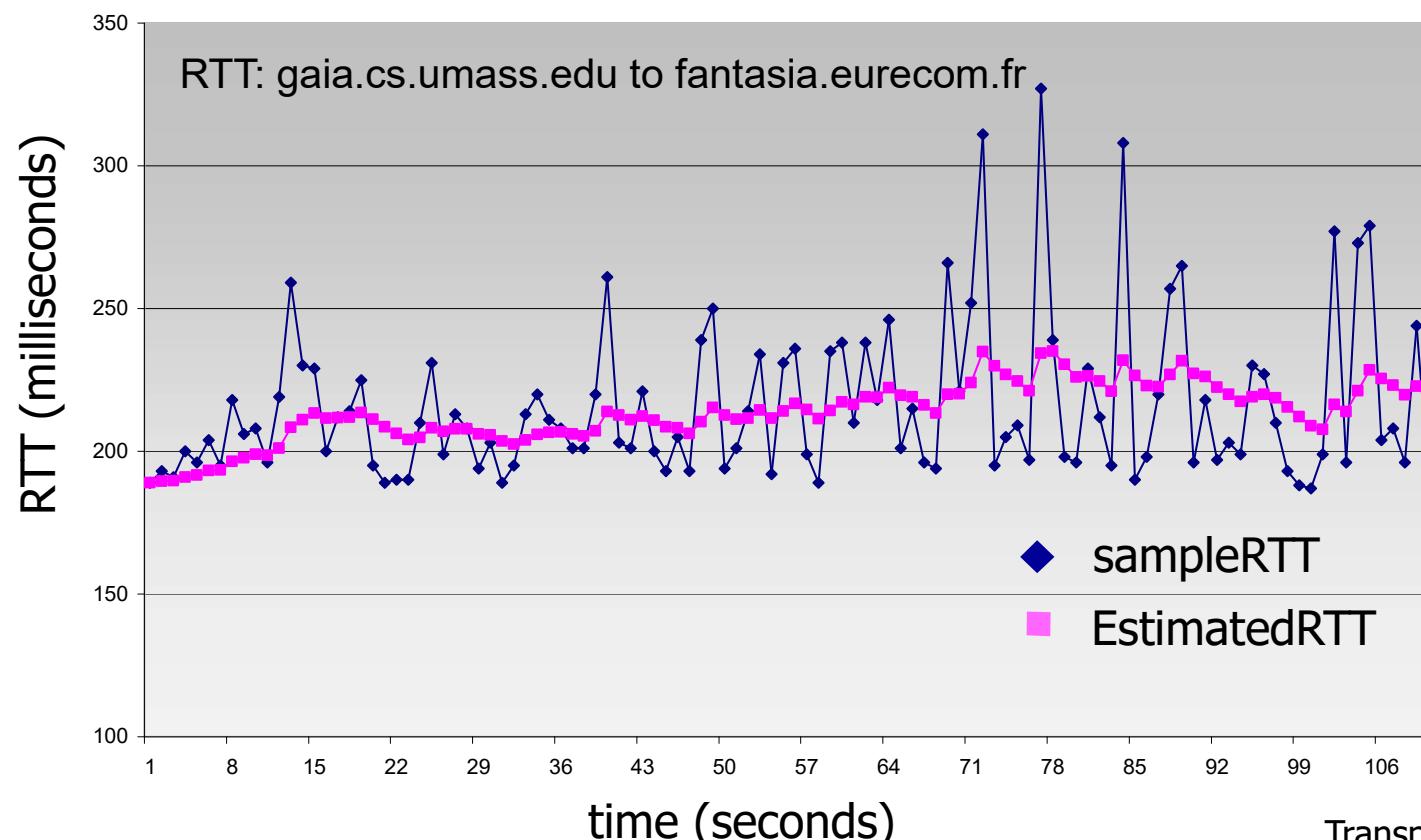
**Q:** how to estimate RTT?

- ❖ **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

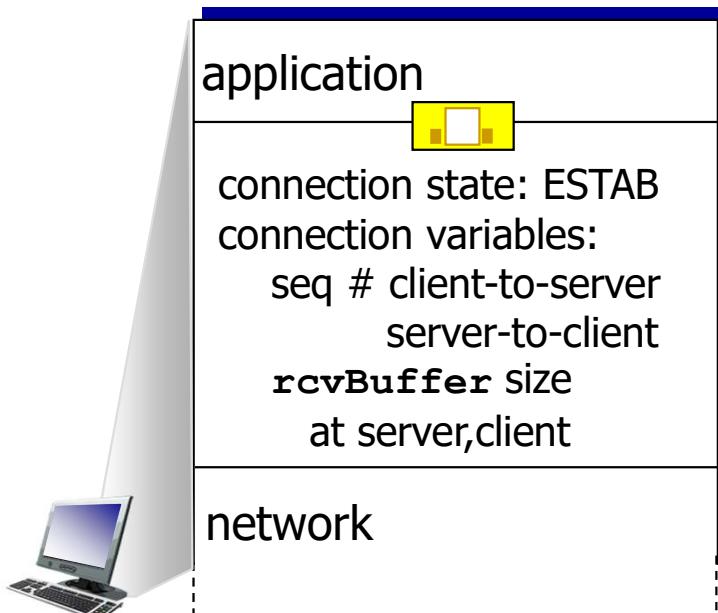
3.6 principles of congestion control

3.7 TCP congestion control

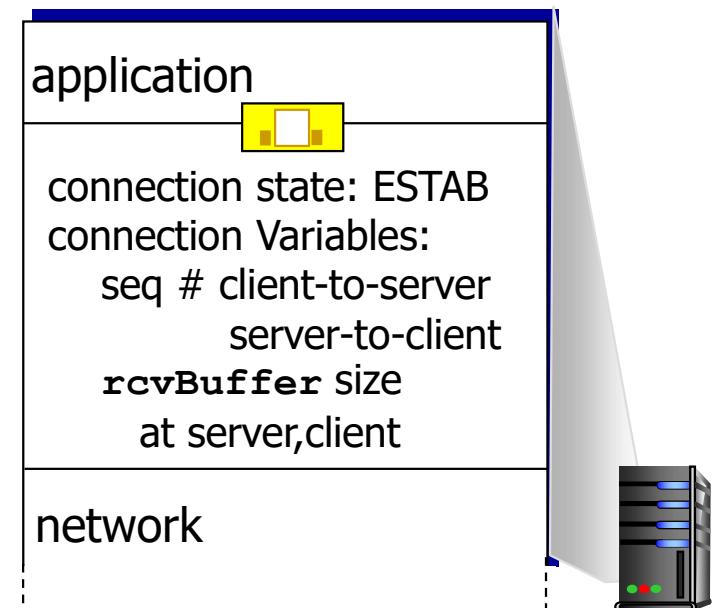
# Connection Management

before exchanging data, sender/receiver “handshake”:

- ❖ agree to establish connection (each knowing the other willing to establish connection)
- ❖ agree on connection parameters



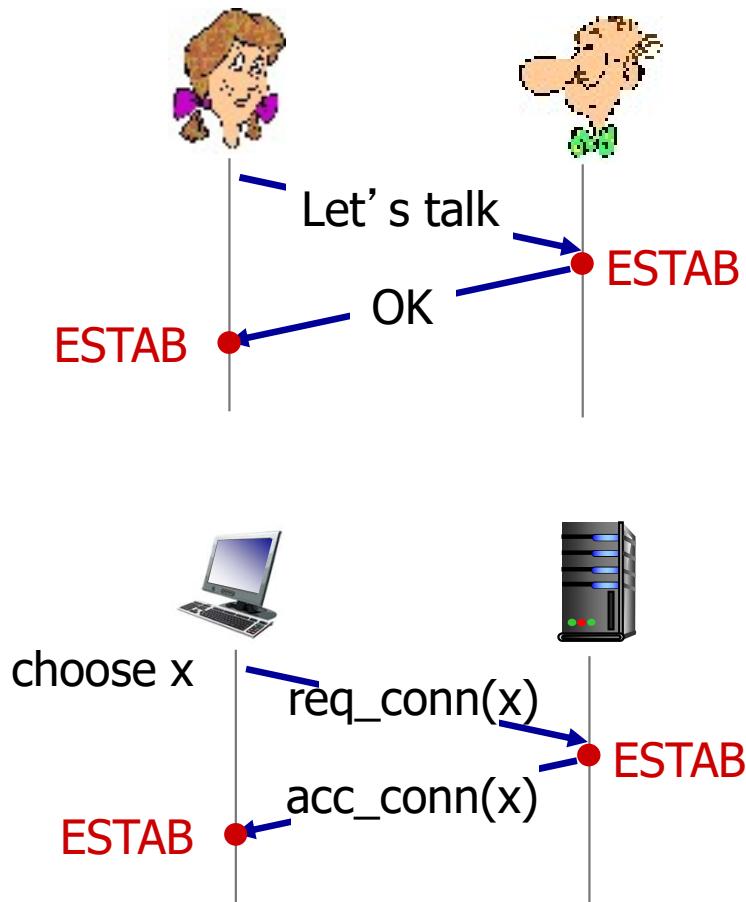
```
Socket clientSocket =  
    newSocket("hostname", "port  
    number");
```



```
Socket connectionSocket =  
    welcomeSocket.accept();
```

# Agreeing to establish a connection

2-way handshake:

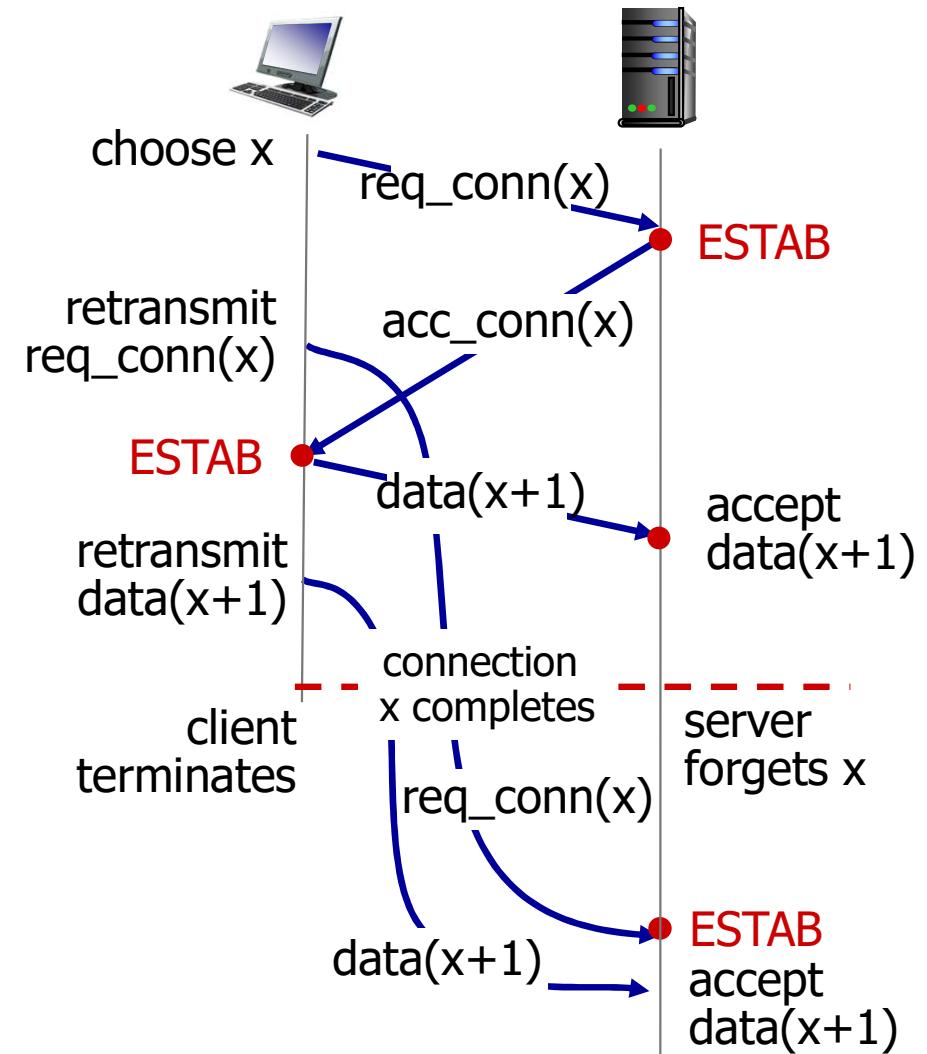
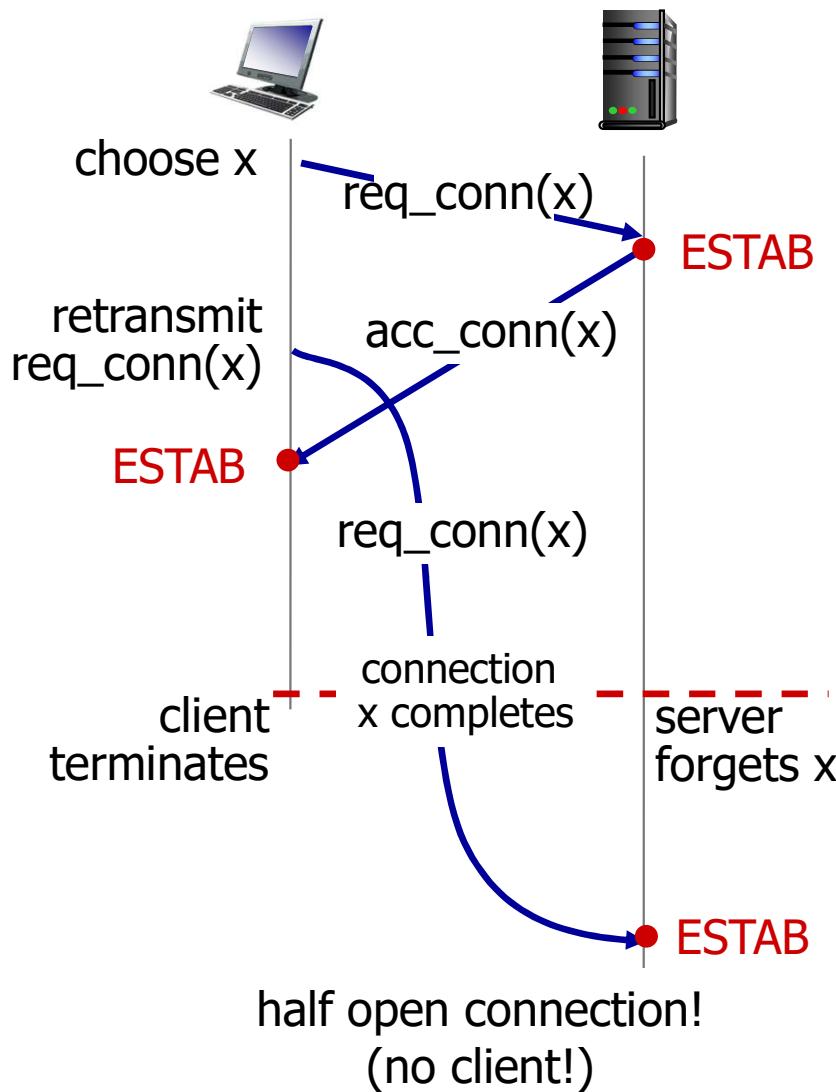


**Q:** will 2-way handshake always work in network?

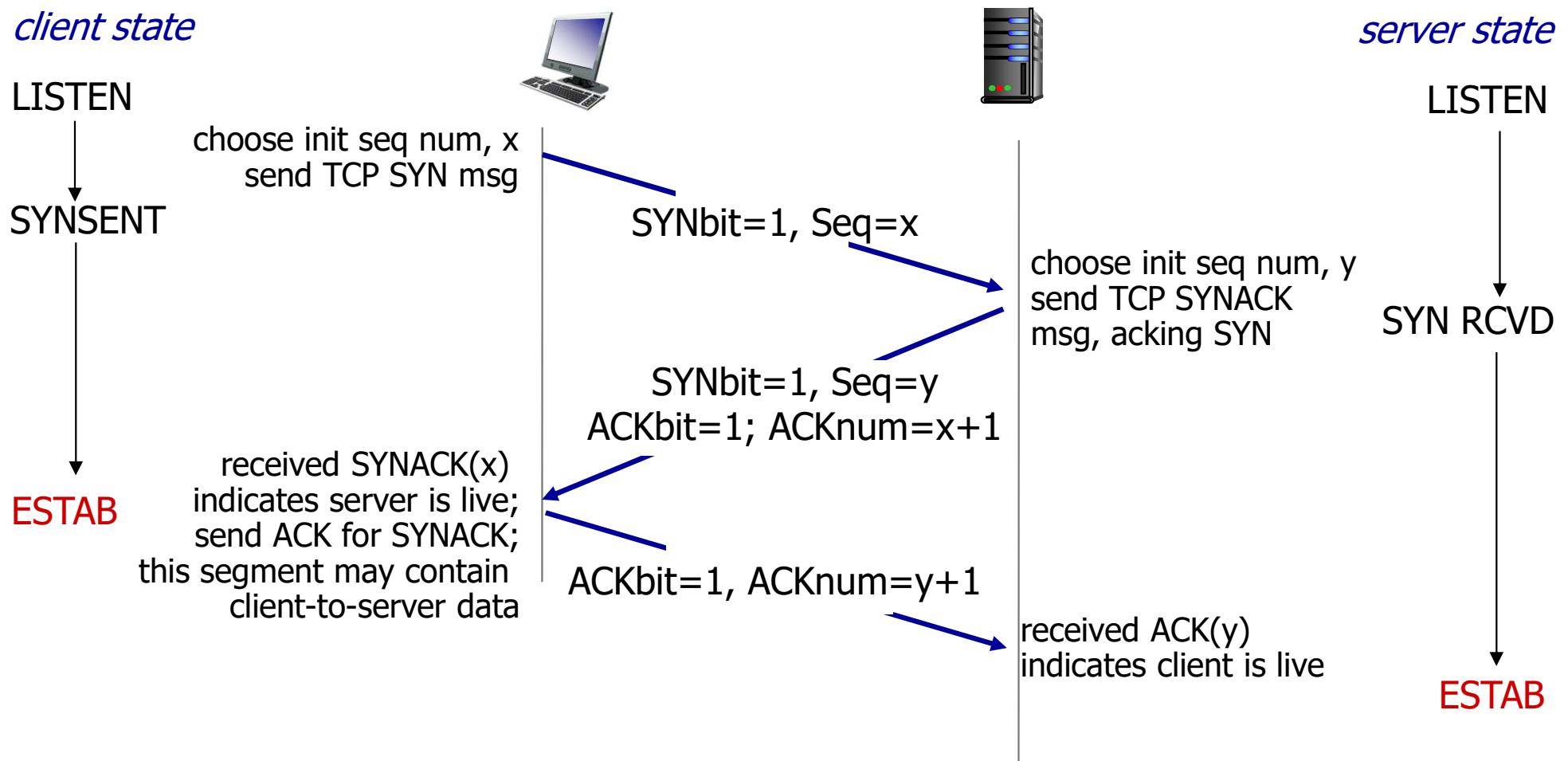
- ❖ variable delays
- ❖ retransmitted messages (e.g. `req_conn(x)`) due to message loss
- ❖ message reordering
- ❖ can't "see" other side

# Agreeing to establish a connection

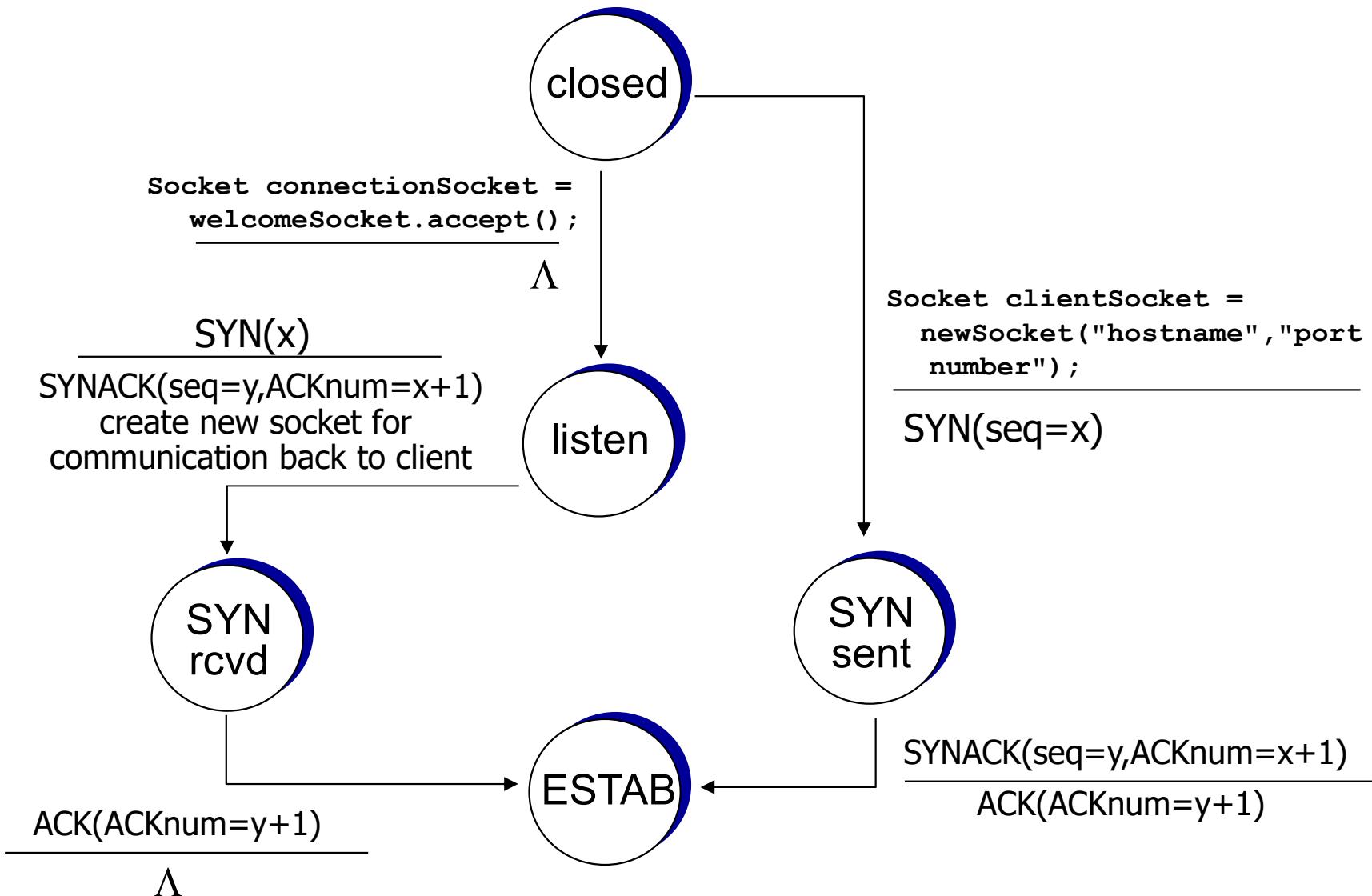
2-way handshake failure scenarios:



# TCP 3-way handshake



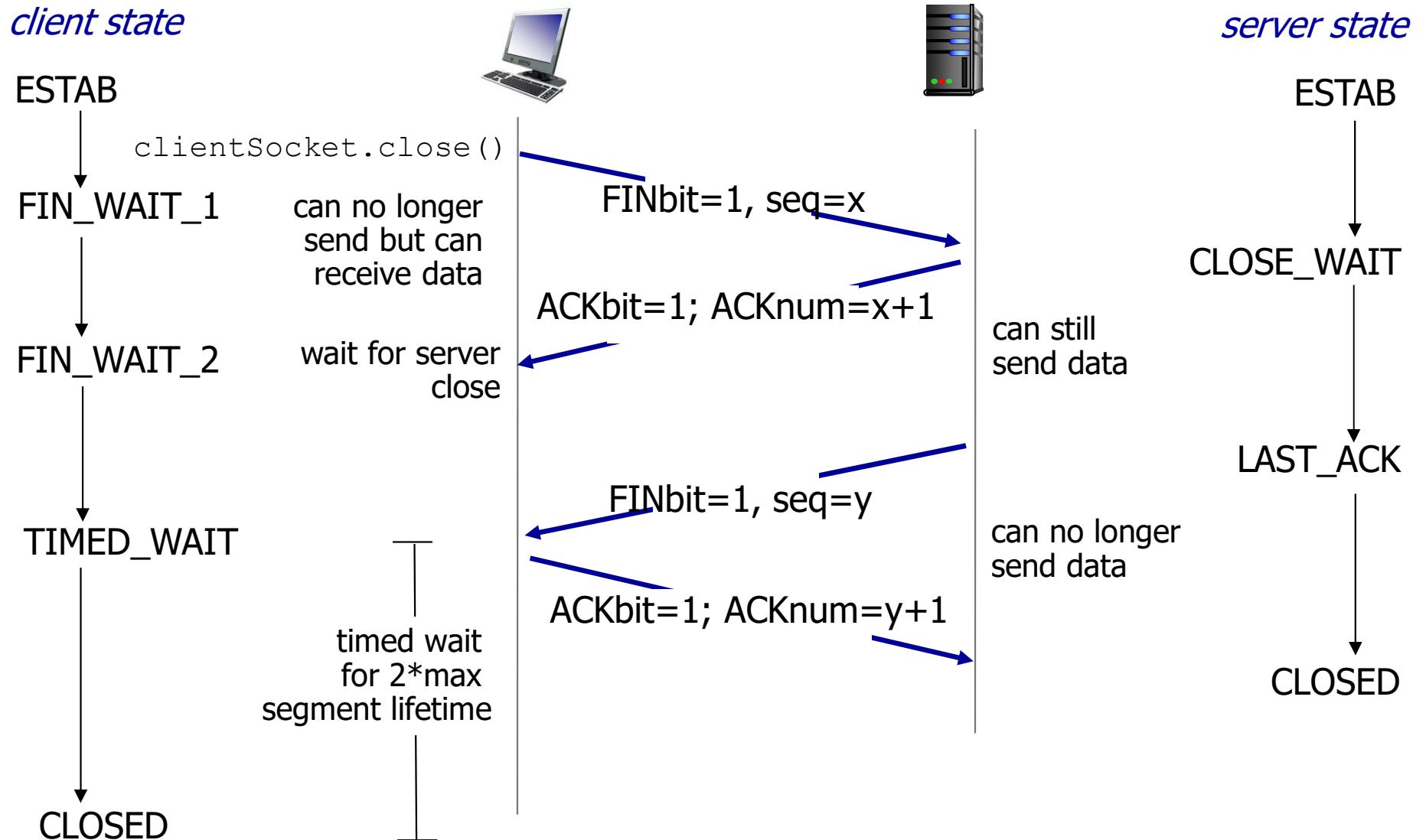
# TCP 3-way handshake: FSM



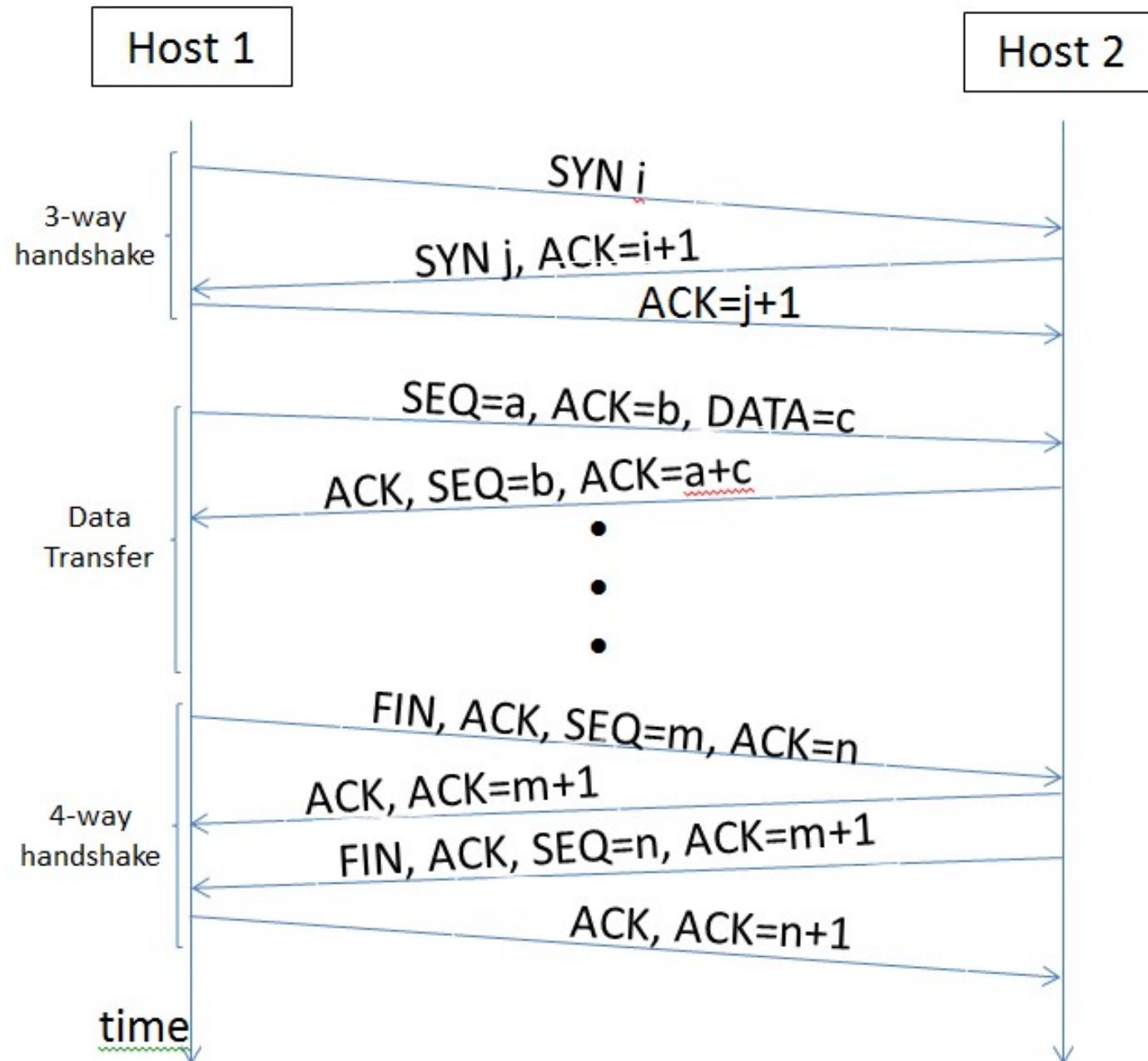
# TCP: closing a connection

- ❖ 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



# TCP: closing a connection



# 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`

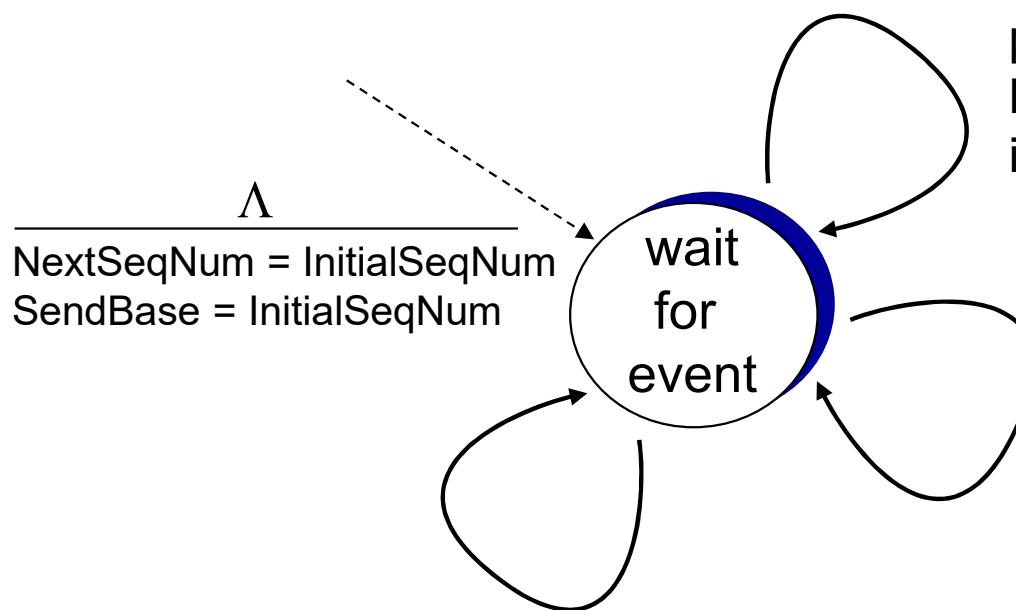
## *timeout:*

- ❖ 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)



$\Lambda$   
NextSeqNum = InitialSeqNum  
SendBase = InitialSeqNum

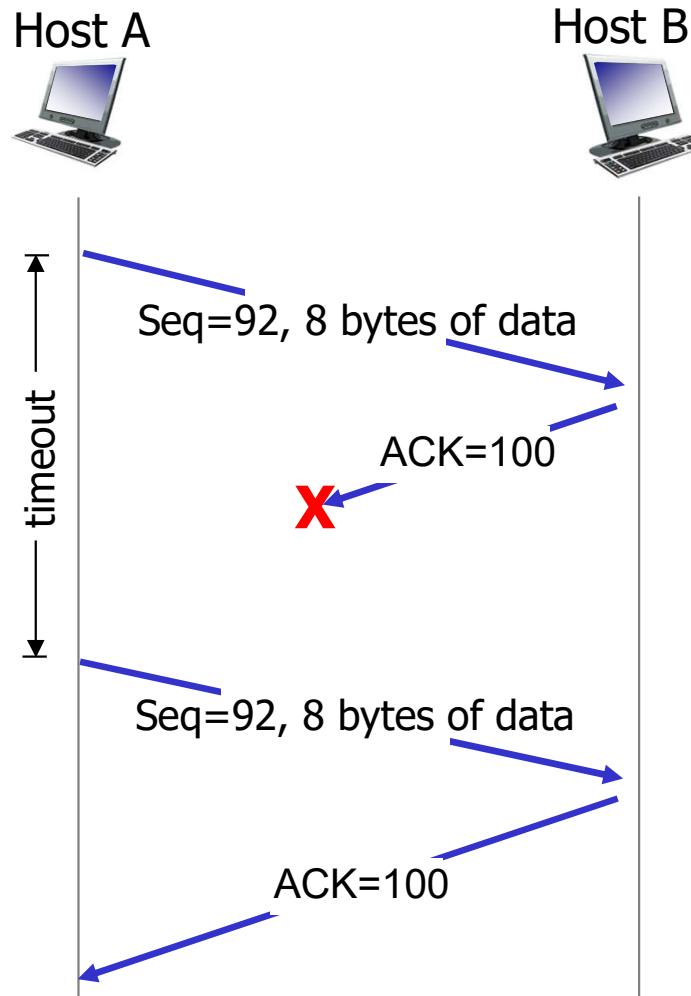
data received from application above  
create segment, seq. #: NextSeqNum  
pass segment to IP (i.e., “send”)  
NextSeqNum = NextSeqNum + length(data)  
if (timer currently not running)  
    start timer

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

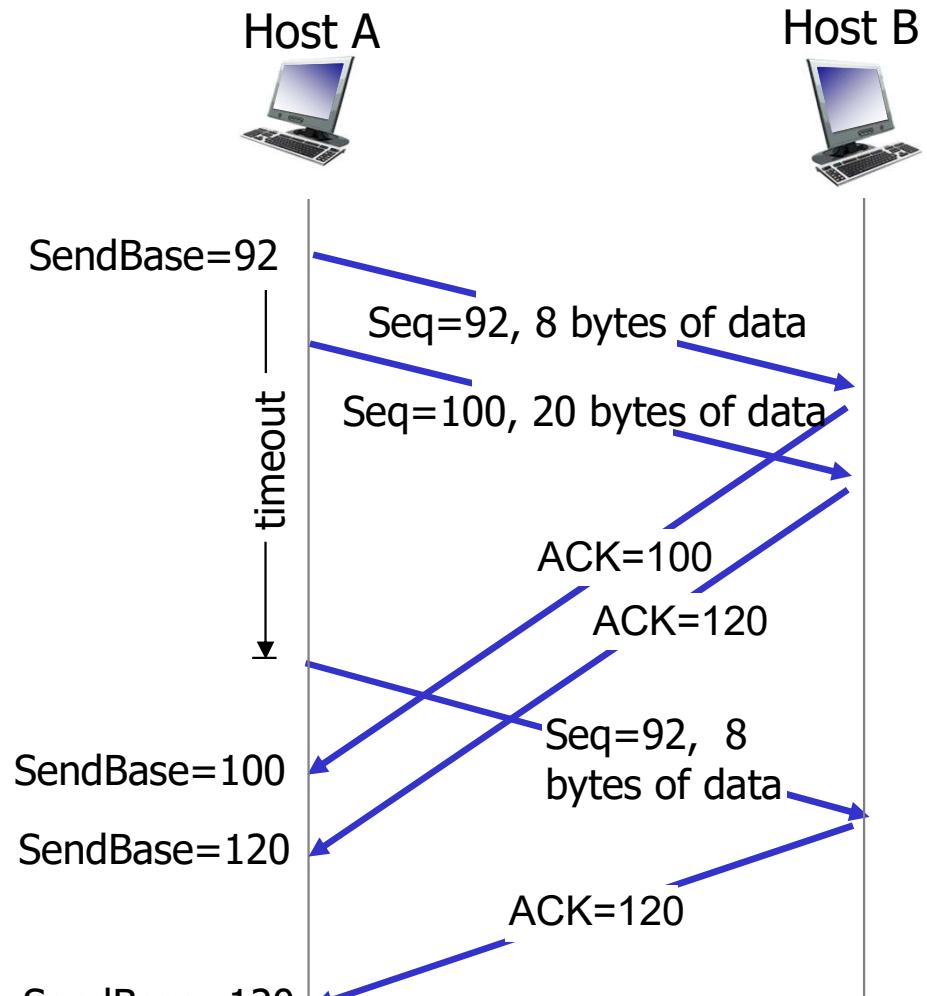
ACK received, with ACK field value y

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

# TCP: retransmission scenarios

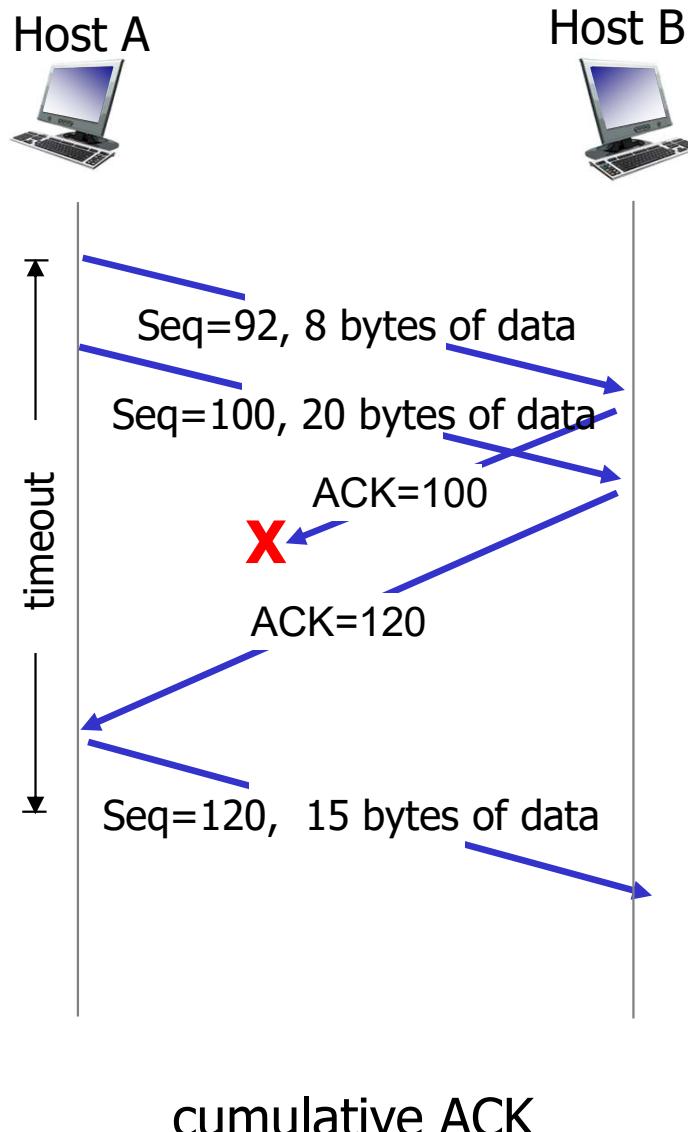


lost ACK scenario



premature timeout

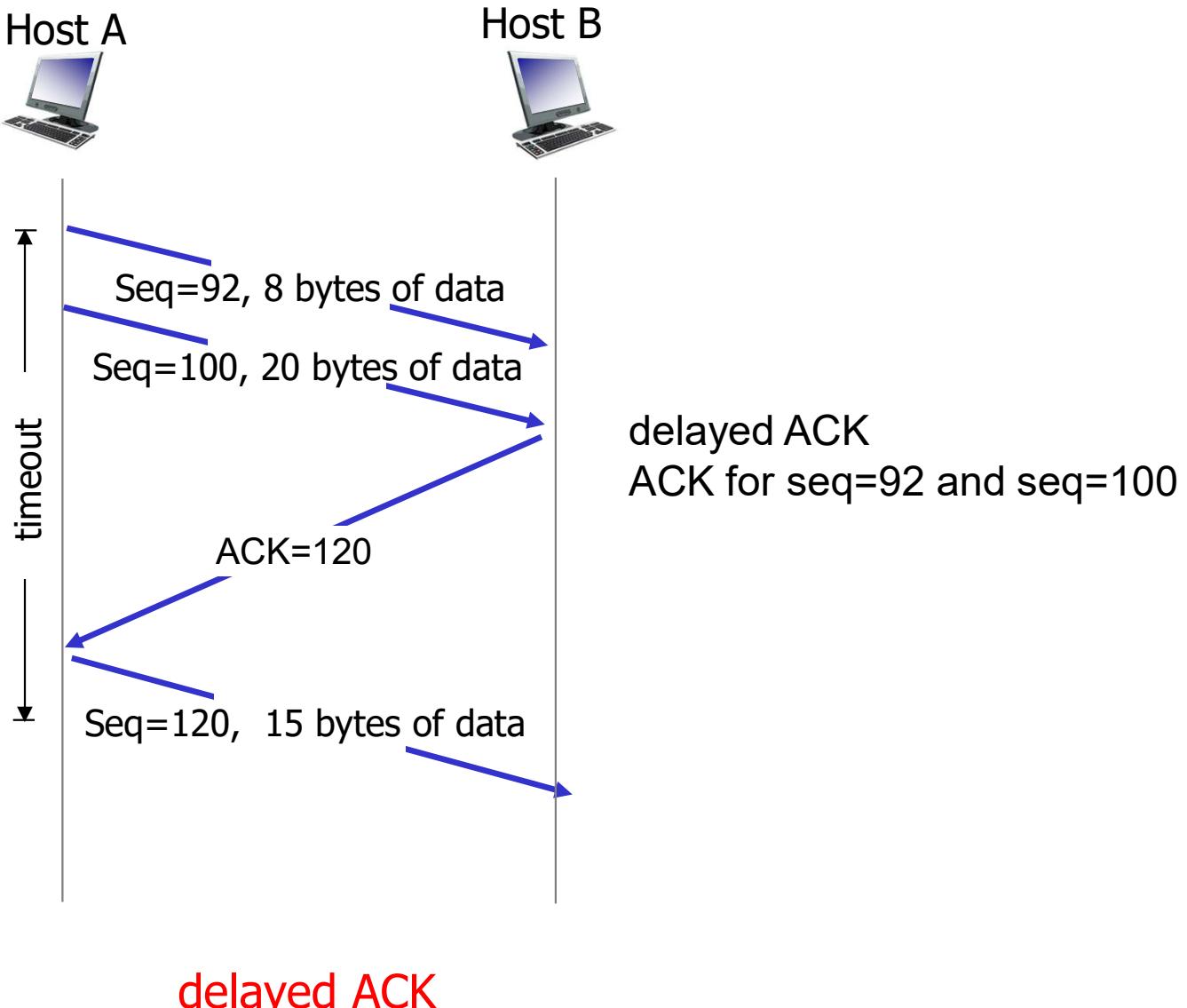
# TCP: retransmission scenarios



# TCP ACK generation [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	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send <i>duplicate ACK</i> , 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 delayed ACK



# TCP fast retransmit

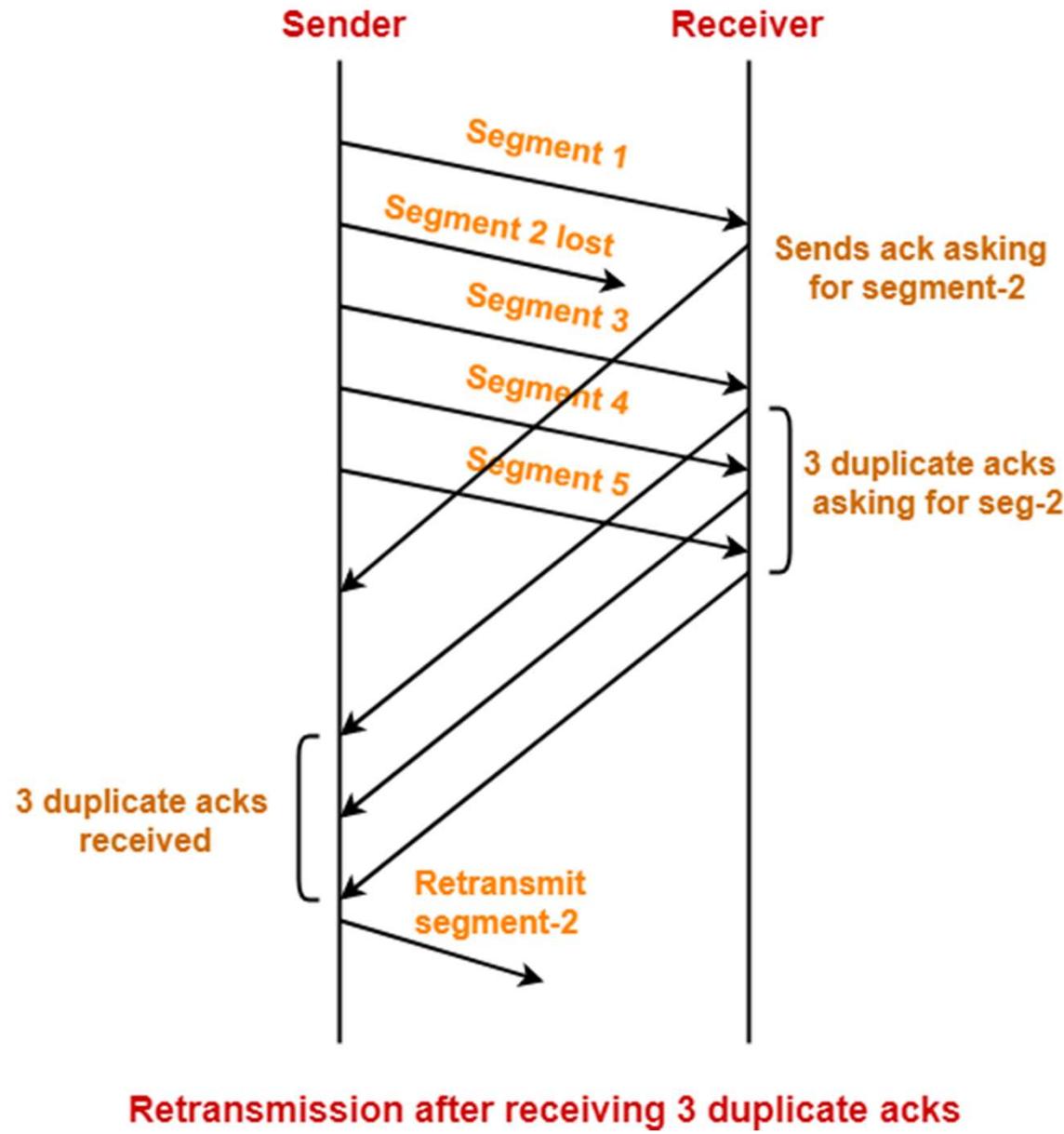
- ❖ time-out period often relatively long:
  - long delay before resending lost packet
- ❖ detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

## *TCP fast retransmit*

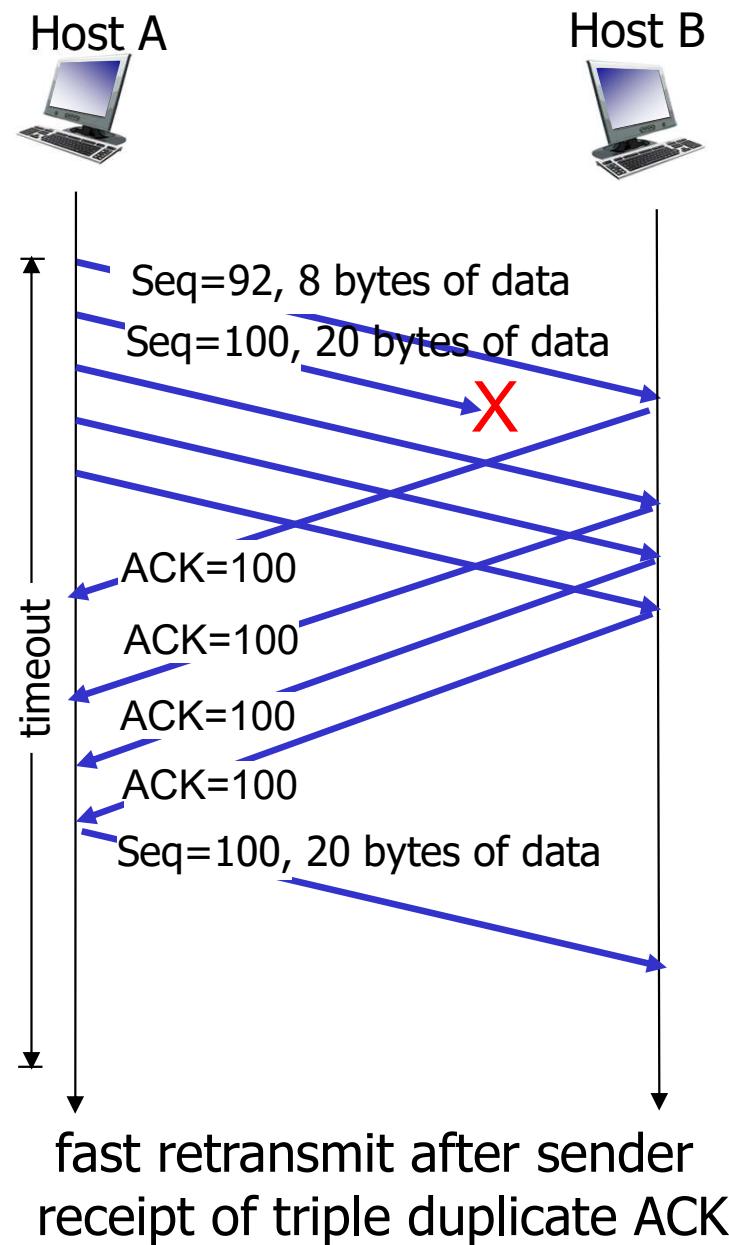
if sender receives 3 ACKs for same data (“triple duplicate ACKs”), resend unacked segment with smallest seq #

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

# TCP fast retransmit



# TCP fast retransmit



# 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

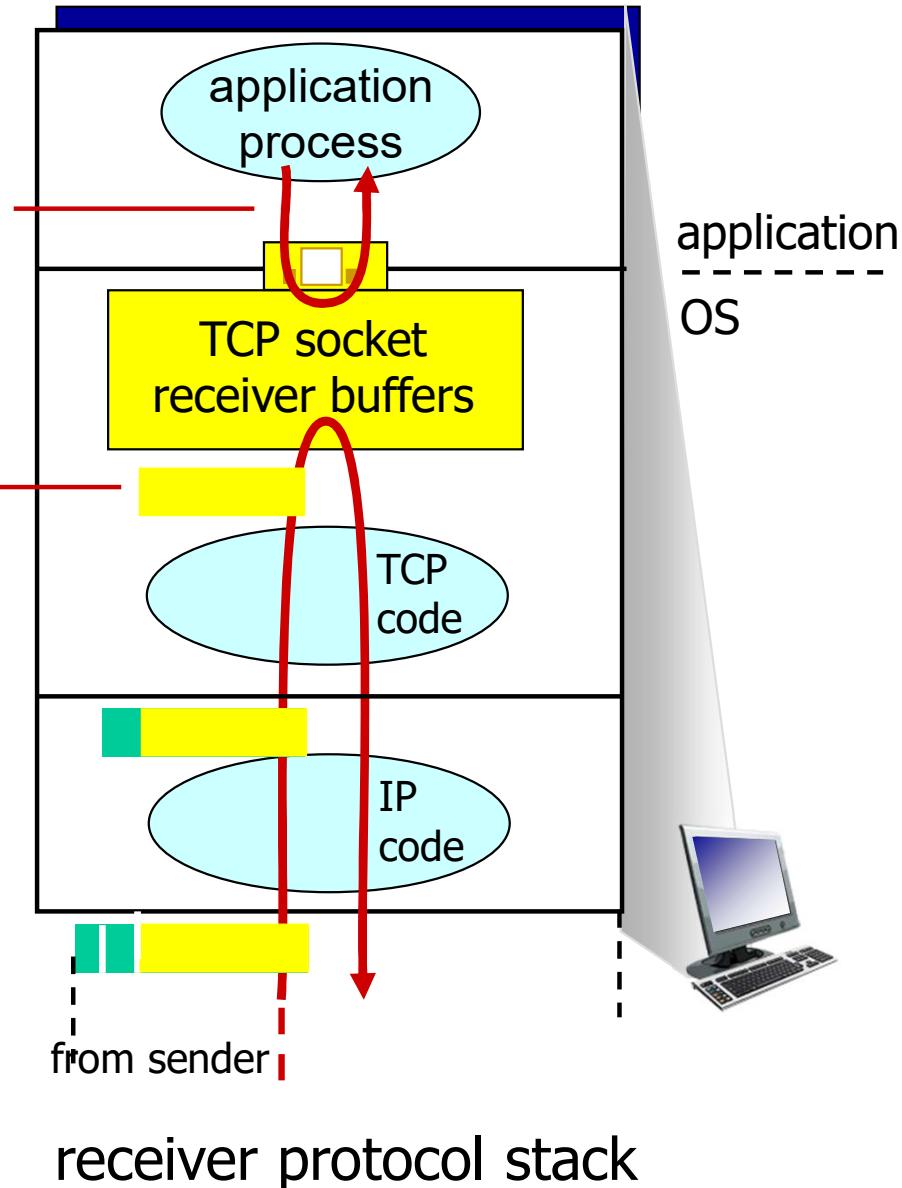
3.7 TCP congestion control

# TCP flow control

application may  
remove data from  
TCP socket buffers ....

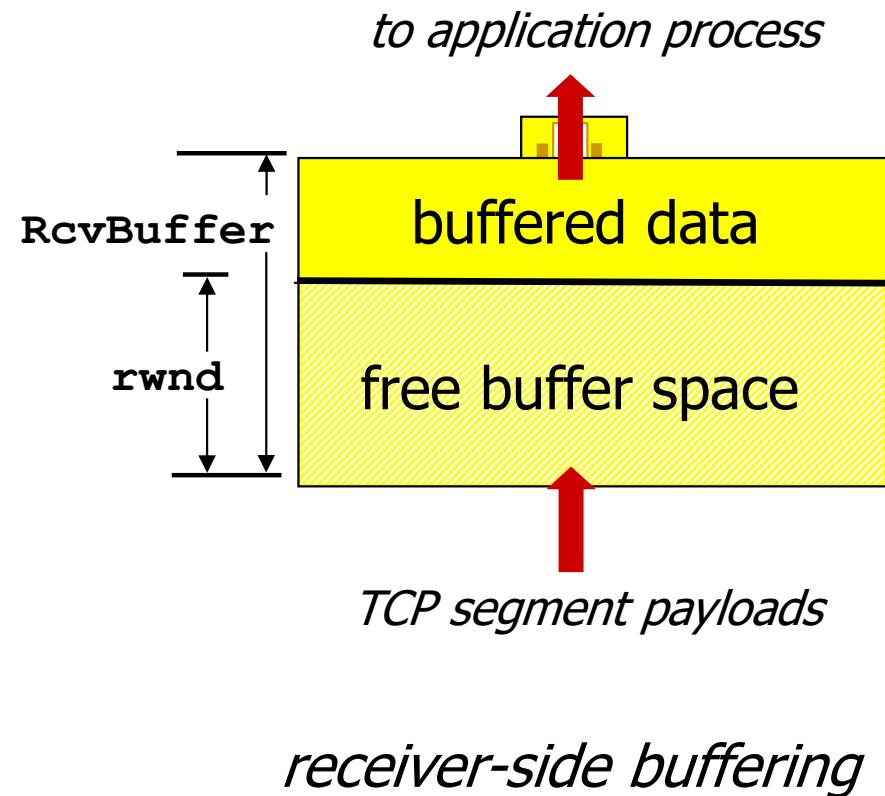
... slower than TCP  
receiver is delivering  
(sender is sending)

**flow control**  
receiver controls sender, so  
sender won't overflow  
receiver's buffer by transmitting  
too much, too fast

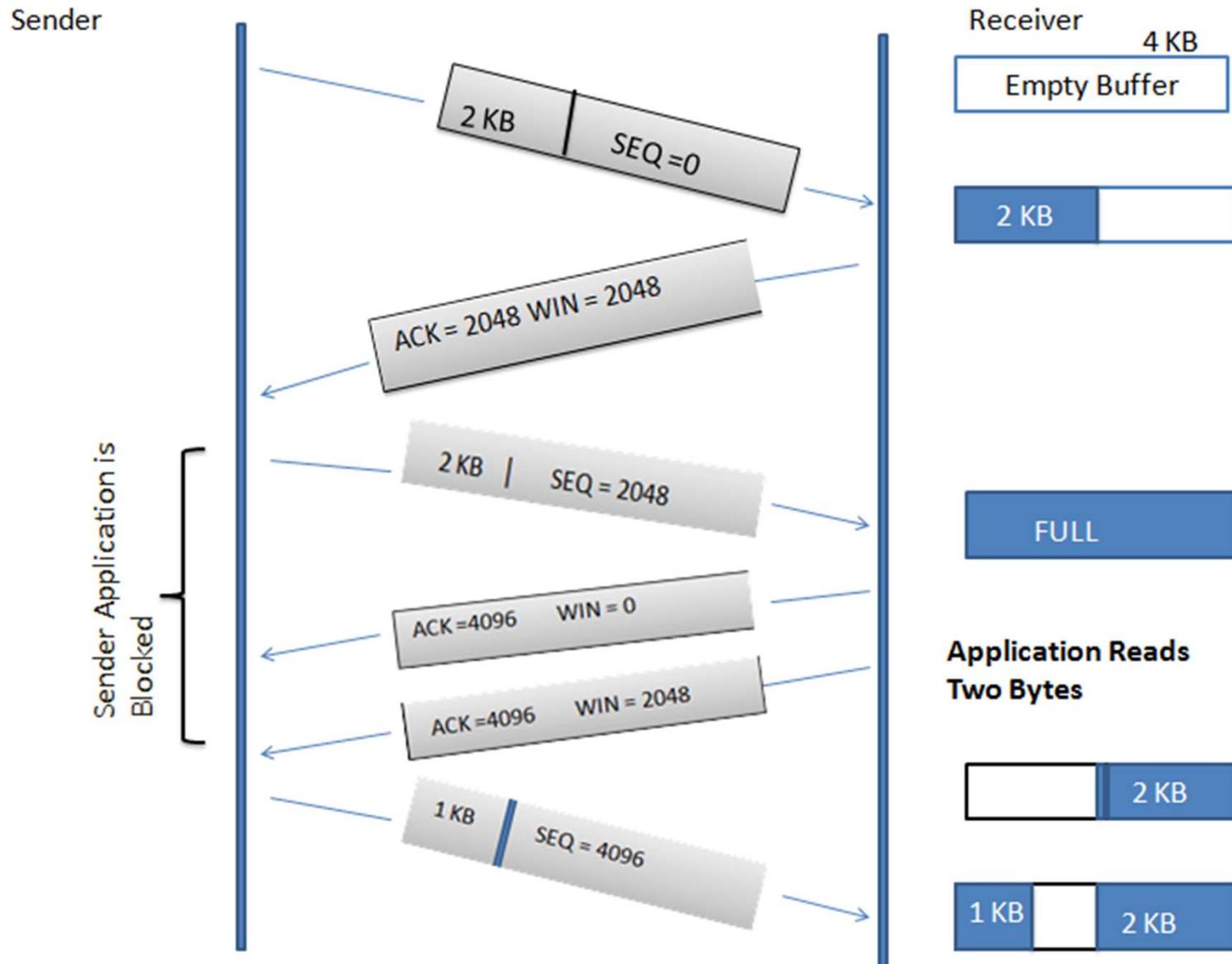


# TCP flow control

- ❖ 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



# TCP flow control



# 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

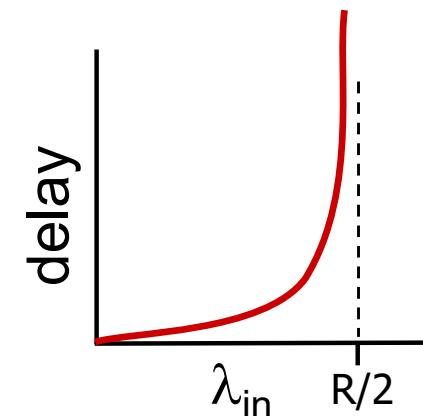
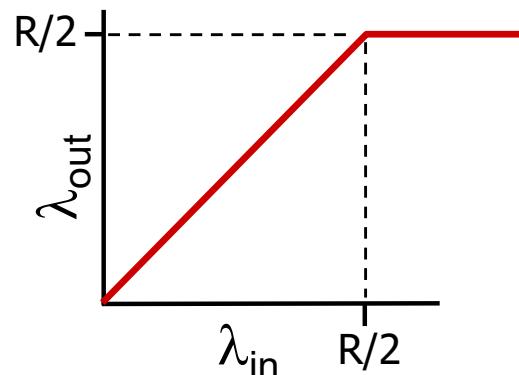
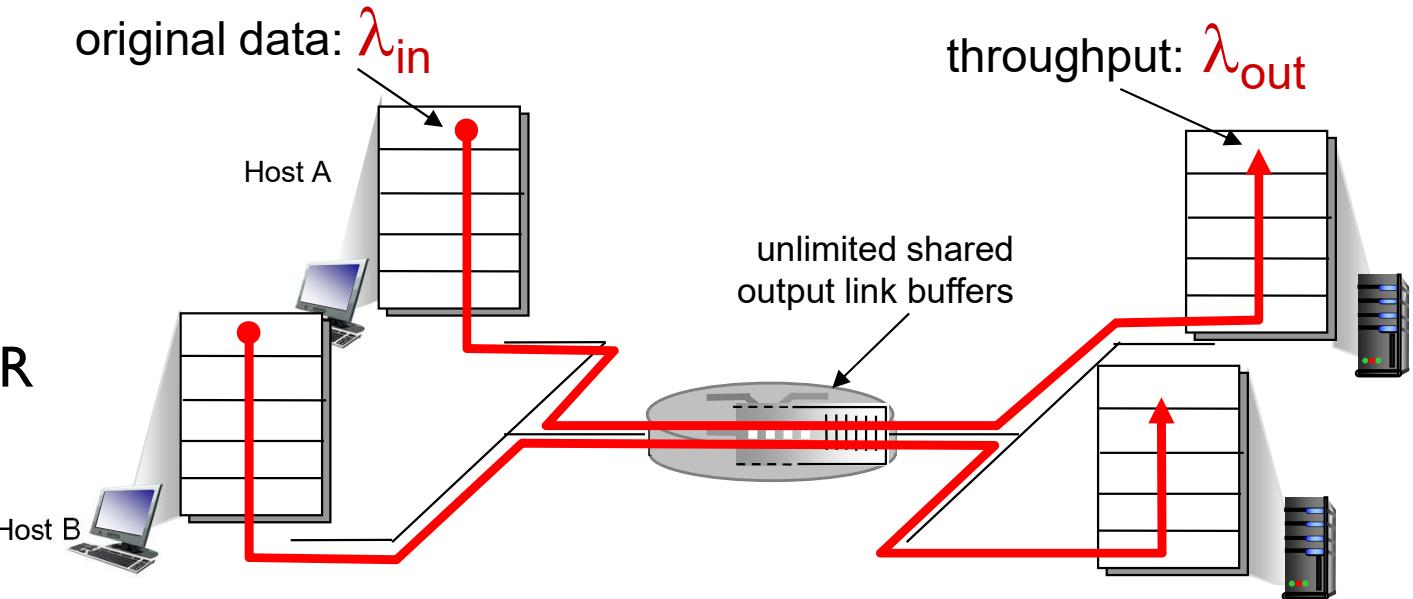
# Principles of congestion control

*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 (queueing in router buffers)
- ❖ a top-10 problem!

# Causes/costs of congestion: scenario I

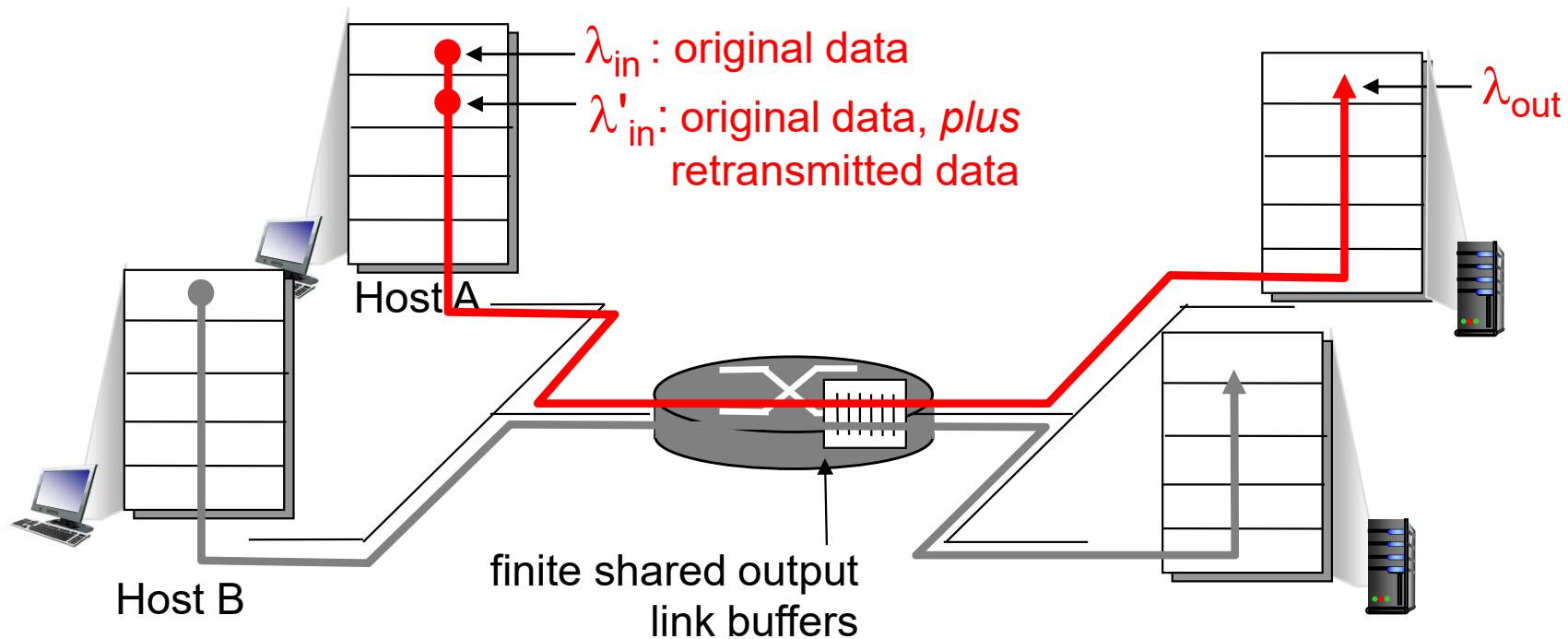
- ❖ two senders, two receivers
- ❖ one router, infinite buffers
- ❖ output link capacity:  $R$
- ❖ no retransmission



- ❖ maximum per-connection throughput:  $R/2$
- ❖ large delays as arrival rate,  $\lambda_{in}$ , approaches capacity

# Causes/costs of congestion: scenario 2

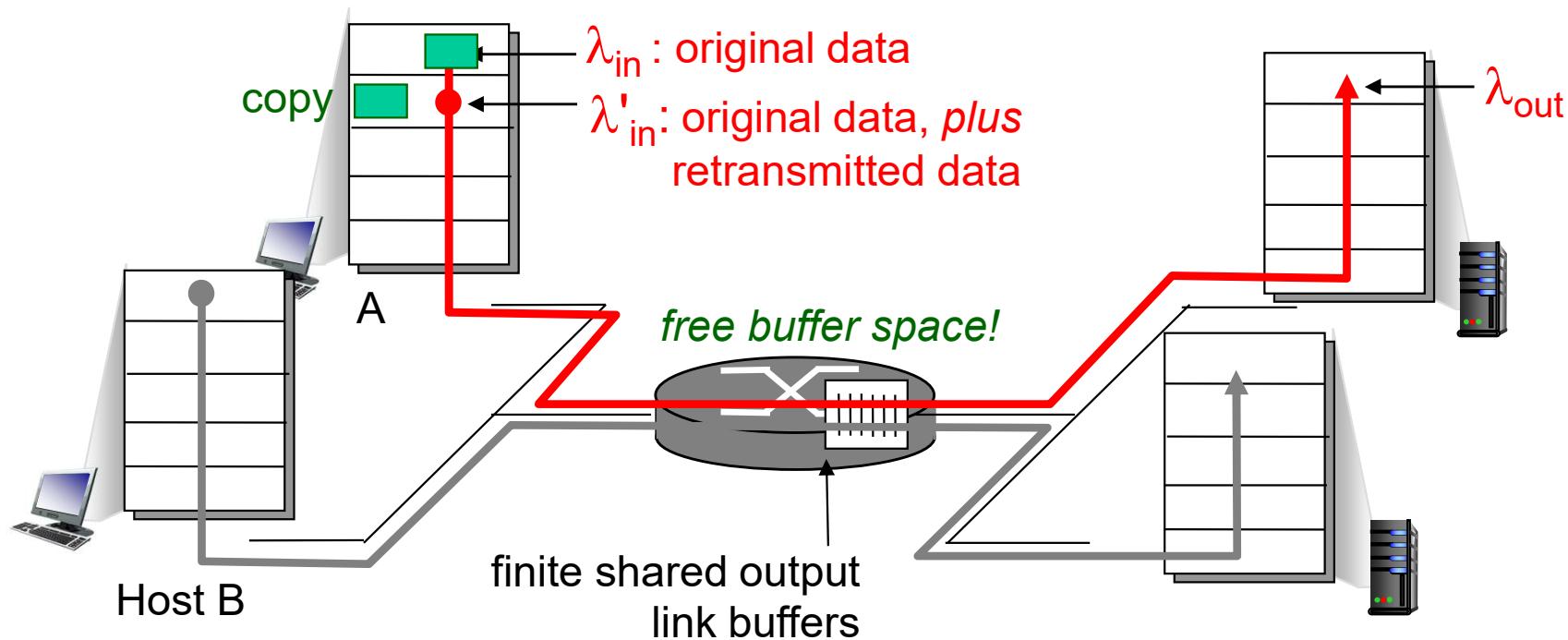
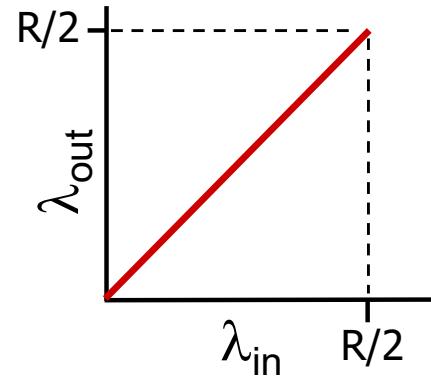
- ❖ one router, *finite buffers*
- ❖ sender retransmission of timed-out packet
  - application-layer input = application-layer output:  $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* :  $\lambda'_{in} \geq \lambda_{in}$



# Causes/costs of congestion: scenario 2

idealization: perfect knowledge

- ❖ sender sends only when router buffers available

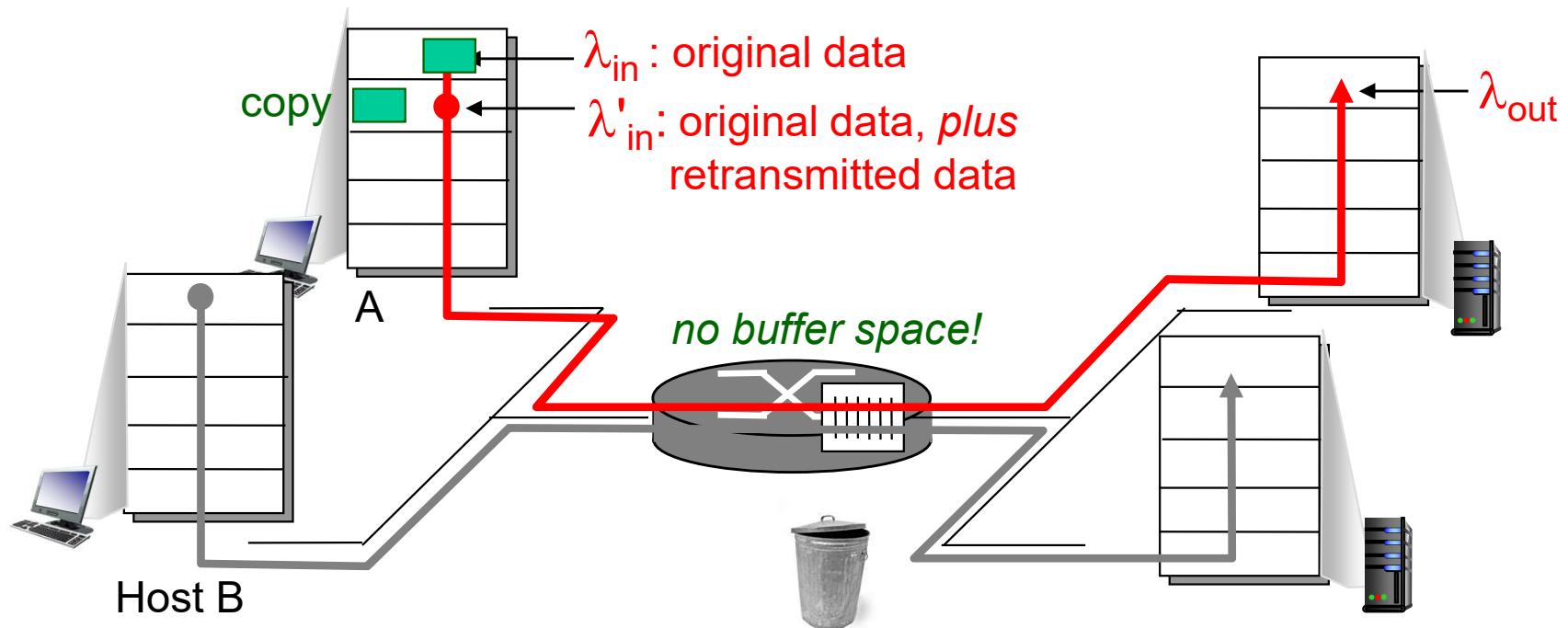


# Causes/costs of congestion: scenario 2

## Idealization: *known loss*

packets can be lost,  
dropped at router due  
to full buffers

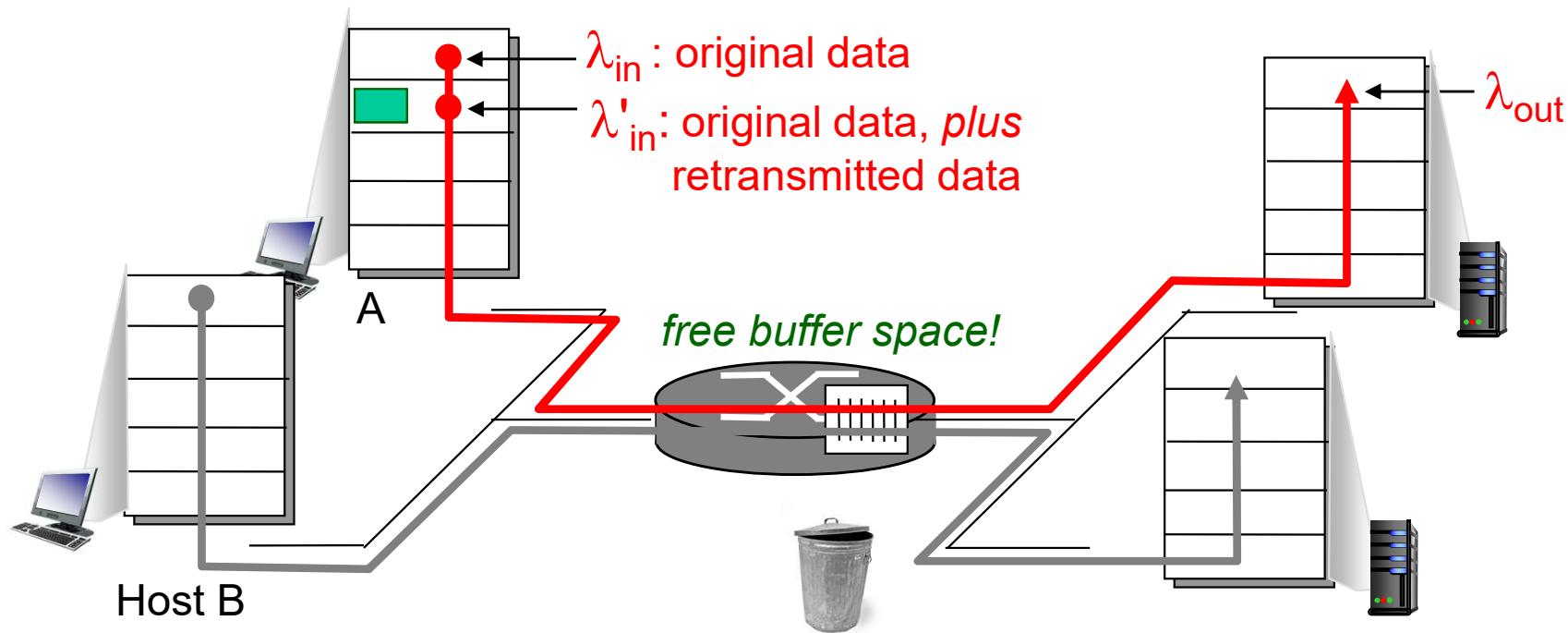
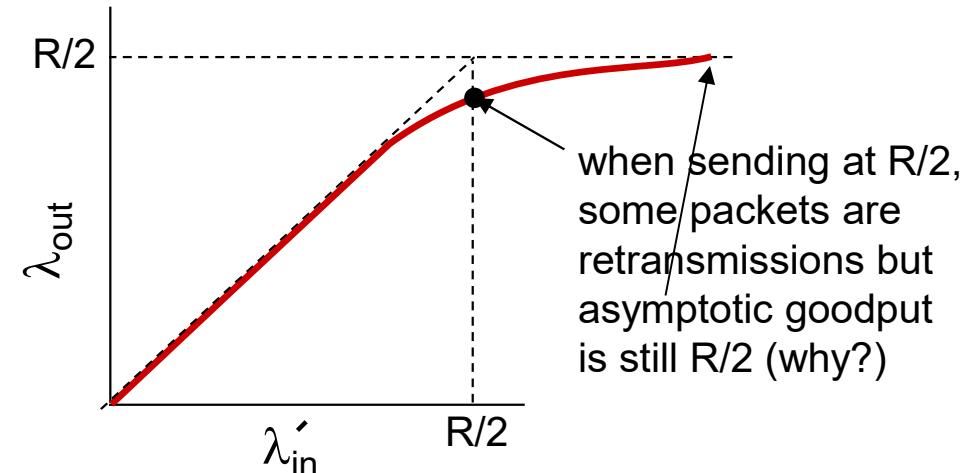
- ❖ sender only resends if  
packet *known* to be lost



# Causes/costs of congestion: scenario 2

## Idealization: *known loss*

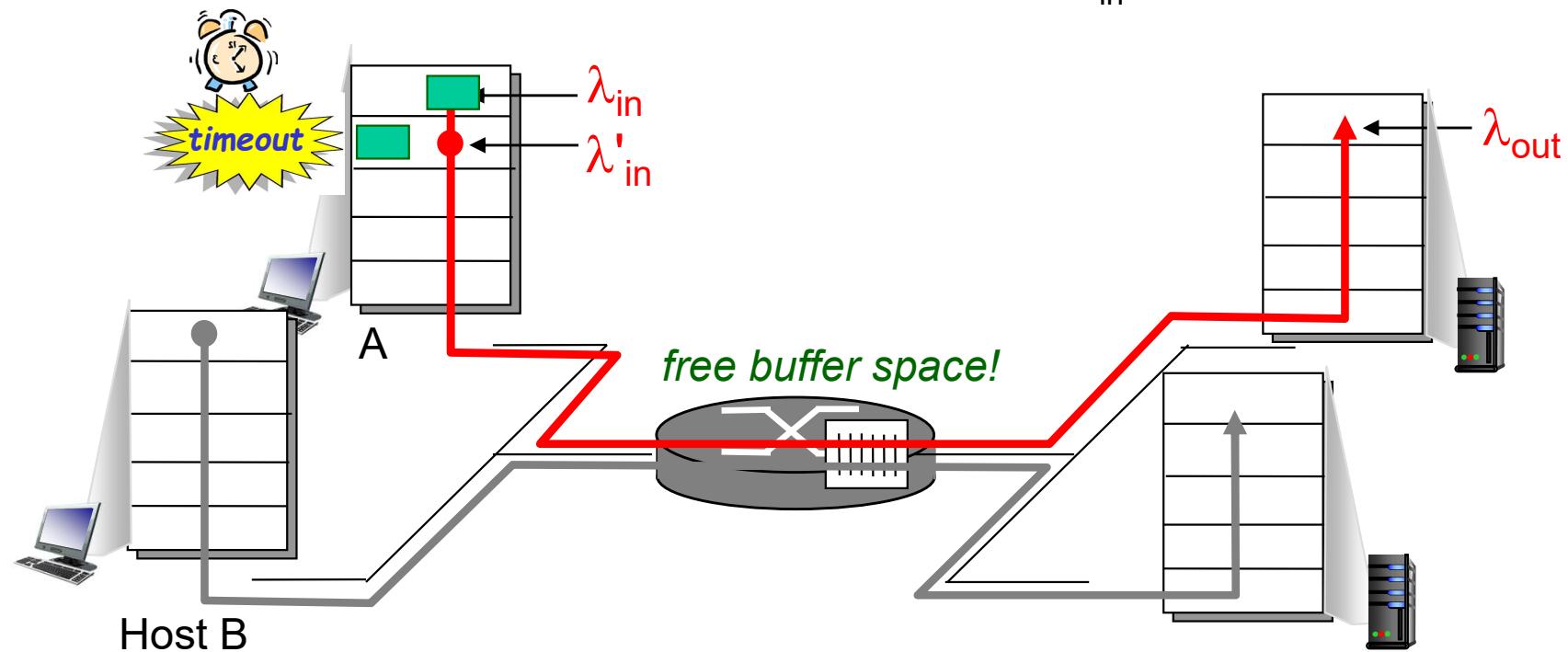
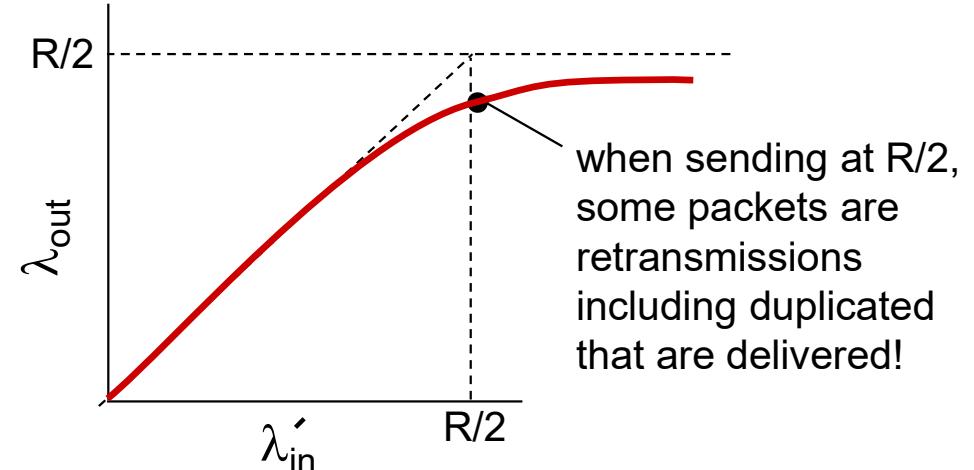
- packets can be lost, dropped at router due to full buffers
- ❖ sender only resends if packet *known* to be lost



# Causes/costs of congestion: scenario 2

## Realistic: *duplicates*

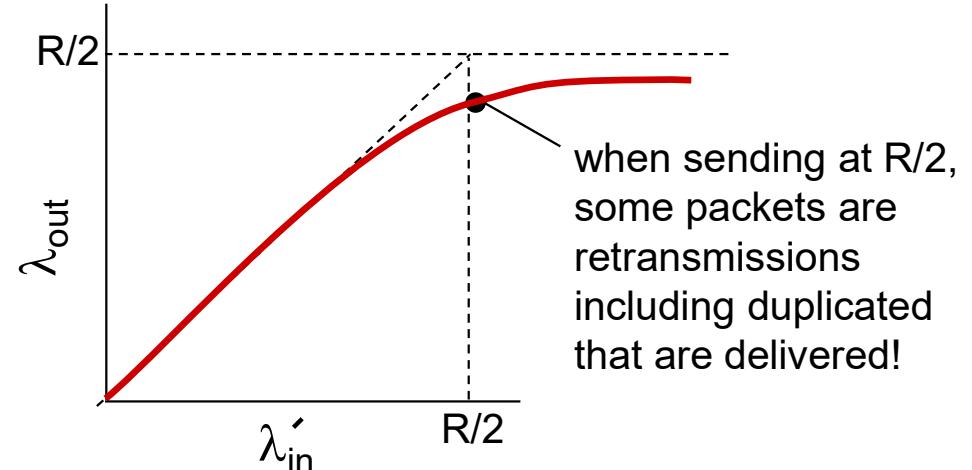
- ❖ packets can be lost, dropped at router due to full buffers
- ❖ sender times out prematurely, sending **two** copies, both of which are delivered



# Causes/costs of congestion: scenario 2

## Realistic: *duplicates*

- ❖ packets can be lost, dropped at router due to full buffers
- ❖ sender times out prematurely, sending **two** copies, both of which are delivered



## “costs” of congestion:

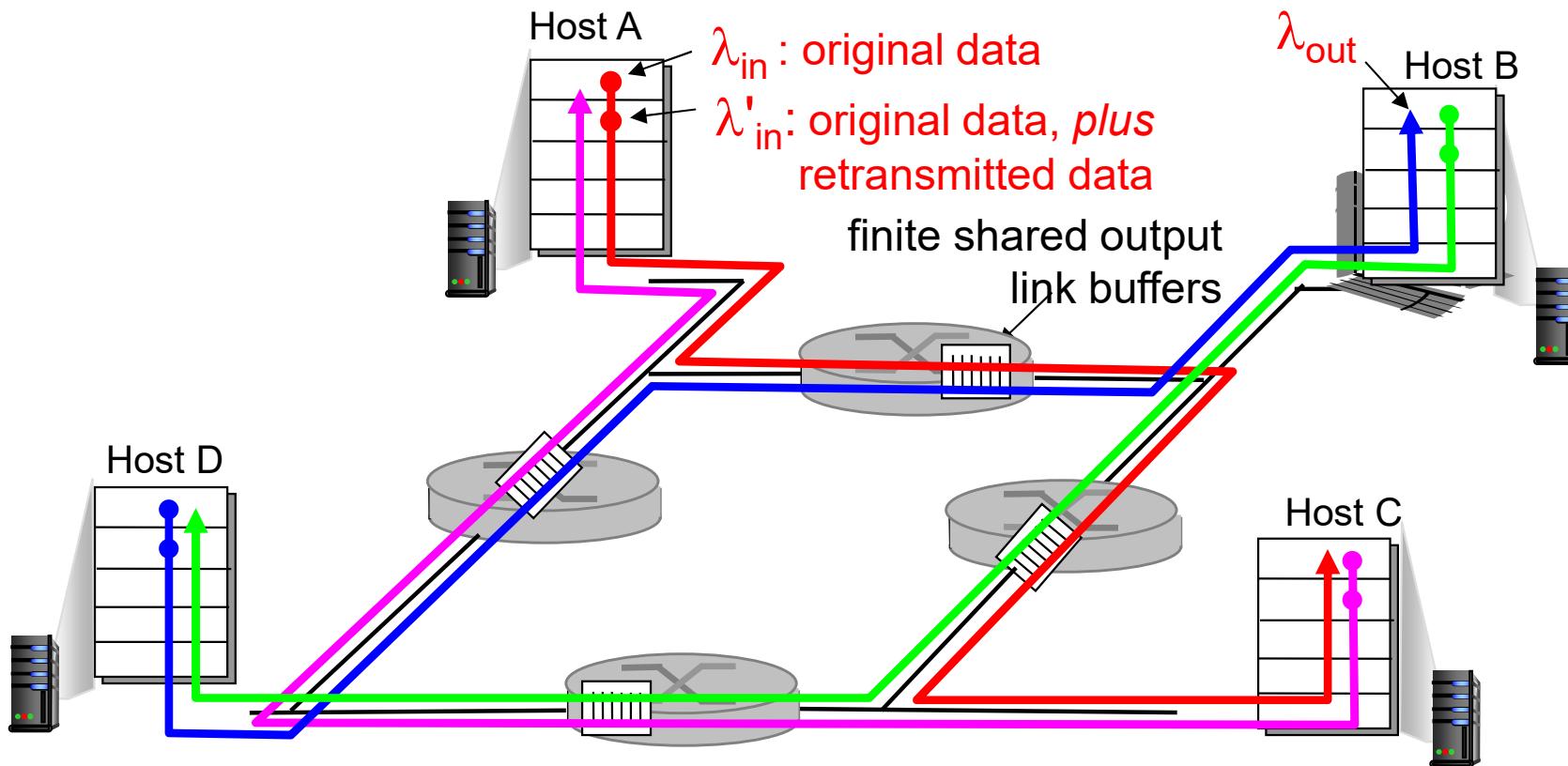
- ❖ more work (retrans) for given “goodput”
- ❖ unneeded retransmissions: link carries multiple copies of pkt
  - decreasing goodput

# Causes/costs of congestion: scenario 3

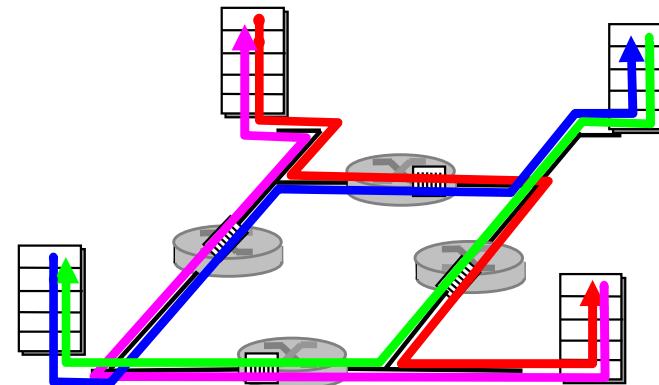
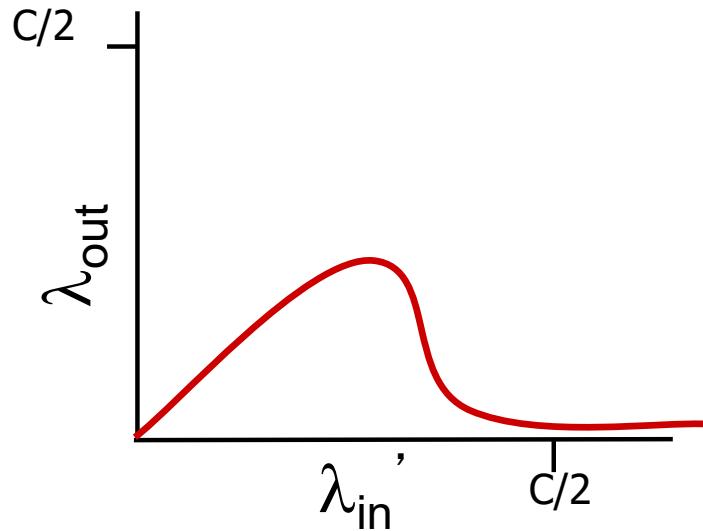
- ❖ four senders
- ❖ multihop paths
- ❖ timeout/retransmit

Q: what happens as  $\lambda_{in}$  and  $\lambda'_{in}$  increase ?

A: as red  $\lambda'_{in}$  increases, all arriving blue pkts at upper queue are dropped, blue throughput  $\rightarrow 0$



## Causes/costs of congestion: scenario 3



another “cost” of congestion:

- ❖ when packet dropped, any “upstream transmission capacity used for that packet was wasted!

# Approaches towards congestion control

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

# 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

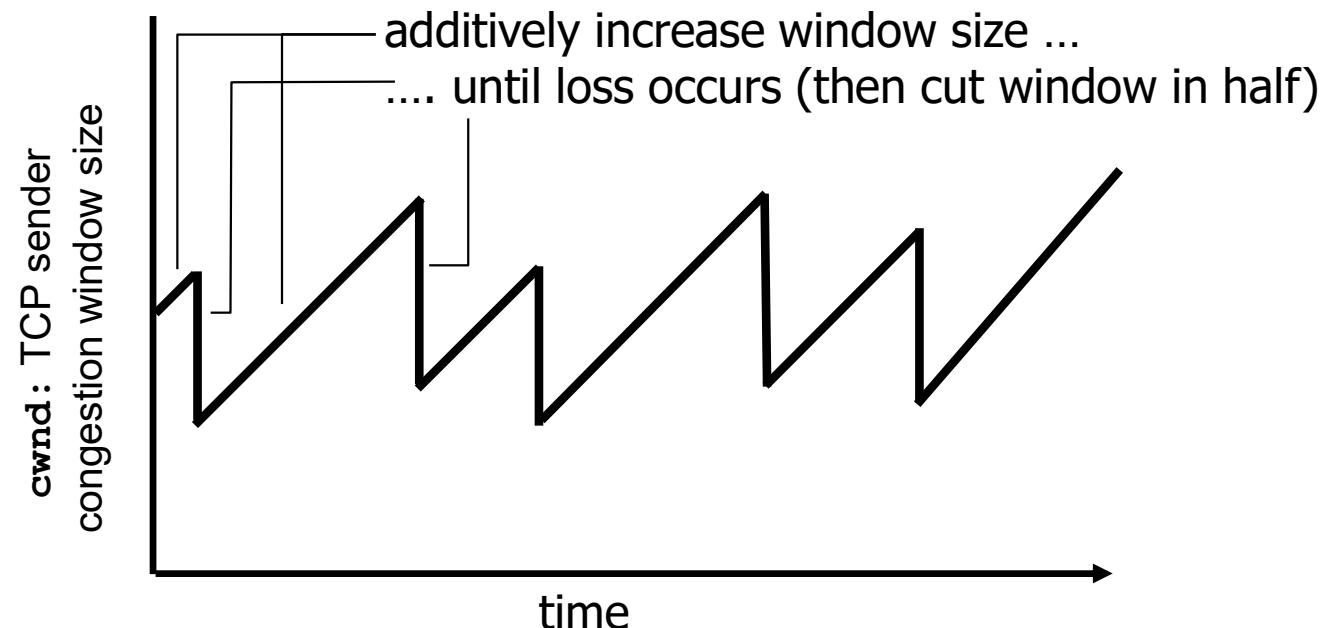
3.6 principles of congestion control

3.7 TCP congestion control

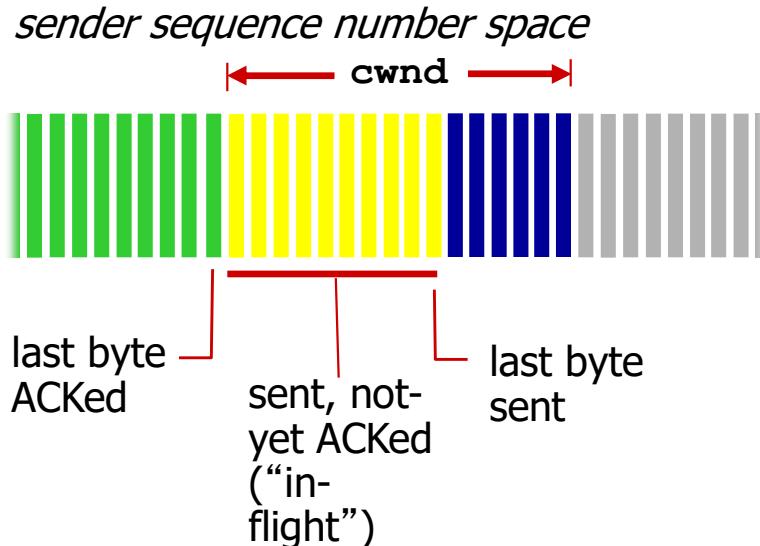
# TCP congestion control: additive increase multiplicative decrease

- ❖ *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:

$$\frac{\text{LastByteSent} - \text{LastByteAcked}}{\text{RTT}} \leq \text{Min}\{\text{cwnd}, \text{rwnd}\}$$

*TCP sending rate:*

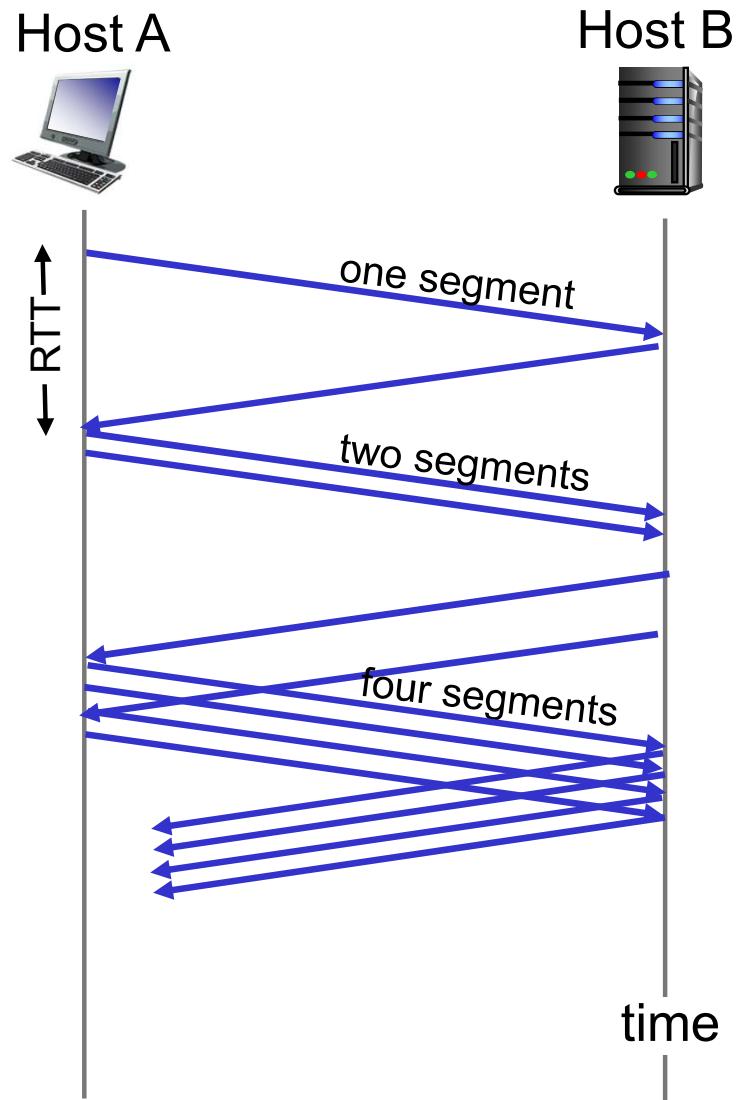
- ❖ roughly: send cwnd bytes, wait RTT for ACKS, then send more bytes

$$\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

- ❖ **cwnd** is dynamic, function of perceived network congestion

# 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)

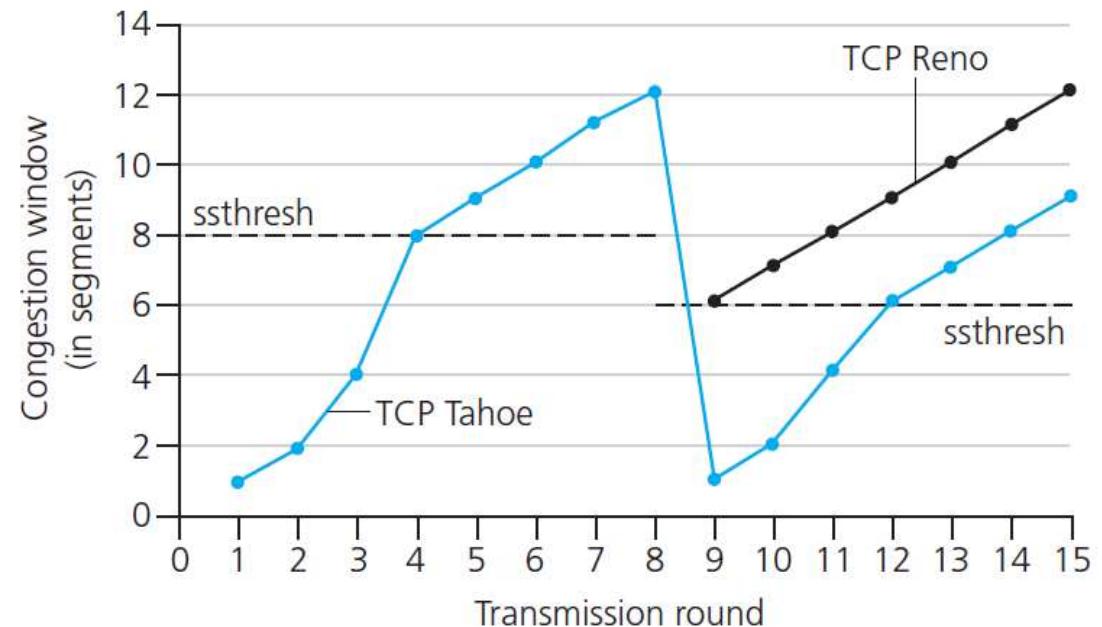
# TCP: switching from slow start to CA

**Q:** when should the exponential increase switch to linear?

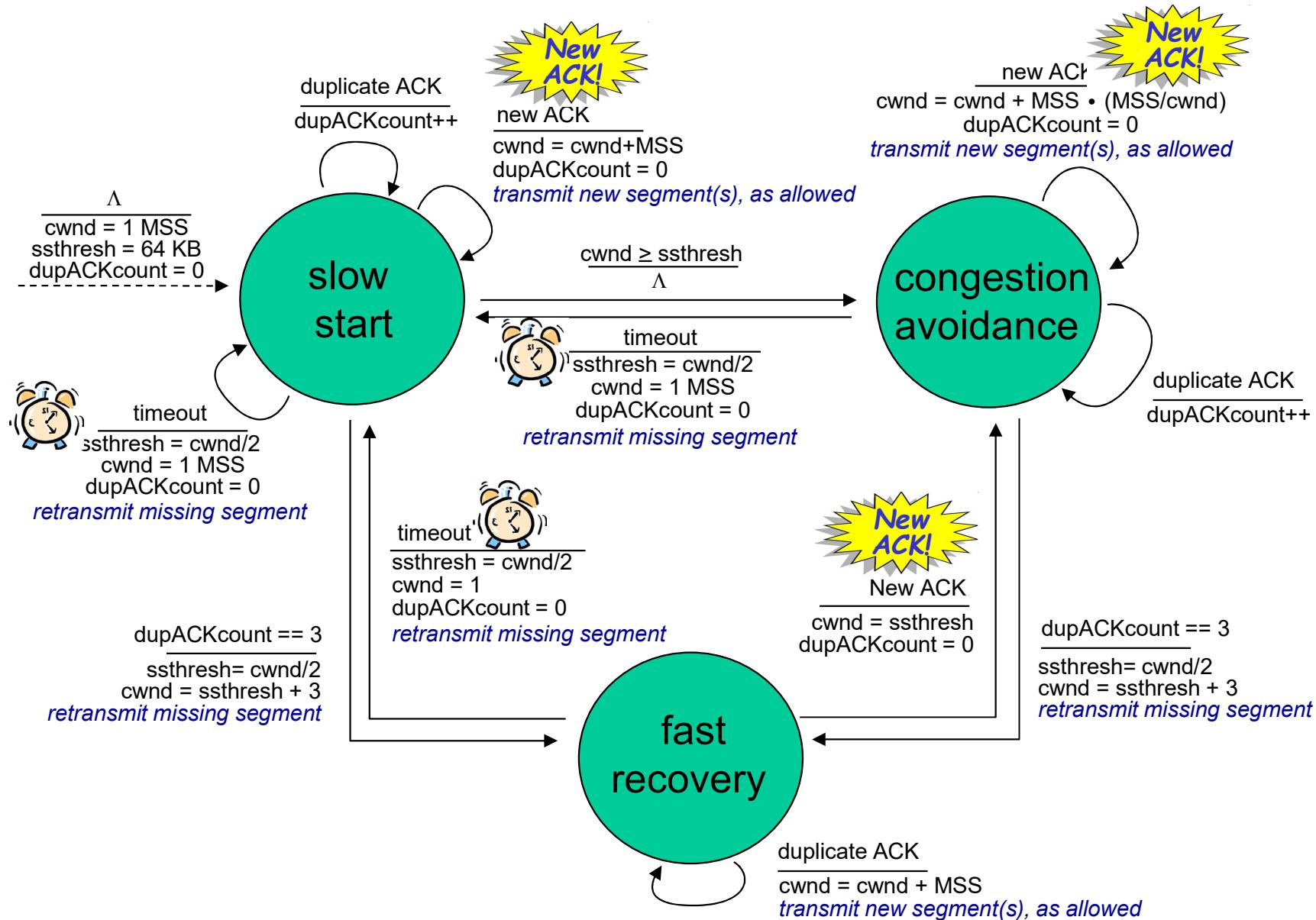
**A:** when **cwnd** gets to 1/2 of its value before timeout.

## Implementation:

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



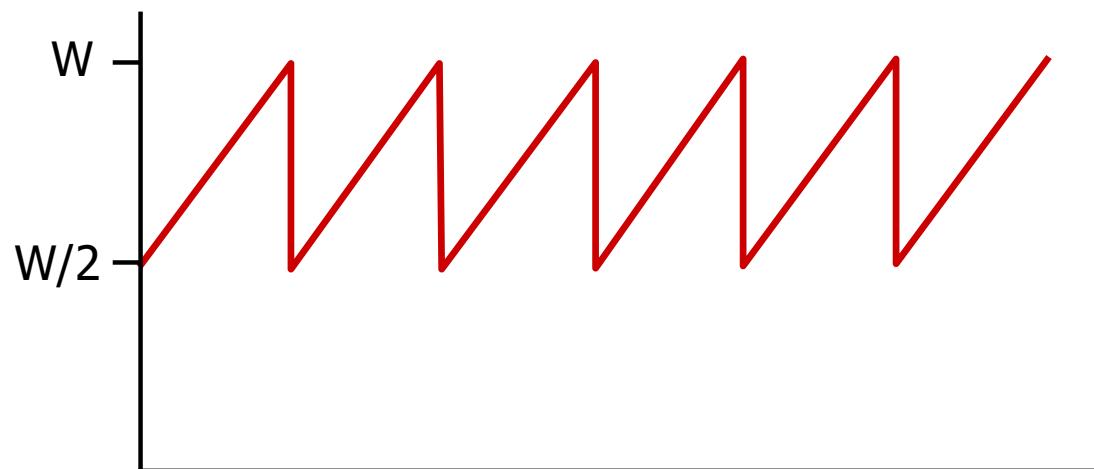
# Summary: TCP Congestion Control



# TCP throughput

- ❖ avg. TCP thruput 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  $\frac{3}{4} W$
  - avg. thruput is  $\frac{3}{4}W$  per RTT

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



# TCP Futures: TCP over “long, fat pipes”

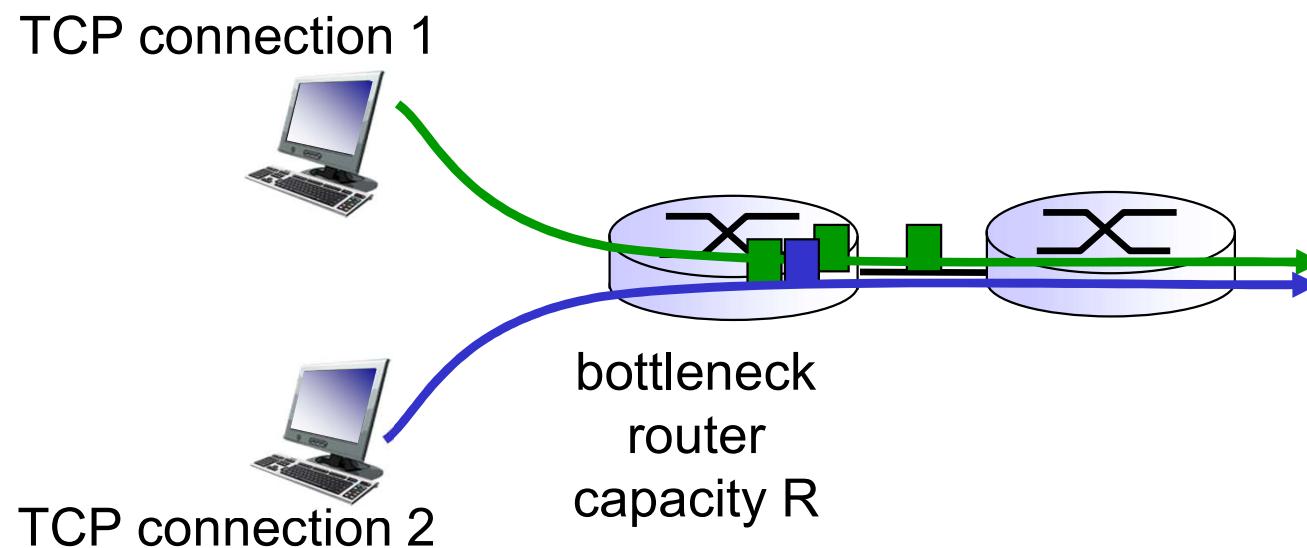
- ❖ example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- ❖ requires  $W = 83,333$  in-flight segments
- ❖ throughput in terms of segment loss probability,  $L$  [Mathis 1997]:

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

- 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

# TCP Fairness

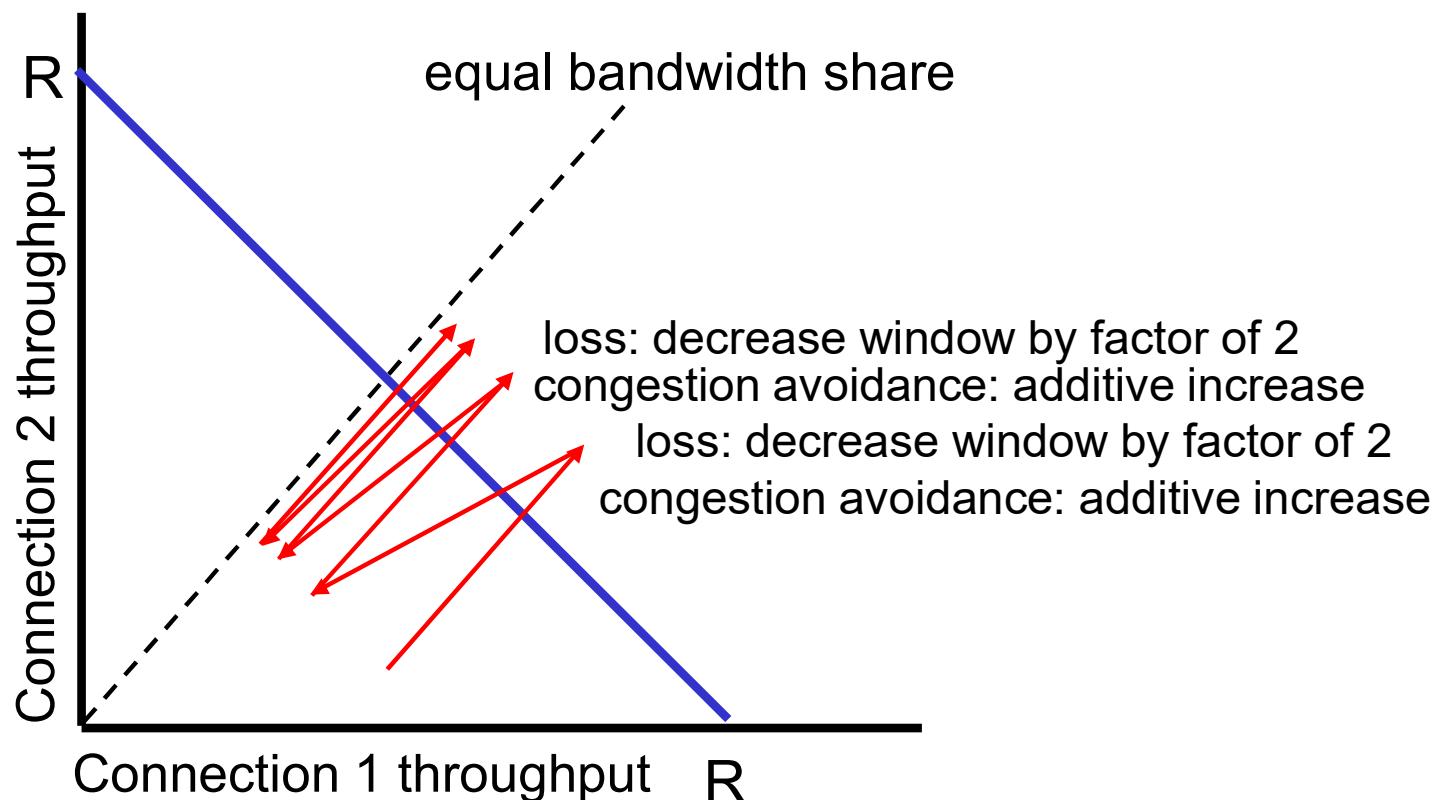
*fairness goal:* if  $K$  TCP sessions share same bottleneck link of bandwidth  $R$ , each should have average rate of  $R/K$



# Why is TCP fair?

two competing sessions:

- ❖ additive increase gives slope of 1, as throughout increases
- ❖ multiplicative decrease decreases throughput proportionally



# Fairness (more)

## *Fairness and UDP*

- ❖ multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- ❖ instead use UDP:
  - send audio/video at constant rate, tolerate packet loss

## *Fairness, parallel TCP connections*

- ❖ application can open multiple parallel connections between two hosts
- ❖ web browsers do this
- ❖ e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate  $R/10$
  - new app asks for 11 TCPs, gets  $R/2$

# Chapter 3: summary

- ❖ principles behind transport layer services:
    - multiplexing, demultiplexing
    - reliable data transfer
    - flow control
    - congestion control
  - ❖ instantiation, implementation in the Internet
    - UDP
    - TCP
- next:
- ❖ leaving the network “edge” (application, transport layers)
  - ❖ into the network “core”