

Chapter 2

Application Layer

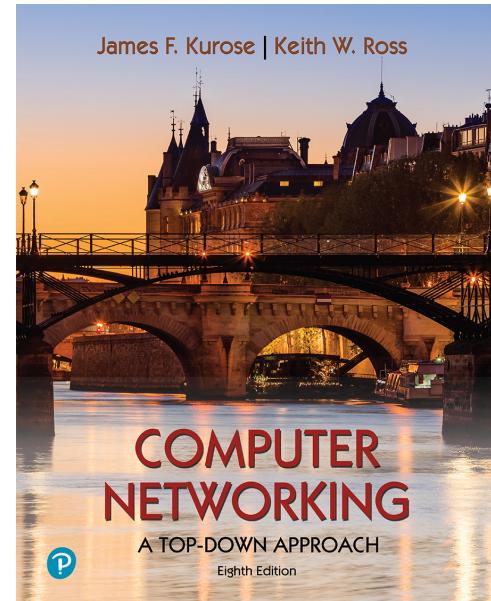
A note on the use of these PowerPoint slides:

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



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

Day 10: Principles of Transport and UDP



CSEE 4119
Computer Networks
Ethan Katz-Bassett

 COLUMBIA UNIVERSITY

IN THE CITY OF NEW YORK

Slides adapted from (and often identical to) slides from Kurose and Ross.

All material copyright 1996-2020

J.F.Kurose and K.W.Ross, All Rights Reserved

Oct 6 admin

- Masks ***strongly recommended***, over nose and mouth
- Attendance & participation ***not*** required
- If you are not feeling well or were exposed to COVID,
please stay home
- Videos of lectures available

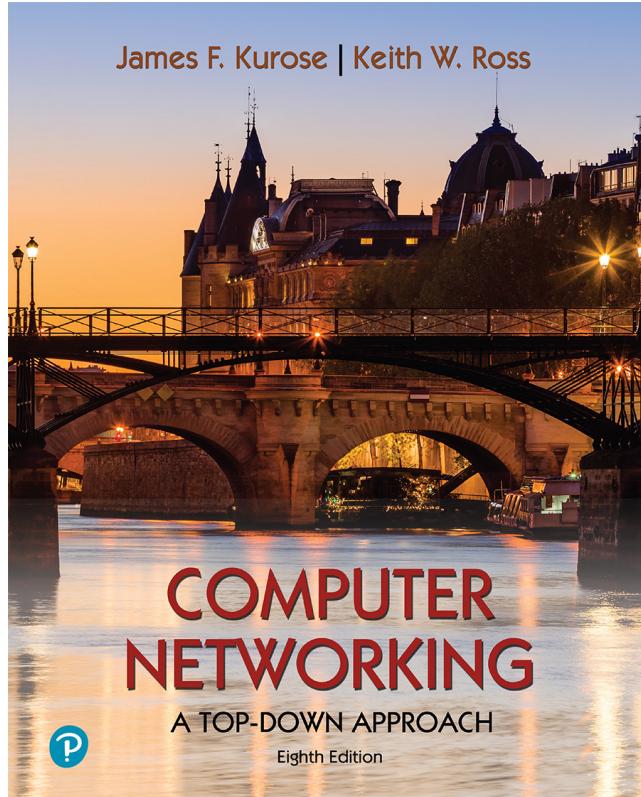
- HW1 due last night
- Project I Preliminary Stage due next week
- HW2 released by tomorrow
- Project I Final Stage released soon

Chapter 3

Transport Layer

Adapted from (and often identical to) slides from Kurose and Ross.

© 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

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

Today's class

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

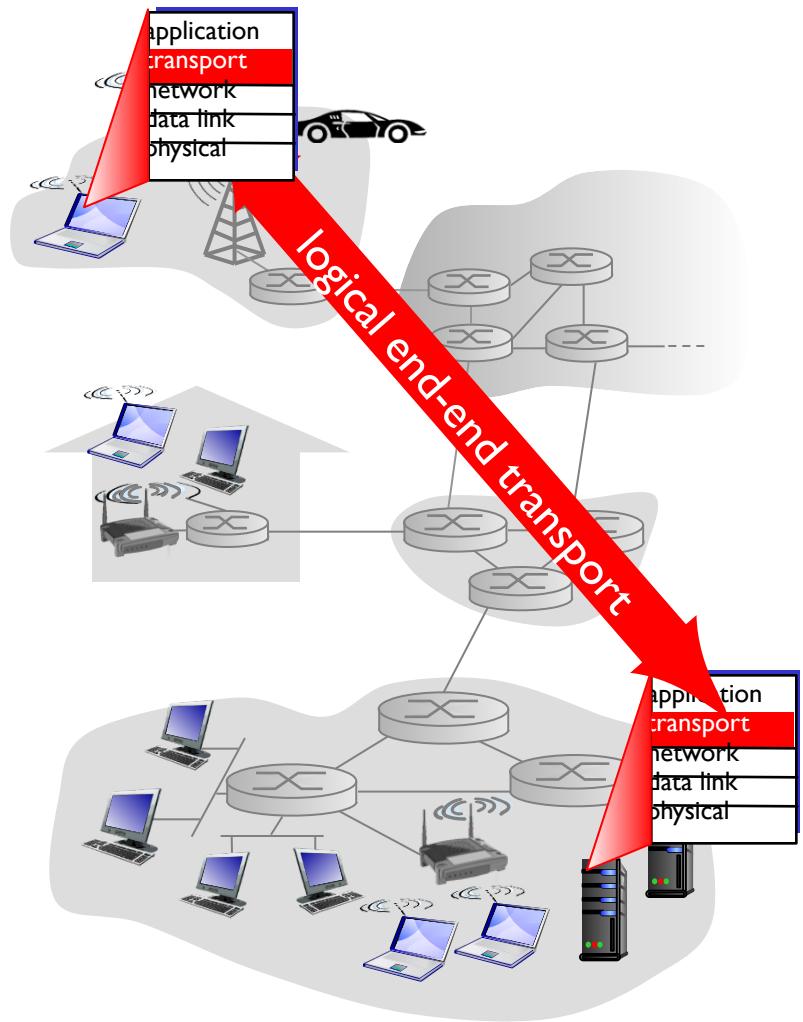
3.6 principles of congestion control

3.7 TCP congestion control

3.8 Evolution of transport-layer functionality

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
 - receive 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

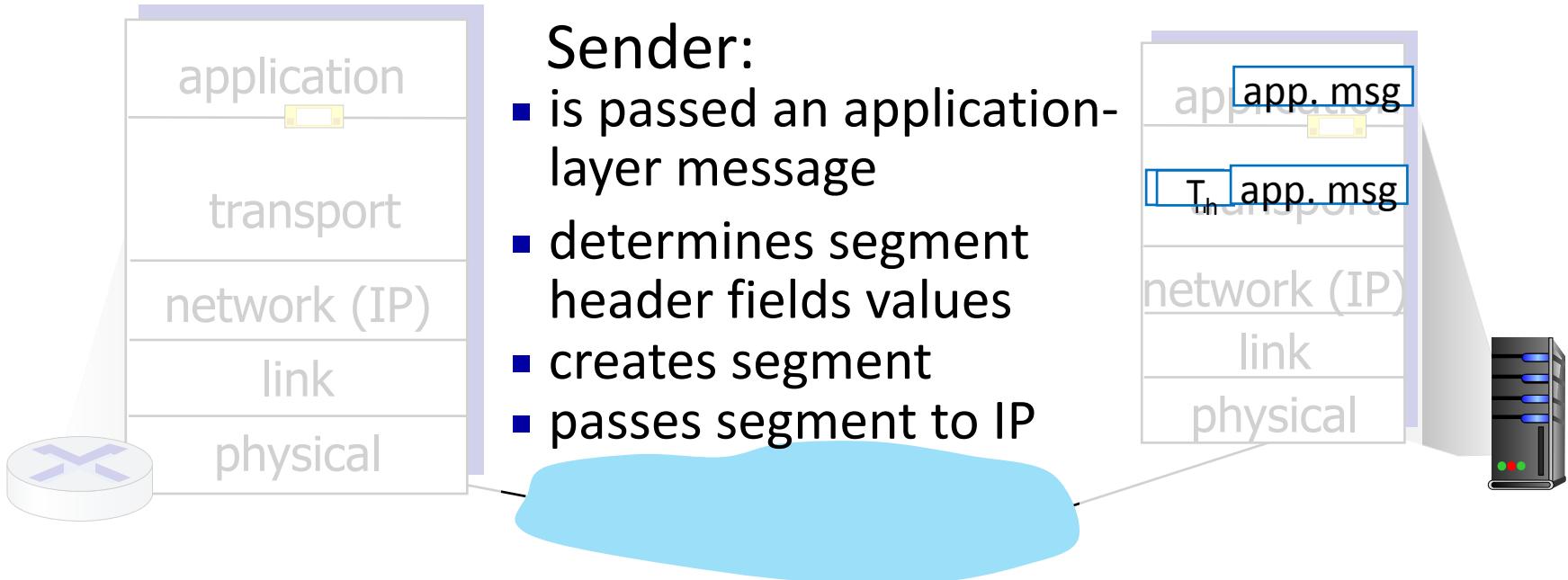
- *network layer*: logical communication between *hosts*
- *transport layer*: logical communication between *processes*
 - relies on and enhances network layer services

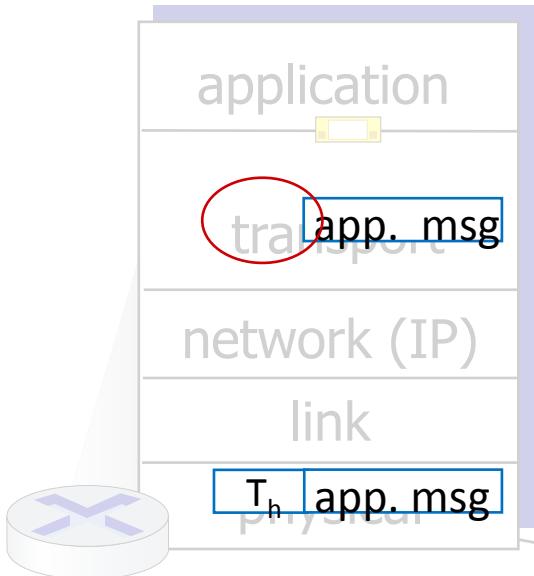
household analogy:

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

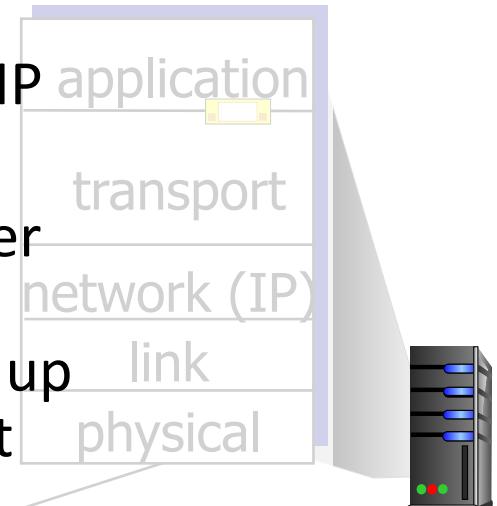
Transport Layer Actions





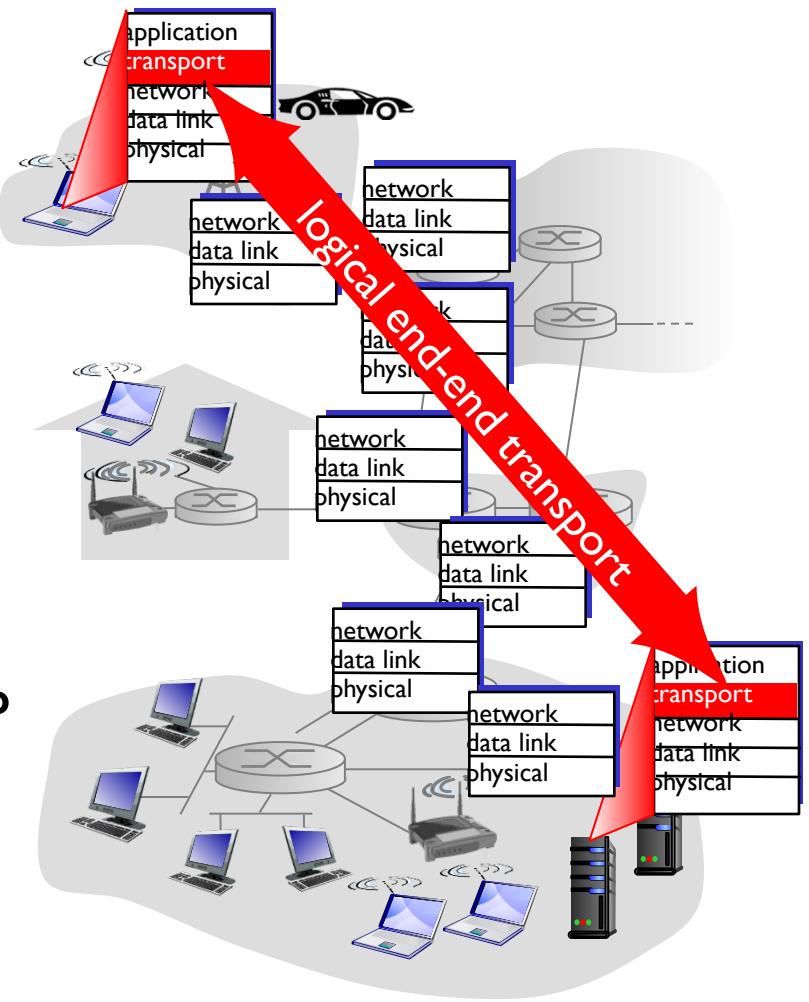
Receiver:

- receives segment from IP
- checks header values
- extracts application-layer message
- demultiplexes message up to application via socket



Internet transport-layer protocols

- TCP: reliable, in-order delivery
(Transmission Control Protocol)
 - congestion control
 - flow control
 - connection setup
 - bytestream abstraction
- UDP: unreliable, unordered delivery
(User Datagram Protocol)
 - no-frills extension of “best-effort” IP
 - datagram abstraction
- 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

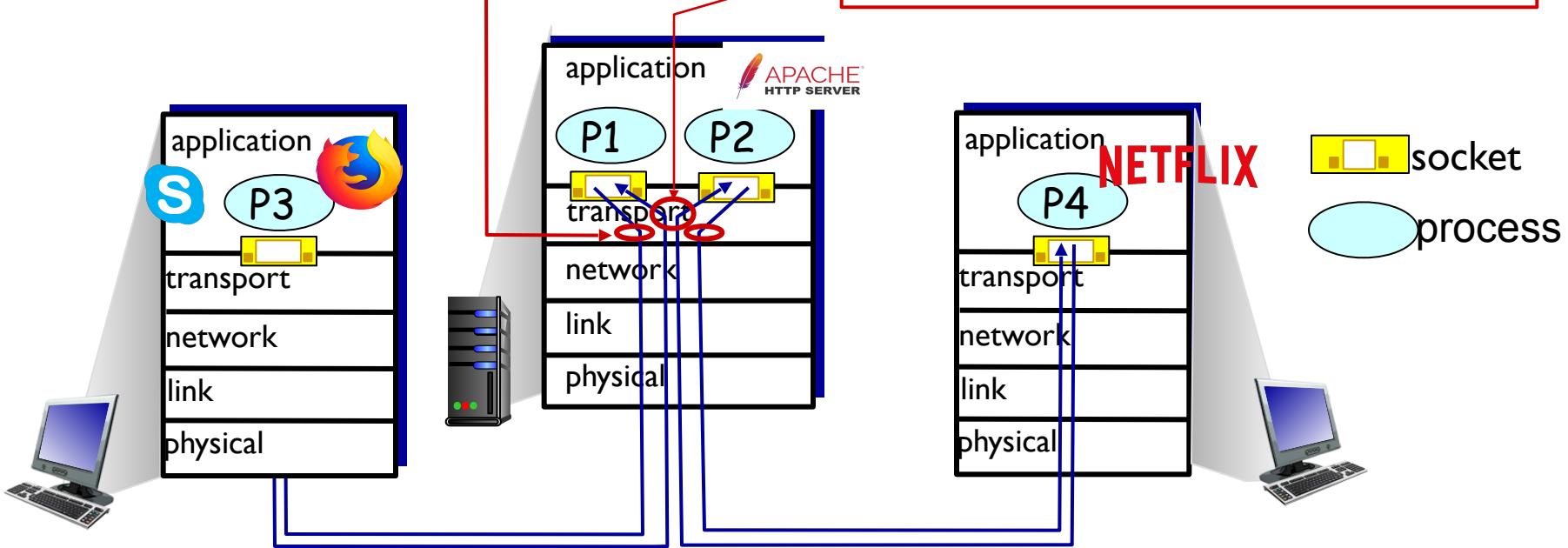
Multiplexing/demultiplexing

multiplexing at sender:

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

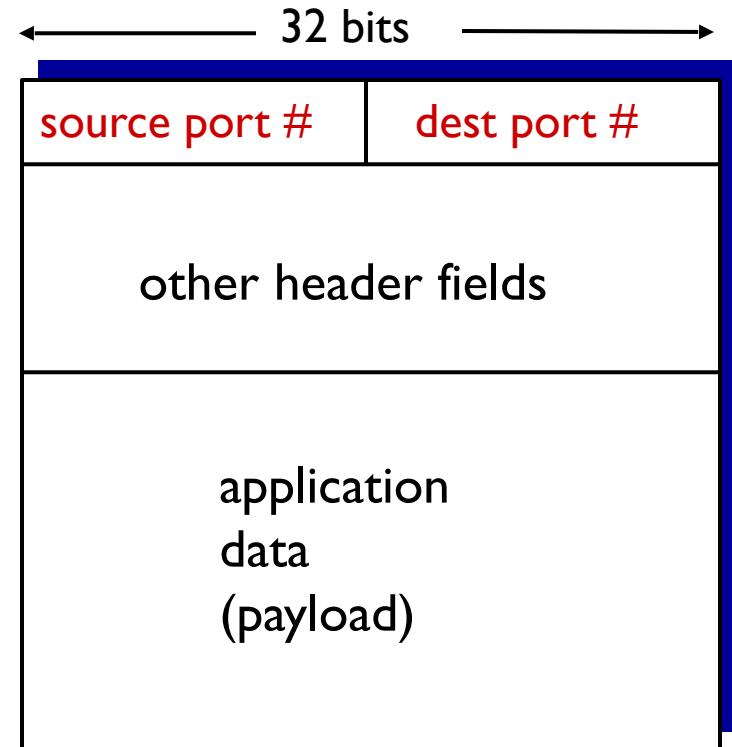
demultiplexing at receiver:

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

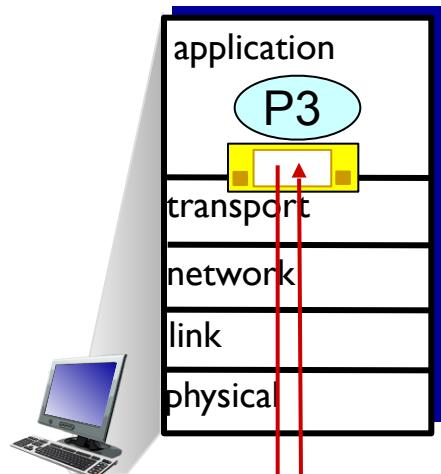
- created socket has host-local port #:

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



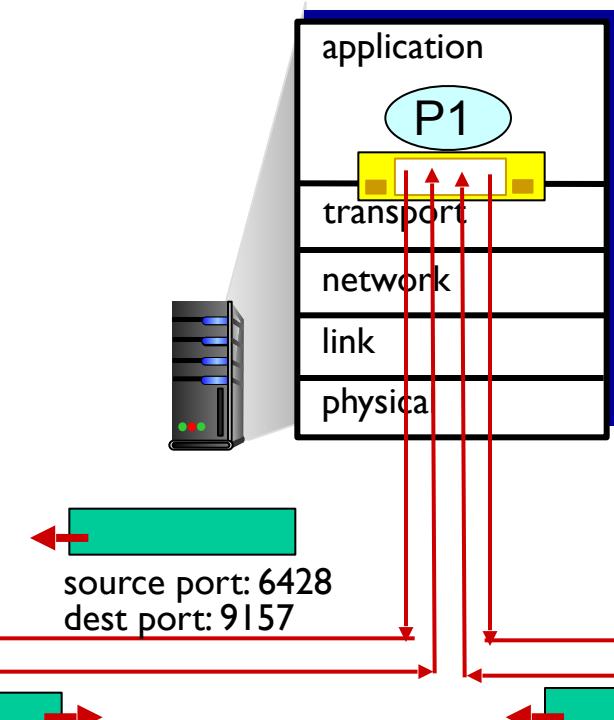
Connectionless demux: example

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



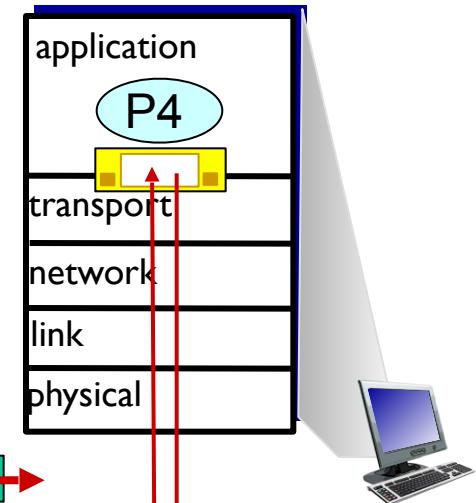
source port: 9157
dest port: 6428

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



source port: ?
dest port: ?

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

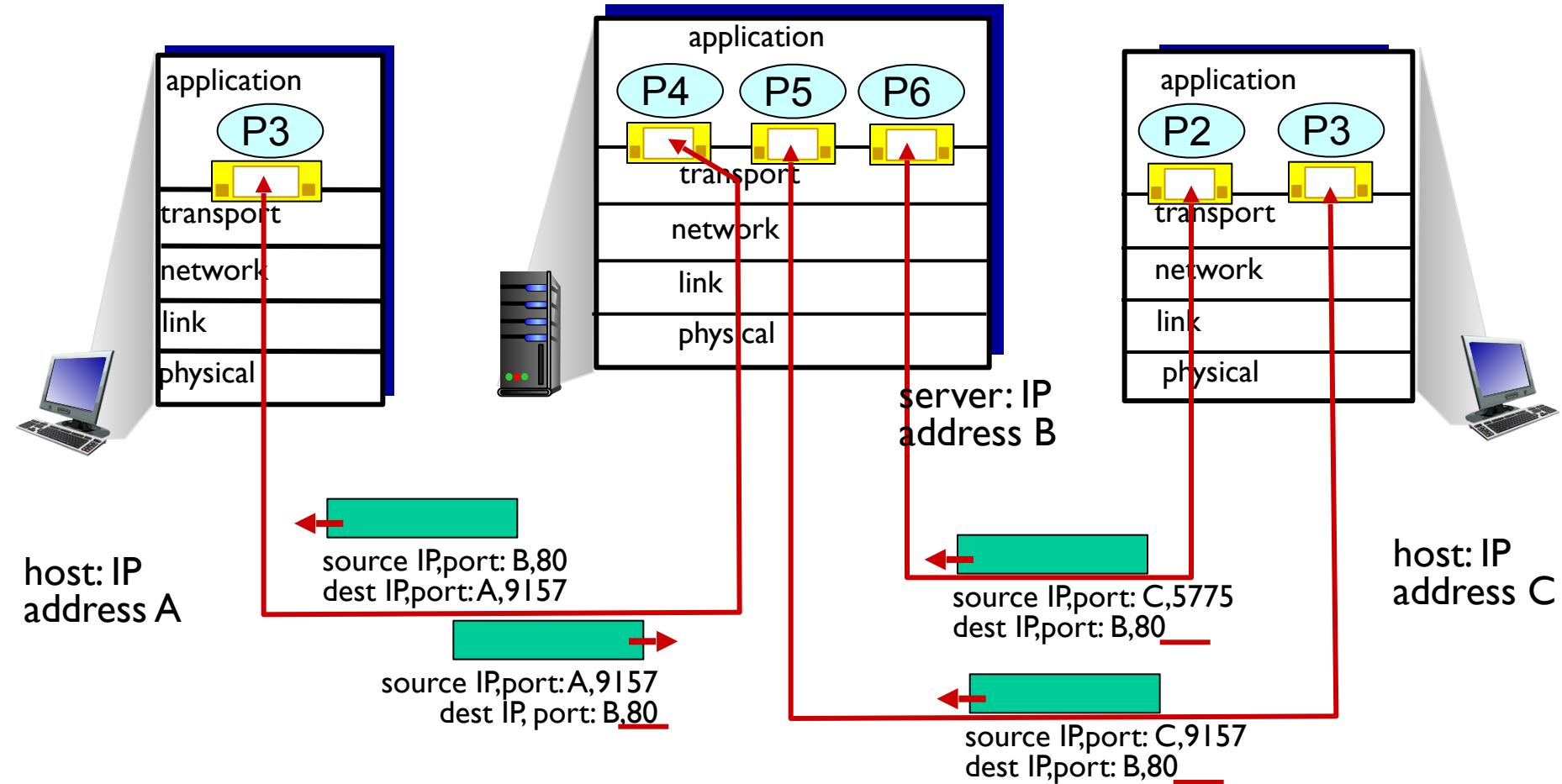


source port: ?
dest port: ?

Connection-oriented demux

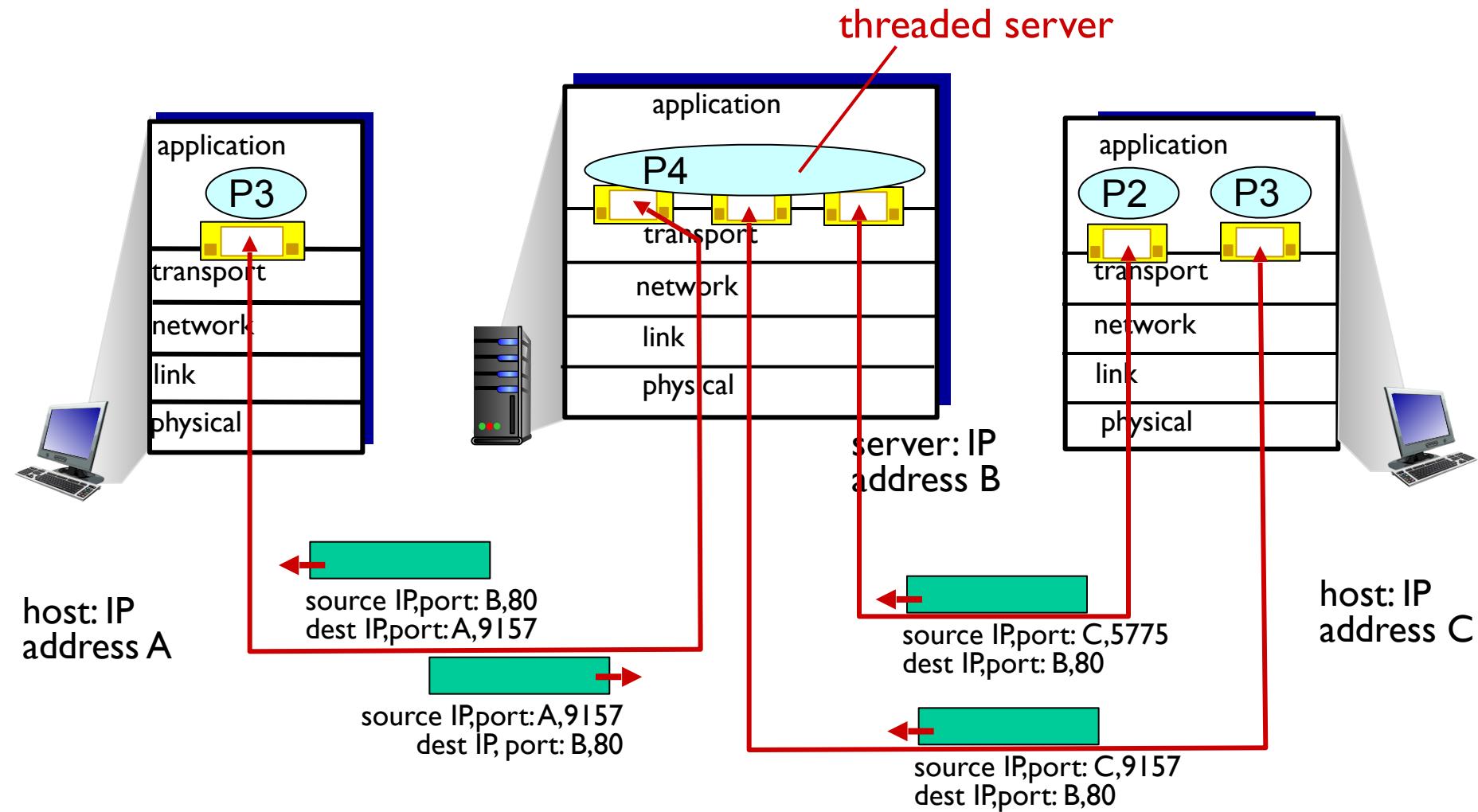
- TCP socket identified by **4-tuple**:
 - source IP address
 - source port number
 - dest IP address
 - dest port number
- demux: receiver uses **all 4 values** to direct segment to appropriate socket
- server host may support many simultaneous TCP sockets:
 - each socket identified by its own 4-tuple
- 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



Summary

- Multiplexing, demultiplexing: based on segment, datagram header field values
- **UDP:** demultiplexing using destination port number (only)
- **TCP:** demultiplexing using 4-tuple: source and destination IP addresses, and port numbers
- Multiplexing/demultiplexing happen at *all* layers

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP
if time allows

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

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

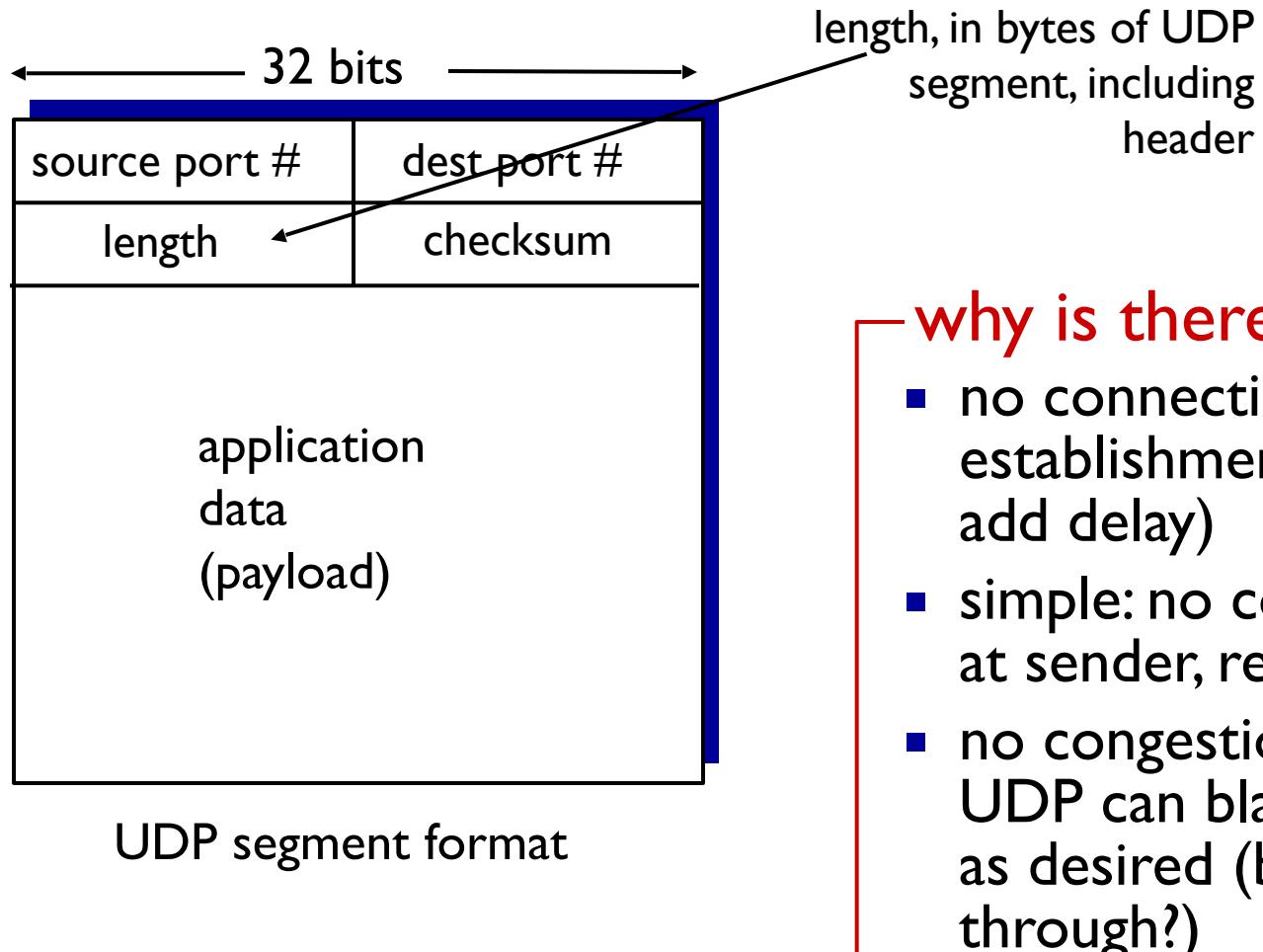
3.6 principles of congestion control

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 uses include:
 - streaming multimedia apps (loss tolerant, rate sensitive)
 - DNS
 - SNMP
 - HTTP/3
- reliable transfer over UDP:
 - add reliability at application layer
 - application-specific error recovery!
 - e.g., QUIC, HTTP/3

UDP: segment header



why is there a UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- no congestion control: UDP can blast away as fast as desired (but will it get through?)

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
	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
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	0	1	1

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

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Summary: UDP

- “no frills” protocol:
 - segments may be lost, delivered out of order
 - best effort service: “send and hope for the best”
- UDP has its plusses:
 - no setup/handshaking needed (no RTT incurred)
 - can function when network service is compromised
 - some applications don’t need TCP guarantees, so don’t need to incur its overheads
- build additional functionality on top of UDP in application layer (e.g., HTTP/3)

Day 11: Principles of Reliable Data Transfer



CSEE 4119
Computer Networks
Ethan Katz-Bassett

 COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

Slides adapted from (and often identical to) slides from Kurose and Ross.

All material copyright 1996-2020

J.F.Kurose and K.W.Ross, All Rights Reserved

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

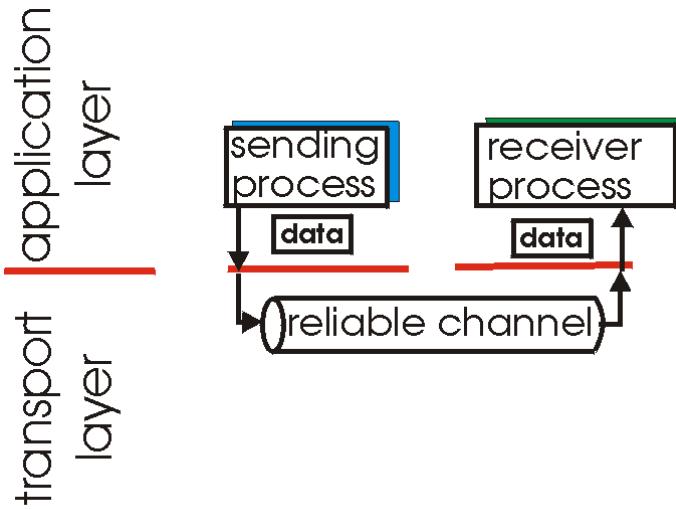
3.6 principles of congestion control

3.7 TCP congestion control

Today's class

Principles of reliable data transfer

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

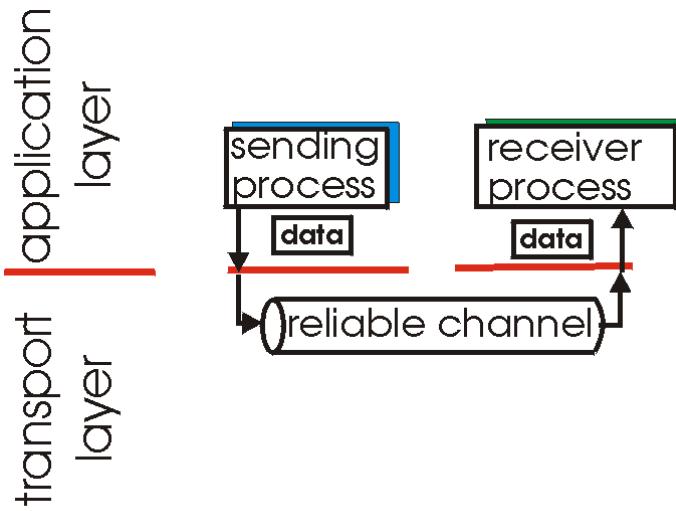


(a) provided service

- What is reliable channel?

Principles of reliable data transfer

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

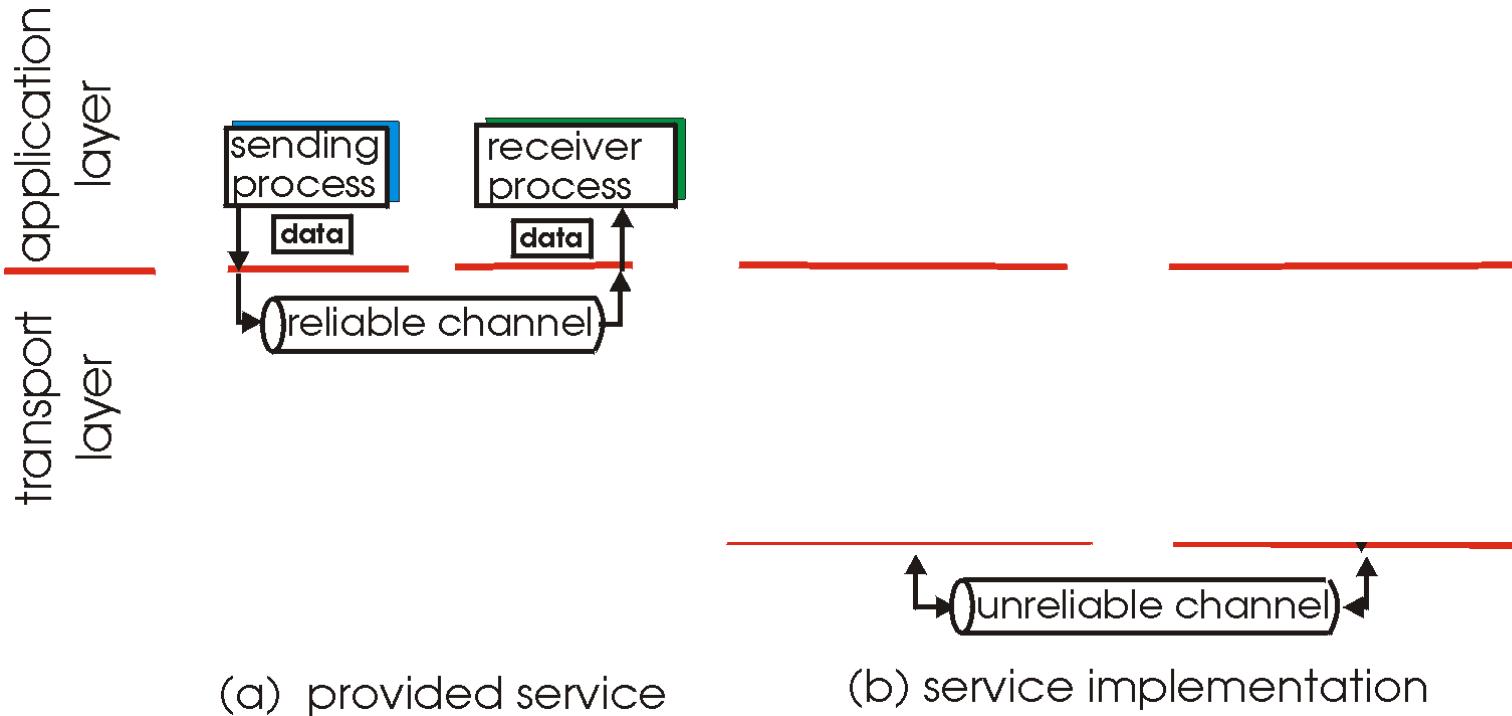


(a) provided service

- What is reliable channel? bits in order, no corruption/loss

Principles of reliable data transfer

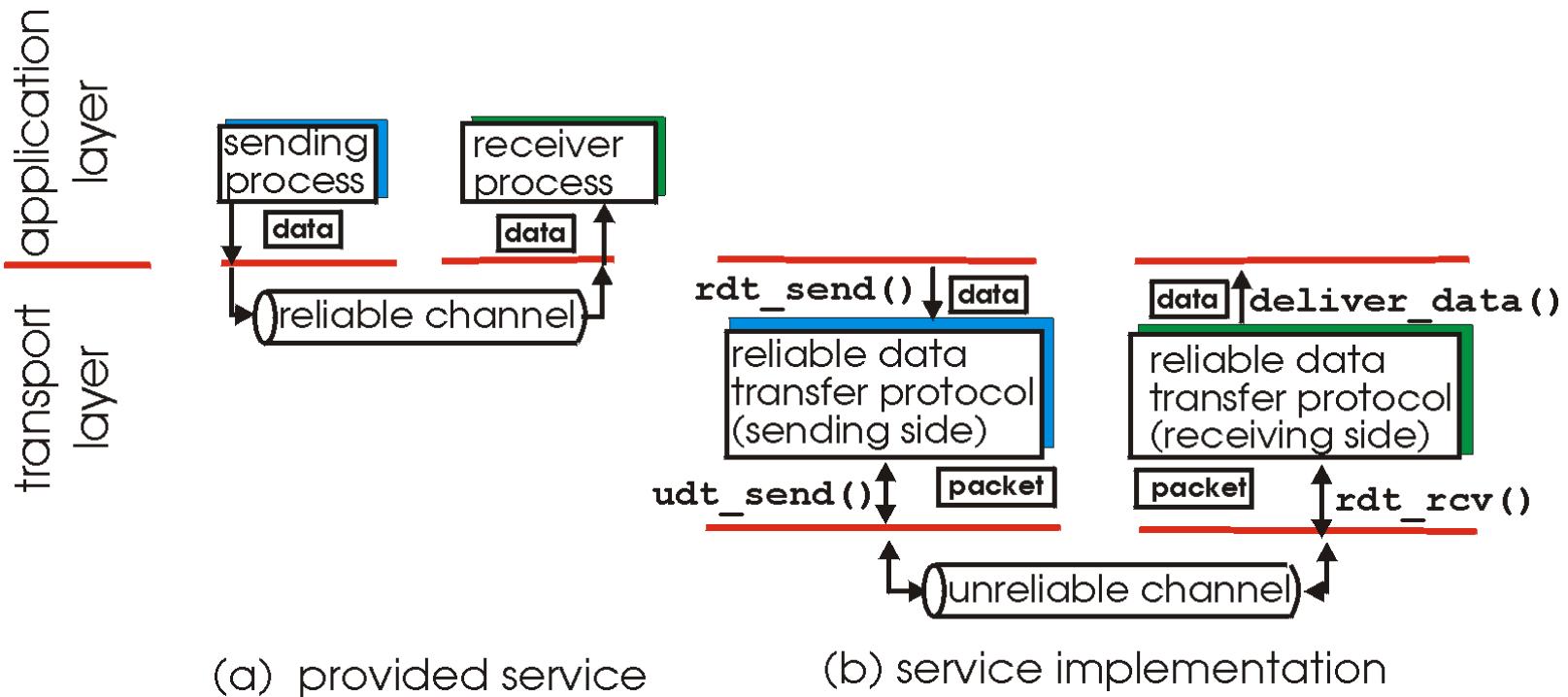
- important in application, transport, link layers
 - top-10 list of important networking topics!



- What is reliable channel? bits in order, no corruption/loss
- characteristics of unreliable channel will determine complexity of reliable data transfer protocol

Principles of reliable data transfer

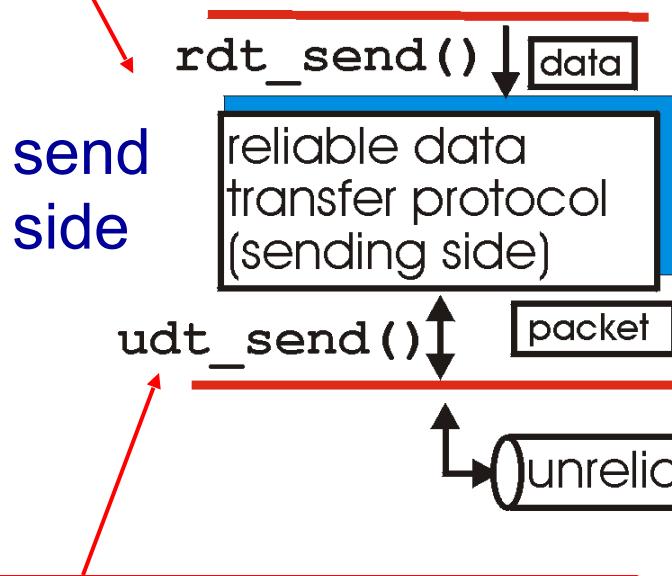
- important in application, transport, link layers
 - top-10 list of important networking topics!



- What is reliable channel? bits in order, no corruption/loss
- characteristics of unreliable channel will determine complexity of reliable data transfer protocol

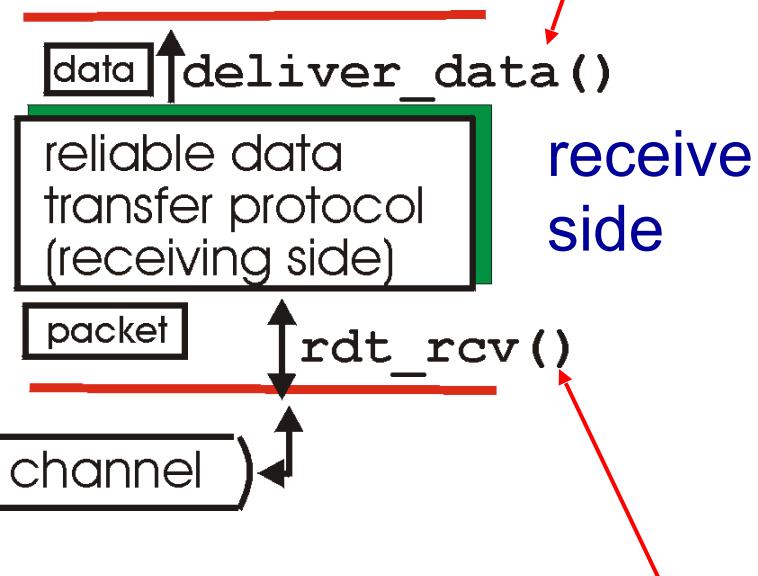
Reliable data transfer: getting started

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



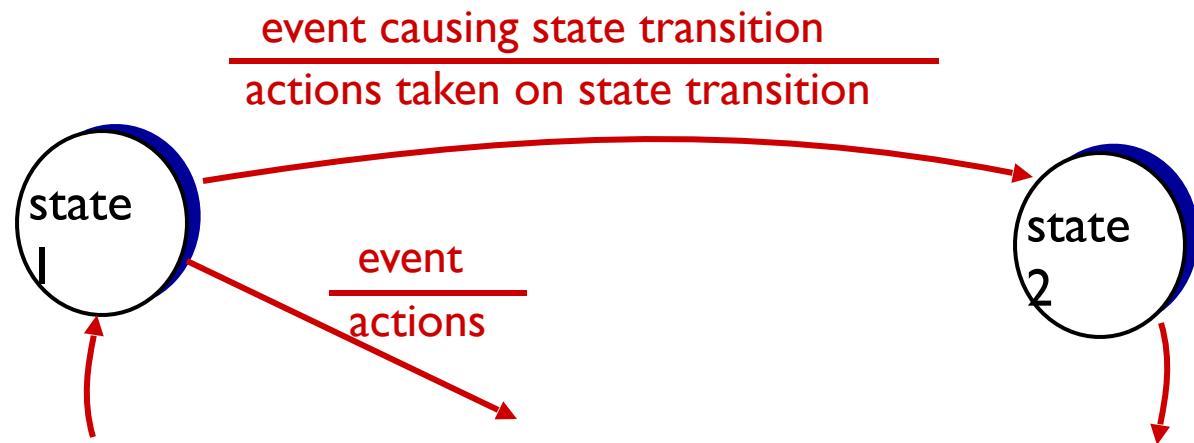
rdt_rcv() : called when packet arrives on rcv-side of channel

Reliable data transfer: getting started

we'll:

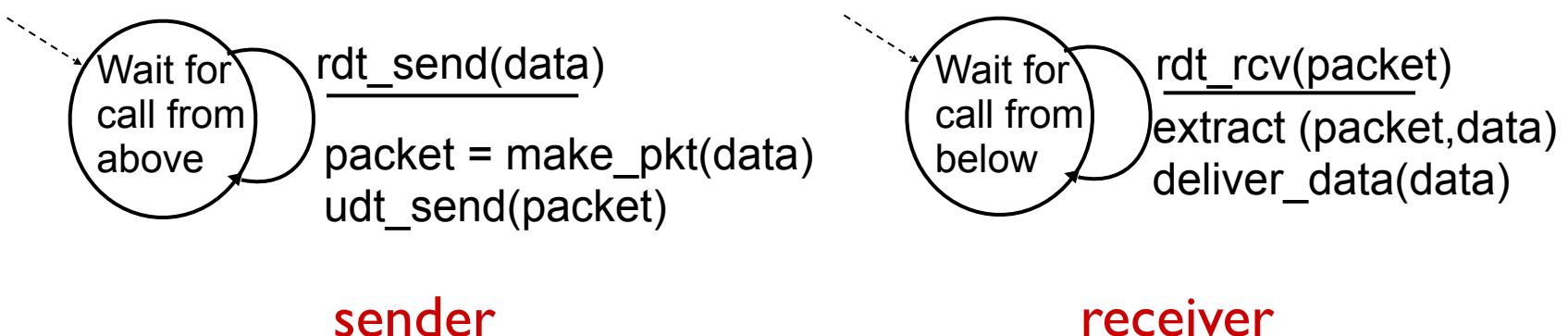
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
 - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver

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
- retransmit packets with errors
- *the question: how to recover from errors:*
 - *checksum* to detect bit 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

rdt_send(data)

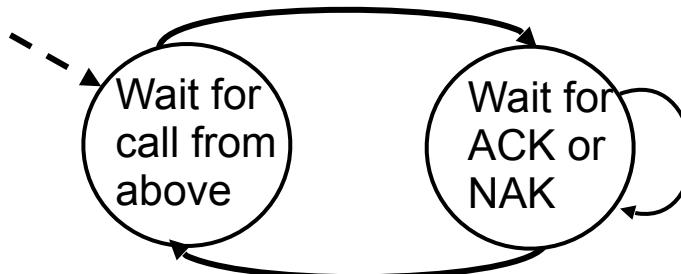
snkpkt = make_pkt(data, checksum)

udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
 Λ

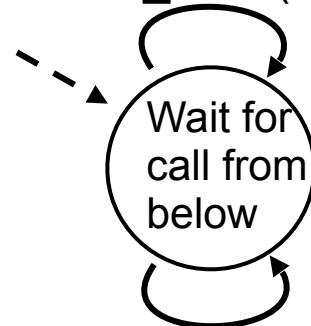
sender



receiver

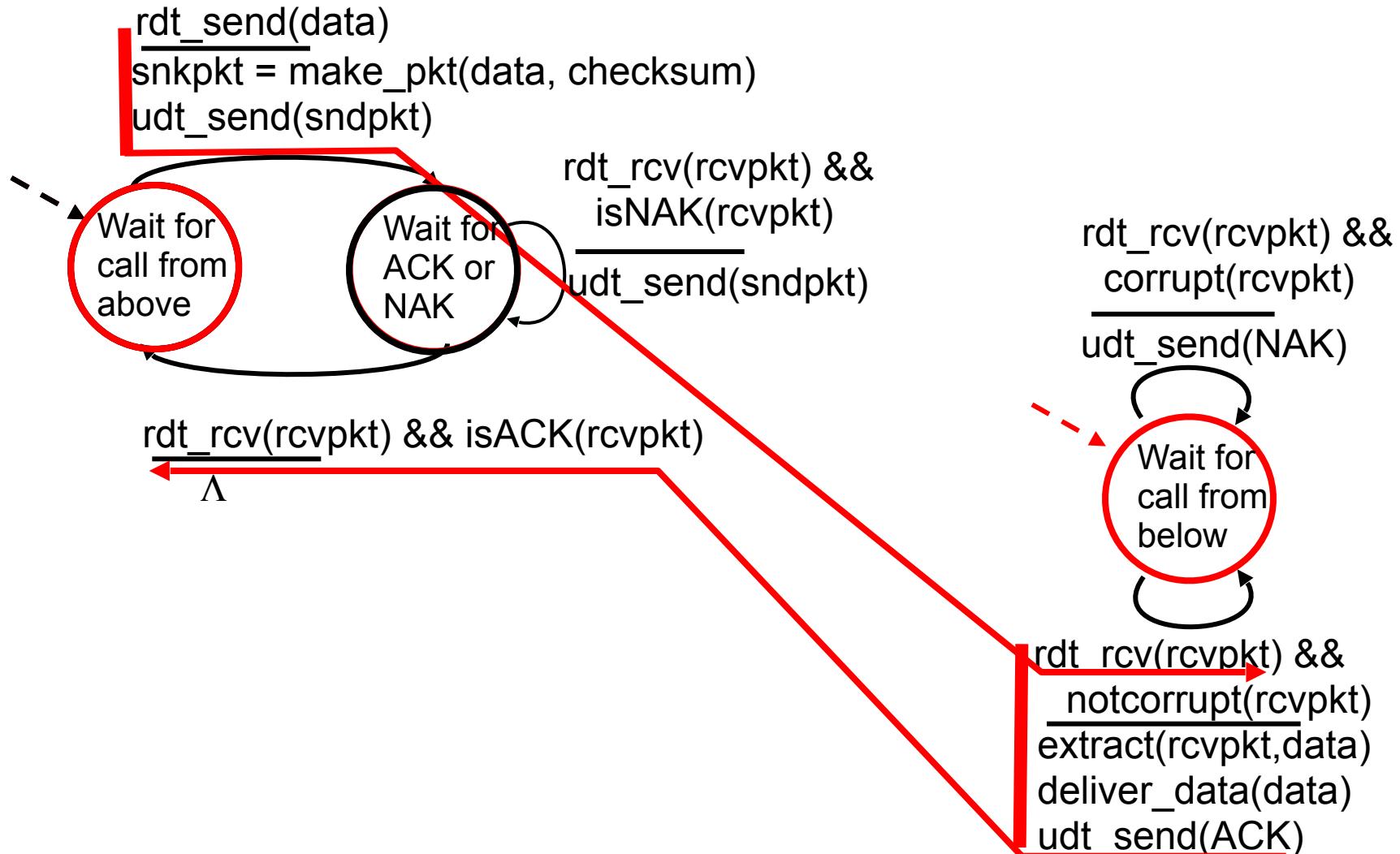
rdt_rcv(rcvpkt) &&
corrupt(rcvpkt)

udt_send(NAK)

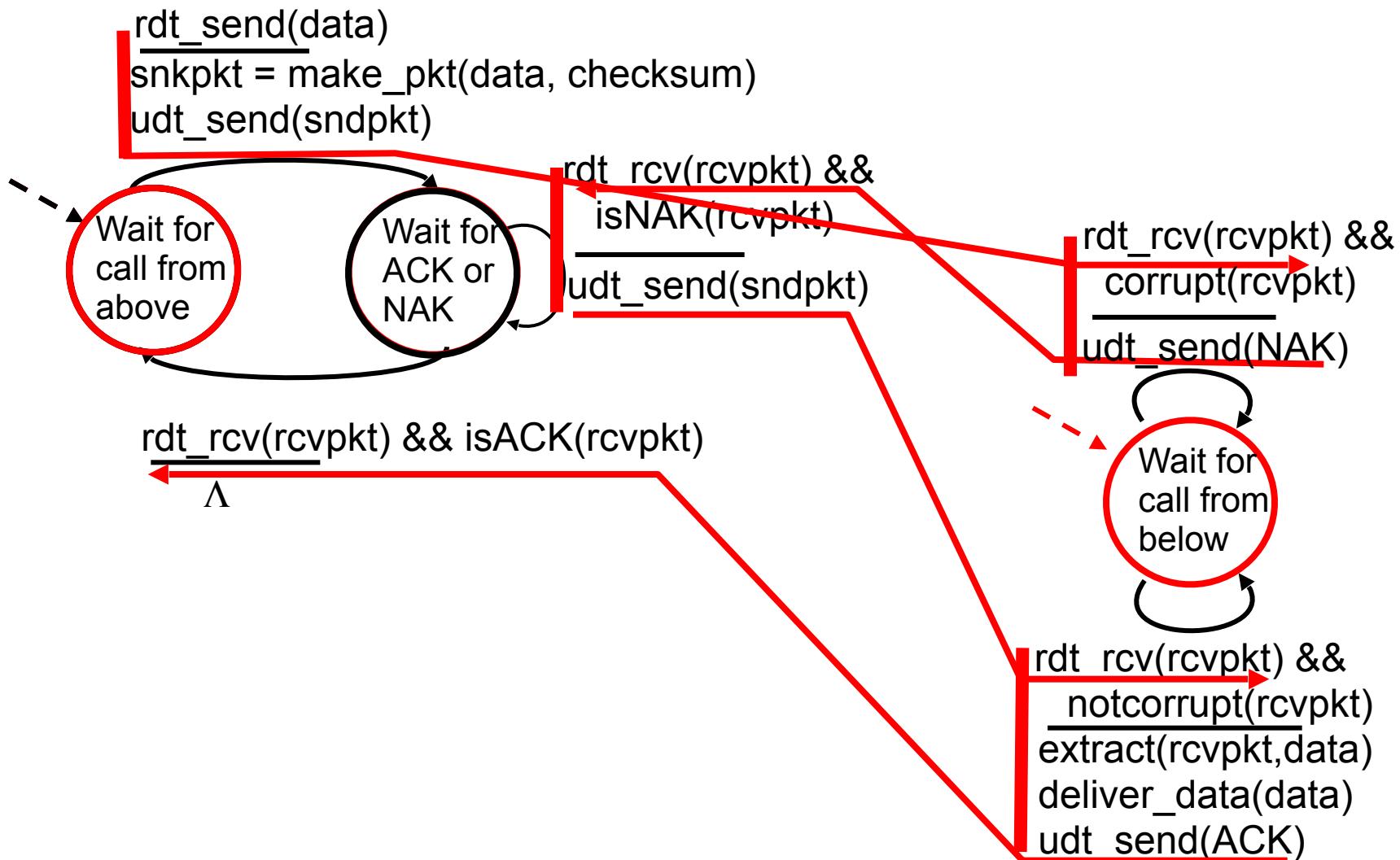


rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

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

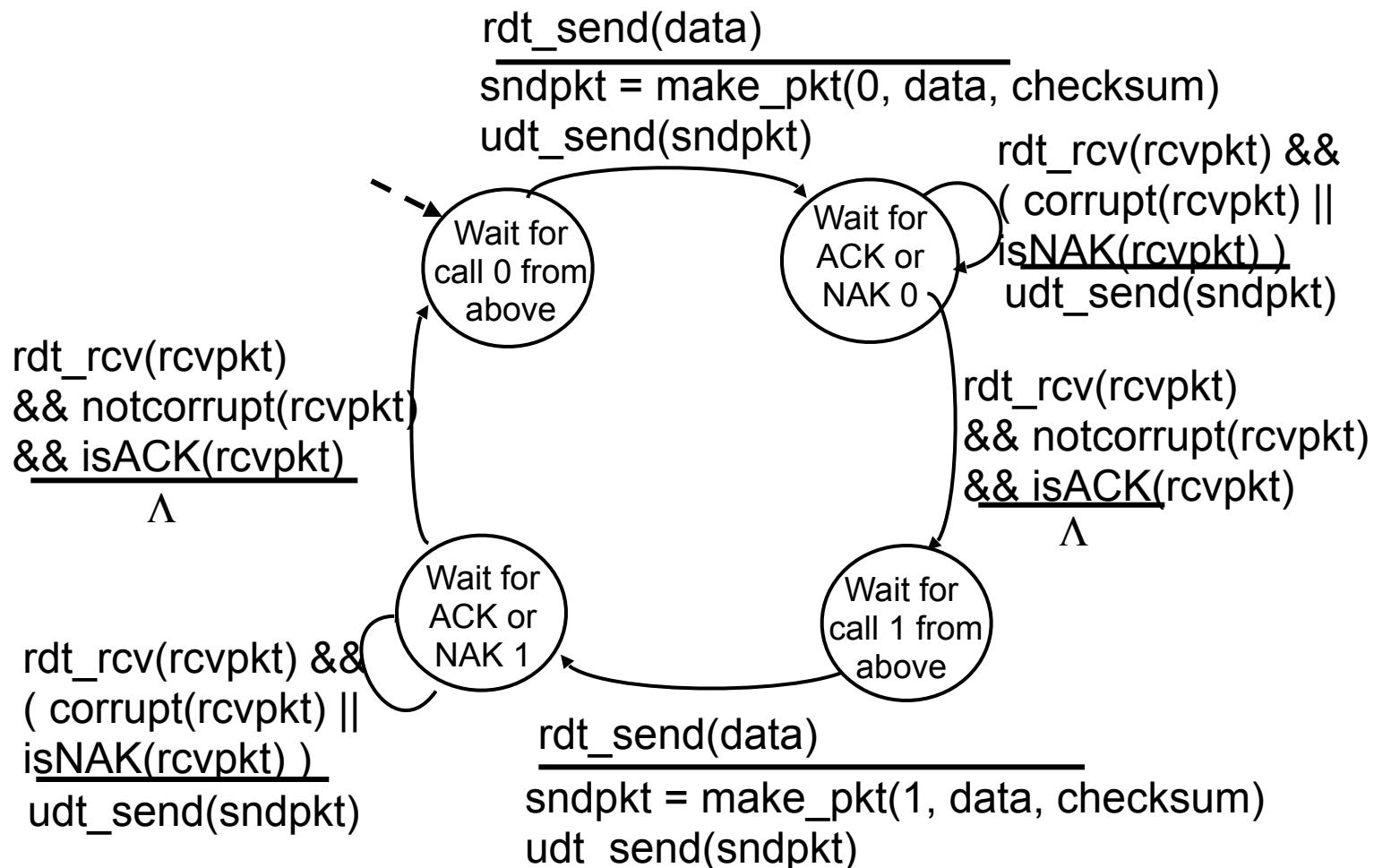
handling duplicates:

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

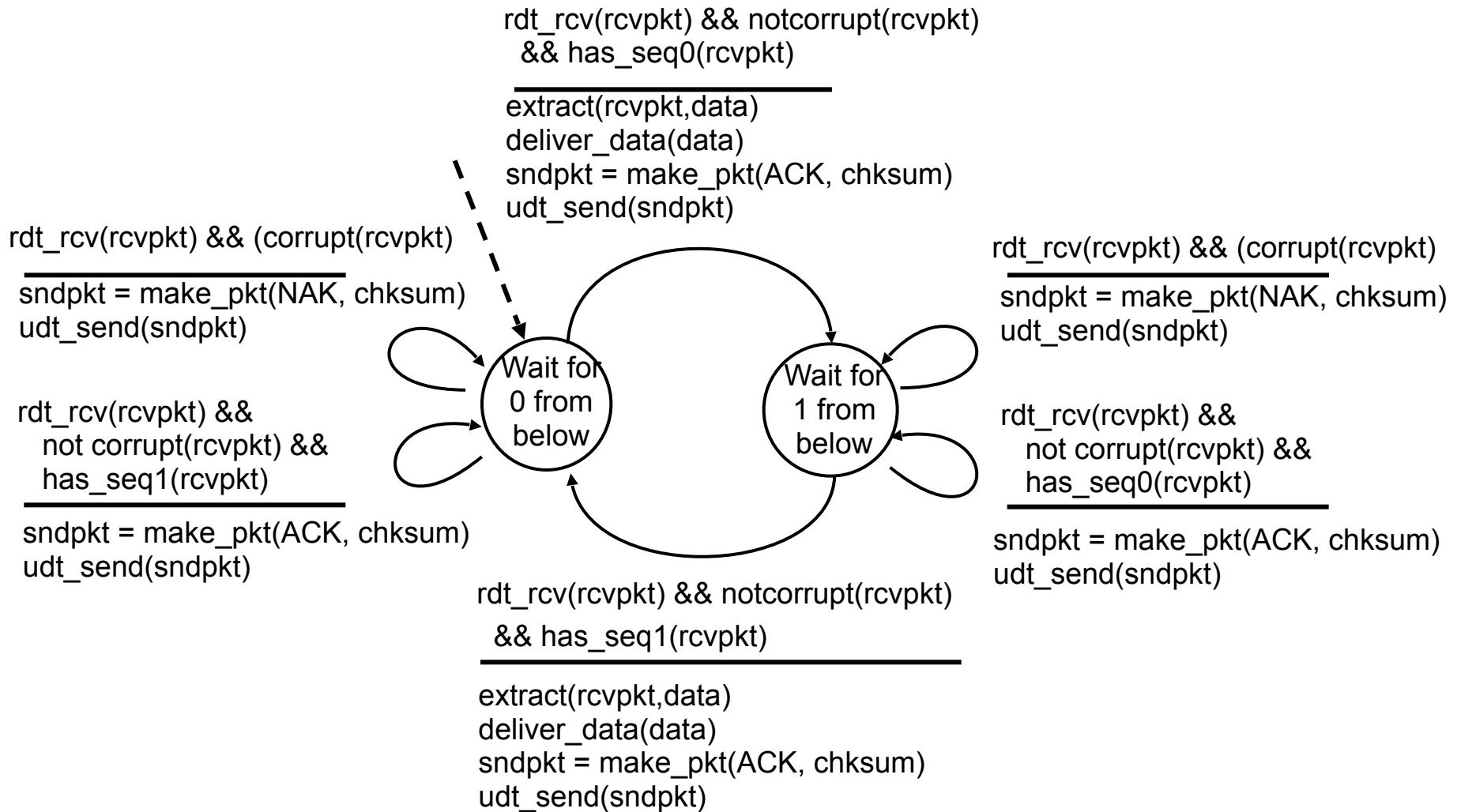
stop and wait

sender sends one packet,
then waits for receiver
response

rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs



Day 12: Principles of Reliable Data Transfer cont.



CSEE 4119
Computer Networks
Ethan Katz-Bassett

 COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

Slides adapted from (and often identical to) slides from Kurose and Ross.

All material copyright 1996-2020

J.F.Kurose and K.W.Ross, All Rights Reserved

recap: 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

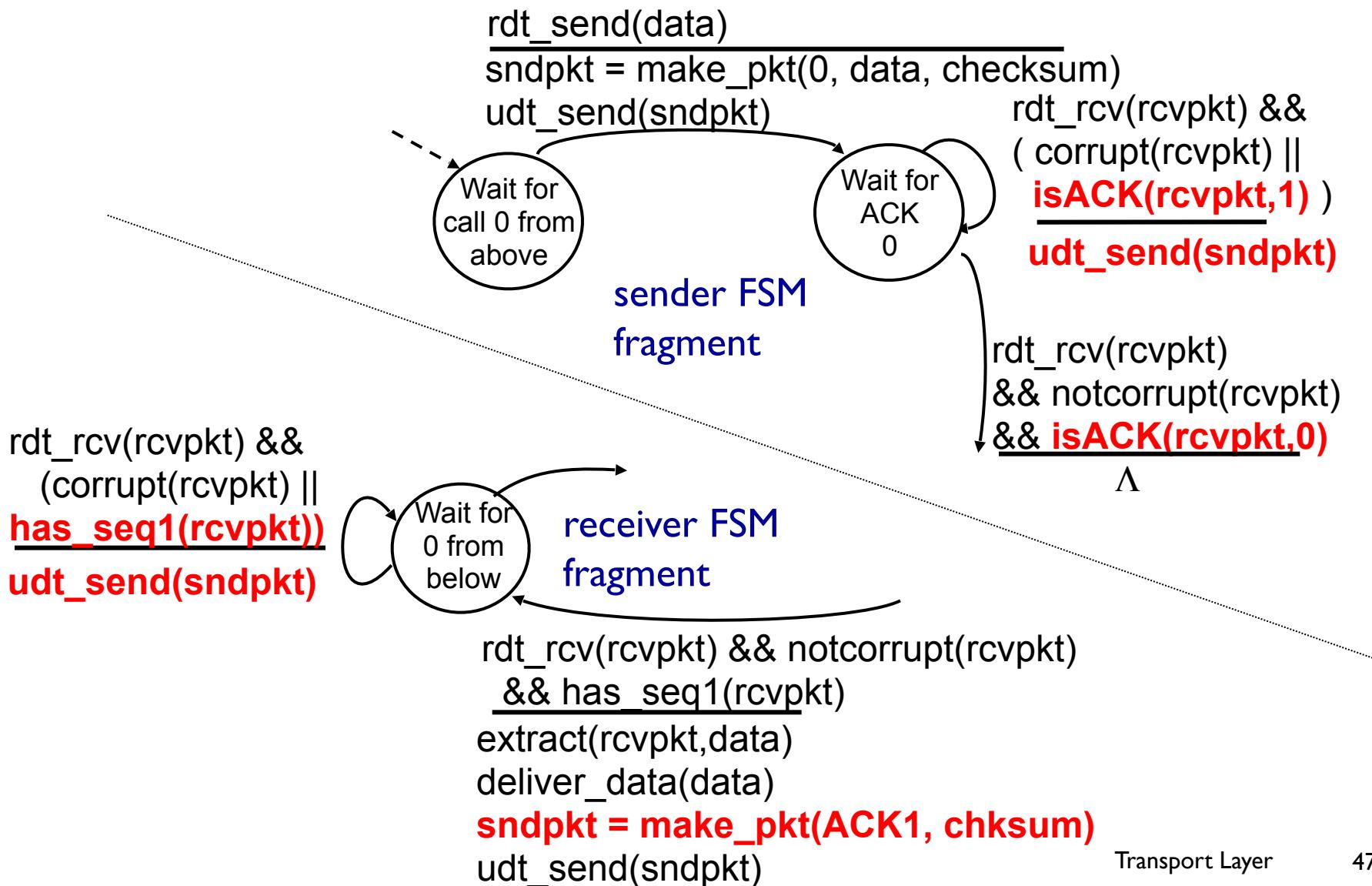
recap: channel with bit errors

- 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
- new mechanisms in `rdt2.1` (beyond `rdt2.0`):
 - deal with garbled ACK/NAK by retransmission
 - sequence numbers to detect duplicates

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)

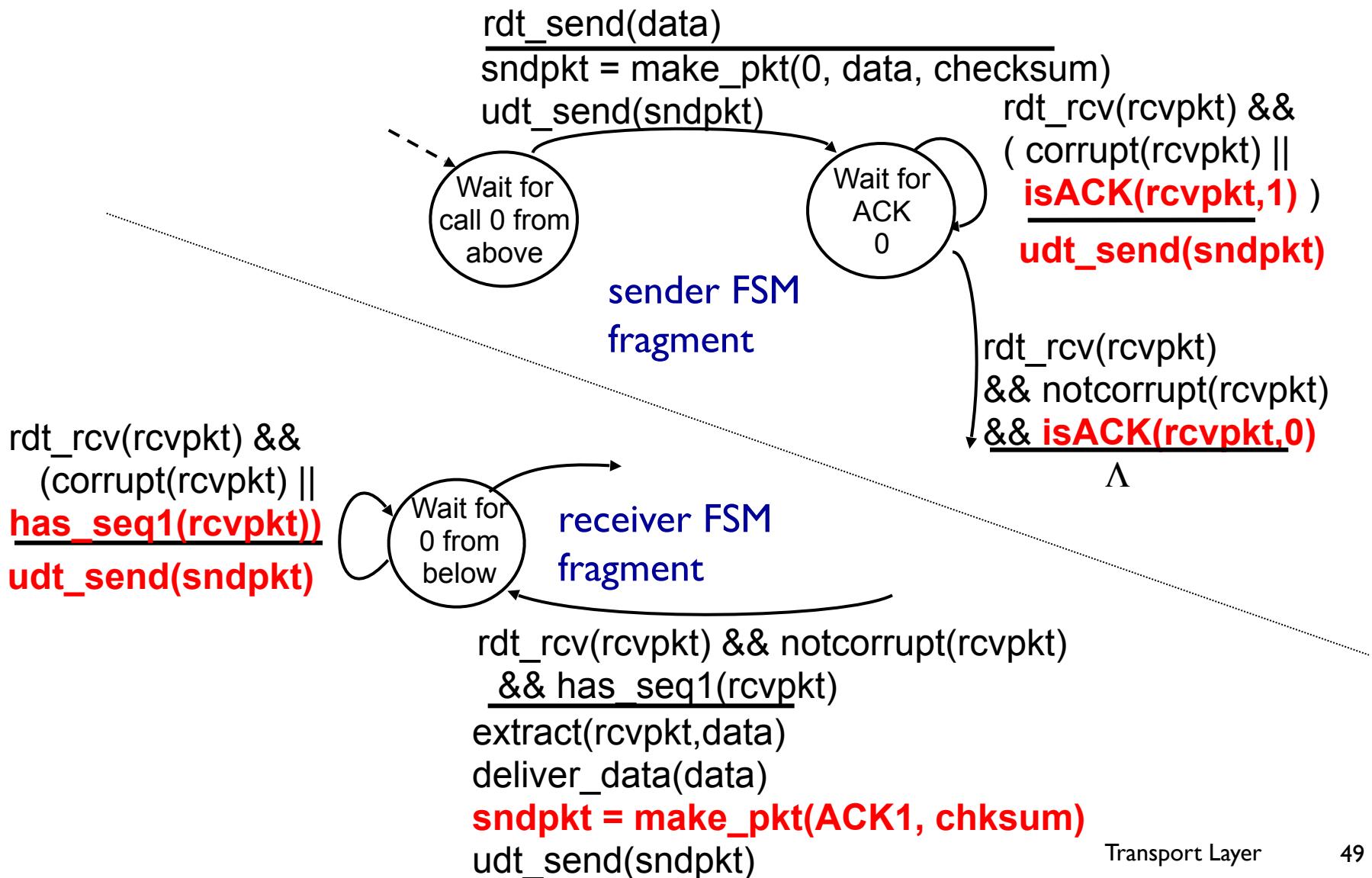
- checksum, seq. #, ACKs, retransmissions will be of help ... but not enough

approach: sender waits

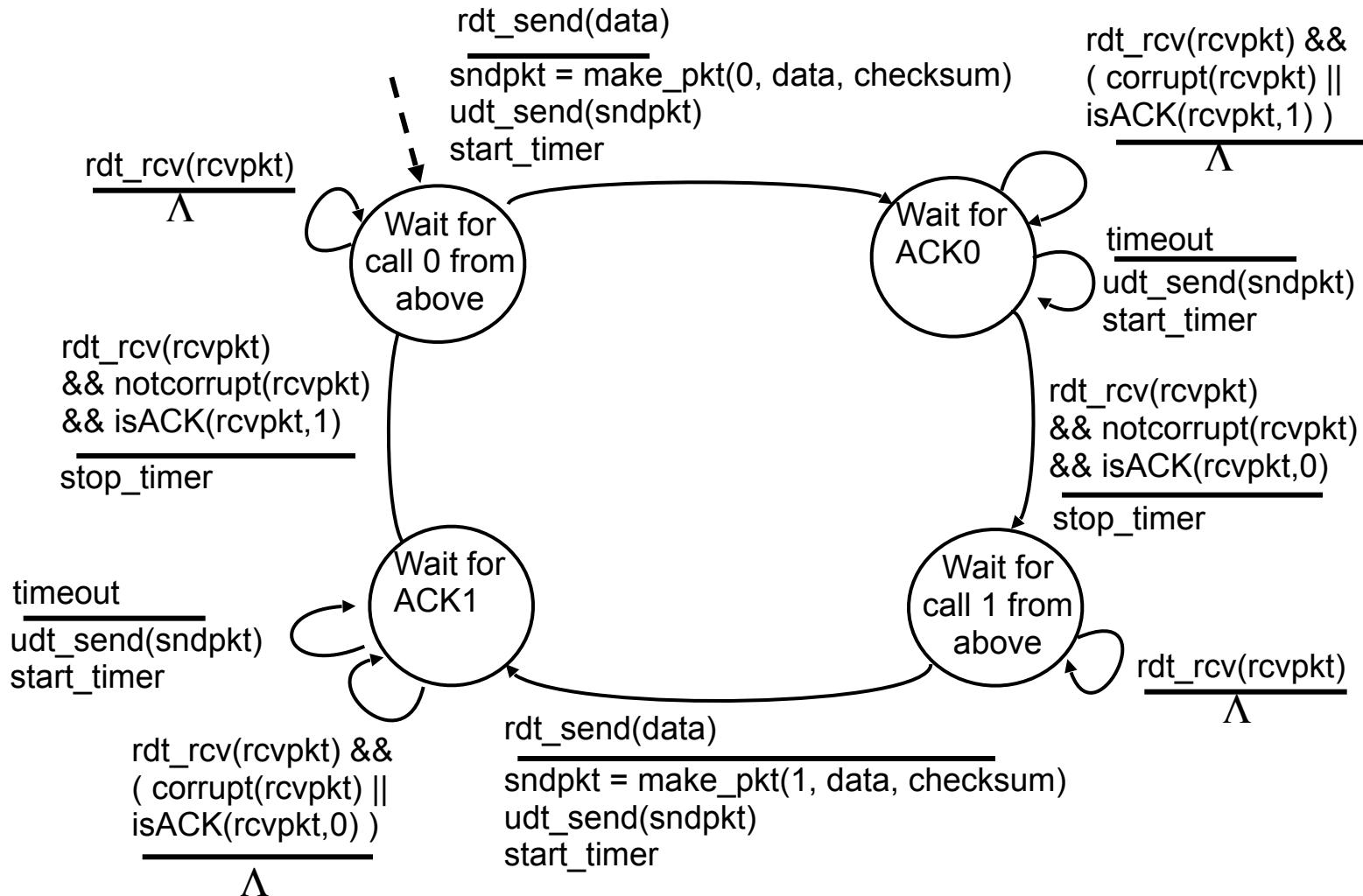
“reasonable” amount of time for ACK

- 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

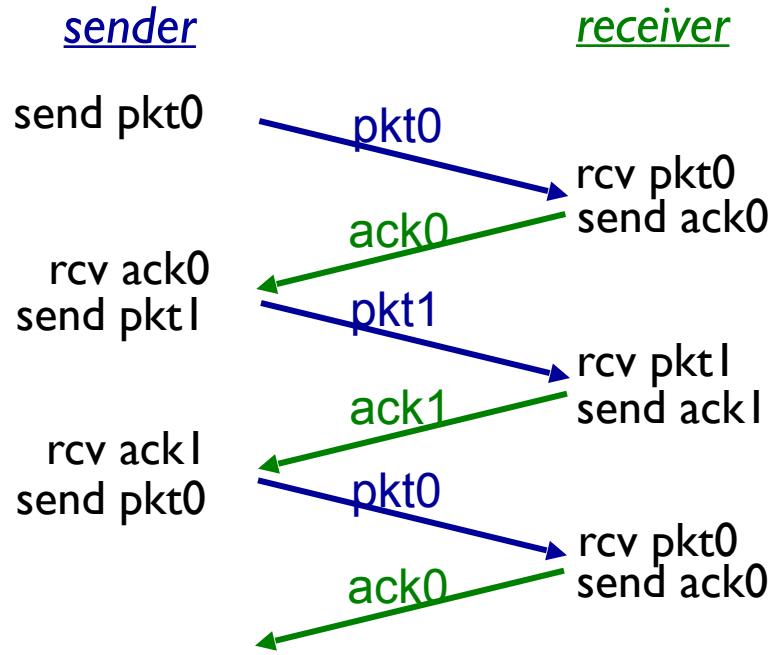
recap: rdt2.2: sender, receiver fragments



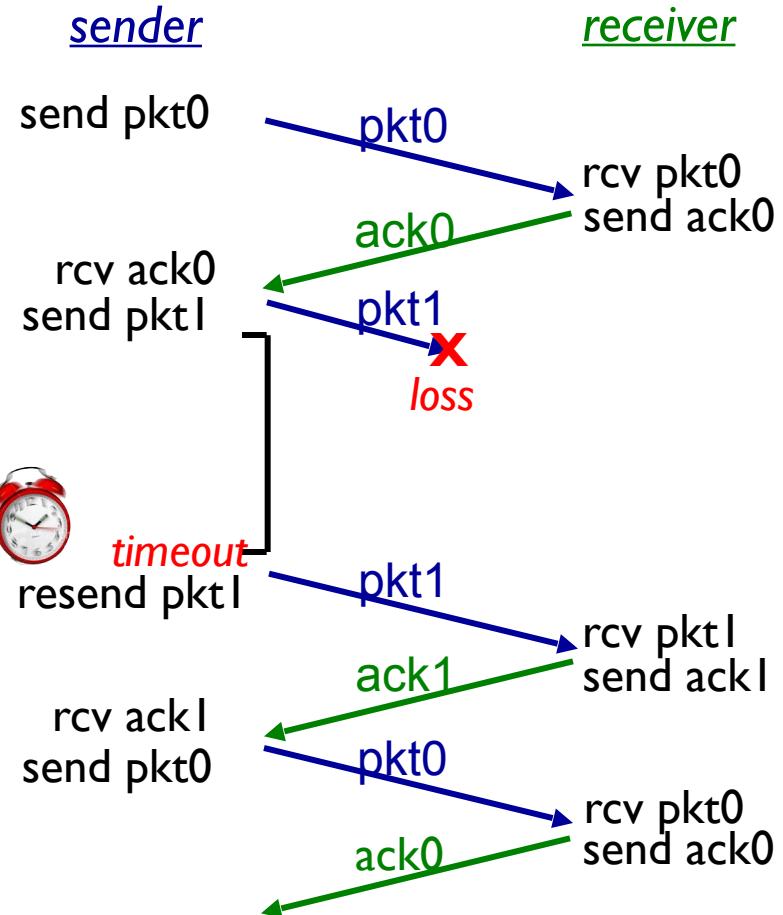
rdt3.0 sender



rdt3.0 in action

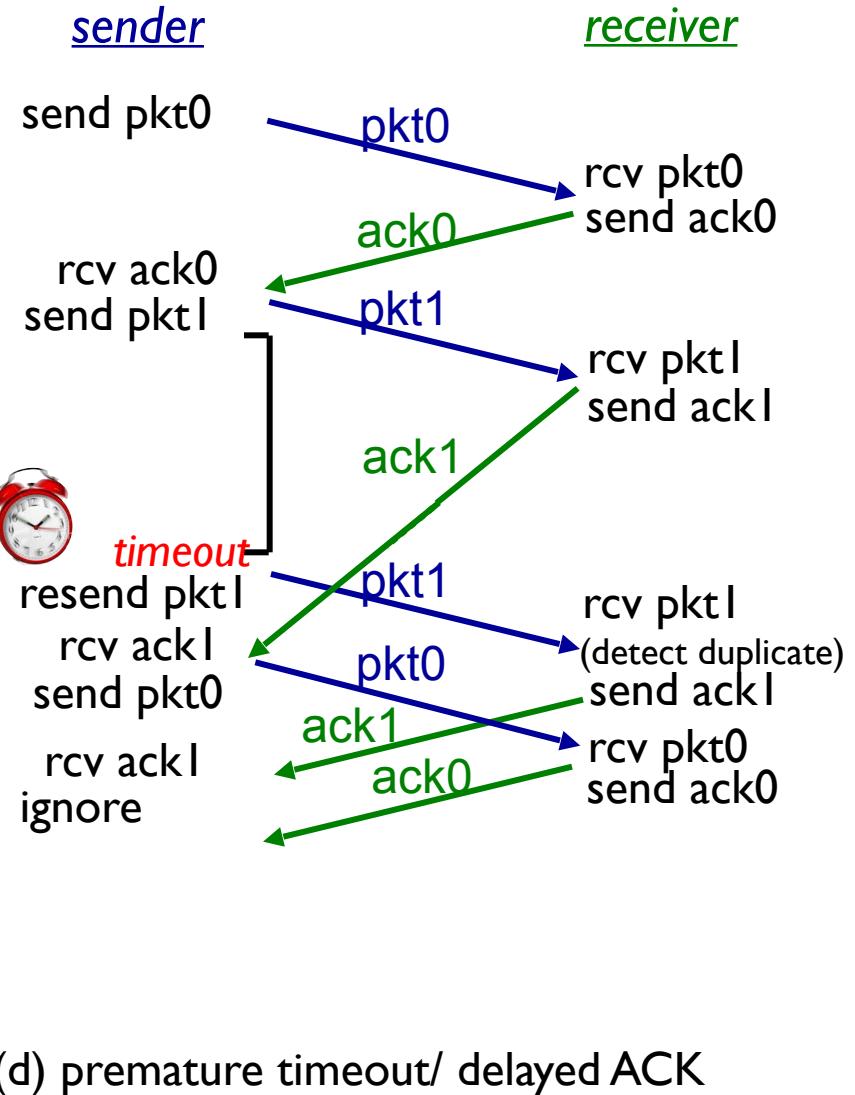
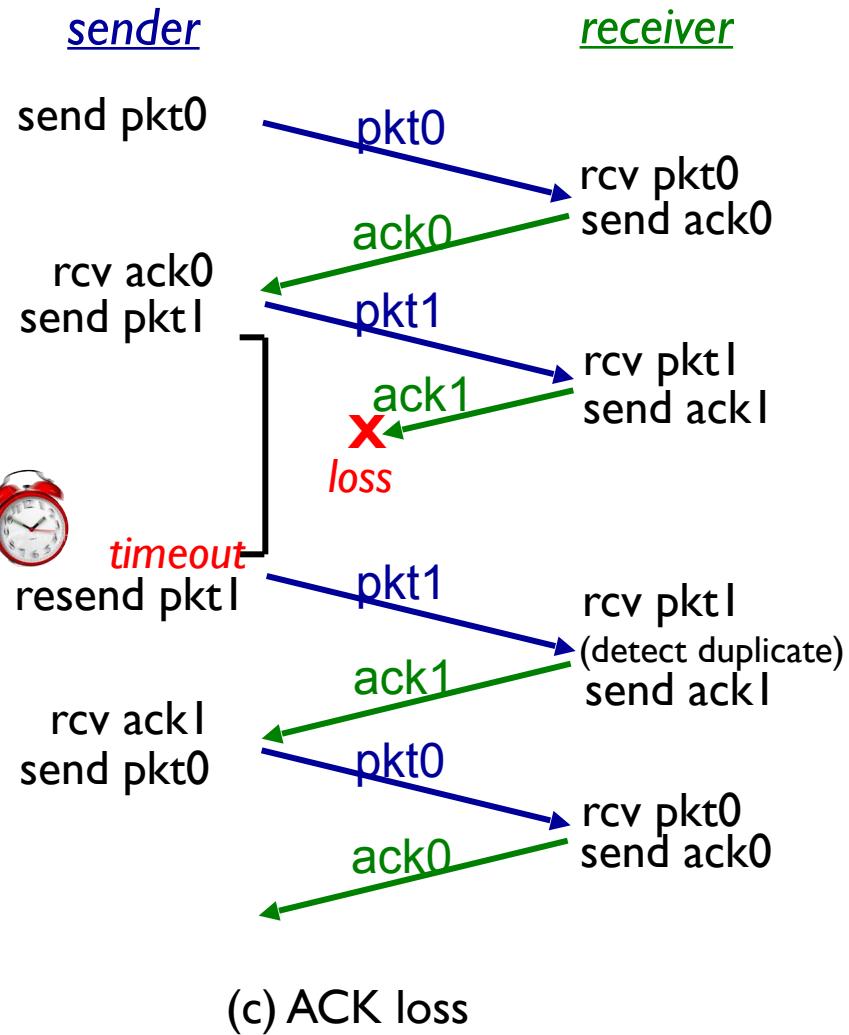


(a) no loss

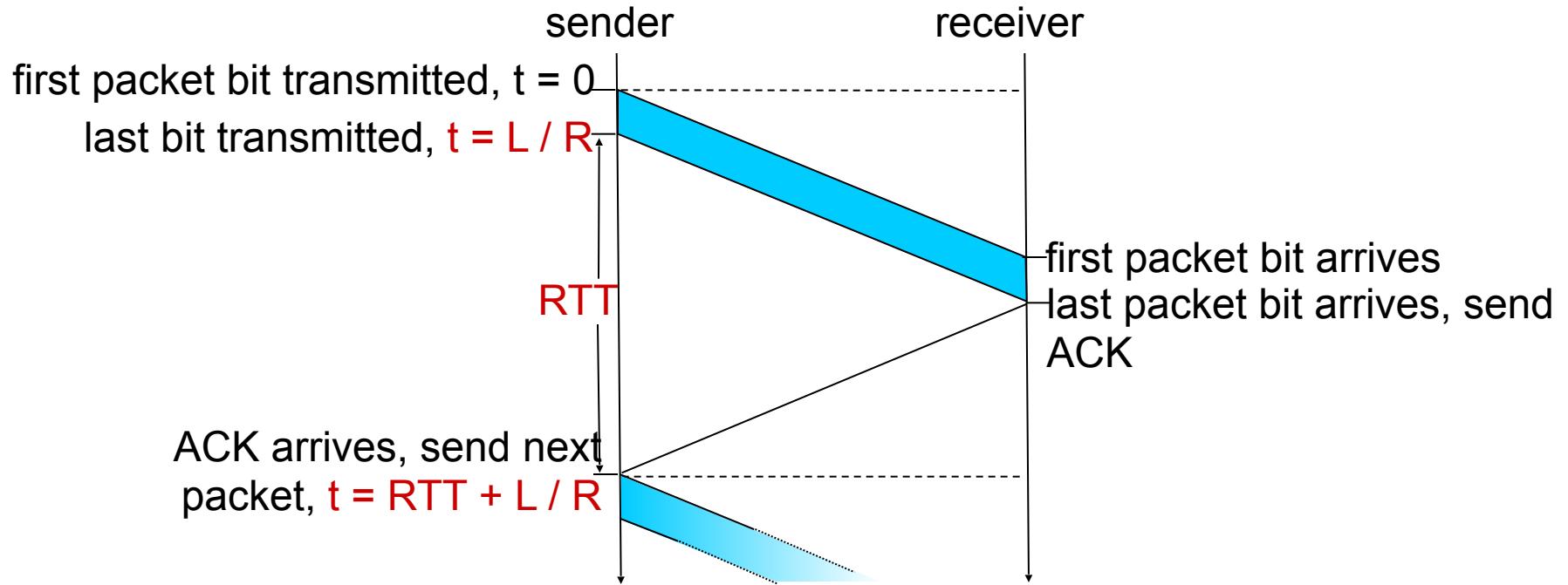


(b) packet loss

rdt3.0 in action



rdt3.0: stop-and-wait operation



L = packet size

R = transmission rate

RTT = round trip time

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

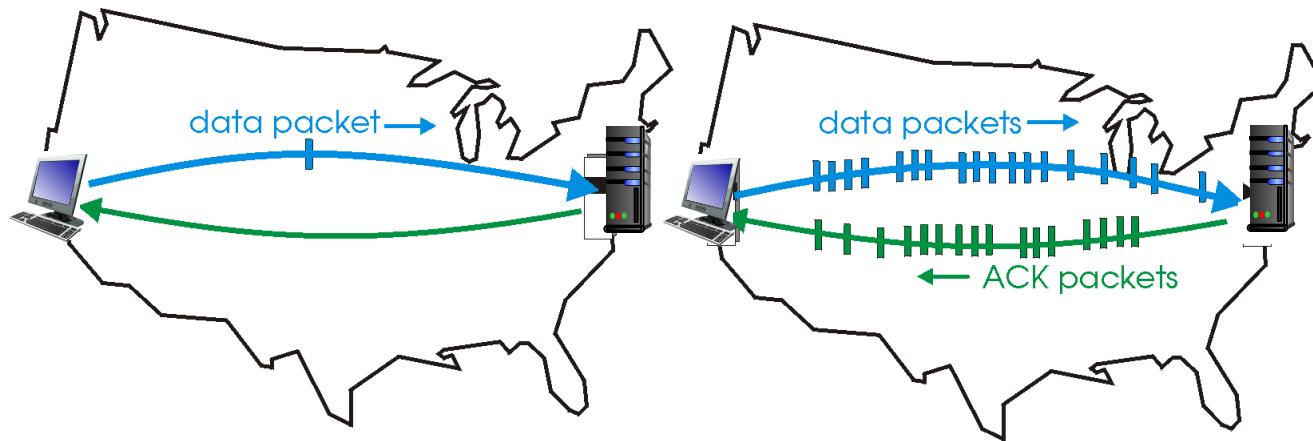
$$U_{\text{sender}} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

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

Pipelined protocols

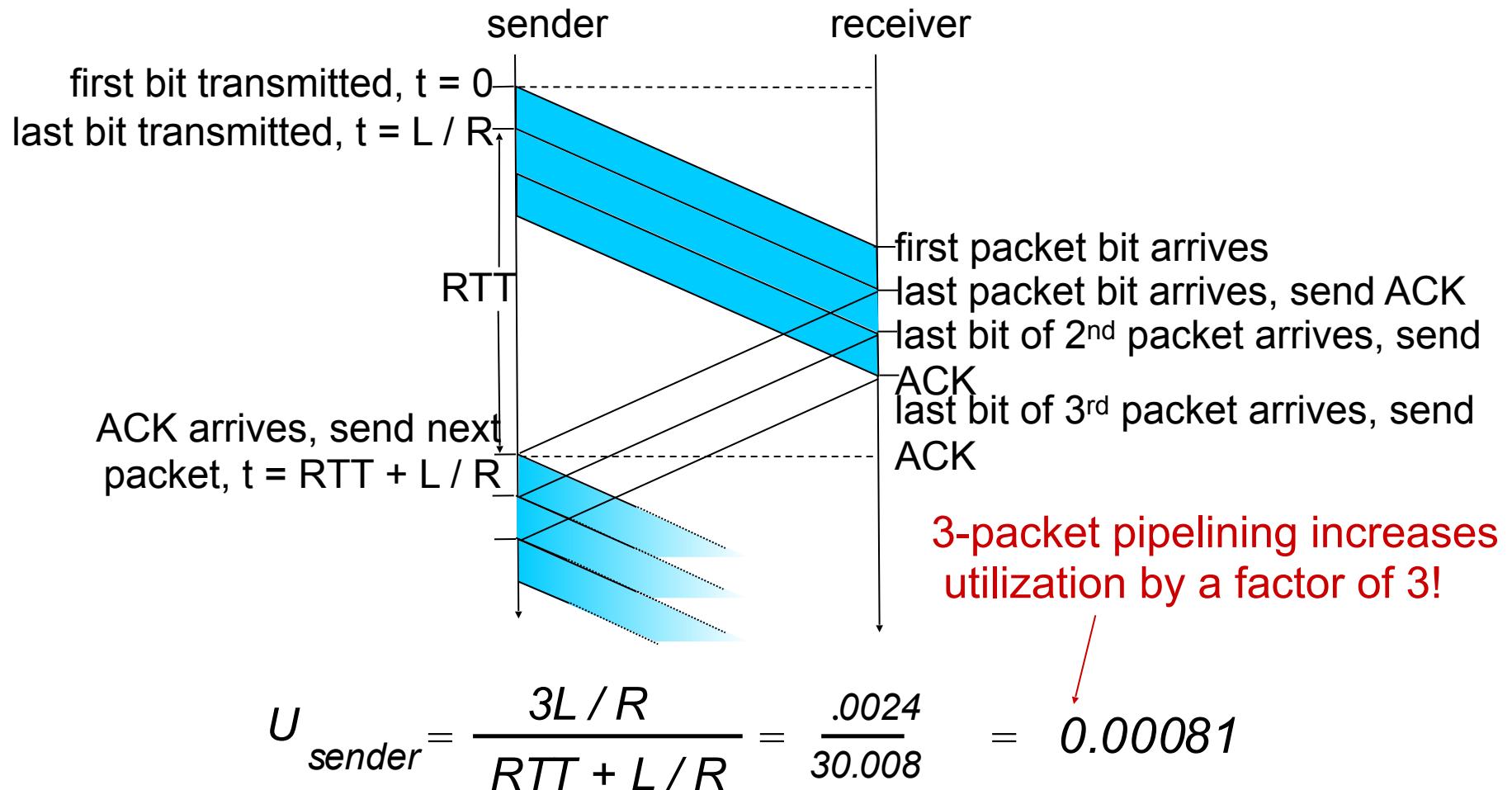
pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

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



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

Pipelining: increased utilization



Pipelined protocols: overview

Go-back-N:

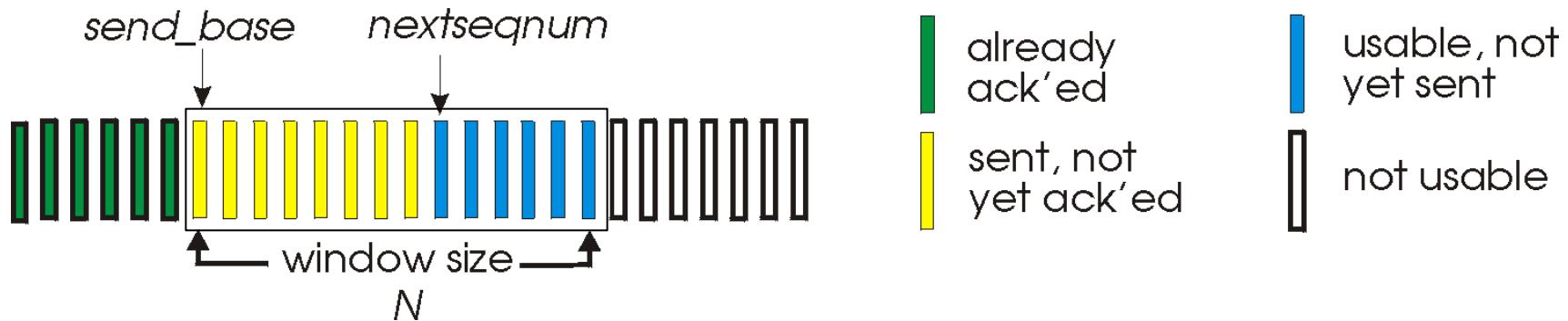
- sender can have up to N unacked packets in pipeline
- receiver only sends **cumulative ack**
 - doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
 - when timer expires, retransmit *all* unacked packets

Selective Repeat:

- sender can have up to N unack'd 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

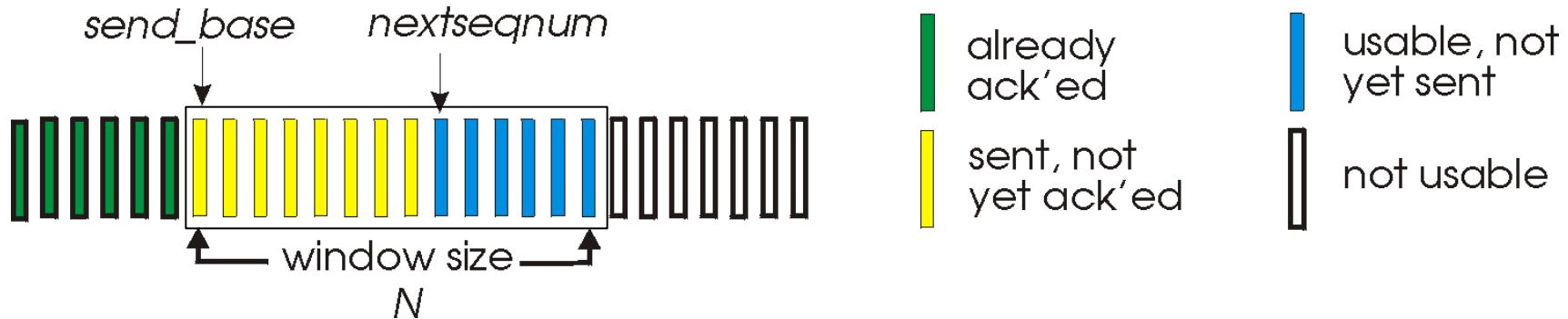
Go-Back-N: sender

- k-bit sequence # in packet header
- “window” of up to N consecutive unack’ed pkts allowed



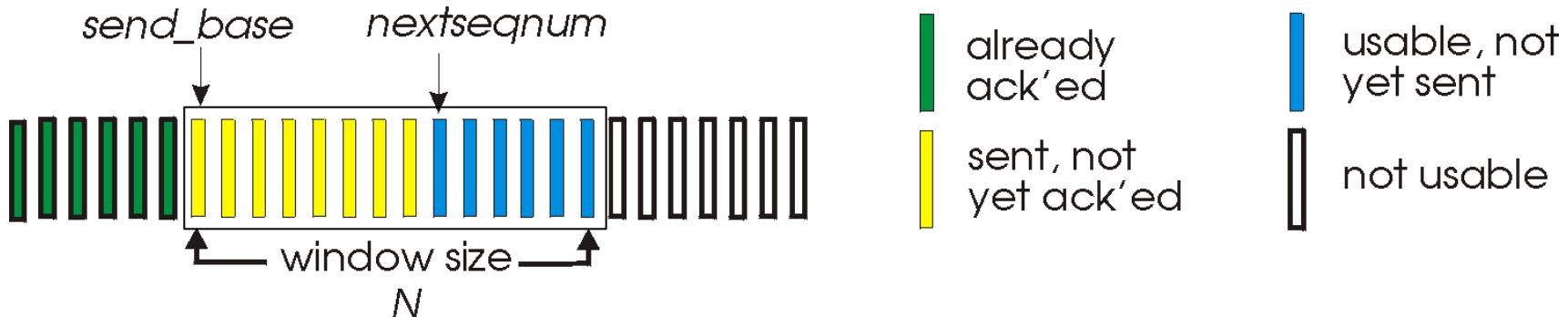
Go-Back-N: sender

- k-bit sequence # in packet header
- “window” of up to N consecutive unack’ed pkts allowed



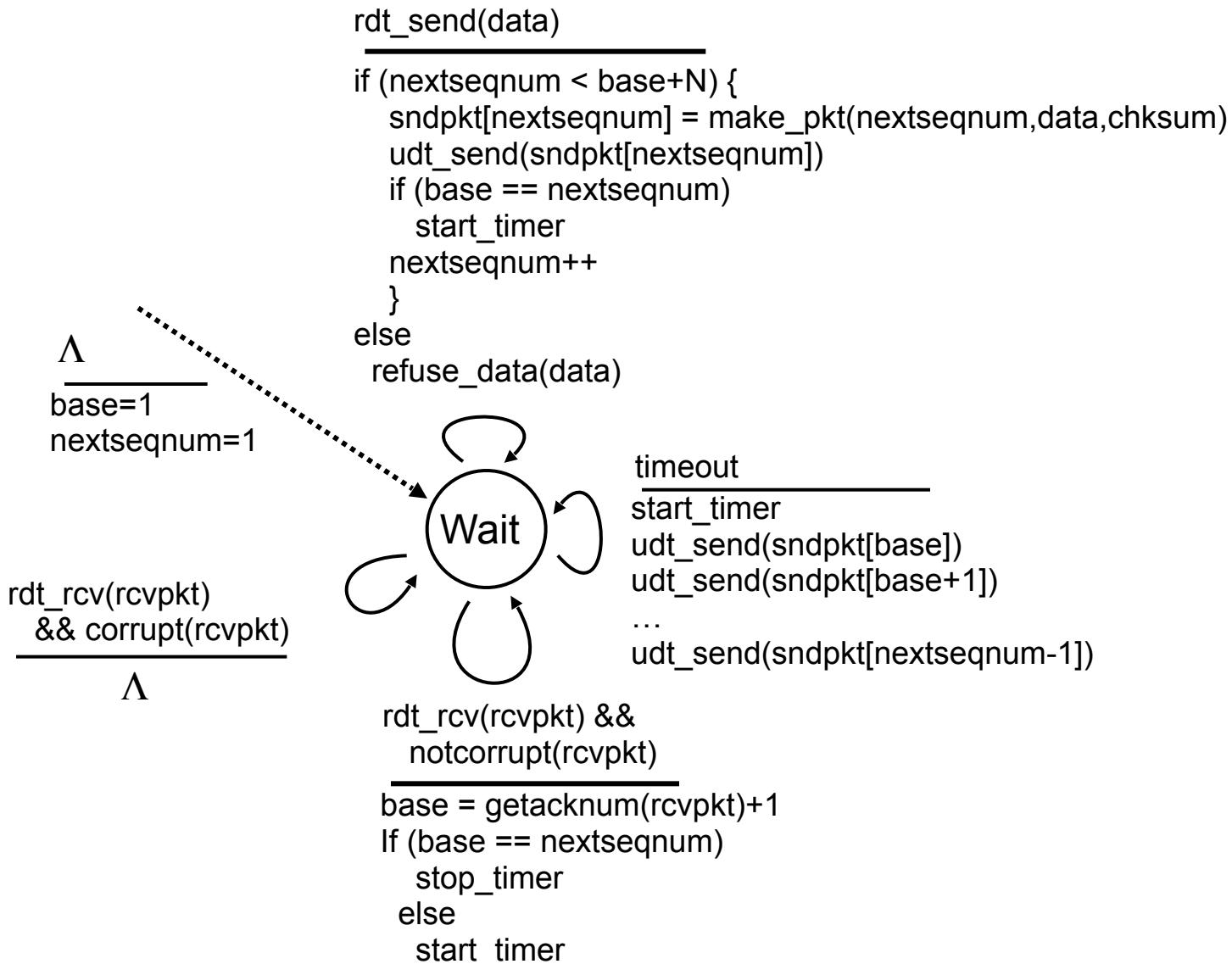
Go-Back-N: sender

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

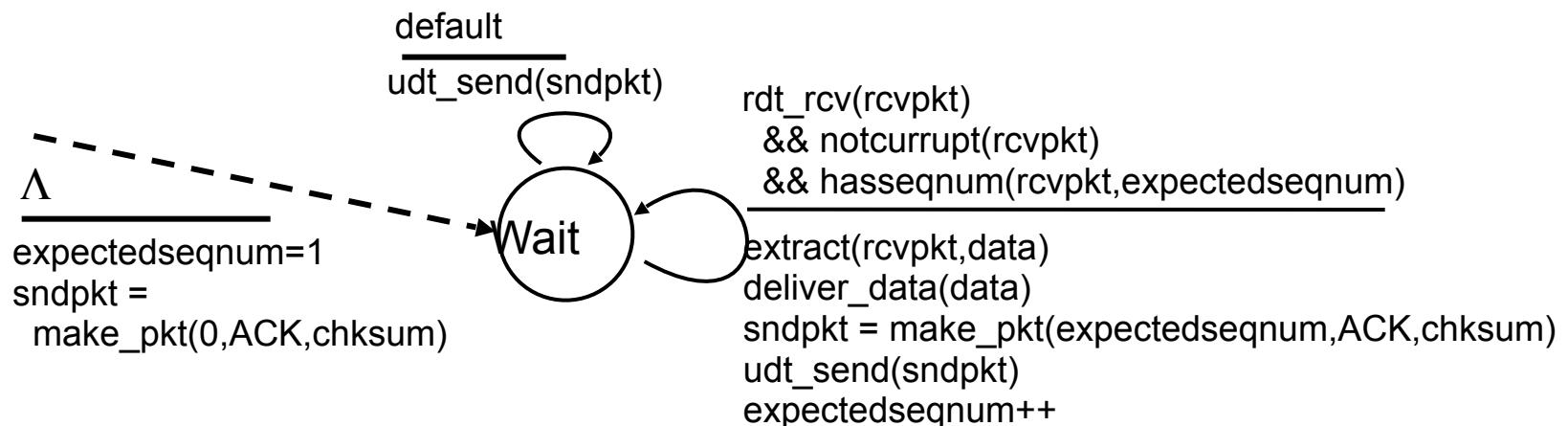


- ACK(n):ACKs all pkts up to, including seq n : “*cumulative ACK*”
 - may receive duplicate ACKs (see receiver)
- Timer for oldest in-flight packet (first yellow one). (Re)starts when:
 - new packet is sent when none are outstanding
 - ACK is received (including duplicate ACKs)
- *timeout(n)*: retransmit 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 packet 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 #

Day 13: Pipelined Transport and TCP



CSEE 4119
Computer Networks
Ethan Katz-Bassett

 COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

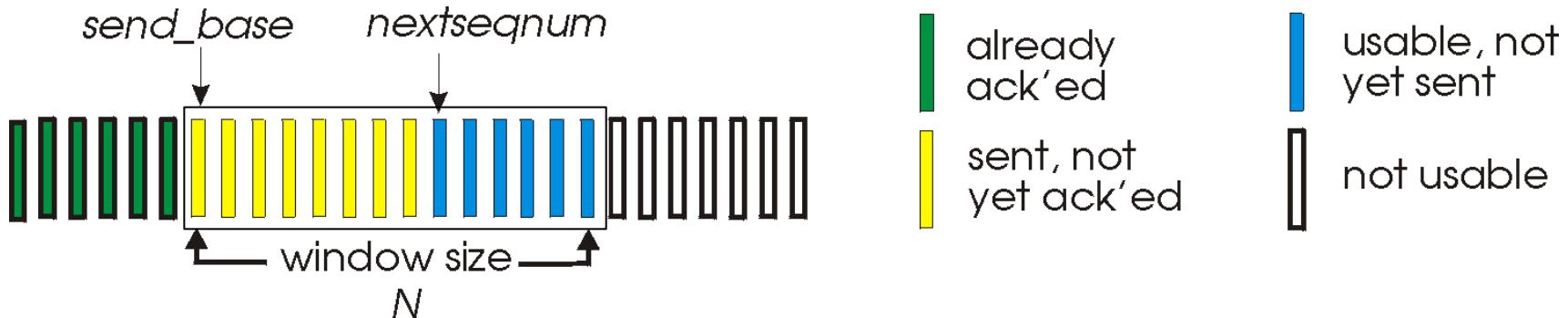
Slides adapted from (and often identical to) slides from Kurose and Ross.

All material copyright 1996-2020

J.F.Kurose and K.W.Ross, All Rights Reserved

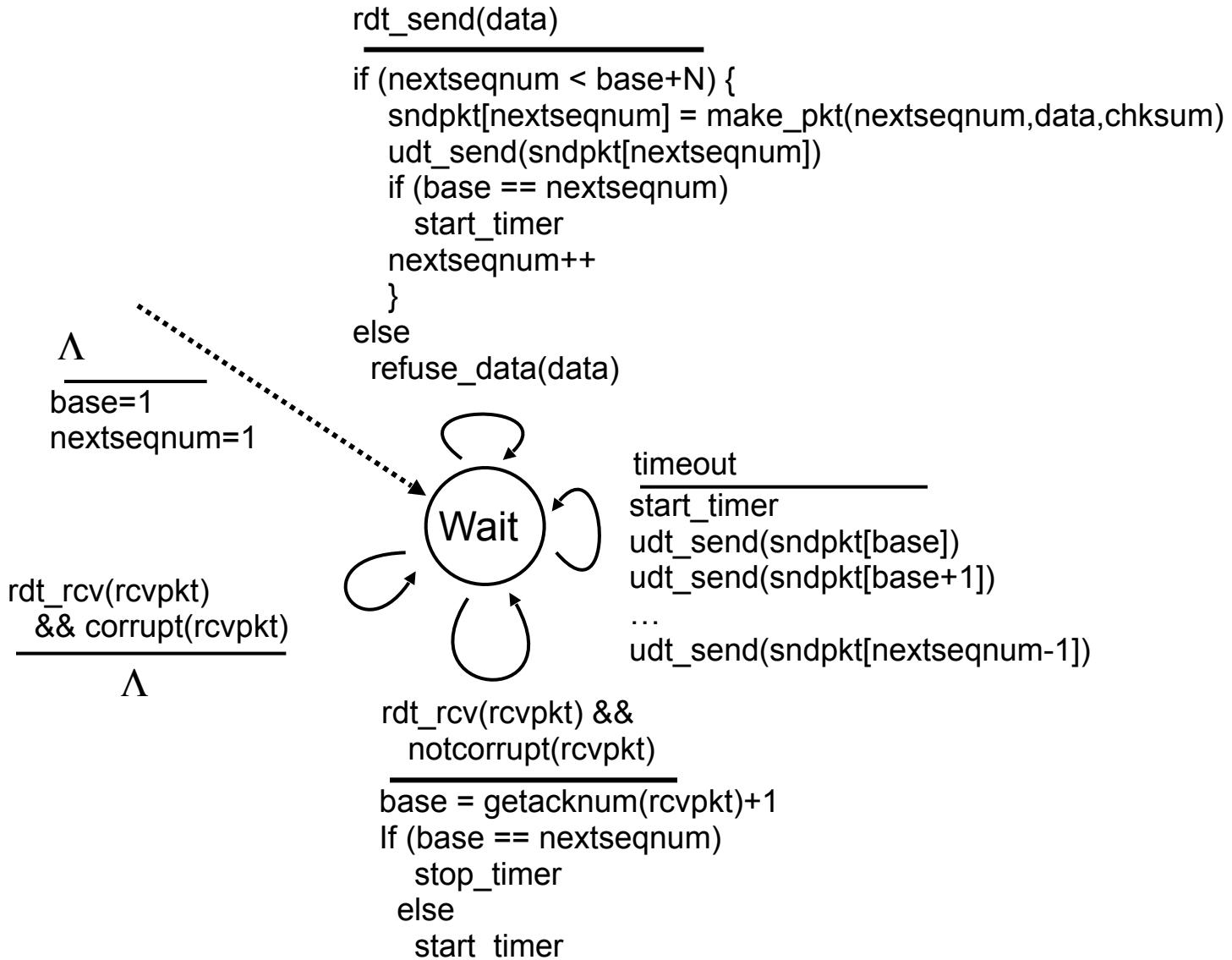
Recap: Go-Back-N: sender

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

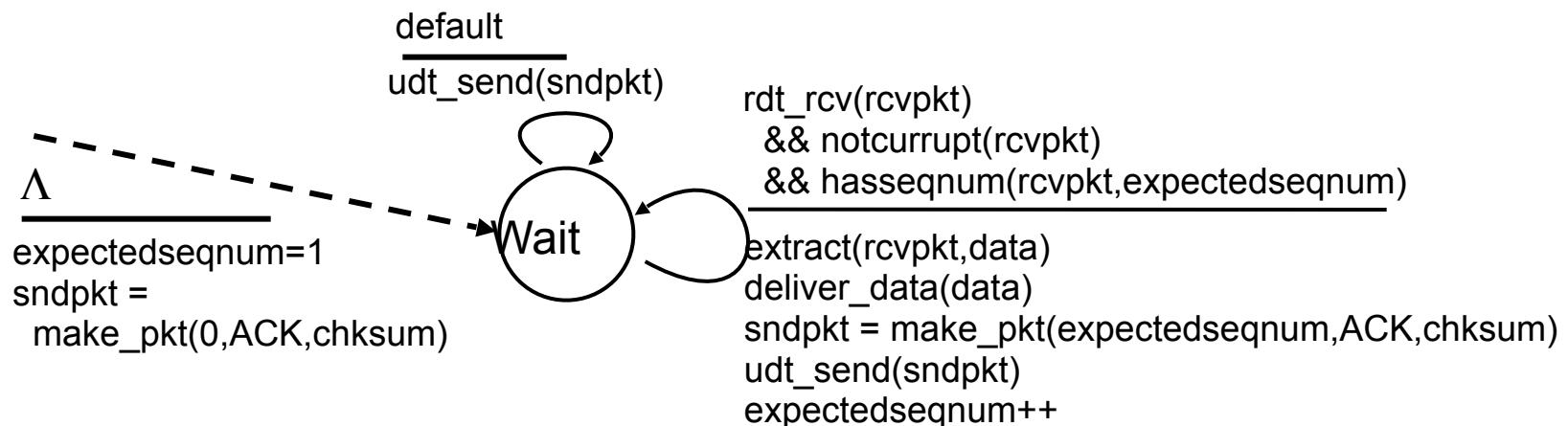


- ACK(n):ACKs all pkts up to, including seq n : “*cumulative ACK*”
 - may receive duplicate ACKs (see receiver)
- Timer for oldest in-flight packet (first yellow one). (Re)starts when:
 - new packet is sent when none are outstanding
 - ACK is received (including duplicate ACKs)
- *timeout(n)*: retransmit packet n and all higher seq # pkts in window

Recap: GBN: sender extended FSM



Recap: GBN: receiver extended FSM



ACK-only: always send ACK for correctly-received packet 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

sender window (N=4)

0 1 2 3 4 5 6 7 8

sender

receiver

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

sender

send pkt0

receiver

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

sender

send pkt0



receiver



GBN in action

sender window (N=4)

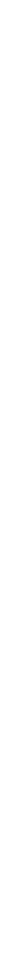
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

sender

send pkt0
send pkt1



receiver



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

sender

send pkt0
send pkt1
send pkt2



receiver



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

sender

send pkt0
send pkt1
send pkt2
send pkt3



receiver

|

|

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

sender

send pkt0
send pkt1
send pkt2
send pkt3
(wait)



receiver



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

sender

send pkt0
send pkt1
send pkt2
send pkt3
(wait)



receiver

Xloss

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

sender

send pkt0
send pkt1
send pkt2
send pkt3
(wait)



receiver

receive pkt0, send ack0
receive pkt1, send ack1

Xloss

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

sender

send pkt0
send pkt1
send pkt2
send pkt3
(wait)



receiver

receive pkt0, send ack0
receive pkt1, send ack1
receive pkt3, discard,
(re)send ack1

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

sender

send pkt0
send pkt1
send pkt2
send pkt3
(wait)

receiver

receive pkt0, send ack0
receive pkt1, send ack1
receive pkt3, discard,
(re)send ack1

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

rcv ack0, send pkt4
rcv ack1, send pkt5

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

sender

send pkt0
send pkt1
send pkt2
send pkt3
(wait)

receiver

receive pkt0, send ack0
receive pkt1, send ack1
receive pkt3, discard,
(re)send ack1

rcv ack0, send pkt4
rcv ack1, send pkt5



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

sender

send pkt0
send pkt1
send pkt2
send pkt3
(wait)

receiver

receive pkt0, send ack0
receive pkt1, send ack1
receive pkt3, discard,
(re)send ack1

rcv ack0, send pkt4
rcv ack1, send pkt5

ignore duplicate ACK



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

sender

send pkt0
send pkt1
send pkt2
send pkt3
(wait)

rcv ack0, send pkt4
rcv ack1, send pkt5

ignore duplicate ACK



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

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

sender

send pkt0
send pkt1
send pkt2
send pkt3
(wait)

rcv ack0, send pkt4
rcv ack1, send pkt5

ignore duplicate ACK



pkt 2 timeout

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

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



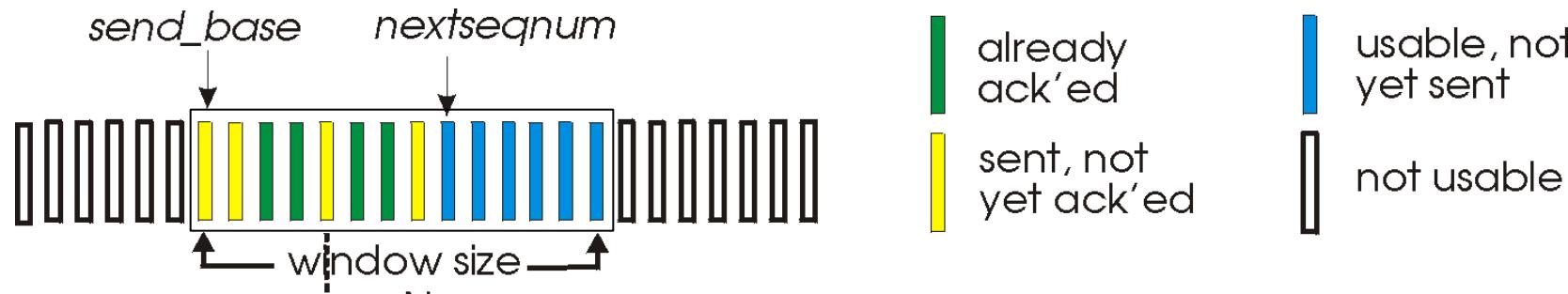
Go-Back-N pros/cons

- What's good about Go-Back-N?
- What's bad about Go-Back-N?

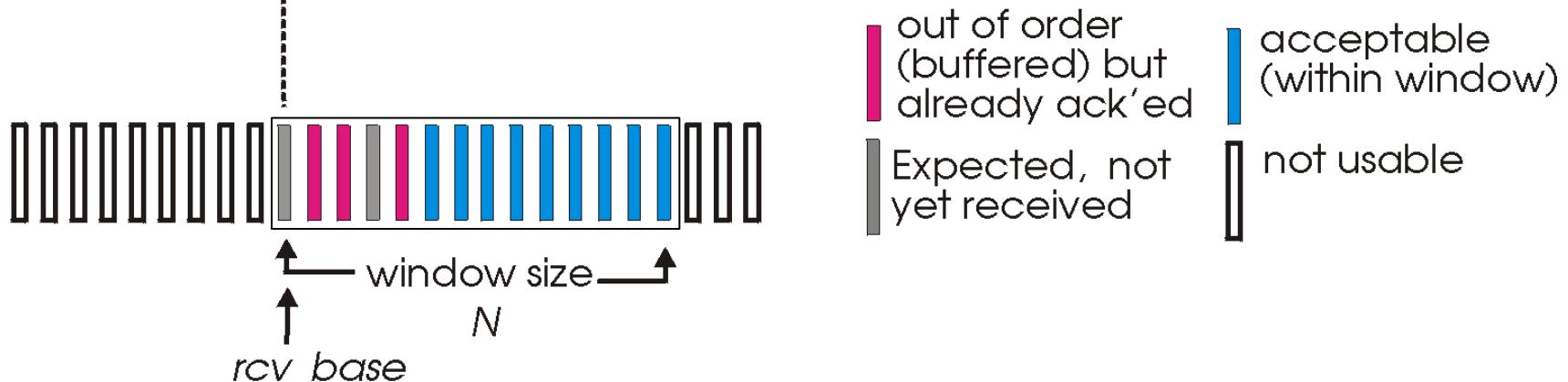
Selective repeat

- receiver *individually* ACKs each packet it receives
 - buffers packets, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
 - separate sender timer for each unACKed packet
- 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 available seq # in window, send packet

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]

- (re)send ACK(n)

otherwise:

- ignore

Selective repeat in action

sender window (N=4)

0 1 2 3 4 5 6 7 8

sender

receiver

Selective repeat 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

sender

send pkt0
send pkt1
send pkt2
send pkt3
(wait)

receiver

Selective repeat in action

sender window (N=4)

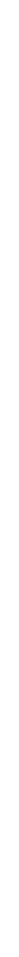
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

sender

send pkt0
send pkt1
send pkt2
send pkt3
(wait)



receiver



Selective repeat 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

sender

send pkt0
send pkt1
send pkt2
send pkt3
(wait)

receiver



Xloss

Selective repeat 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

sender

send pkt0
send pkt1
send pkt2
send pkt3
(wait)

receiver

receive pkt0, send ack0
receive pkt1, send ack1



Selective repeat 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

sender

send pkt0
send pkt1
send pkt2
send pkt3

(wait)

receiver

receive pkt0, send ack0
receive pkt1, send ack1



Selective repeat 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

sender

send pkt0
send pkt1
send pkt2
send pkt3

(wait)

receiver

receive pkt0, send ack0
receive pkt1, send ack1
receive pkt3, buffer,
send ack3

Xloss



Selective repeat 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

sender

send pkt0
send pkt1
send pkt2
send pkt3

(wait)

rcv ack0, send pkt4
rcv ack1, send pkt5

receiver

receive pkt0, send ack0
receive pkt1, send ack1
receive pkt3, buffer,
send ack3



Selective repeat 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

sender

send pkt0
send pkt1
send pkt2
send pkt3

(wait)

rcv ack0, send pkt4
rcv ack1, send pkt5

receiver

receive pkt0, send ack0
receive pkt1, send ack1
receive pkt3, buffer,
send ack3



Selective repeat 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

sender

send pkt0
send pkt1
send pkt2
send pkt3
(wait)

receiver

receive pkt0, send ack0
receive pkt1, send ack1
receive pkt3, buffer,
send ack3

rcv ack0, send pkt4
rcv ack1, send pkt5

record ack3 arrived



Selective repeat 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

sender

send pkt0
send pkt1
send pkt2
send pkt3

(wait)

rcv ack0, send pkt4
rcv ack1, send pkt5

record ack3 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

Selective repeat 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

sender

send pkt0
send pkt1
send pkt2
send pkt3

(wait)

rcv ack0, send pkt4
rcv ack1, send pkt5

record ack3 arrived



pkt 2 timeout

receiver

receive pkt0, send ack0
receive pkt1, send ack1

receive pkt3, buffer,
send ack3

receive pkt4, buffer,
send ack4
receive pkt5, buffer,
send ack5

Selective repeat in action

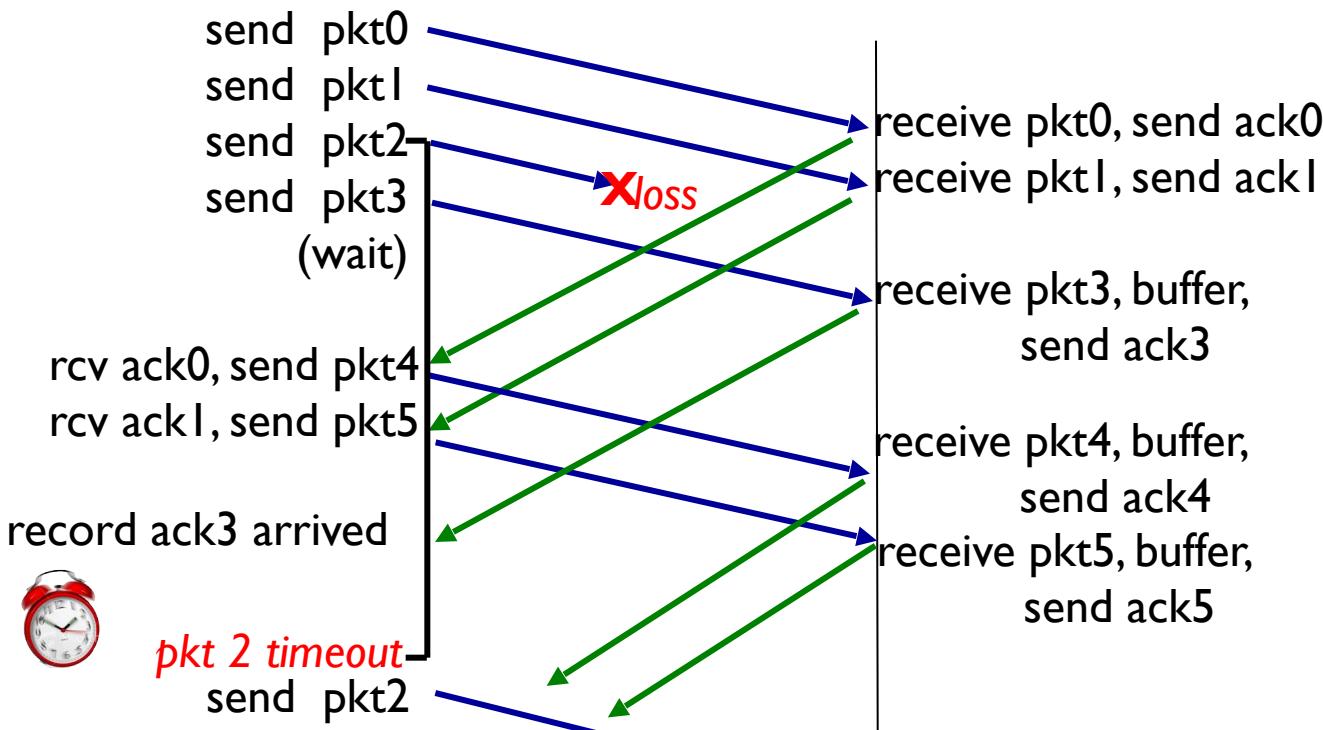
sender window (N=4)

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

sender



receiver

Selective repeat 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

record ack3 arrived



pkt 2 timeout
send pkt2

record ack4 arrived
record ack5 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

Selective repeat 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

pkt 2 timeout
send pkt2

record ack4 arrived
record ack5 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;



Selective repeat in action

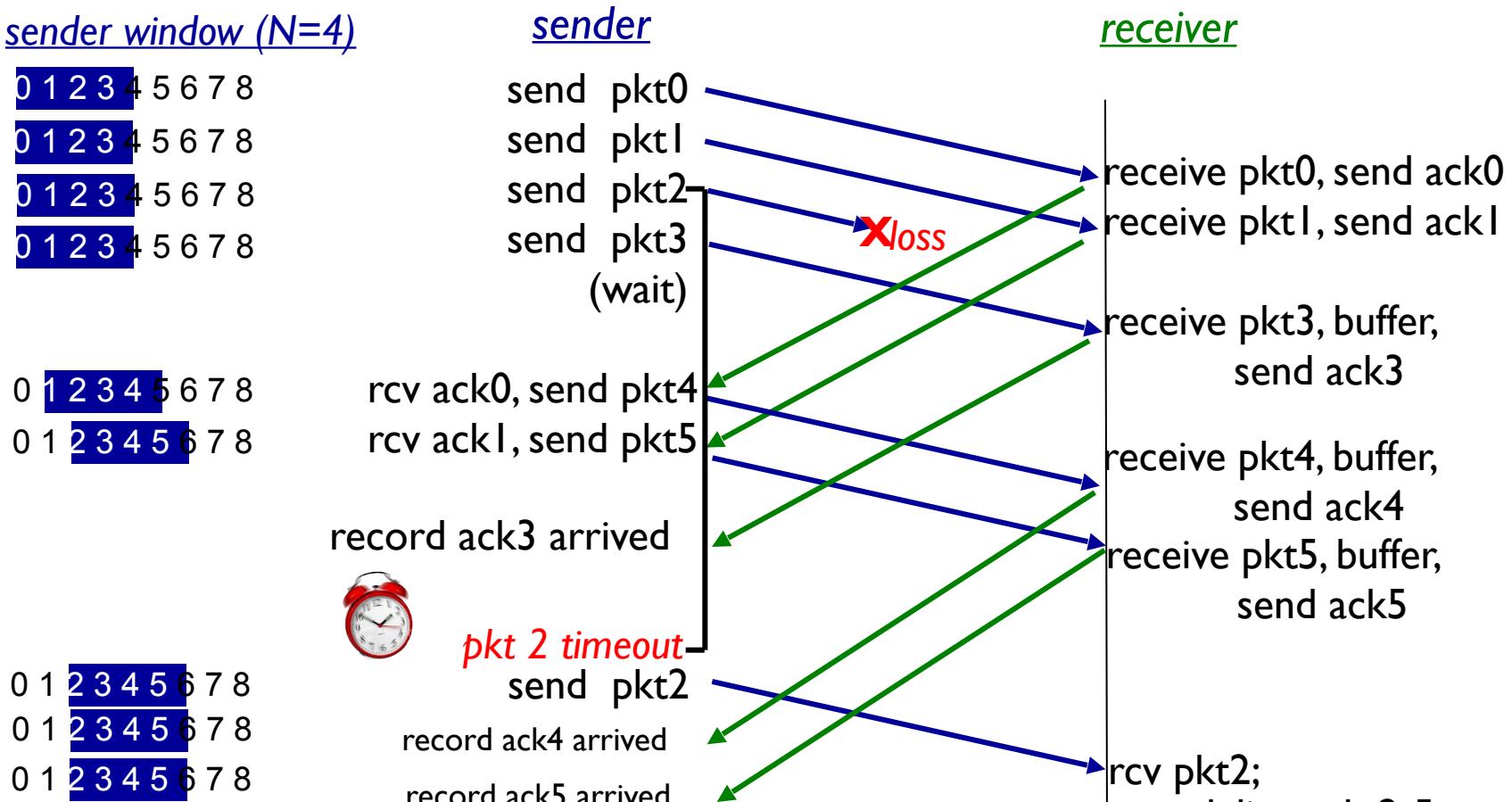
sender window (N=4)

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



Selective repeat in action

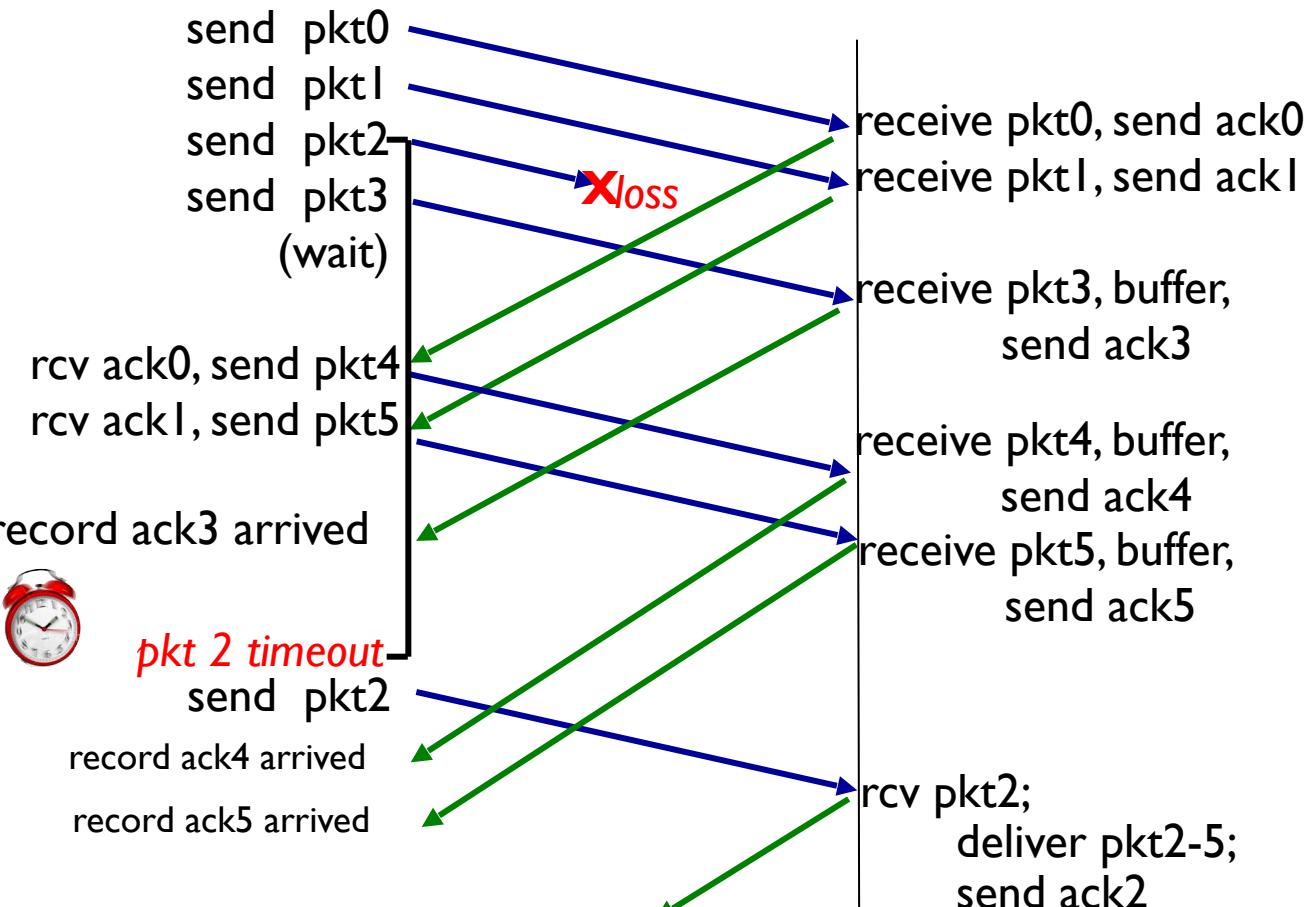
sender window (N=4)

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



Selective repeat 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

pkt 2 timeout
send pkt2

record ack4 arrived
record ack5 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 pkt2-5;
send ack2

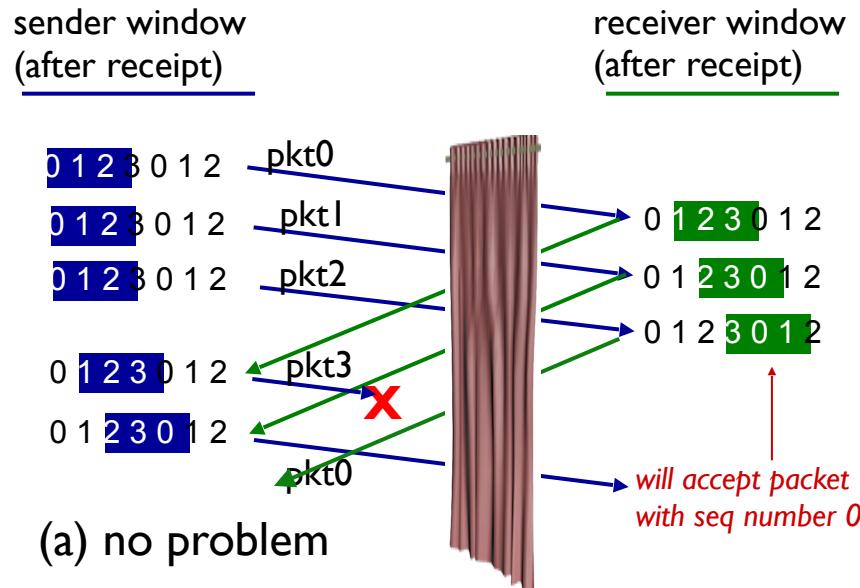
Q: what happens when ack2 arrives?

Selective repeat: dilemma

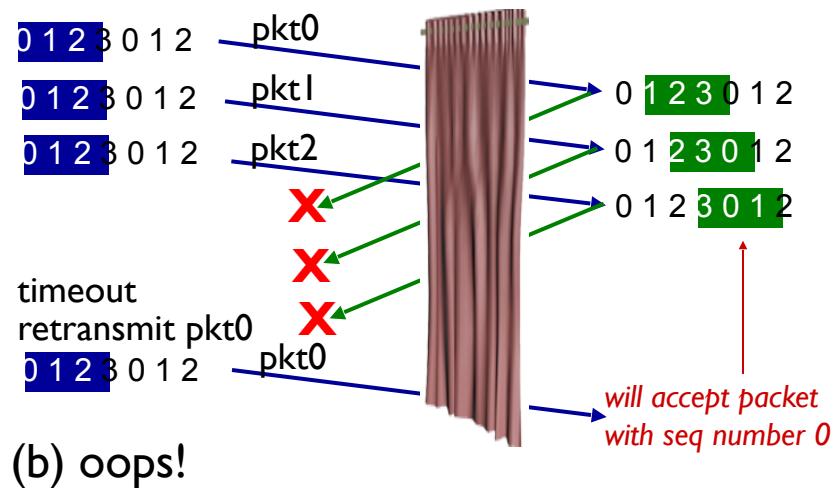
example:

- 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

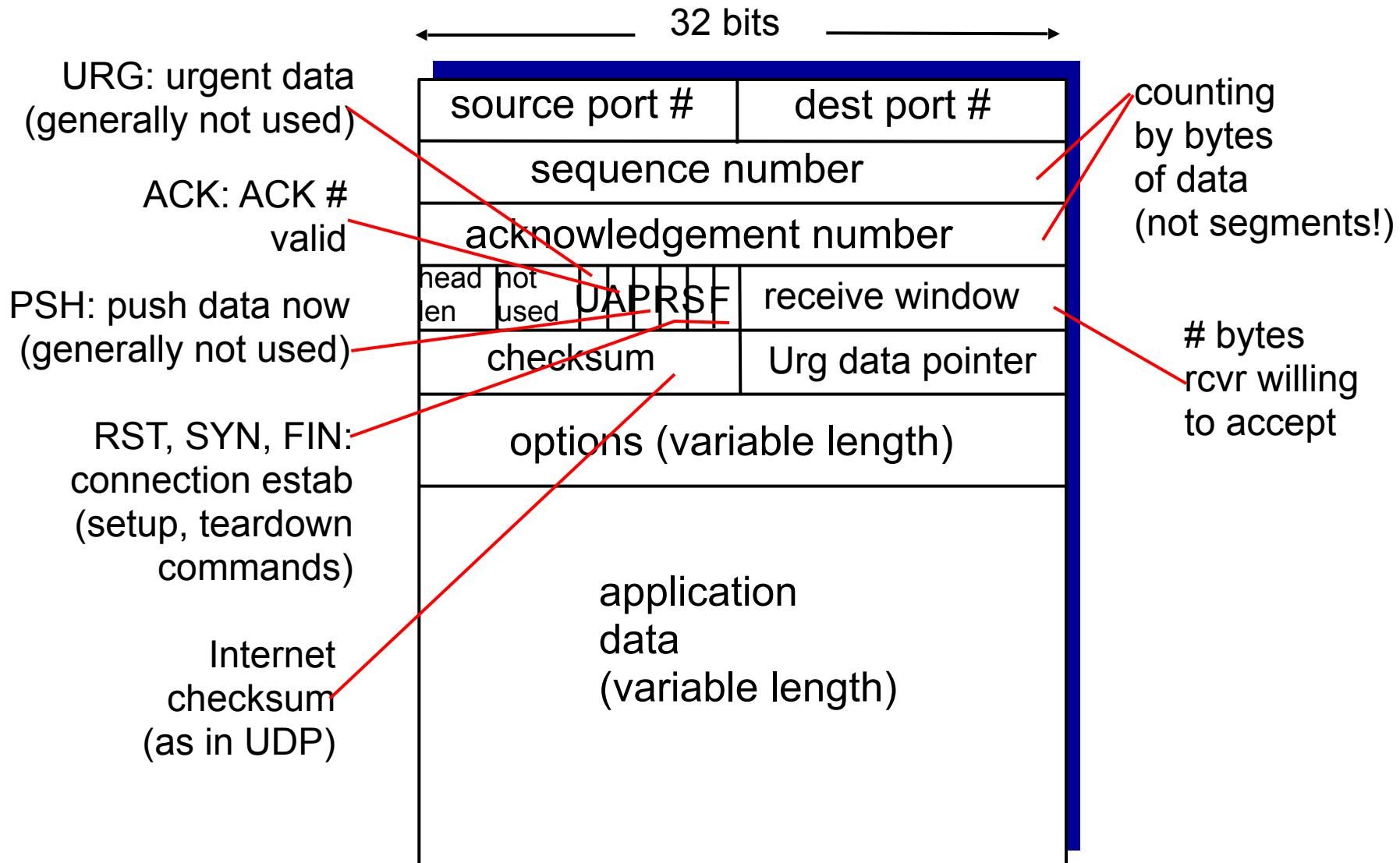
Today's class

TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- **point-to-point:**
 - one sender, one receiver
- **reliable, in-order byte stream:**
 - no “message boundaries”
 - Sent as segments, delivered to application as byte stream
 - MSS: maximum segment size
- **pipelined:**
 - TCP congestion and flow control set window size
- **full duplex data:**
 - bi-directional data flow in same connection
- **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:

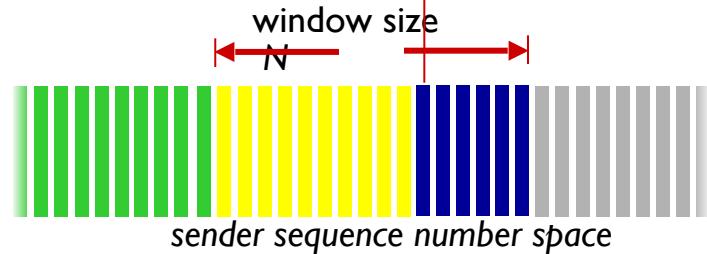
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

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

outgoing segment from sender

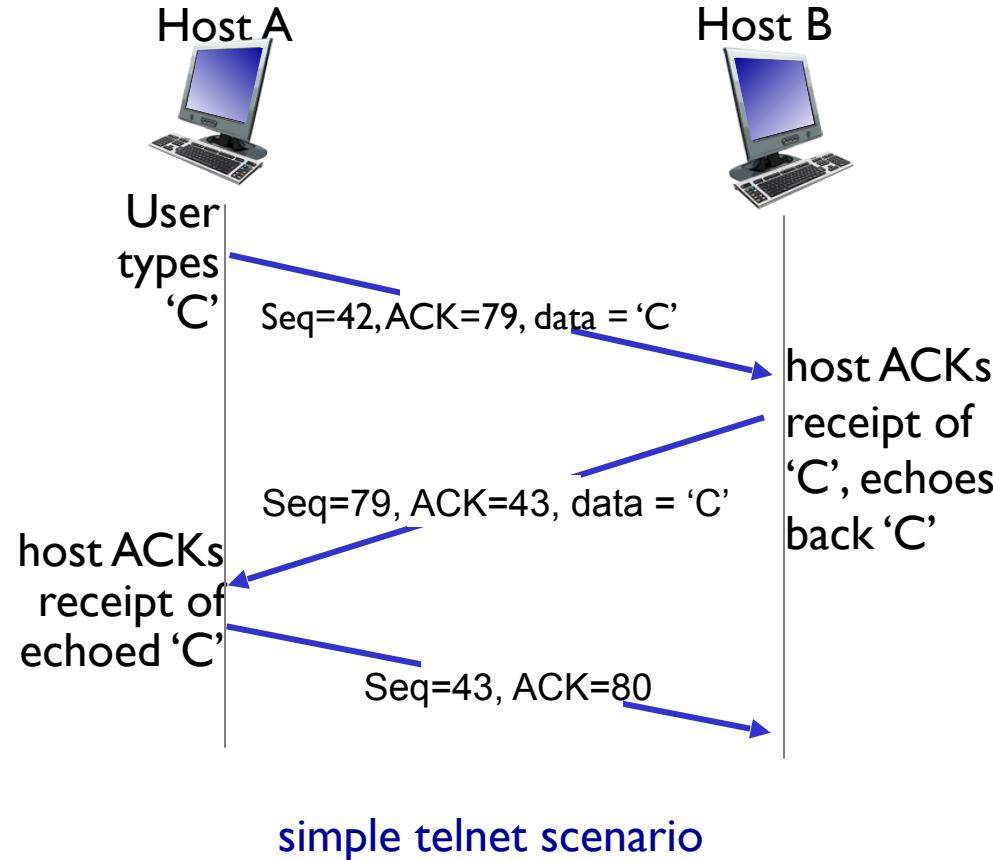
source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer



incoming segment to sender

source port #	dest port #
sequence number	
acknowledgement number	
	rwnd

TCP seq. numbers, ACKs



Day 14: TCP and Congestion Control



CSEE 4119
Computer Networks
Ethan Katz-Bassett

 COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

Slides adapted from (and often identical to) slides from Kurose and Ross.

All material copyright 1996-2020

J.F.Kurose and K.W.Ross, All Rights Reserved

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

TCP round trip time, timeout

- **timeout interval:** EstimatedRTT plus “safety margin”
 - large variation in EstimatedRTT -> larger safety margin

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



- **details:**
 - exponential weighted moving average (EWMA)
 - EstimatedRTT = EWMA of RTT measurements
 - DevRTT = EWMA of deviation from EstimatedRTT
 - For details: Skipped slides next, 3.5.3 in book, video from 2018/10/18 started ~53:30

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

SKIPPED: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

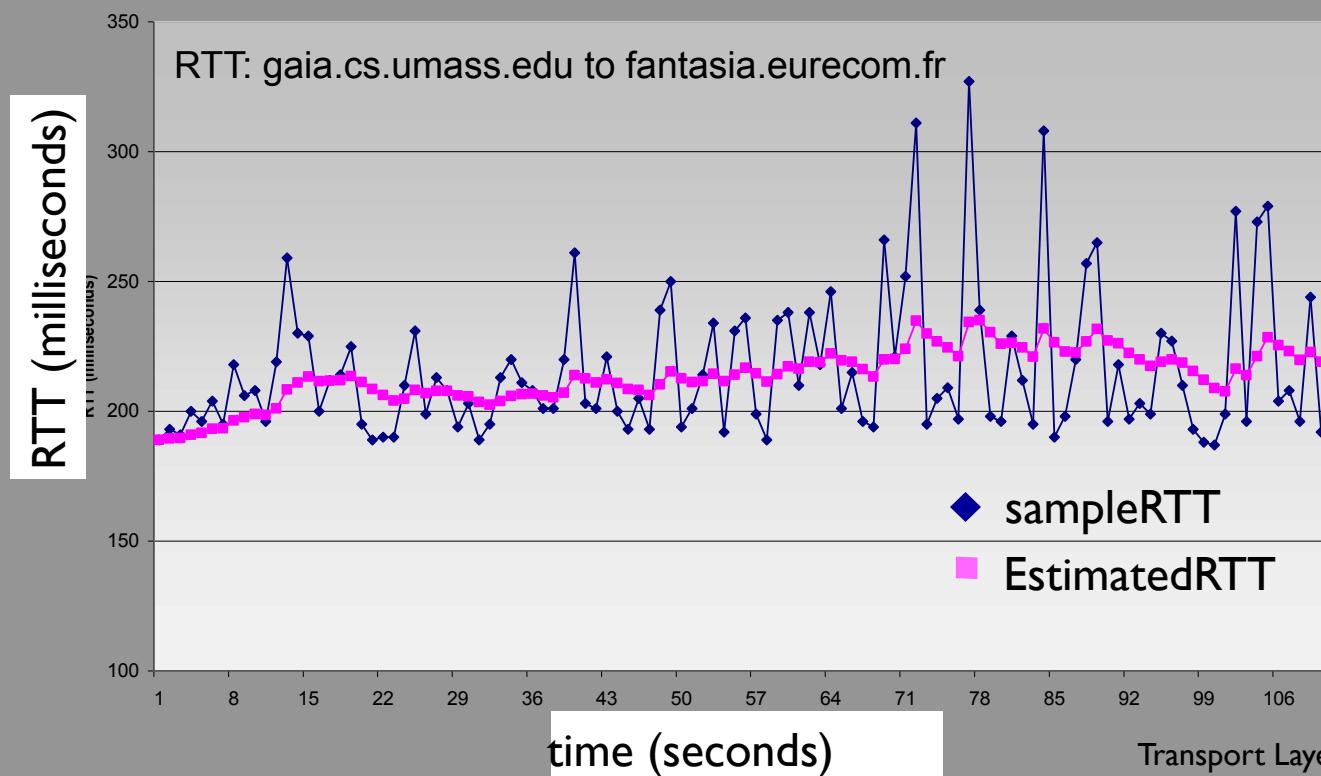
Q: how to estimate RTT?

- **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**

SKIPPED: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: $\alpha = 0.125$



SKIPPED: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”

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

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:

event: data received 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`

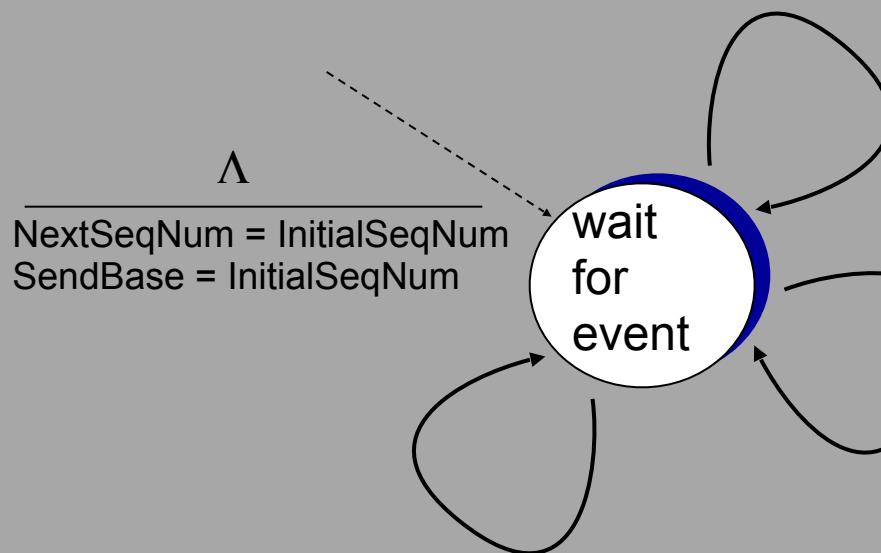
event: timeout:

- 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)



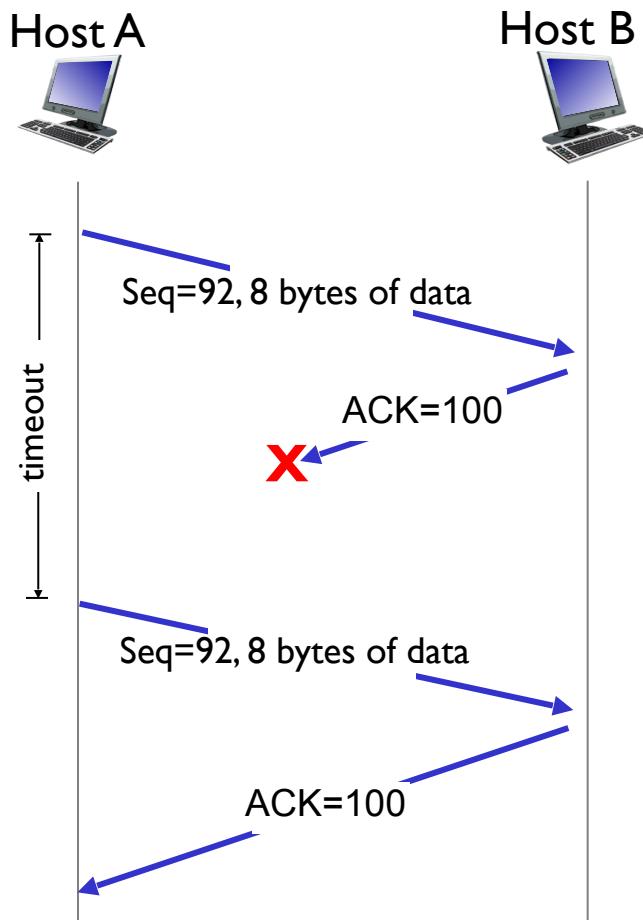
data received from application above
create segment, seq. #: NextSeqNum
pass segment to IP (i.e., “send”)
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
 start timer

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

ACK received, with ACK field value y

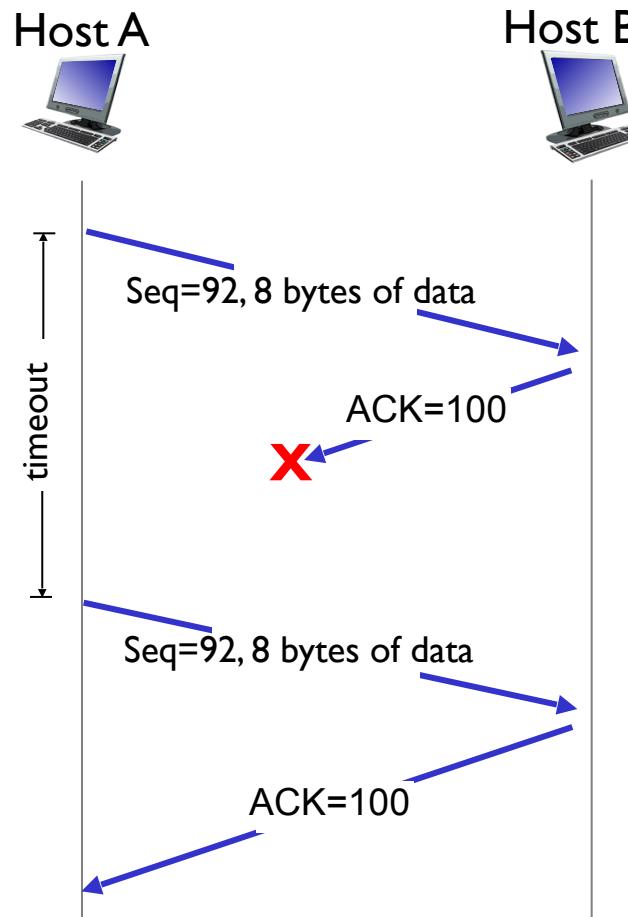
```
if (y > SendBase) {  
    SendBase = y  
    /* SendBase-1: last cumulatively ACKed byte */  
    if (there are currently not-yet-acked segments)  
        start timer  
    else stop timer  
}
```

TCP: retransmission scenarios

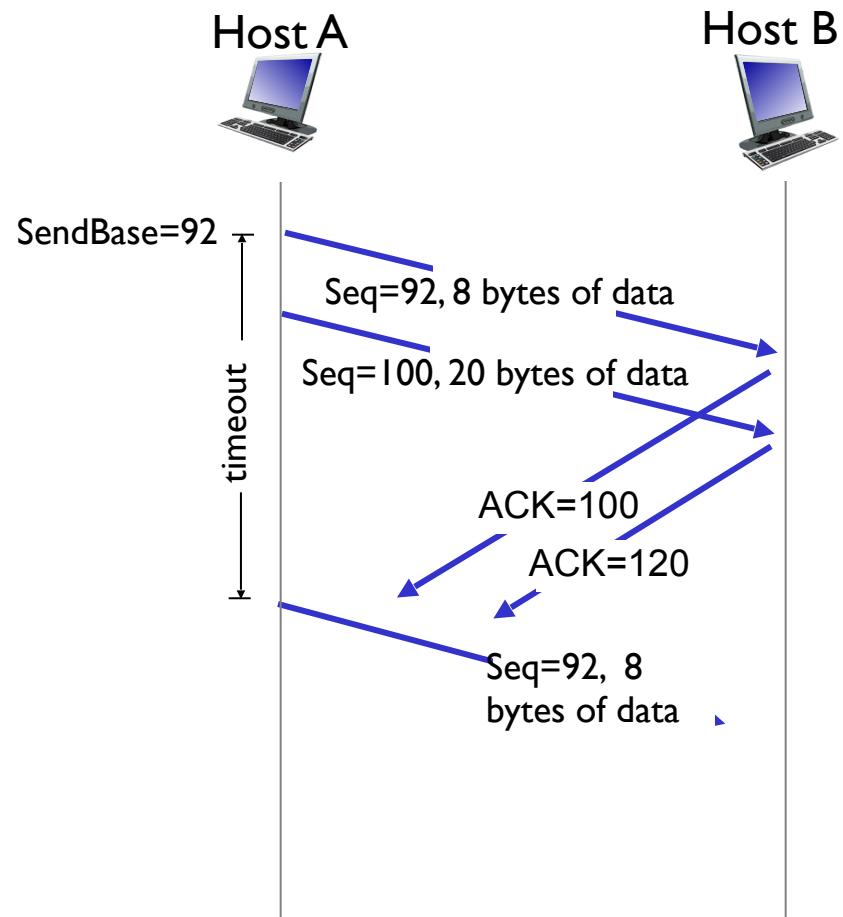


lost ACK scenario

TCP: retransmission scenarios

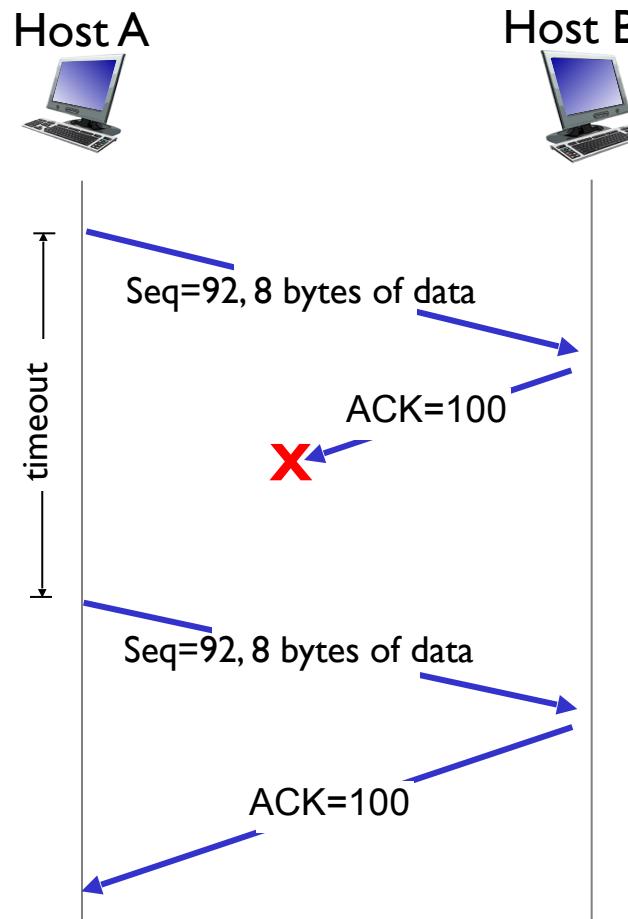


lost ACK scenario

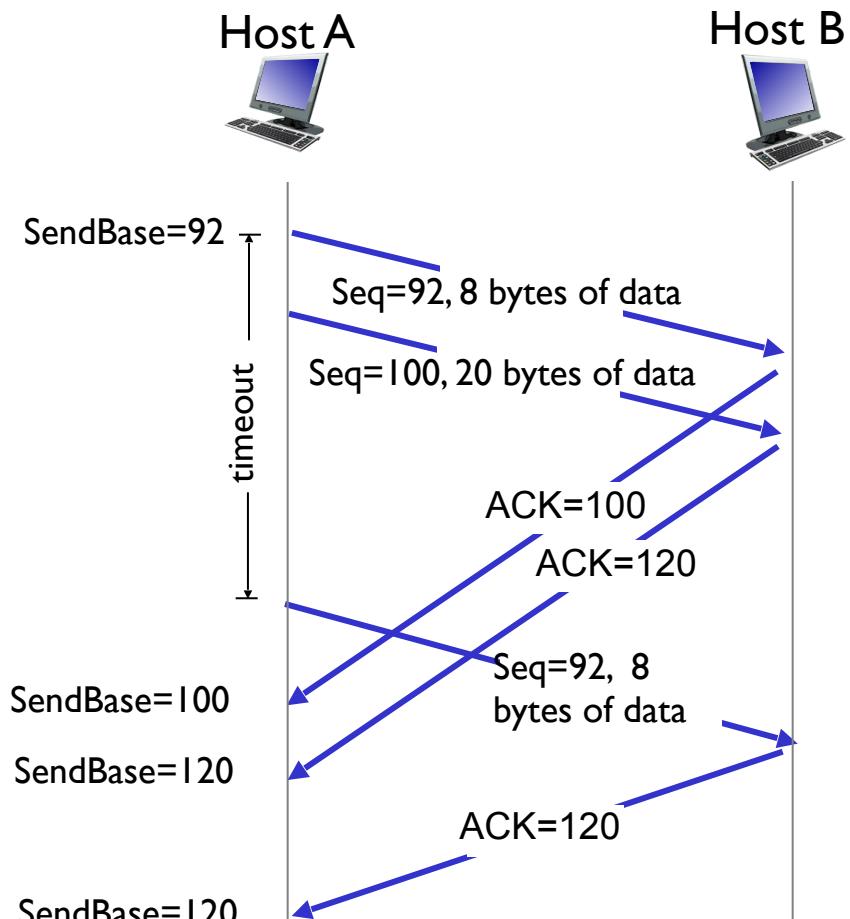


premature timeout

TCP: retransmission scenarios

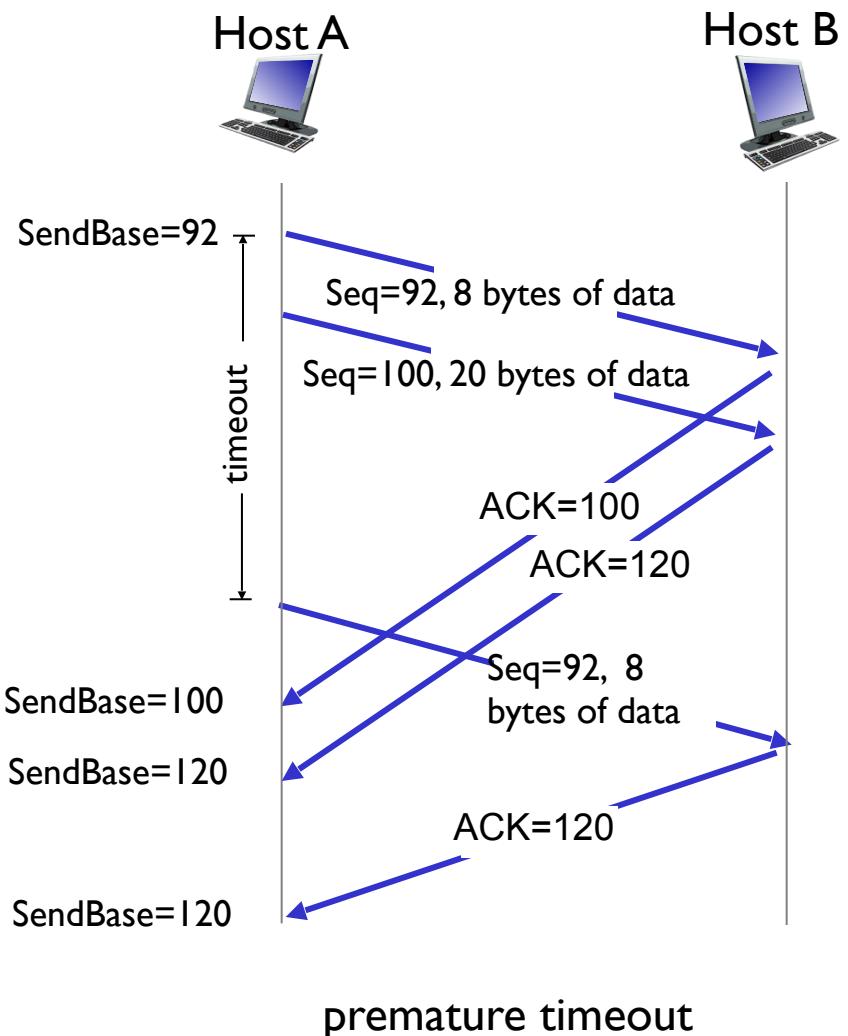


lost ACK scenario

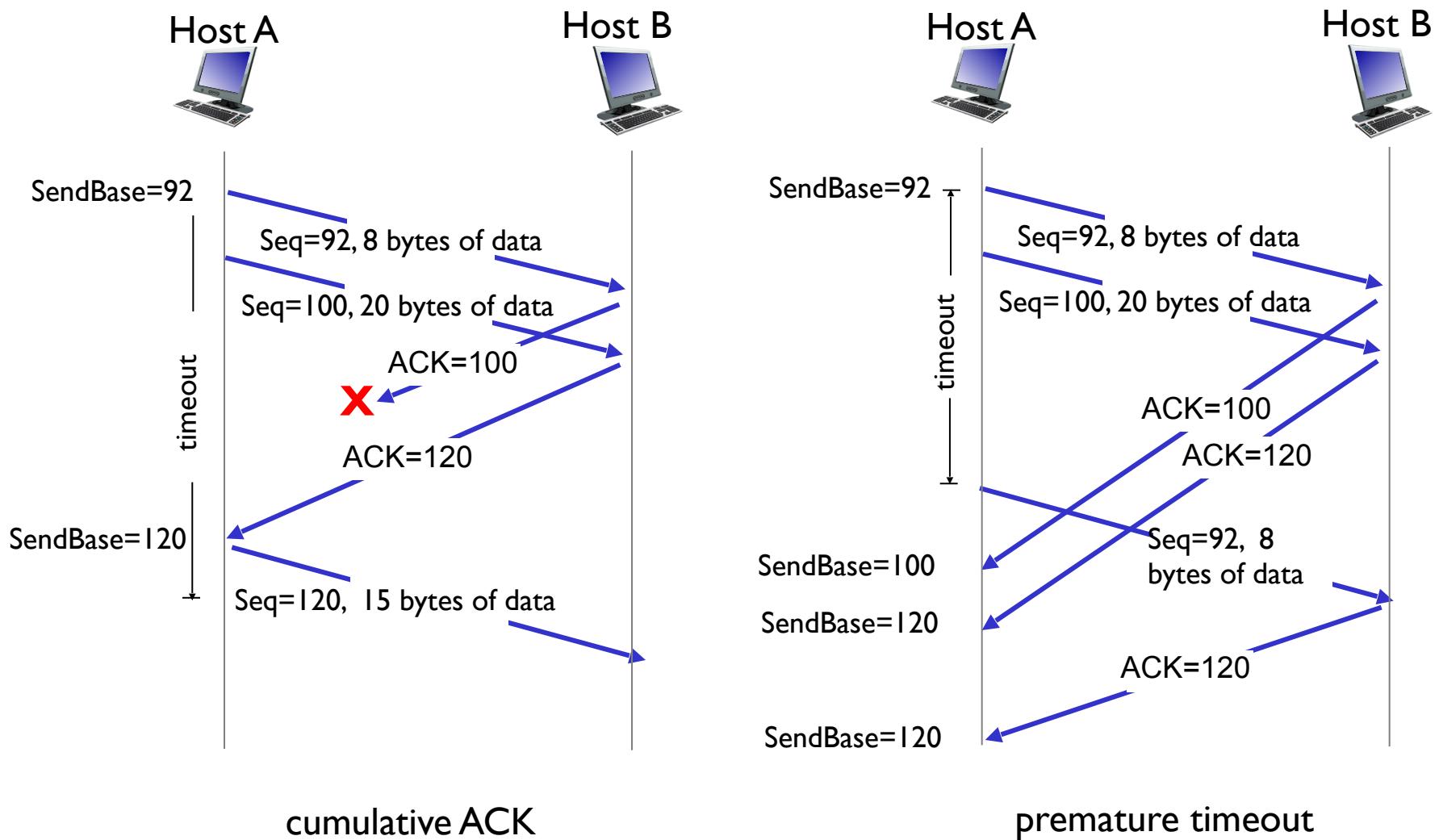


premature timeout

TCP: retransmission scenarios



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 fast retransmit

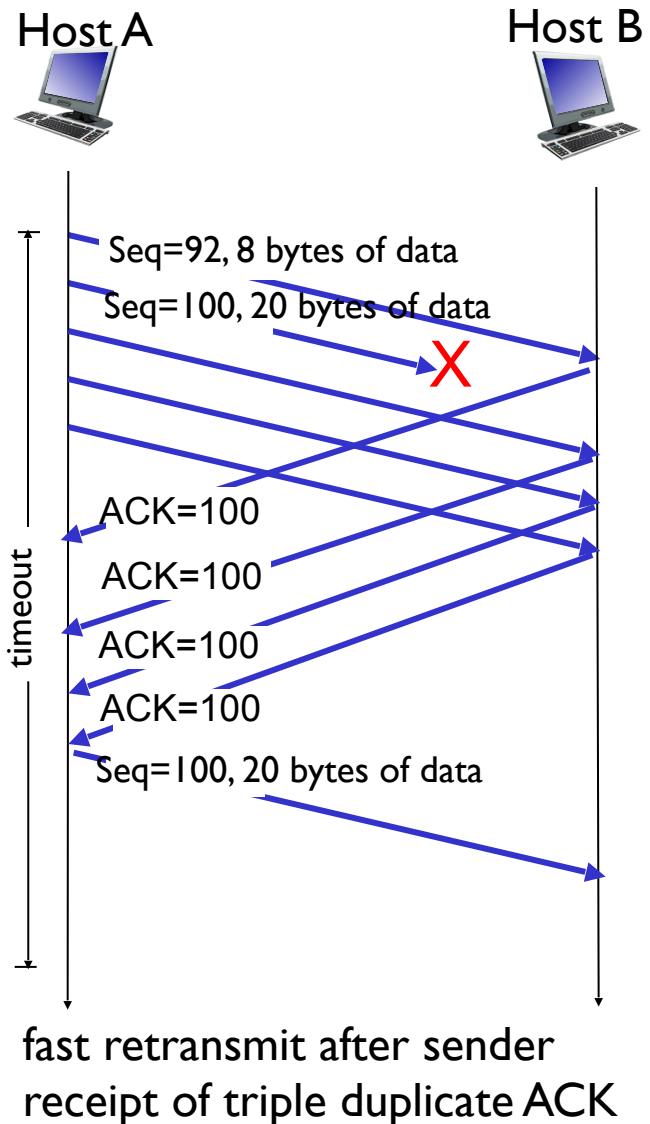
- time-out period often relatively long:
 - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
 - sender often sends many segments back-to-back
 - if segment is lost, there will likely be many duplicate ACKs.

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



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
- connection management
- flow control

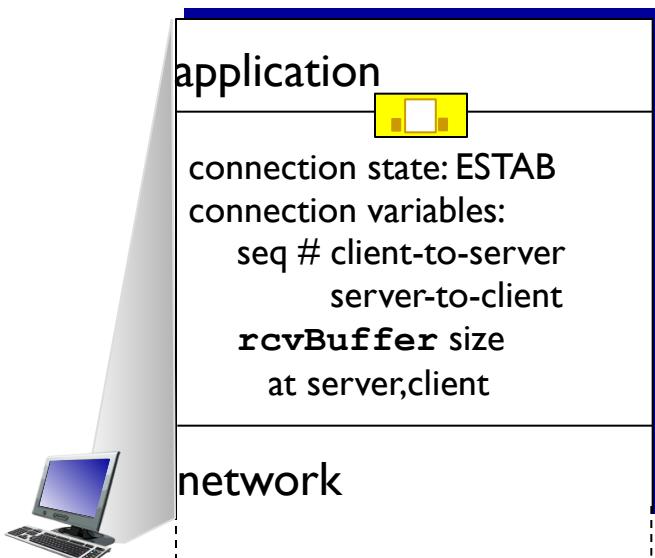
3.6 principles of congestion control

3.7 TCP congestion control

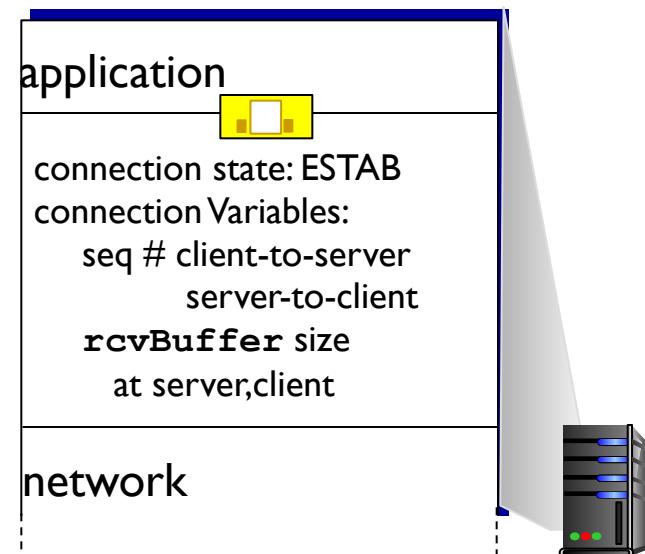
Connection Management

before exchanging data, sender/receiver “handshake”:

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



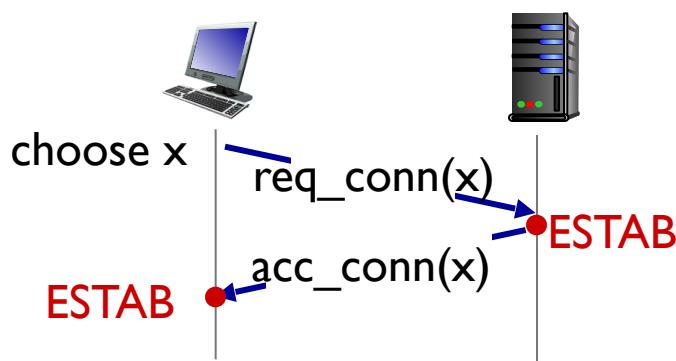
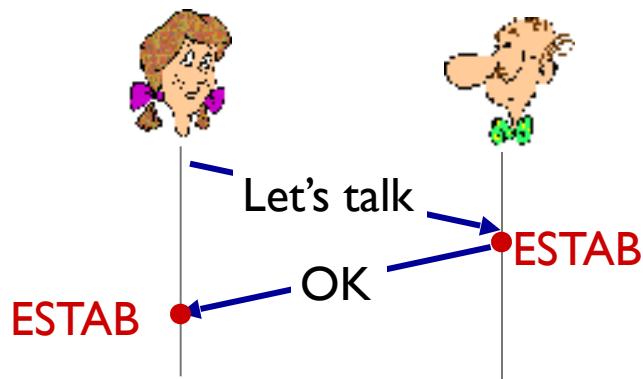
```
Socket clientSocket =  
    newSocket("hostname", "port number");
```



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

Agreeing to establish a connection

2-way handshake:

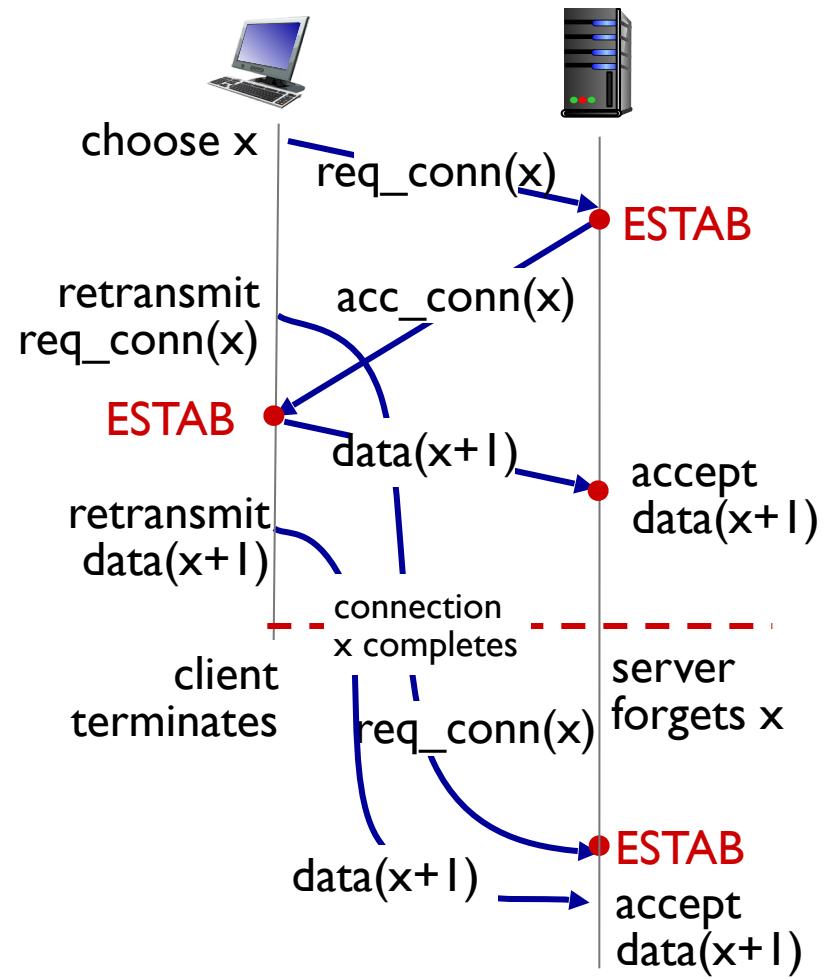
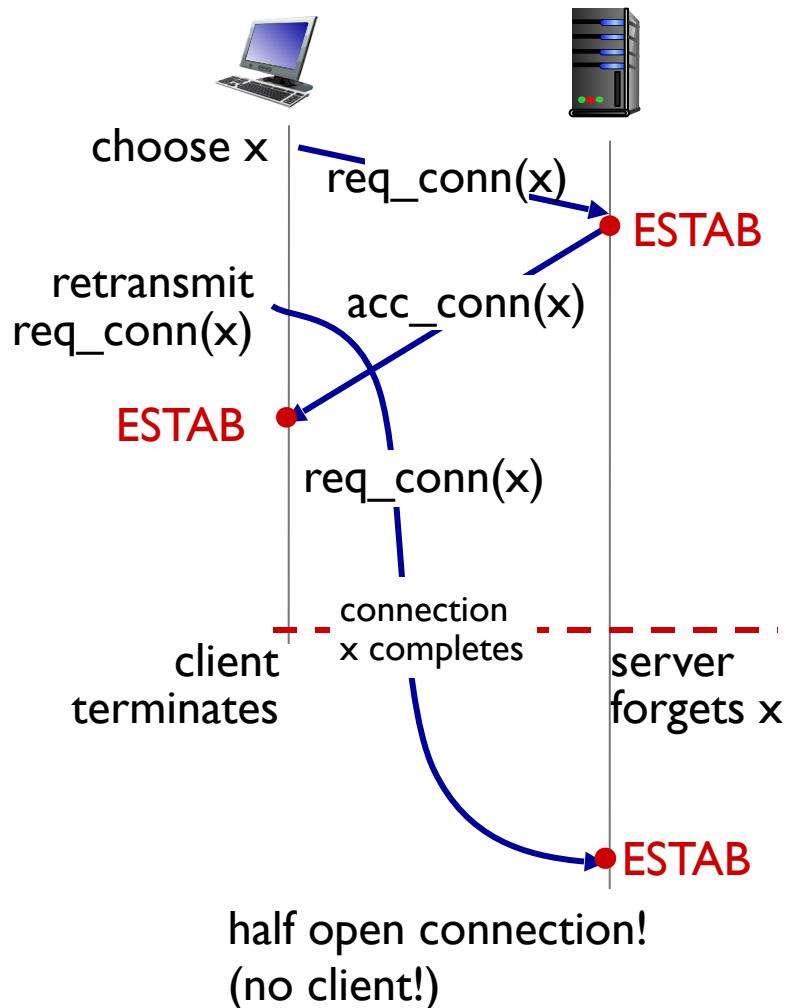


Q: will 2-way handshake always work in network?

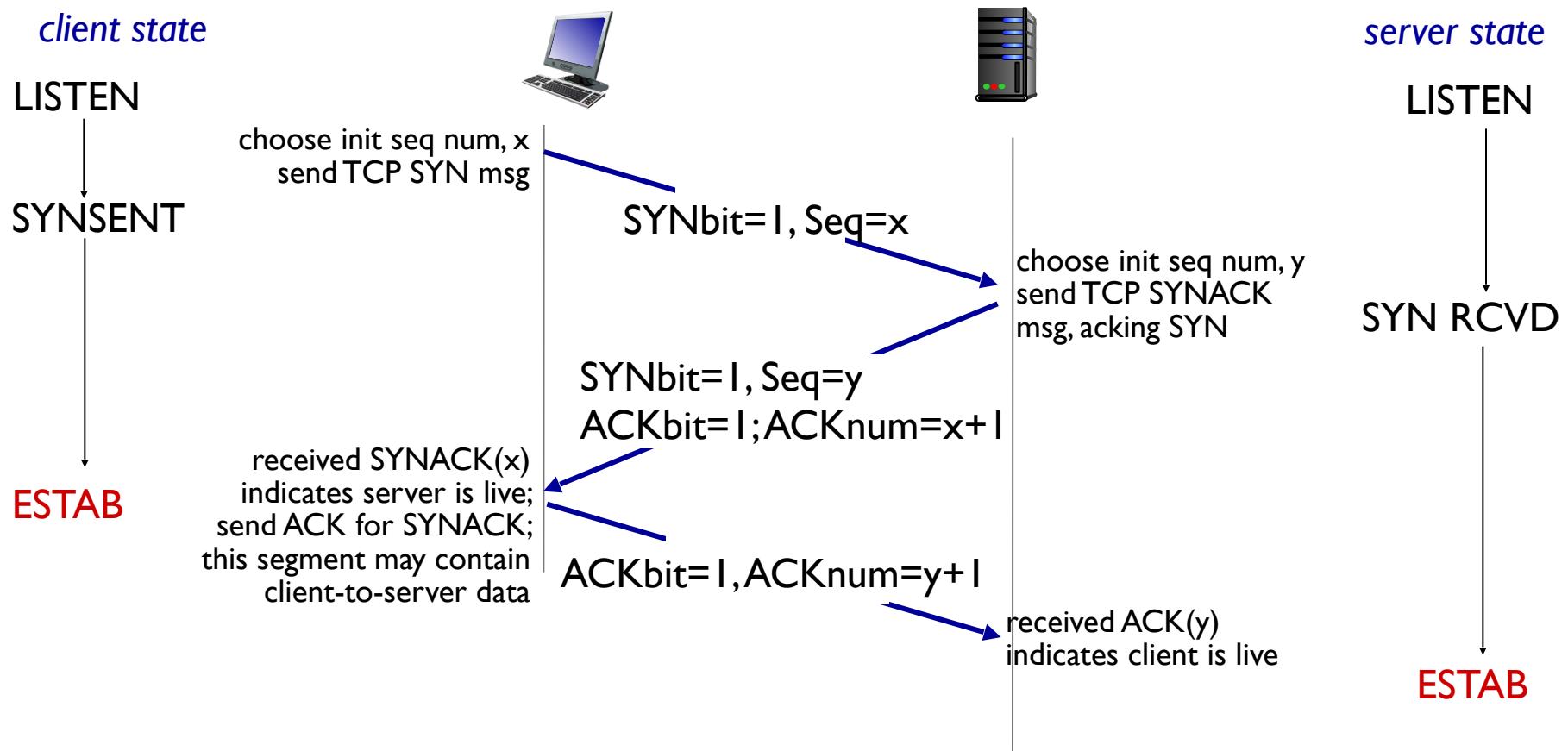
- variable delays
- retransmitted messages (e.g. `req_conn(x)`) due to message loss
- message reordering
- can't "see" other side

Agreeing to establish a connection

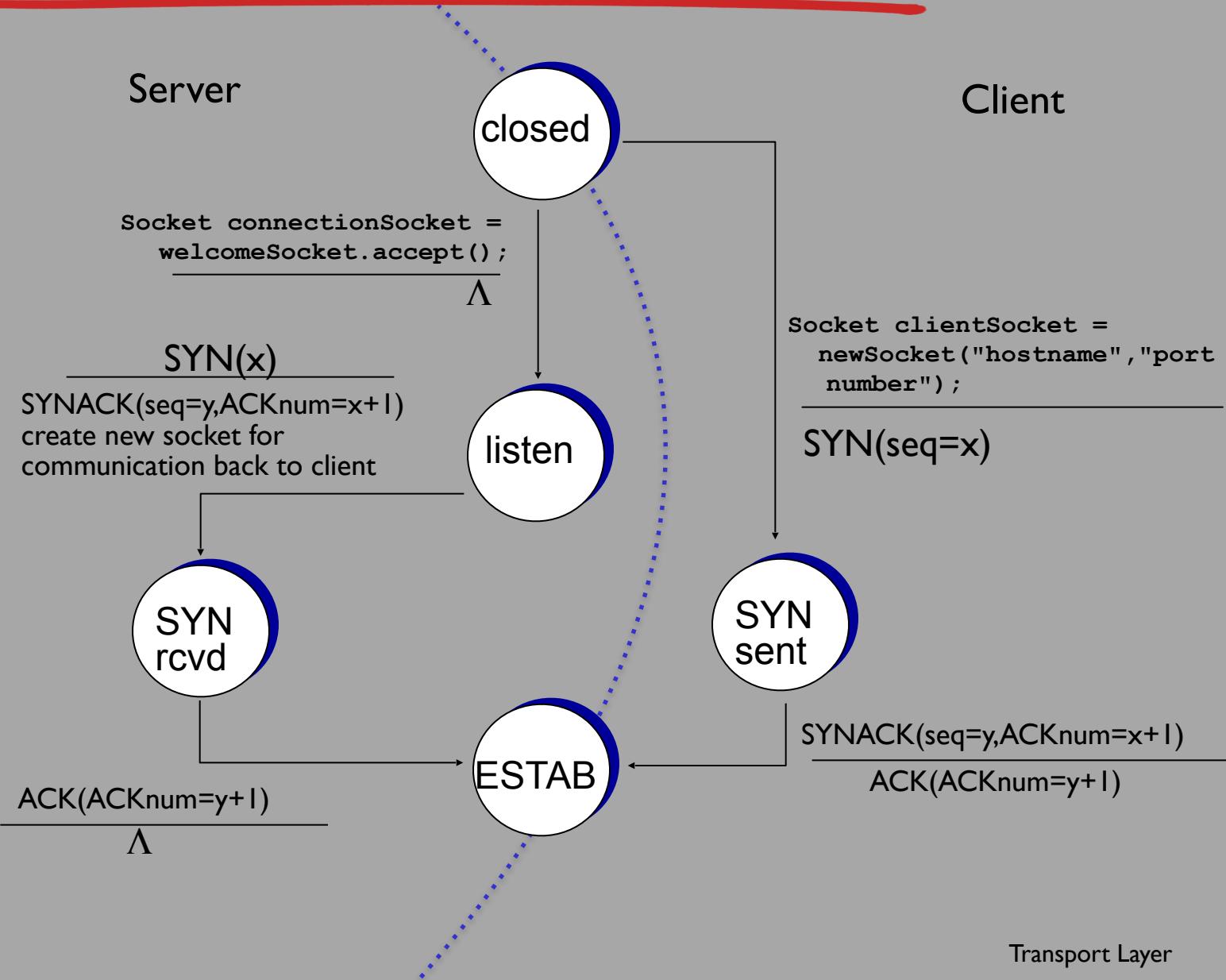
2-way handshake failure scenarios:



TCP 3-way handshake



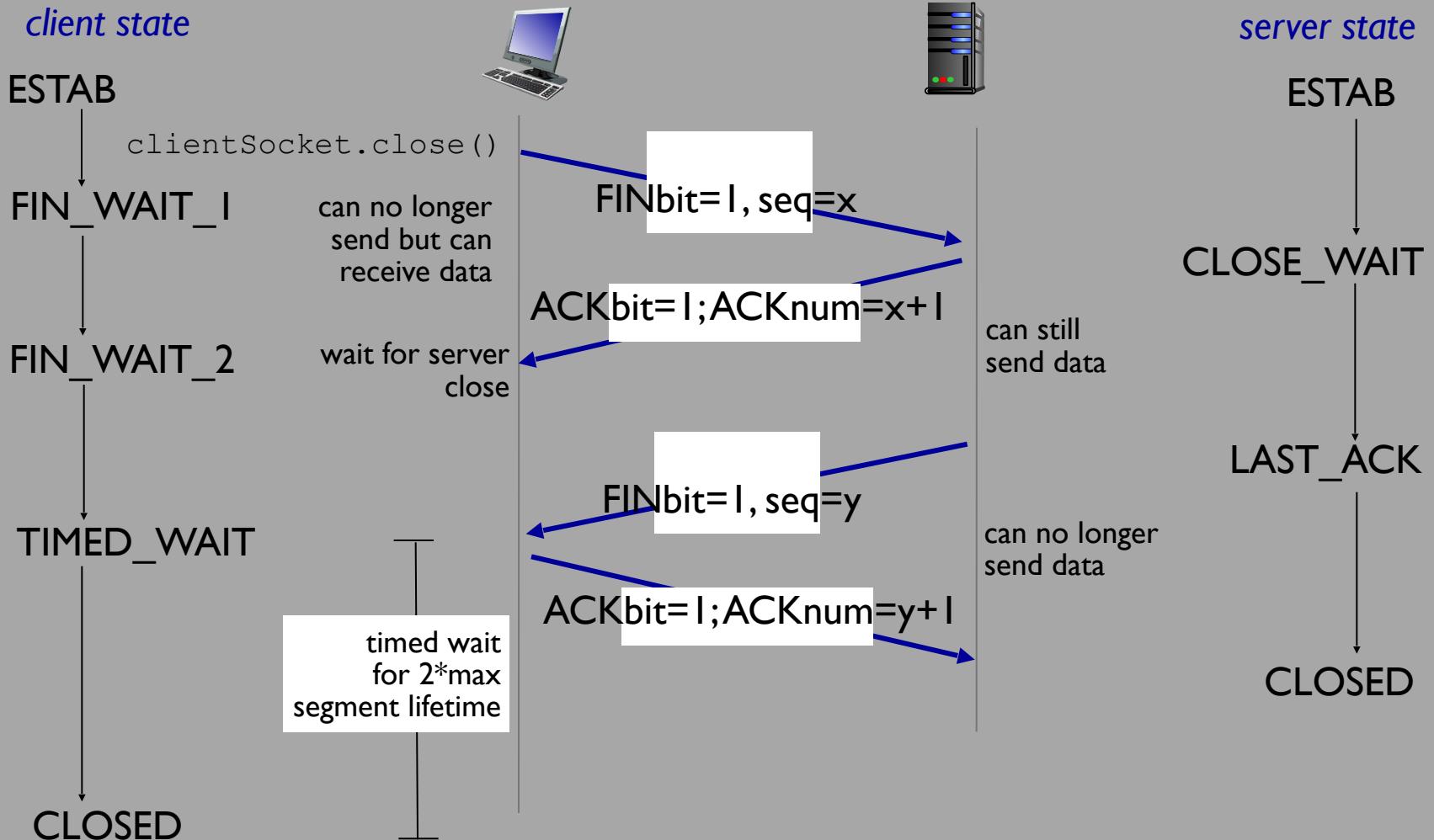
TCP 3-way handshake: FSM



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



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
- connection management
- **flow control**

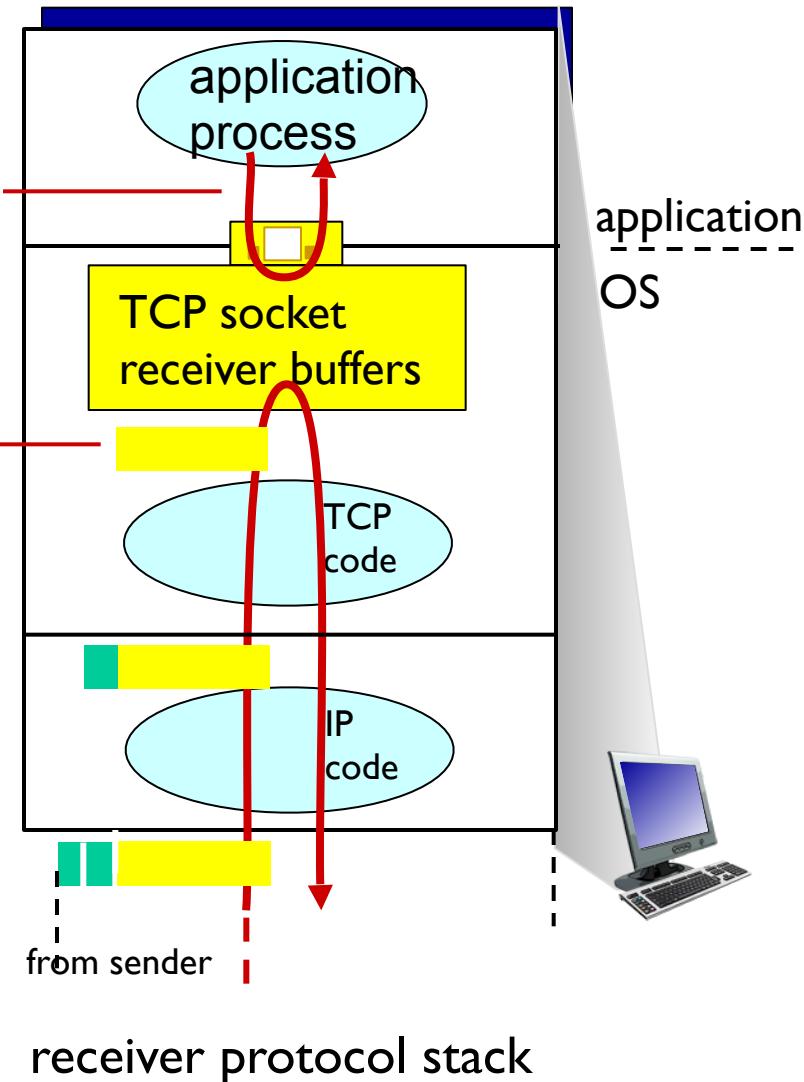
3.6 principles of congestion control

3.7 TCP congestion control

TCP flow control

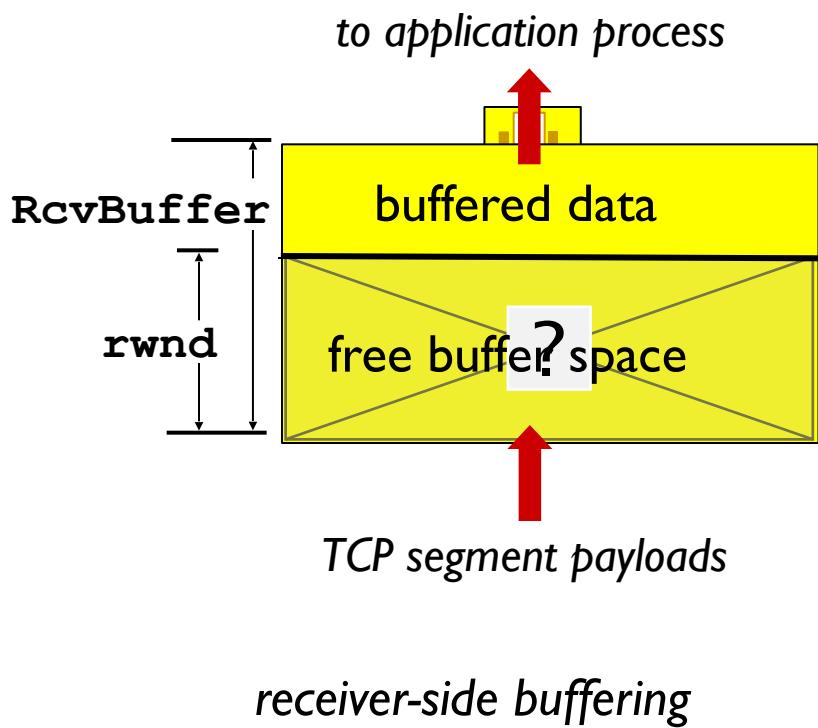
application may
remove data from
TCP socket buffers

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



TCP flow control

- receiver “advertises” free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments
- sender limits amount of unacked (“in-flight”) data to receiver’s **rwnd** value
- guarantees receive buffer will not overflow
- Details
 - RcvBuffer size set via socket options (typical default is 4096 bytes)
 - many operating systems autoadjust **RcvBuffer**



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

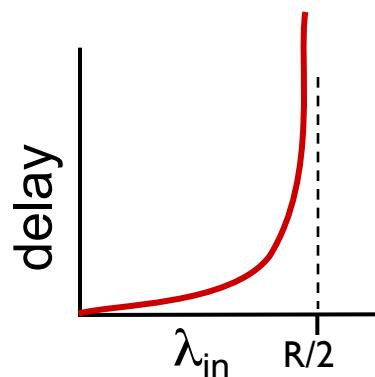
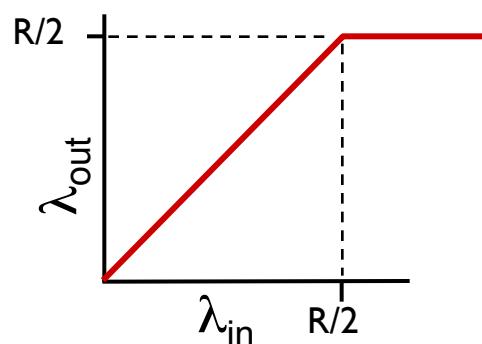
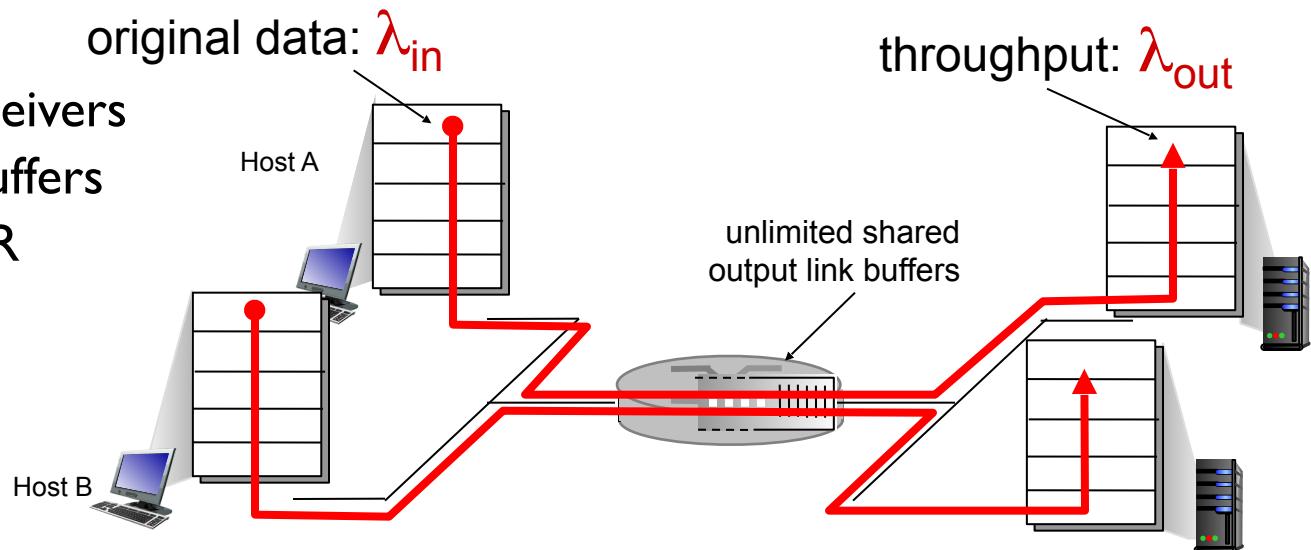
Principles of congestion control

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

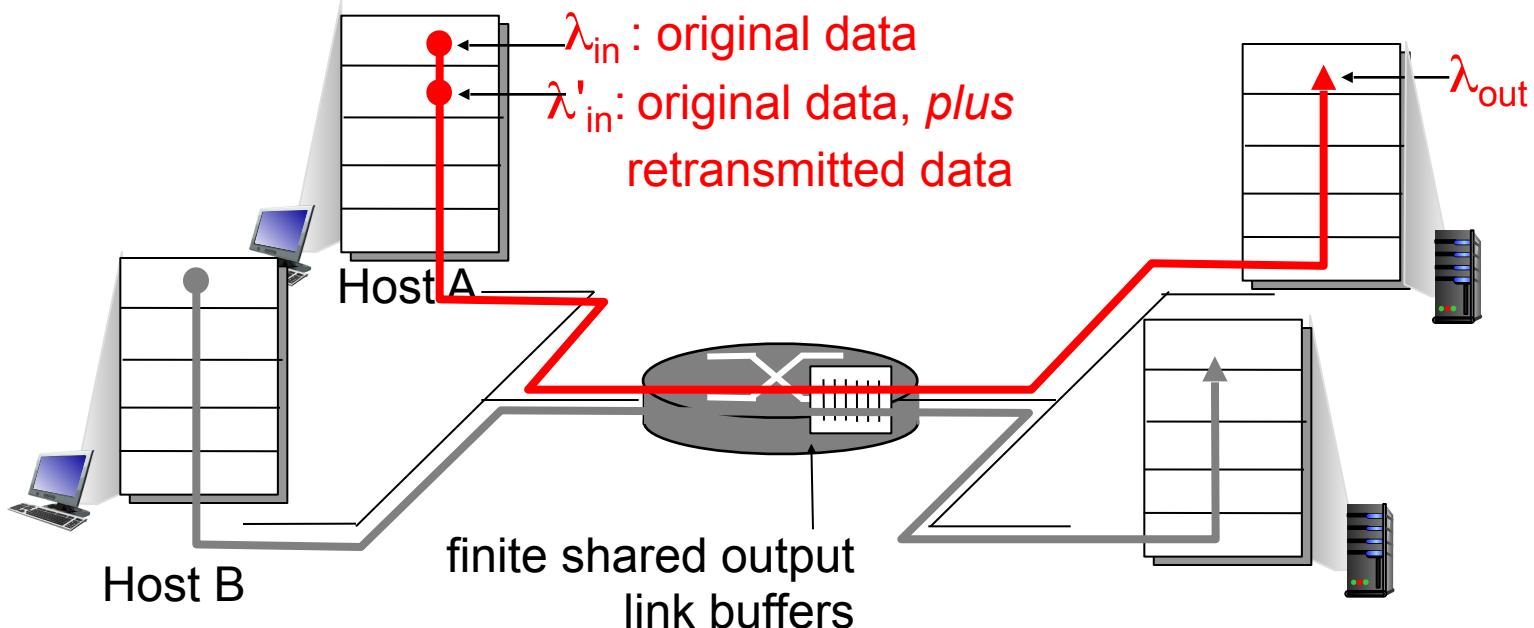
- two senders, two receivers
- one router, infinite buffers
- output link capacity: R
- no retransmission



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

Causes/costs of congestion: scenario 2

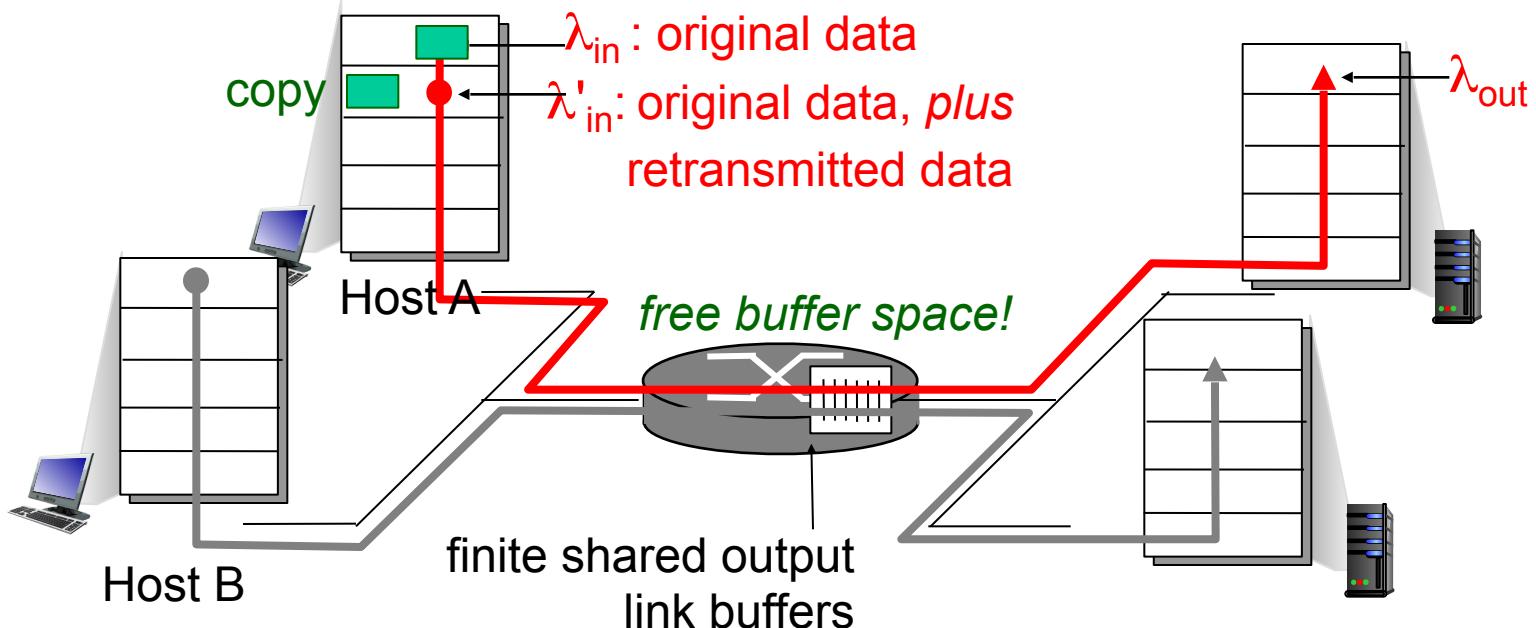
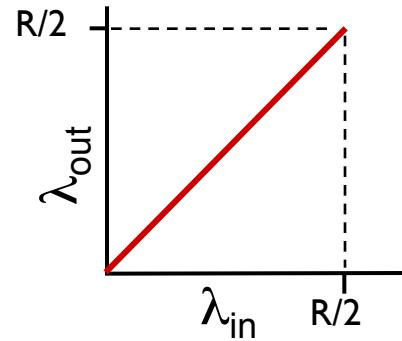
- one router, *finite* buffers
- sender retransmission of timed-out packet
 - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
 - transport-layer input includes *retransmissions* : $\lambda'_{in} > \lambda_{in}$



Causes/costs of congestion: scenario 2a

idealization: perfect knowledge

- sender sends only when router buffers available



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

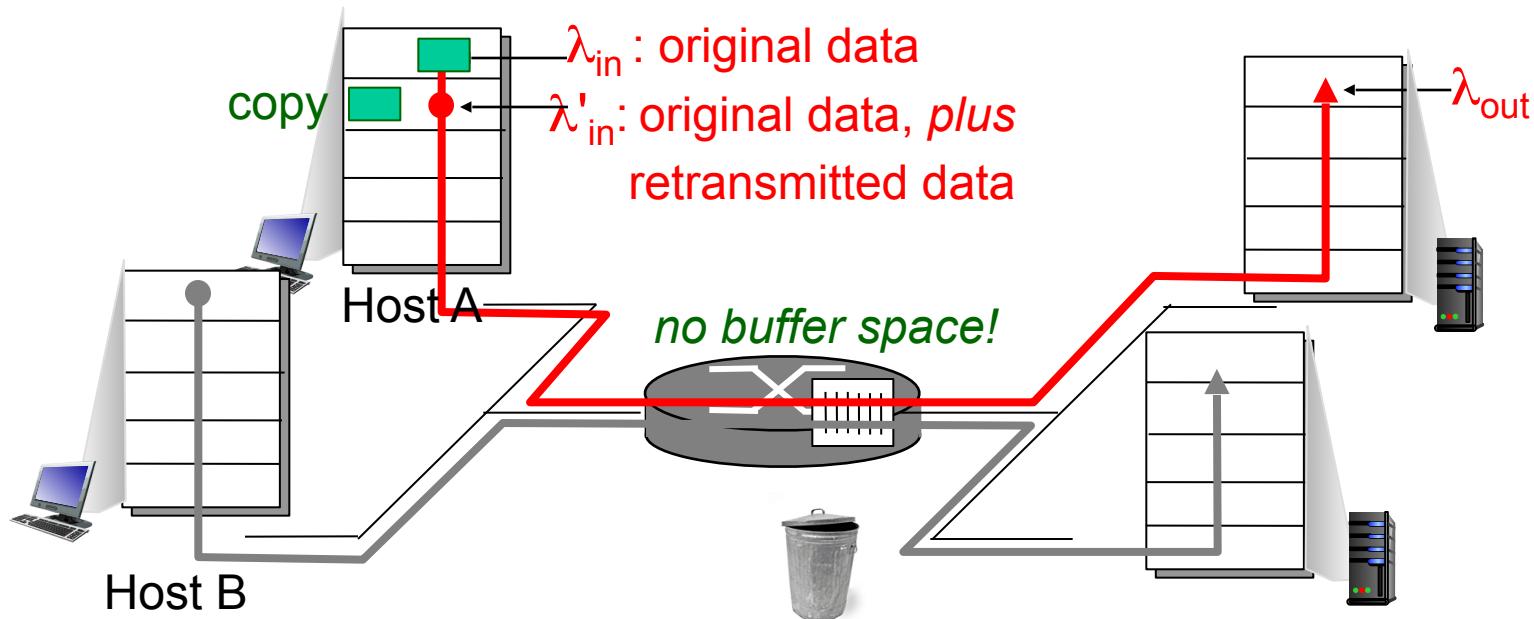
Today's class

Causes/costs of congestion: scenario 2b

Idealization: known loss

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

- sender only resends if
packet known to be lost

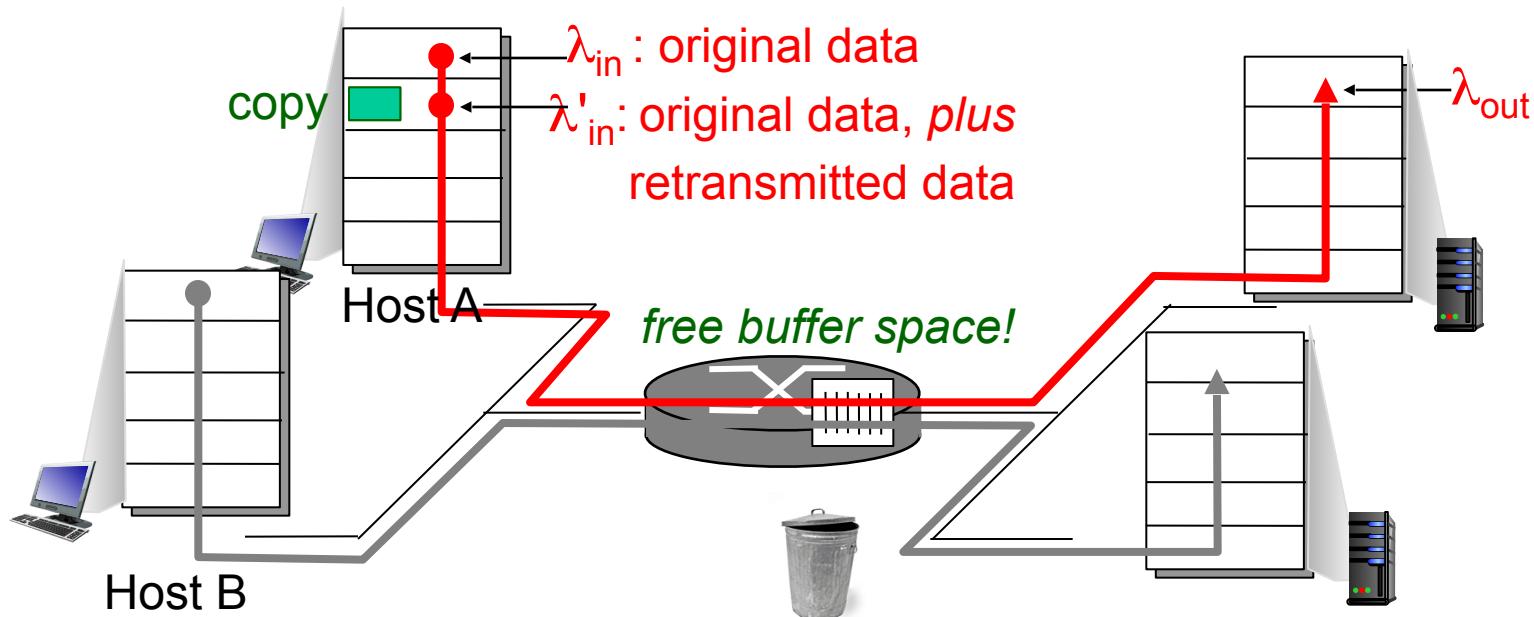
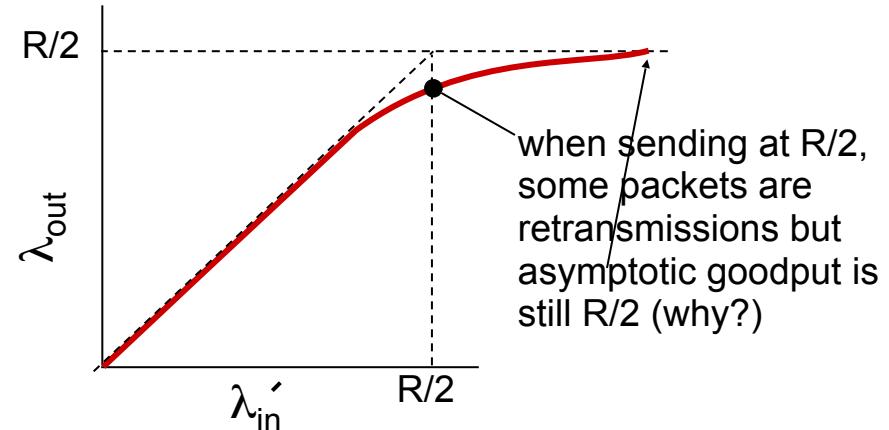


Causes/costs of congestion: scenario 2b

Idealization: known loss

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

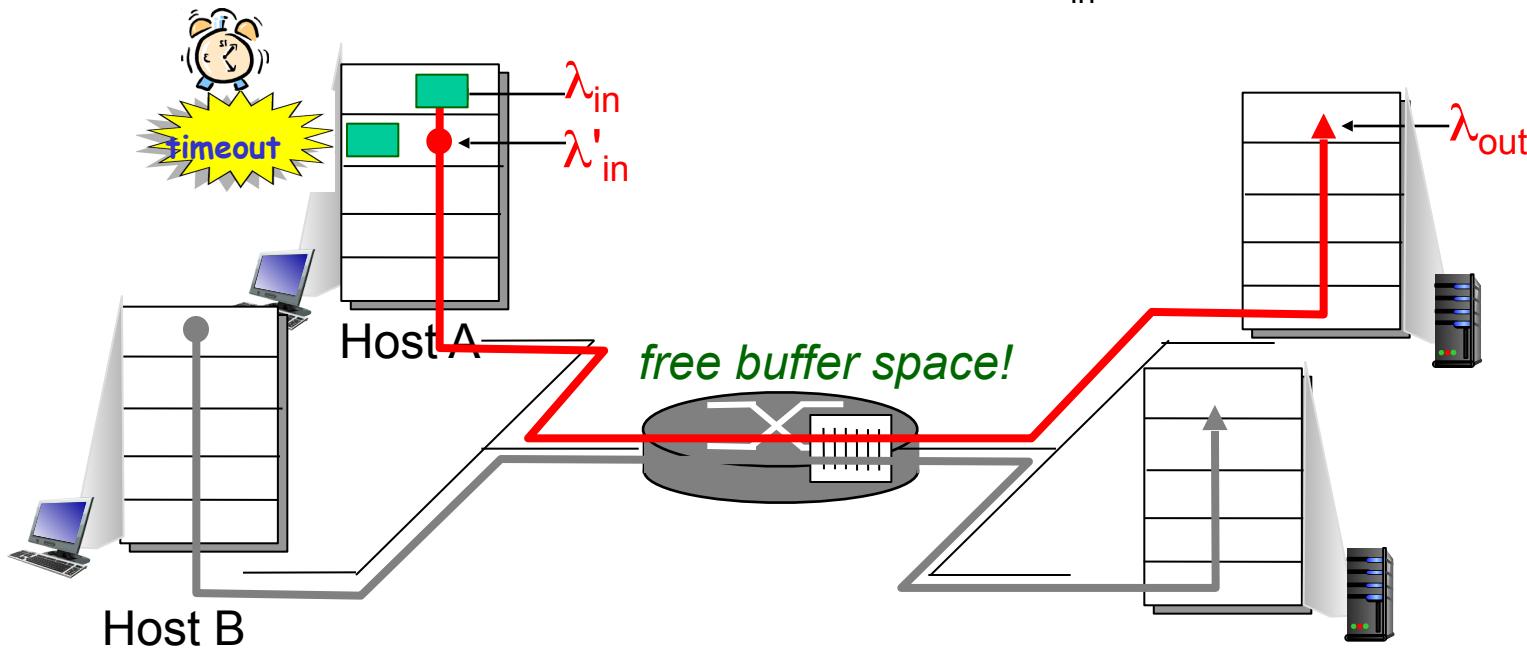
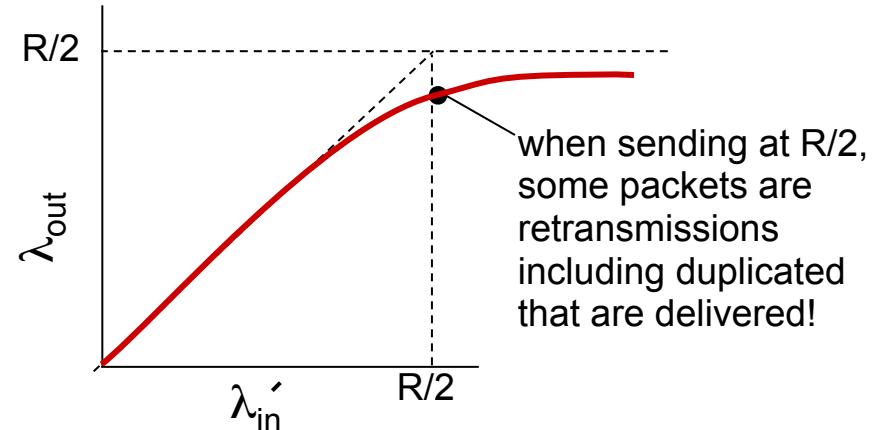
- sender only resends if
packet known to be lost



Causes/costs of congestion: scenario 2c

Realistic: *duplicates*

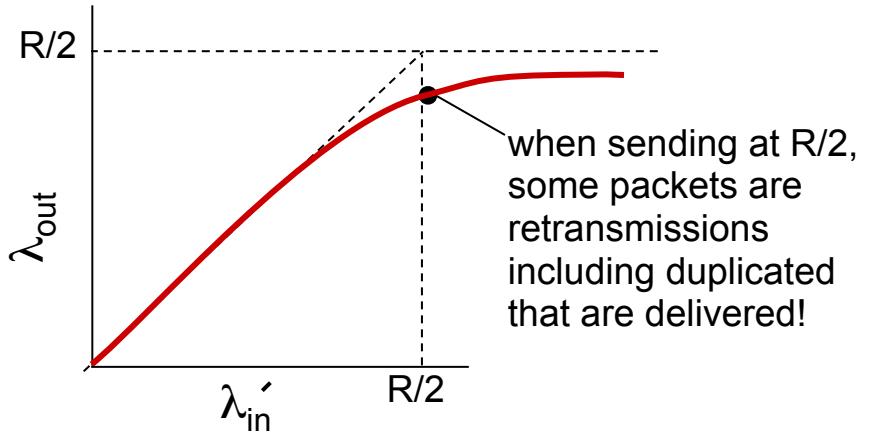
- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending **two** copies, both of which are delivered



Causes/costs of congestion: scenario 2c

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:

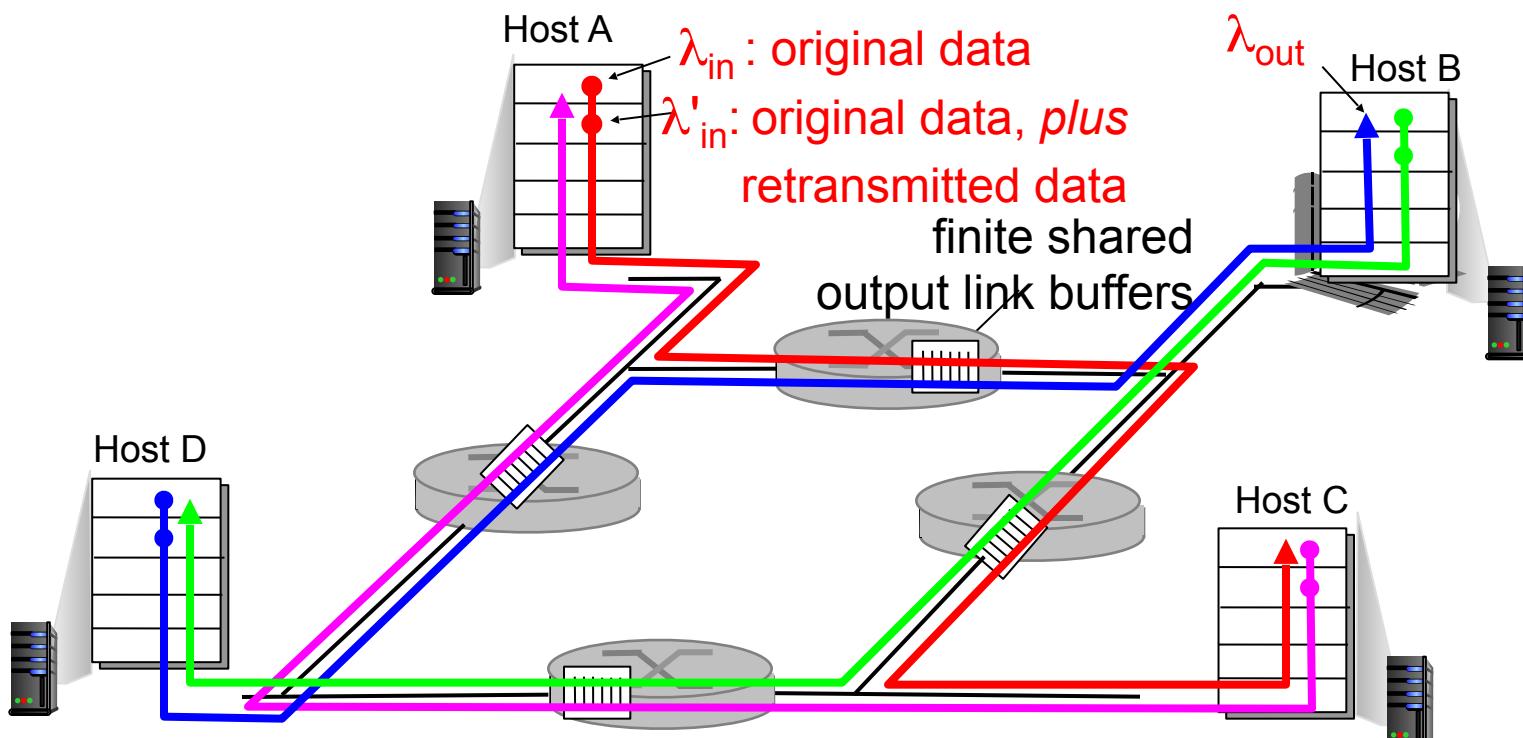
- more work (retransmissions) for given “goodput”
- unneeded retransmissions: link carries multiple copies of packet
 - decreasing goodput

Causes/costs of congestion: scenario 3

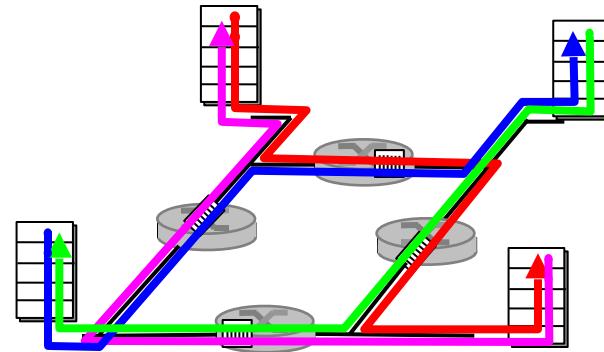
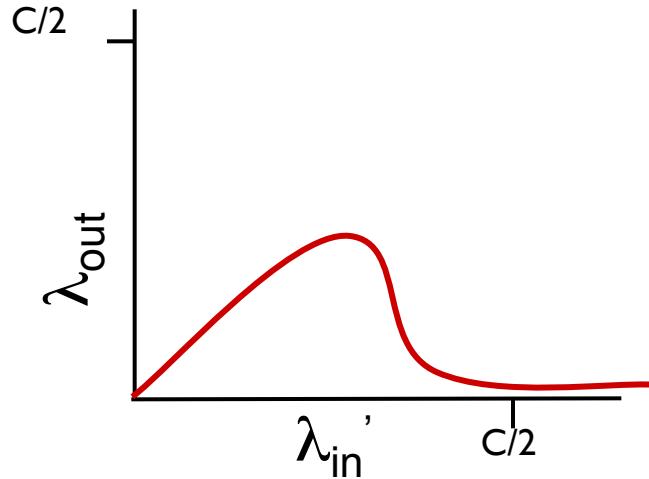
- four senders
- multihop paths
- timeout/retransmit

Q: what happens as λ_{in} and λ'_{in} increase ?

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



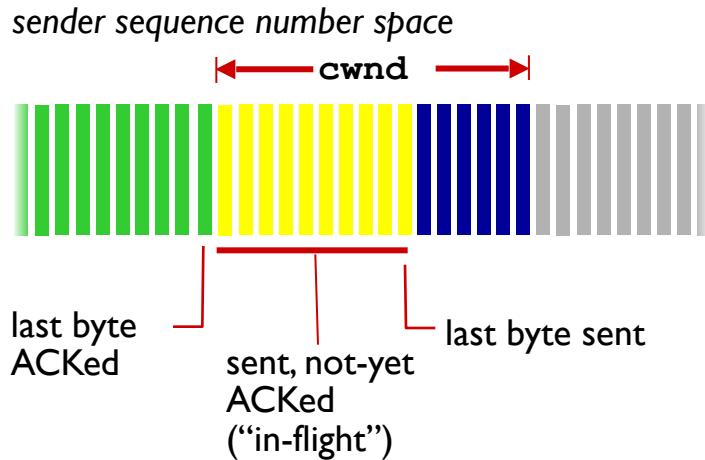
Causes/costs of congestion: scenario 3



another “cost” of congestion:

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

TCP Congestion Control: details



- sender limits transmission:

$$\frac{\text{LastByteSent} - \text{LastByteAcked}}{\text{cwnd}} \leq 1$$

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

TCP sending rate:

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

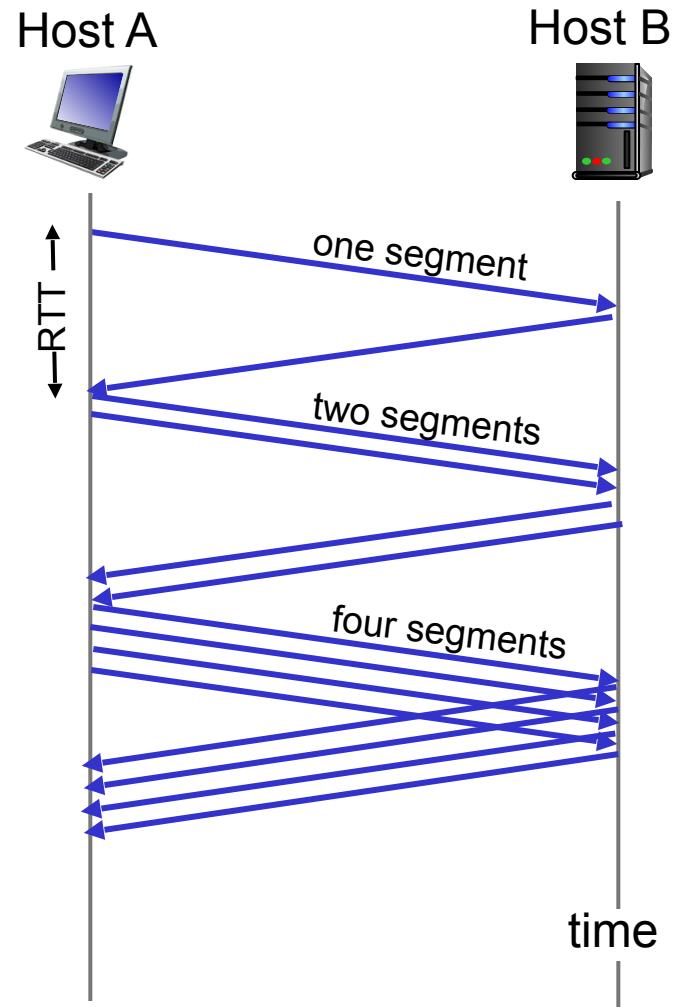
$$\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

TCP Stages

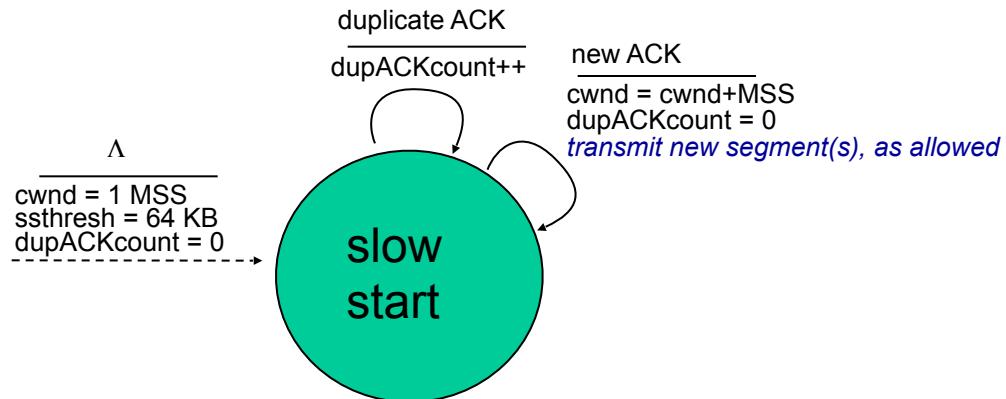
- slow start
- congestion avoidance
- fast recovery

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 Slow Start



TCP: detecting, reacting to loss

- on loss:
 - cwnd set to 1 MSS

Switching from Slow Start (exponential) to Congestion Avoidance (linear)

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

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

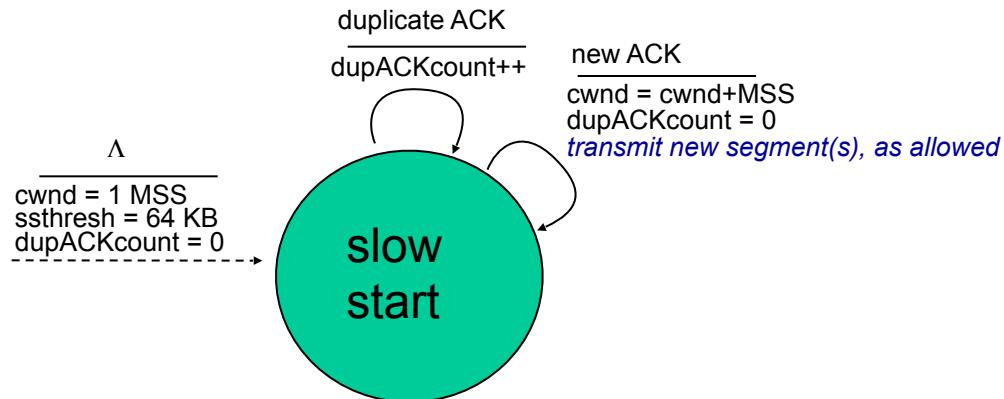
Implementation:

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

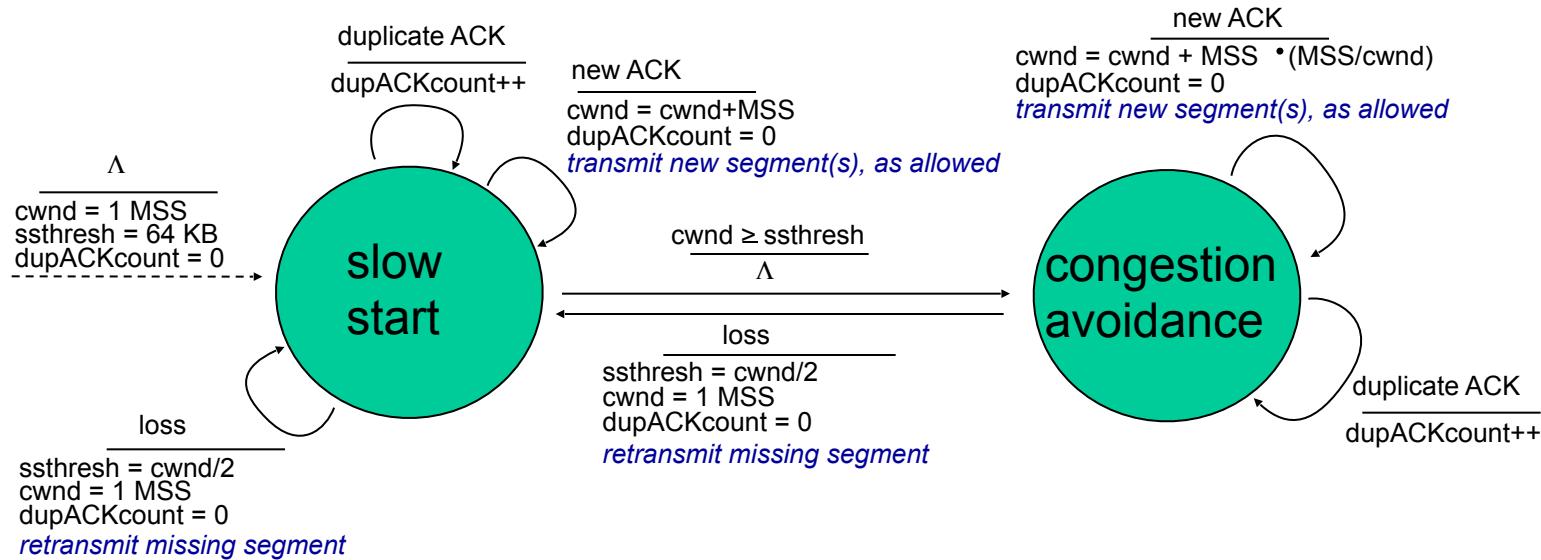


* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

TCP Loss Reaction & Congestion Avoidance



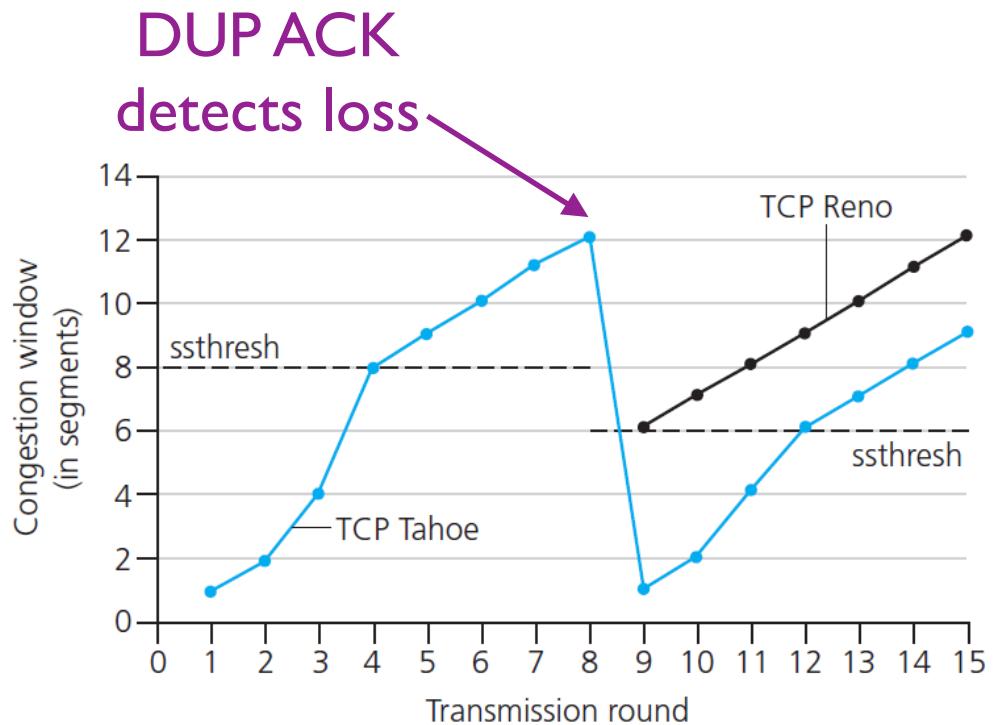
TCP Loss Reaction & Congestion Avoidance



TCP: detecting, reacting to loss

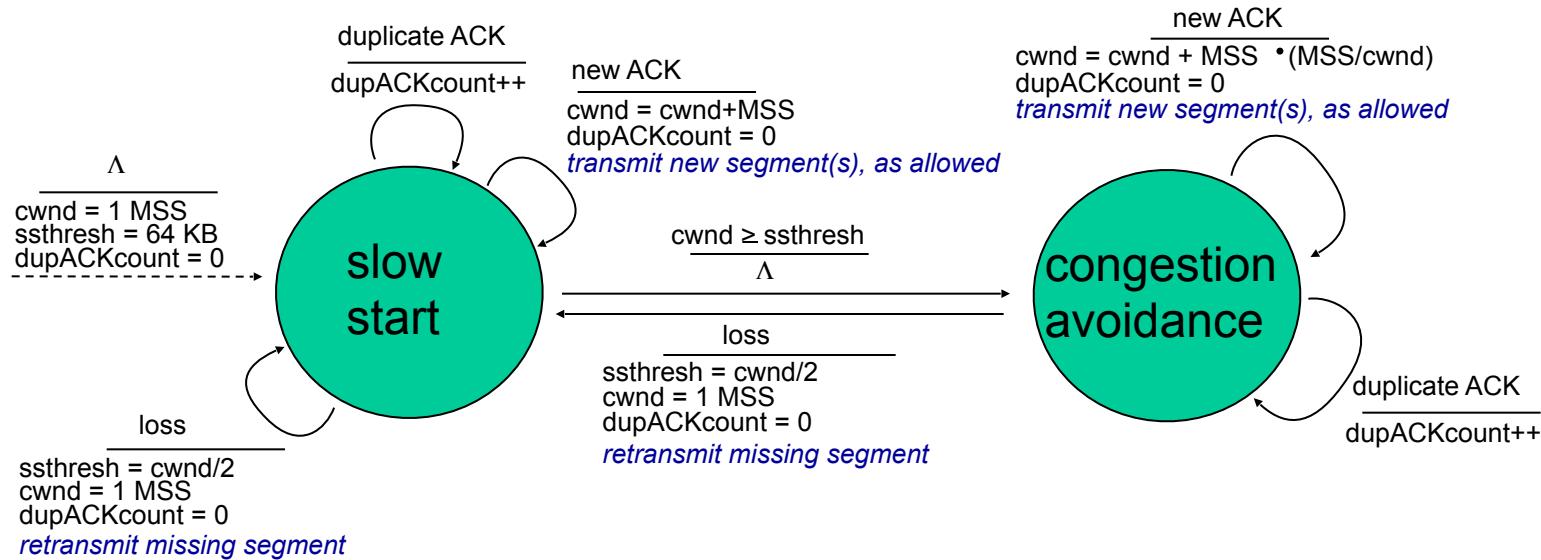
- on loss indicated by timeout:
 - **ssthresh** set to half of **cwnd**
 - **cwnd** set to 1 MSS
 - *slow start* (exponentially growth) up to **ssthresh**, then *congestion avoidance* (linear growth)
- loss indicated by 3 duplicate ACKs:
 - dup ACKs indicate network delivering some segments
 - TCP Tahoe: same as for timeouts above
 - TCP Reno's *fast recovery*:
 - **cwnd** is only cut in half
 - why 3 dup ACKs? Why not 1? Why not 5?

TCP Reno's Fast Recovery

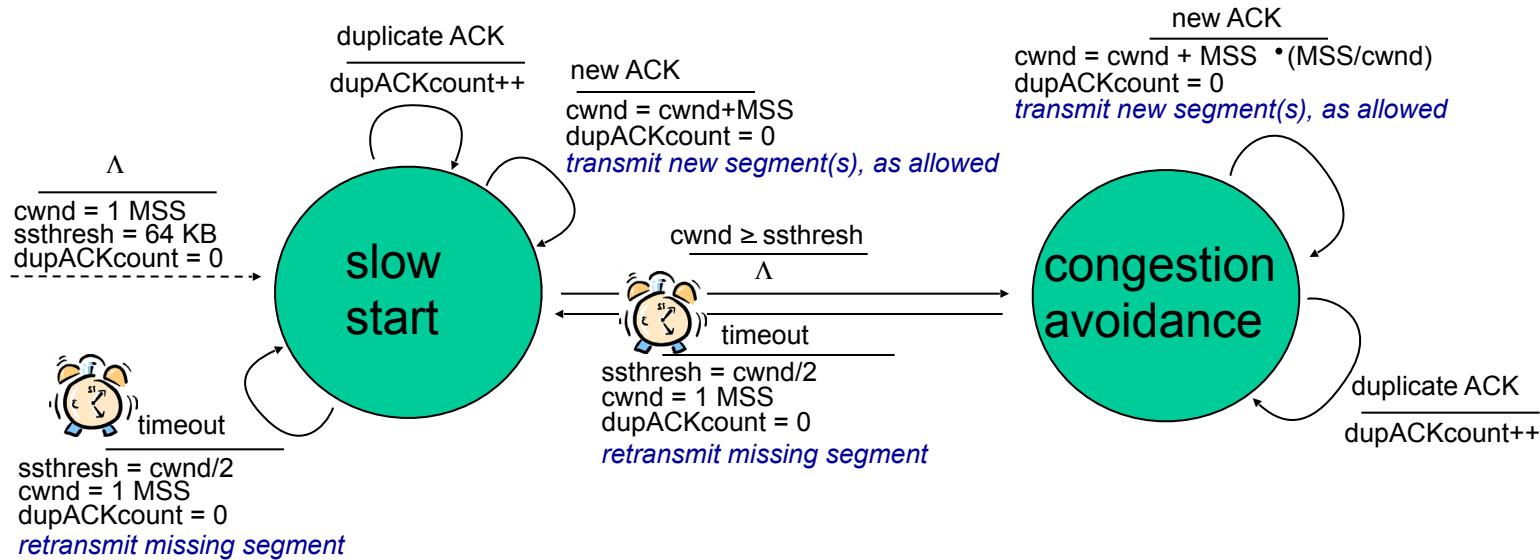


* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

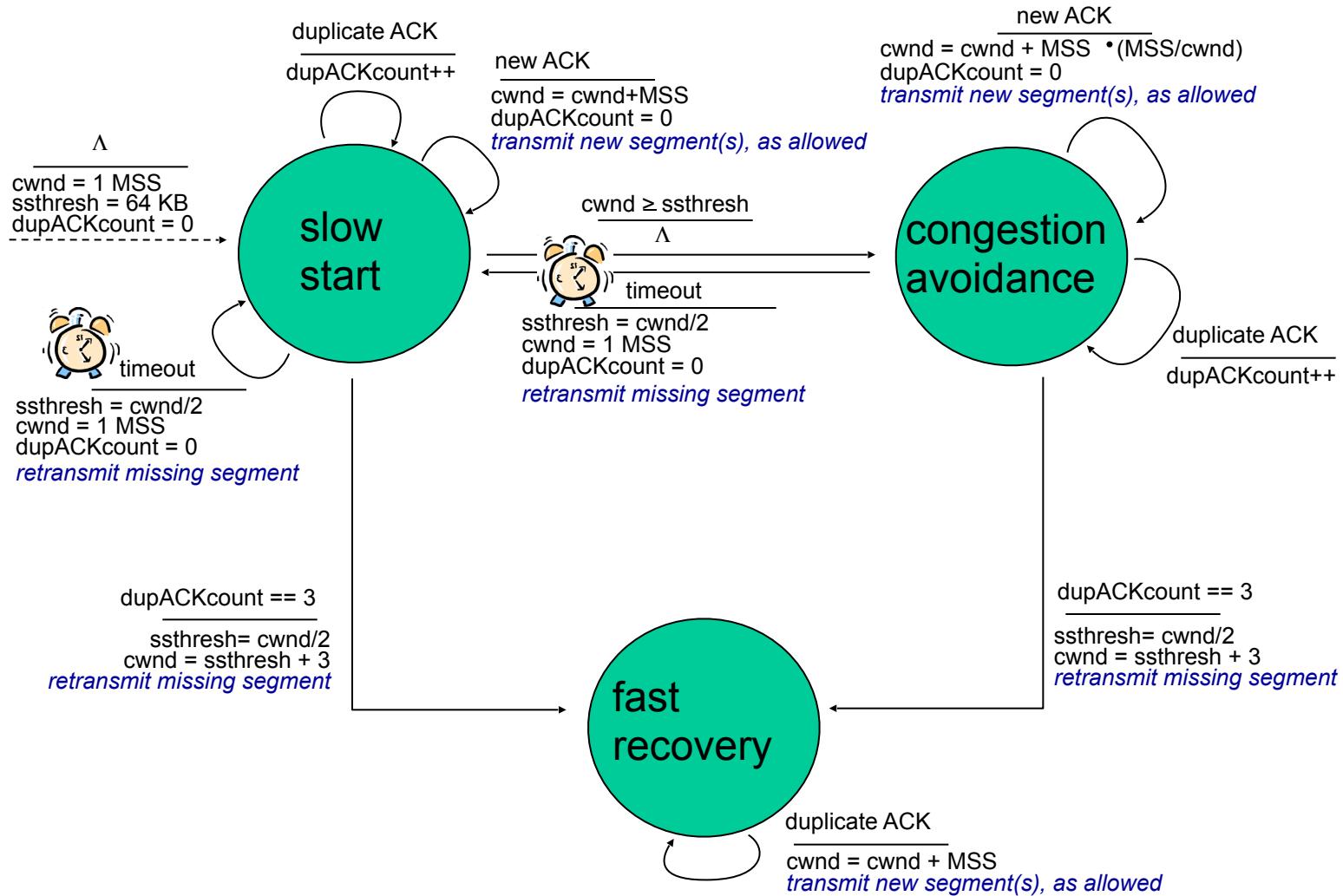
Summary: TCP Congestion Control



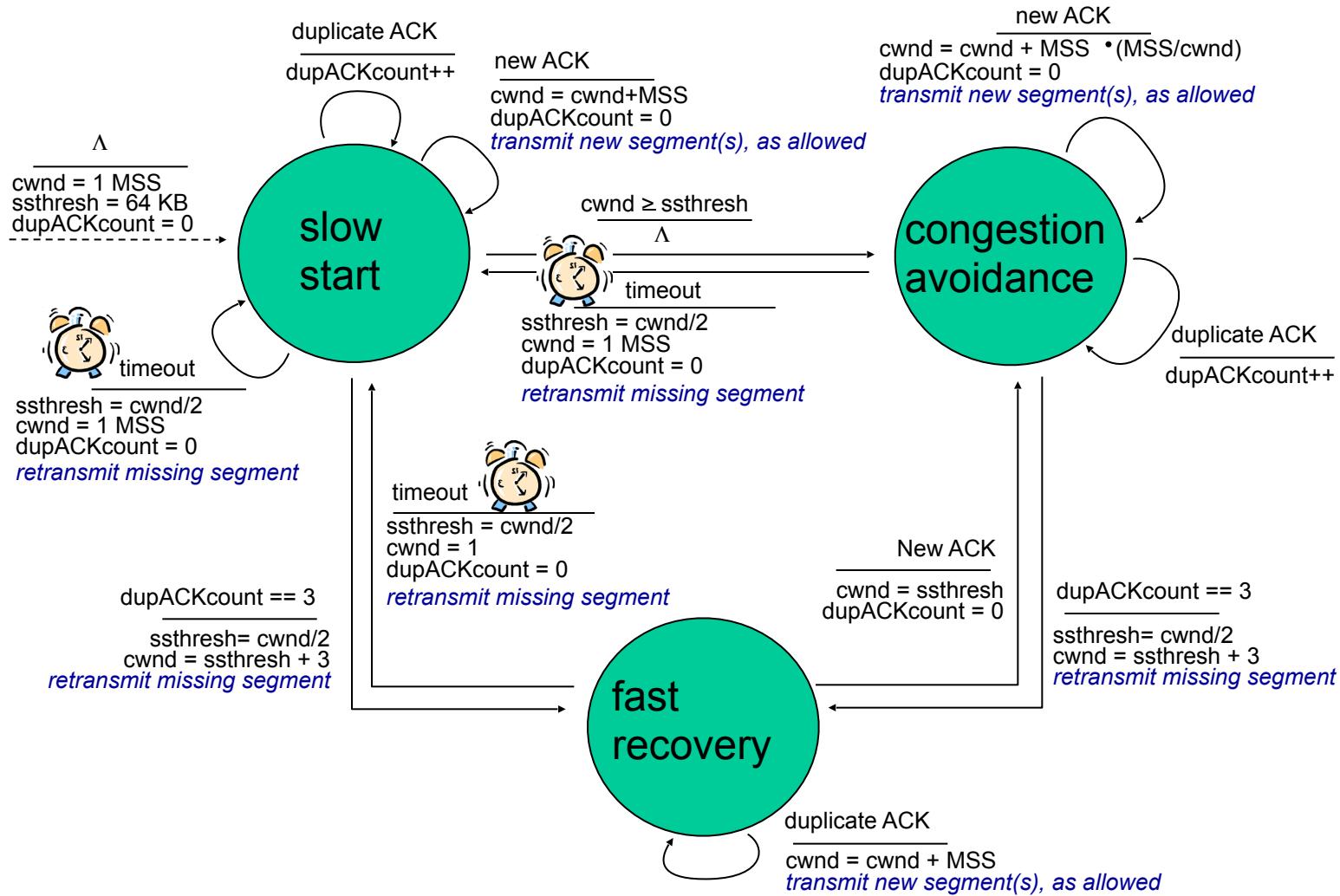
Summary: TCP Congestion Control



Summary: TCP Congestion Control

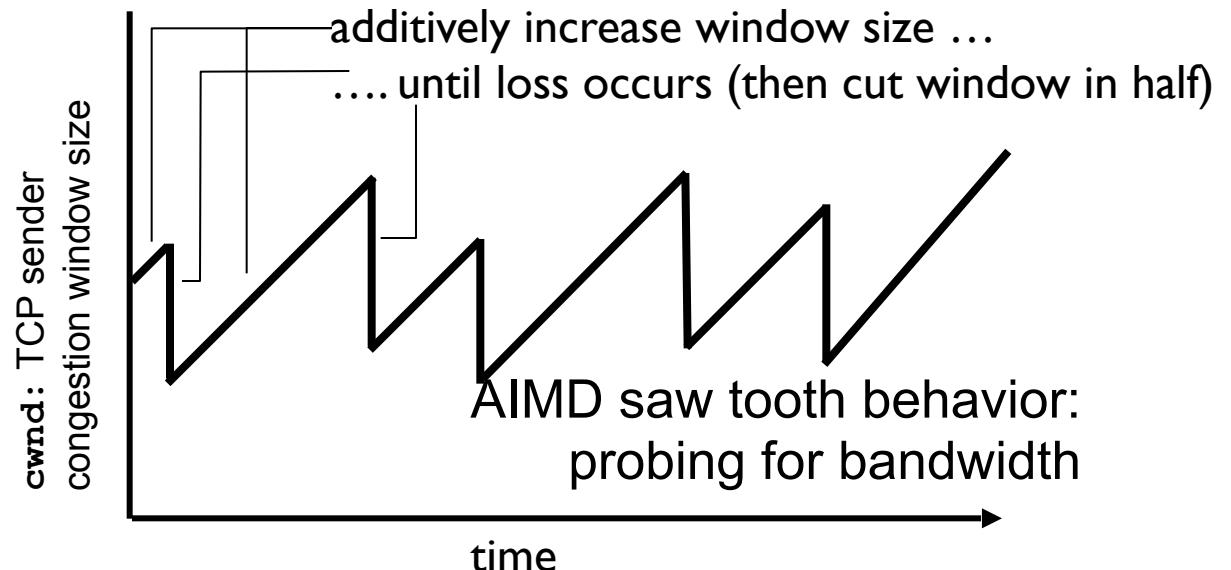


Summary: TCP Congestion Control



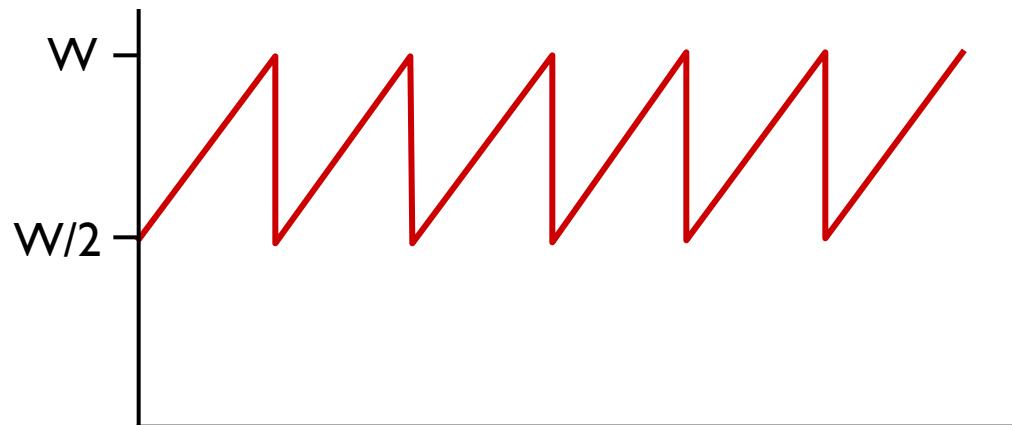
TCP congestion control: additive increase multiplicative decrease

- **approach:** sender increases 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



TCP throughput

- avg TCP throughput as function of window size, RTT?
 - ignore slow start, assume always data to send
- W : window size (measured in bytes) where loss occurs
 - window grows linearly to W , then drops to $W/2$
 - avg. window size (# in-flight bytes) is $\frac{3}{4}W$
 - avg. throughput is $\frac{3}{4}W$ per RTT



TCP Futures: TCP over “long, fat pipes”

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

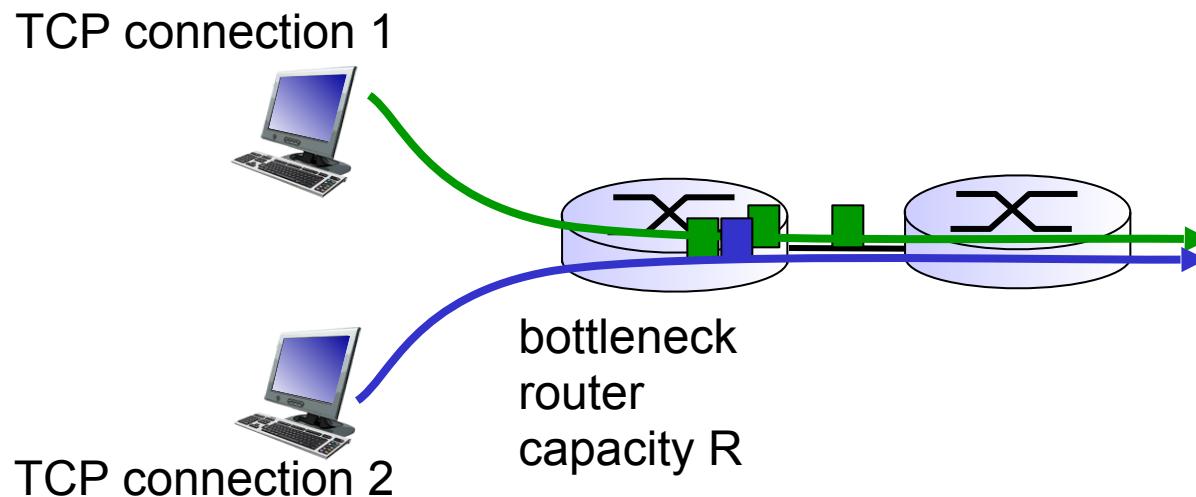
$$\text{TCP throughput} = \frac{1.22 \cdot \text{MSS}}{\text{RTT} \sqrt{L}}$$

→ 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

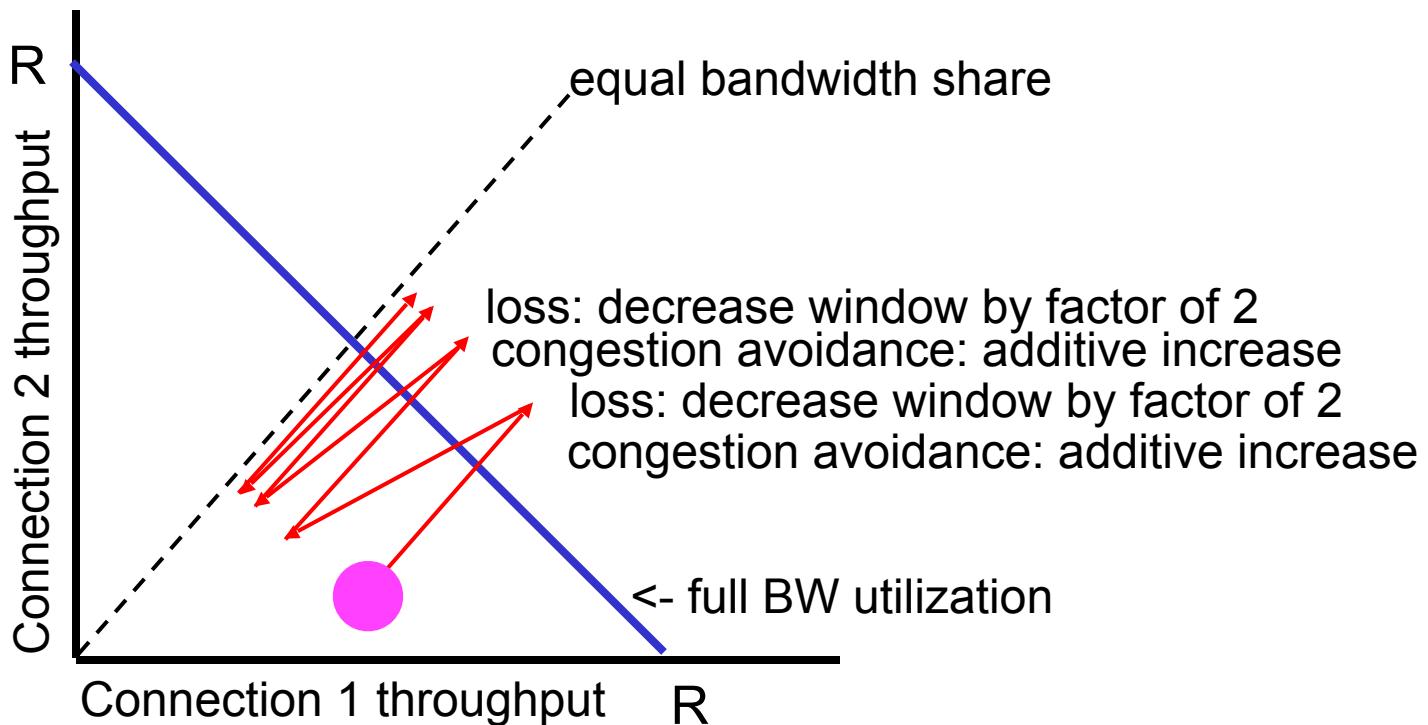
fairness goal: if K TCP sessions share same bottleneck link of bandwidth R , each should have average rate of R/K



Why is TCP fair?

two competing sessions:

- additive increase gives slope of 1, as throughput 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

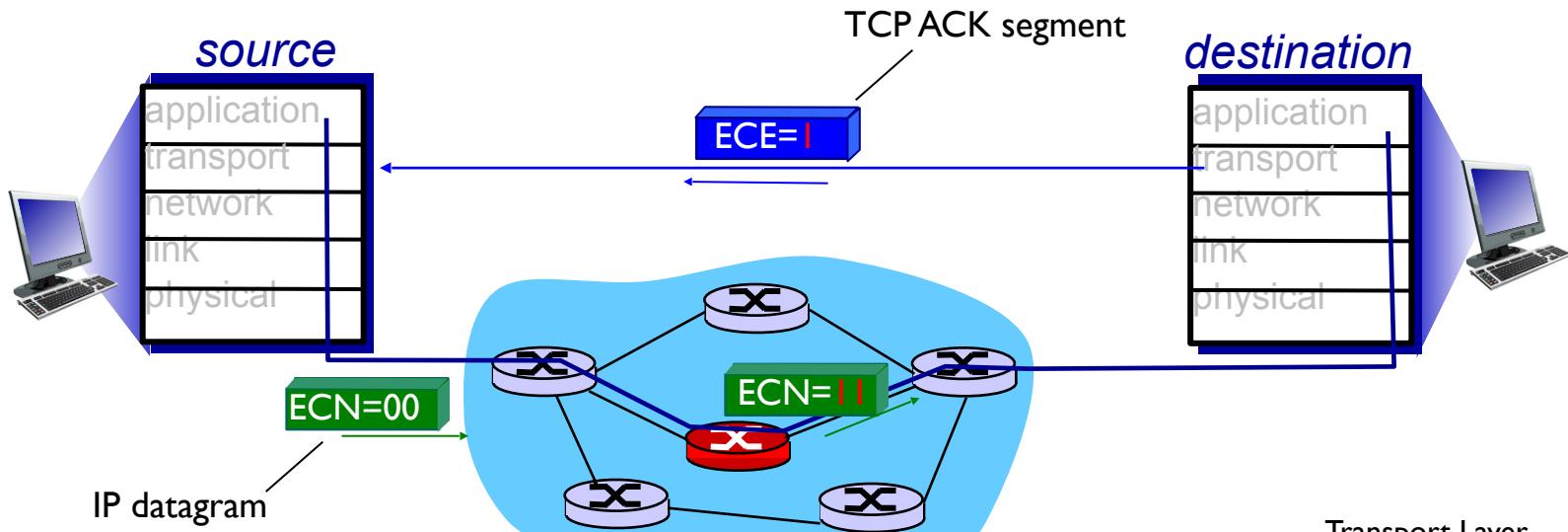
Fairness, parallel TCP connections

- 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 $R/2$

Explicit Congestion Notification (ECN)

network-assisted congestion control:

- two bits in IP header (ToS field) marked *by network router* to indicate congestion
- congestion indication carried to receiving host
- receiver (seeing congestion indication in IP datagram)) sets ECE bit in TCP header of receiver-to-sender ACK segment to notify sender of congestion
- Why does it need to modify both IP header *and* TCP header?



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

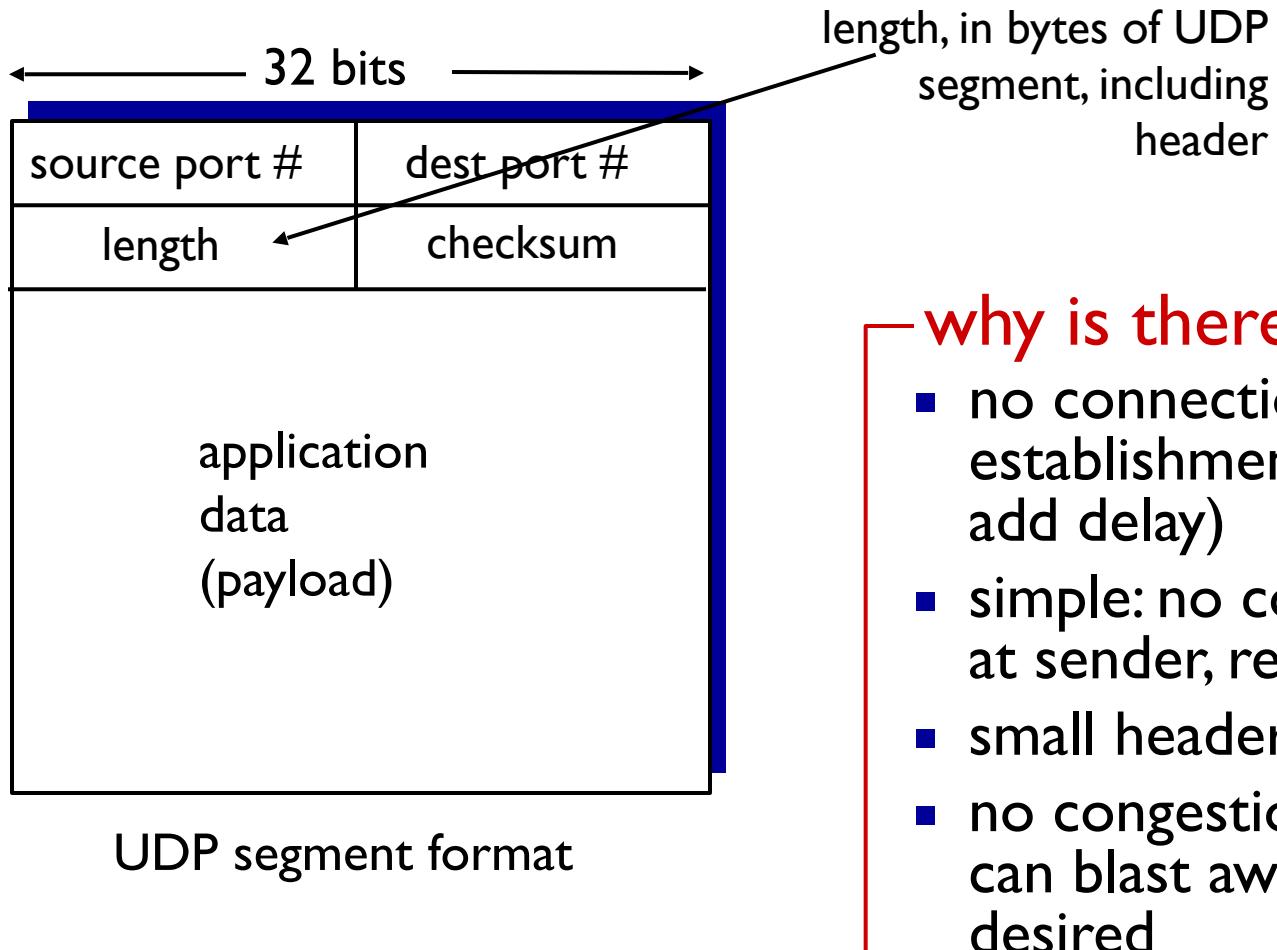
3.6 principles of congestion control

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 uses include:
 - streaming multimedia apps (loss tolerant, rate sensitive)
 - DNS
 - SNMP
- reliable transfer over UDP:
 - add reliability at application layer
 - application-specific error recovery!
 - e.g., QUIC from HW

UDP: segment 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

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
	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
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	0	1	1

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

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

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”
- two network layer chapters:
 - data plane
 - control plane



**DO NOT SHARE
SLIDES AND CLASS MATERIALS
ON ONLINE SITES**

Course Hero

Uploading course materials to sites such as CourseHero, Chegg or Github is academic misconduct at Columbia (see [pg 10](#) of [Columbia guide](#)).