Computer Networks
The Transport Layer

Prof. Navrati Saxena

San Jose State University
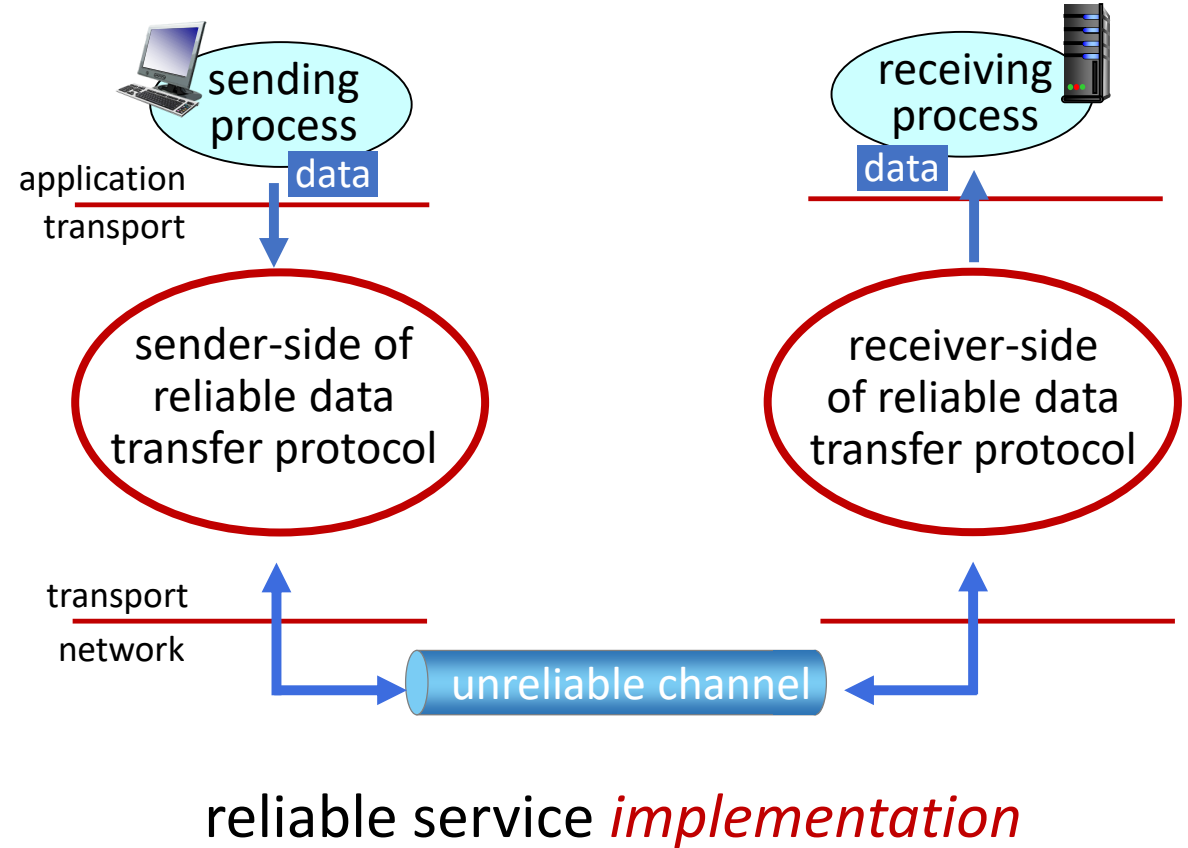
Chapter 3: roadmap

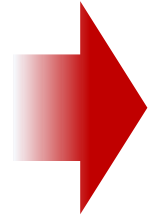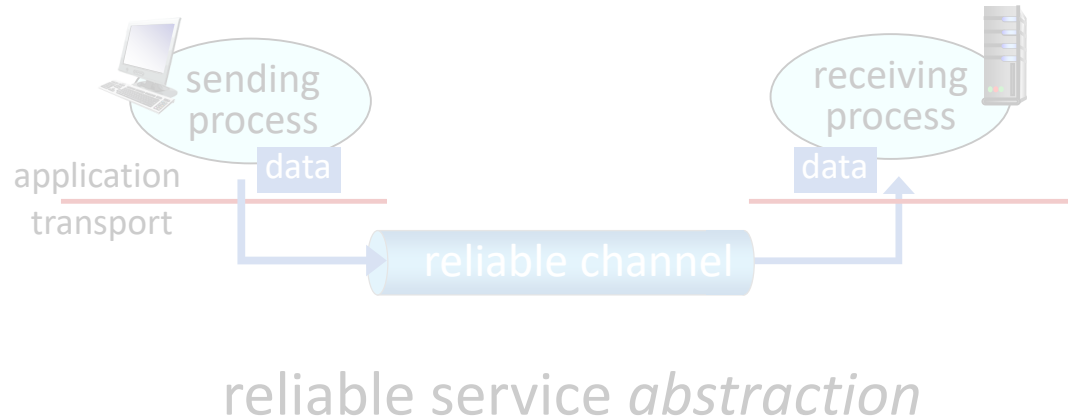# Principles of reliable data transfer

reliable service *abstraction*

Principles of reliable data transfer

reliable service *abstraction*

reliable service *implementation*
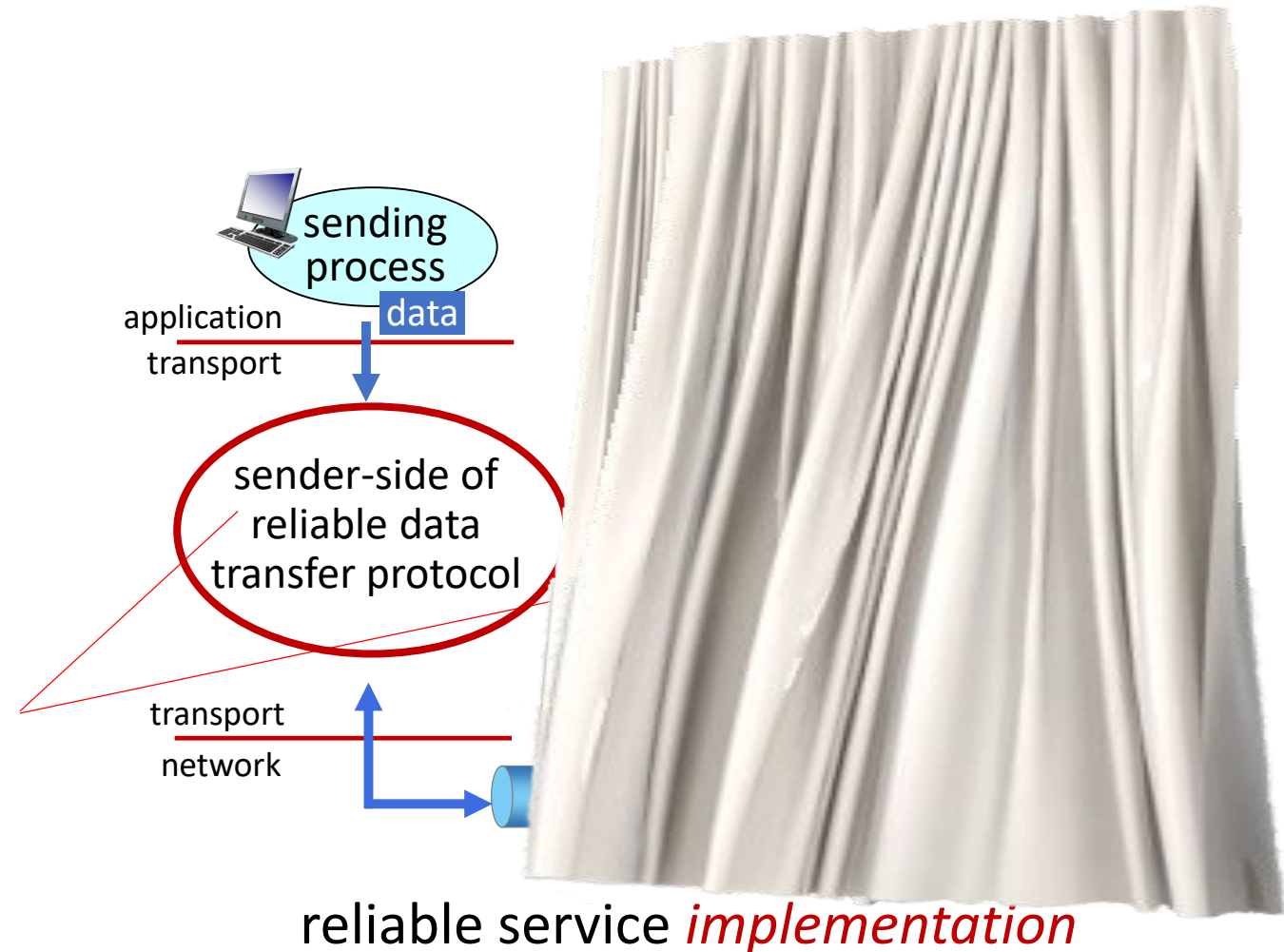
Principles of reliable data transfer

Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data?)
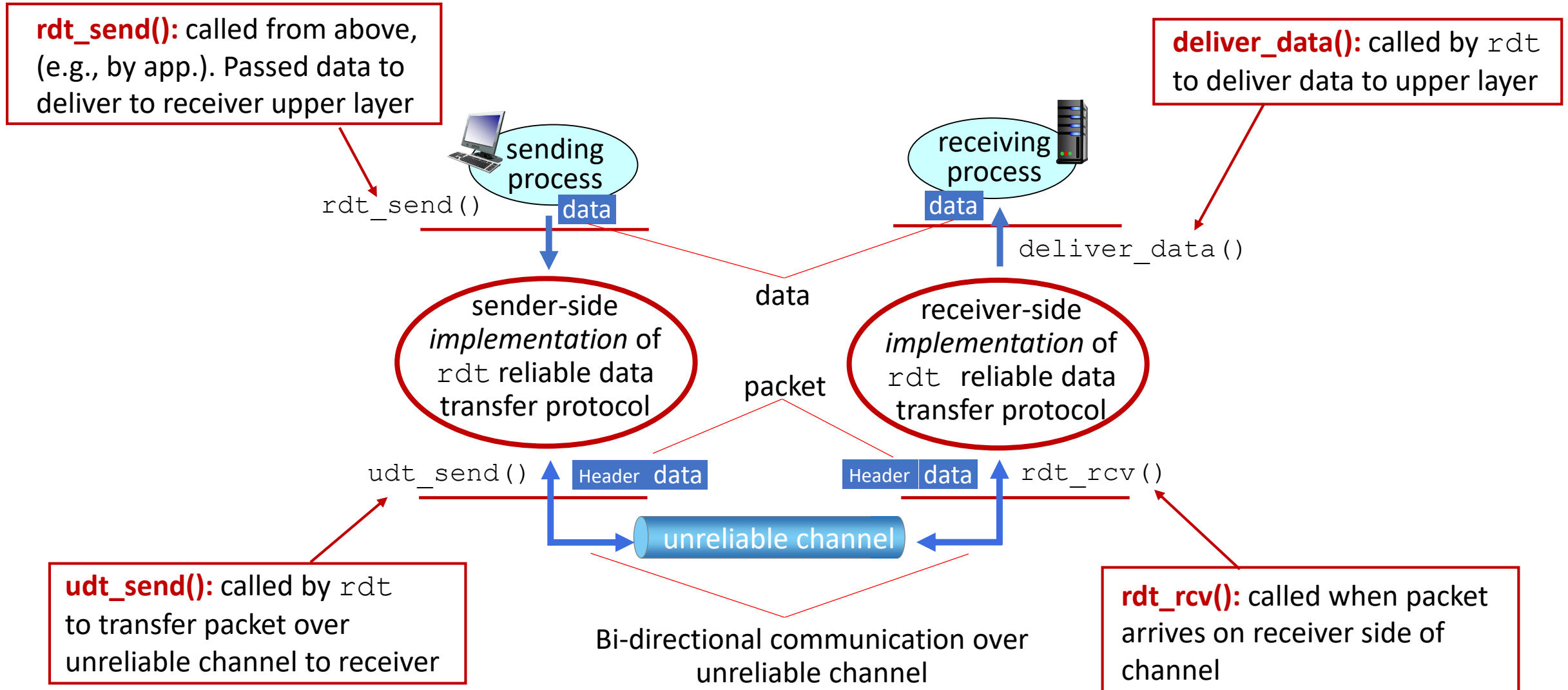

reliable service *implementation*
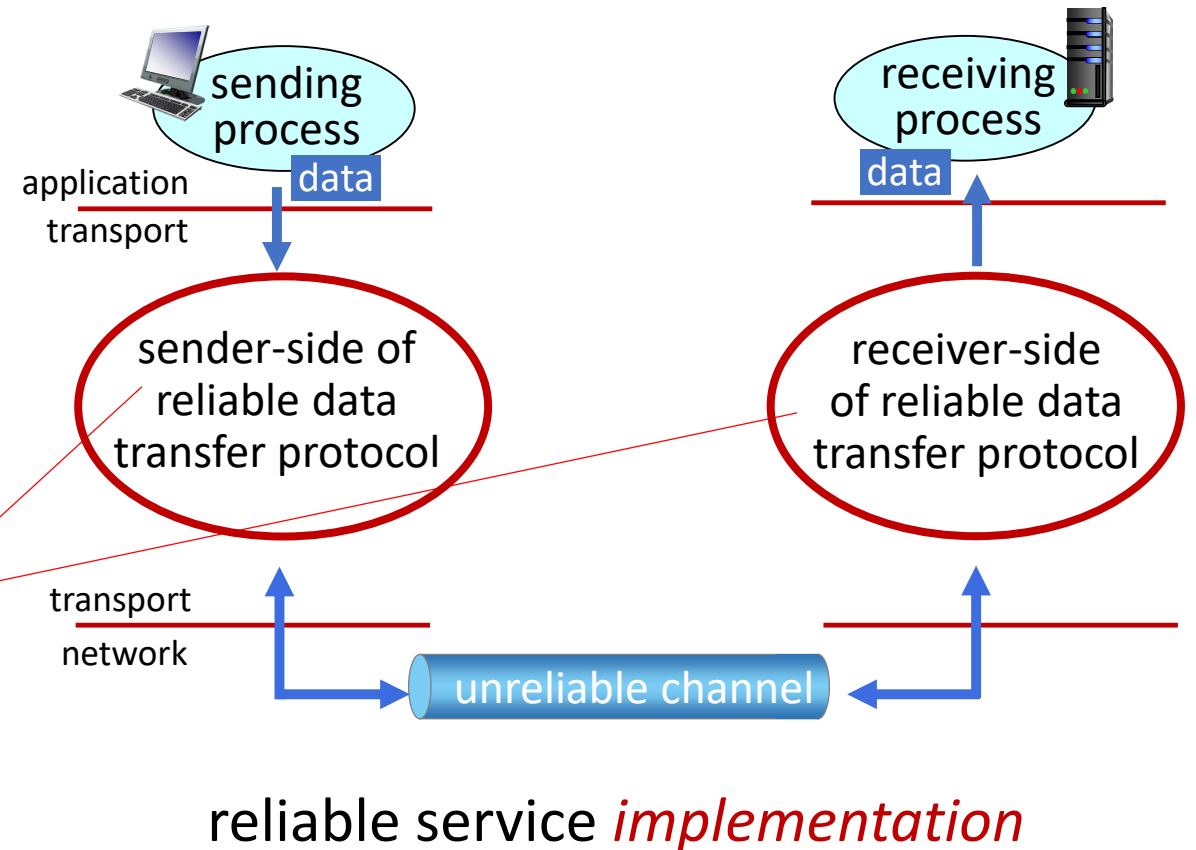
Principles of reliable data transfer

Sender, receiver do *not* know the "state" of each other, e.g., was a message received?
- unless communicated via a message

Reliable data transfer protocol (rdt): interfaces

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

deliver_data(): called by `rdt` to deliver data to upper layer

sending process

receiving process

`rdt_send()`

data

data

`deliver_data()`

data

sender-side *implementation* of `rdt` reliable data transfer protocol

receiver-side *implementation* of `rdt` reliable data transfer protocol

packet

`udt_send()`

Header data

Header data

`rdt_rcv()`

unreliable channel

udt_send(): called by `rdt` to transfer packet over unreliable channel to receiver

Bi-directional communication over unreliable channel

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

Reliable data transfer: getting started

We will:

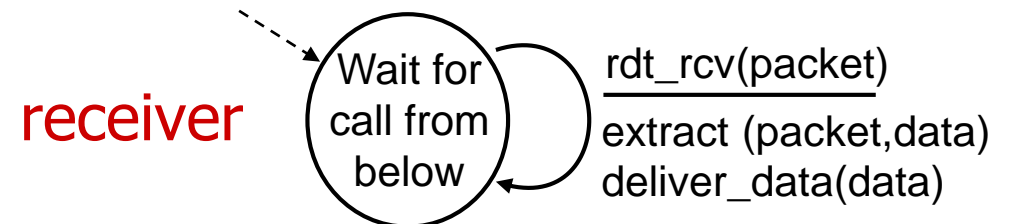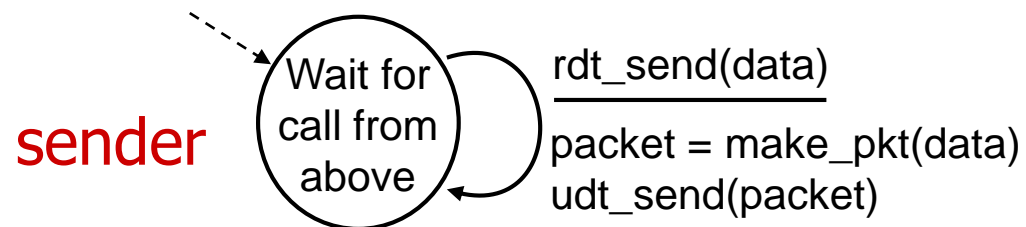- incrementally develop sender, receiver sides of reliable data transfer protocol (`rdt`)
- consider only unidirectional data transfer
  - but control info will flow in both directions!
- use finite state machines (FSM) to specify sender, receiver

state: when in this "state" next state uniquely determined by next event

event causing state transition
actions taken on state transition

event
actions

state 1

state 2
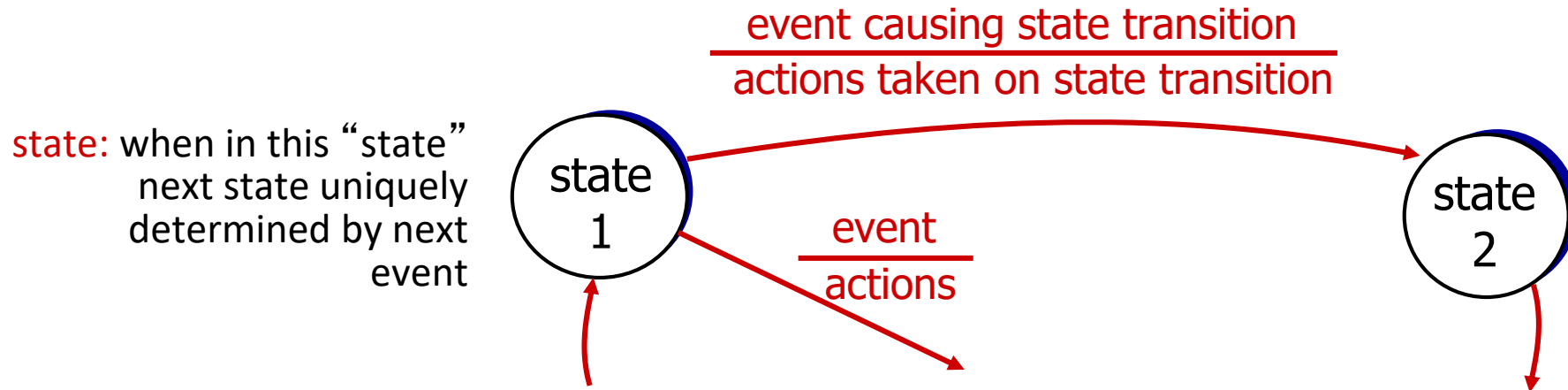
rdt1.0: reliable transfer over a reliable channel

- 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

sender

Wait for call from above

rdt_send(data)
_____
packet = make_pkt(data)
udt_send(packet)

receiver

Wait for call from below

rdt_rcv(packet)
_____
extract (packet,data)
deliver_data(data)

rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum (e.g., Internet checksum) to detect bit errors
- *the* question: how to recover from errors?

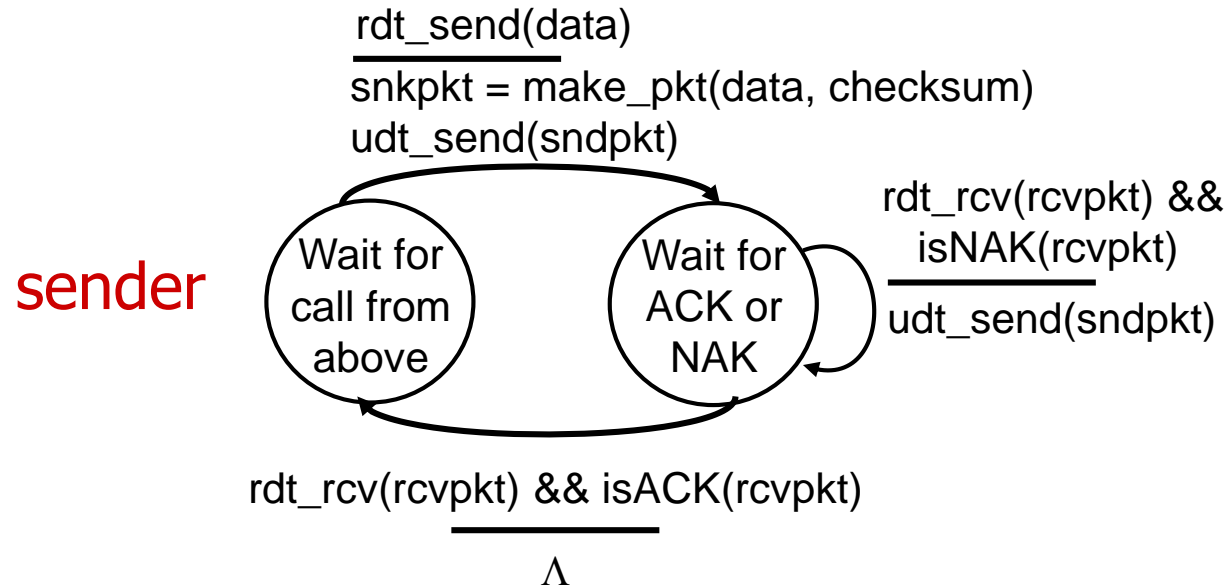*How do humans recover from "errors" during conversation?*

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

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

rdt2.0: FSM specifications

sender

rdt_send(data)
---
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) && isNAK(rcvpkt)
---
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)

Λ

rdt2.0: FSM specification

sender

rdt_send(data)
_____
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
_____
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
_____
Λ
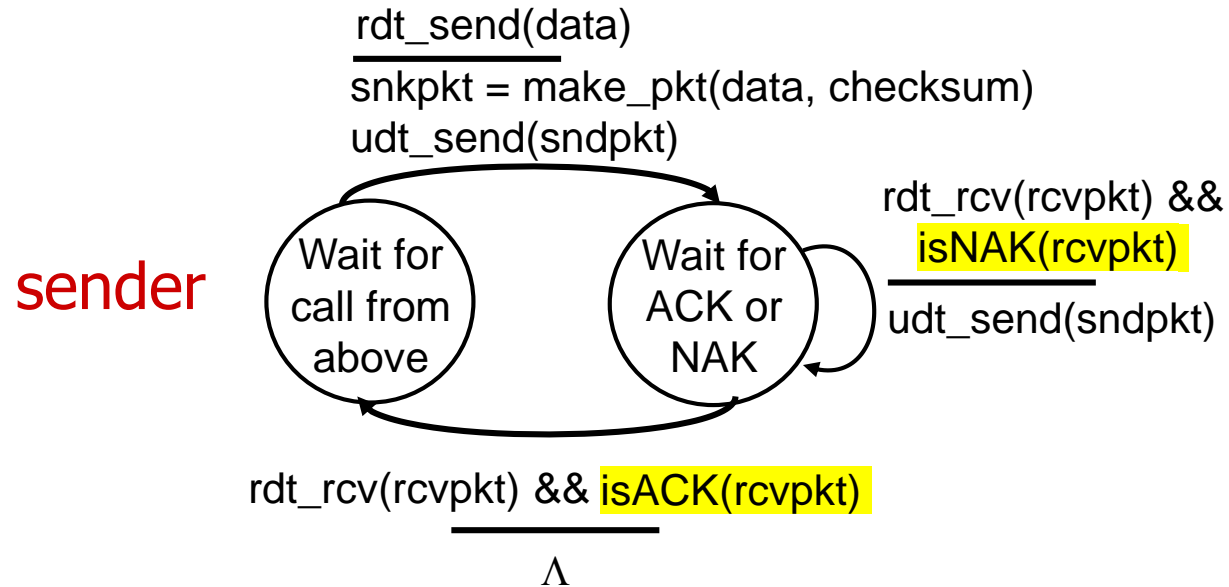
Note: "state" of receiver (did the receiver get my message correctly?) isn't known to sender unless somehow communicated from receiver to sender

- that's why we need a protocol!

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: 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*

As we will see, TCP uses this approach to be NAK-free

rdt3.0: channels with errors *and* loss

New channel assumption: underlying channel can also *lose* packets (data, ACKs)

- checksum, sequence #s, ACKs, retransmissions will be of help … but not quite enough

Q: How do *humans* handle lost sender-to-receiver words in conversation?

rdt3.0: channels with errors *and* loss

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 packet being ACKed
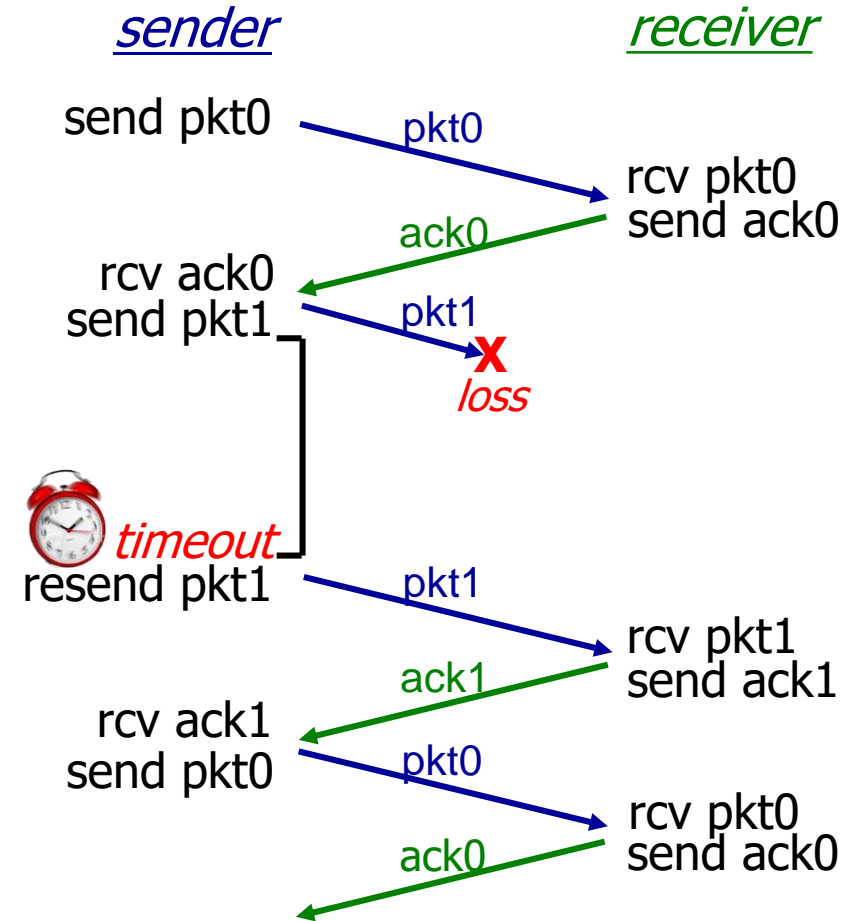- use countdown timer to interrupt after "reasonable" amount of time
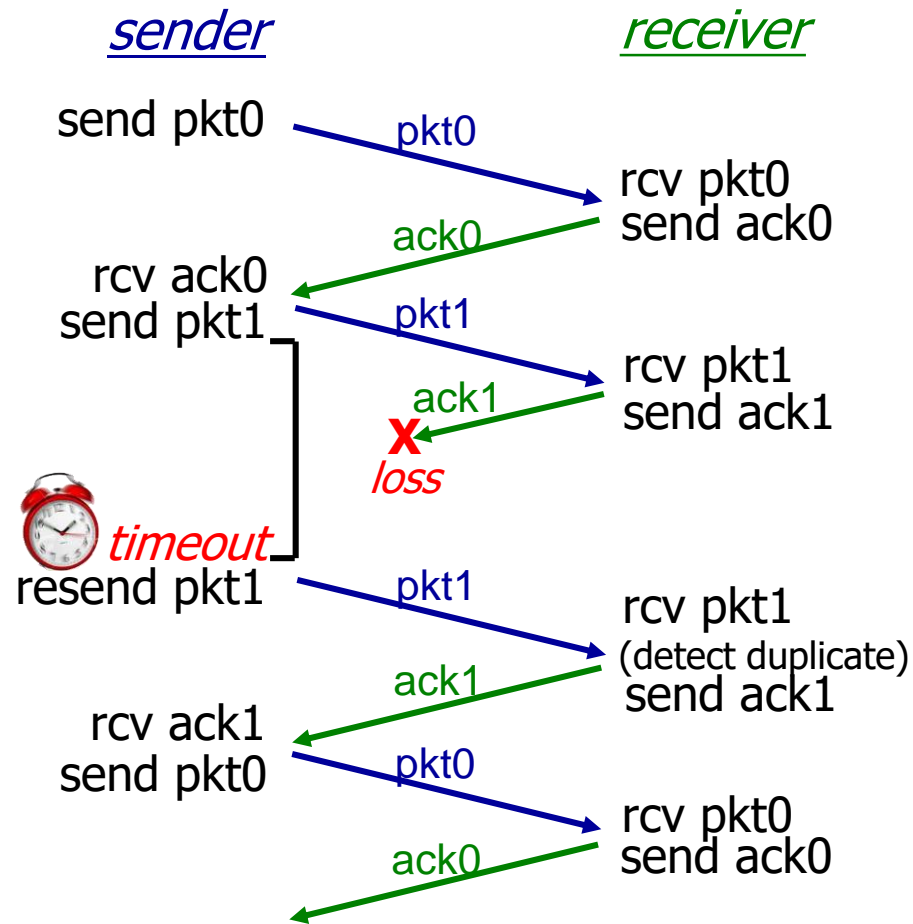
timeout

rdt3.0 in action
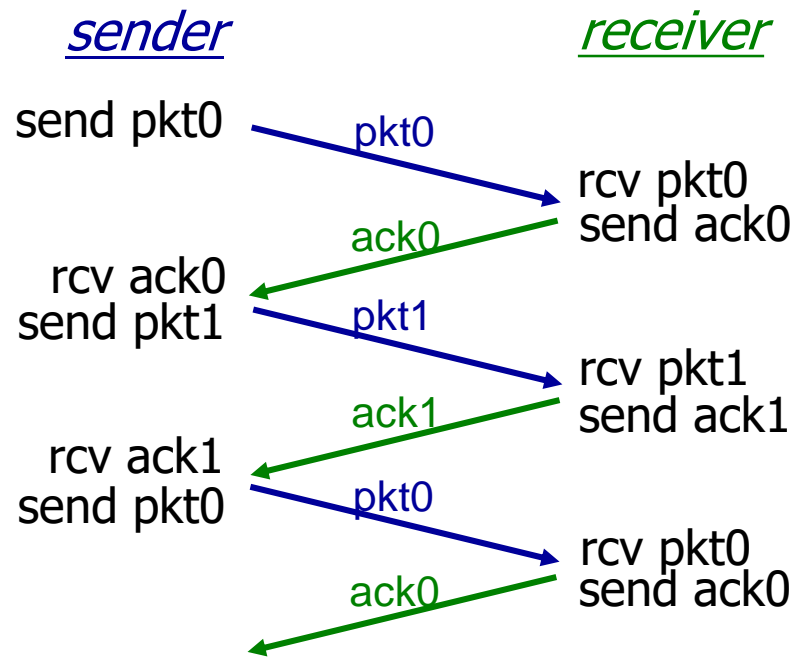
(a) no loss

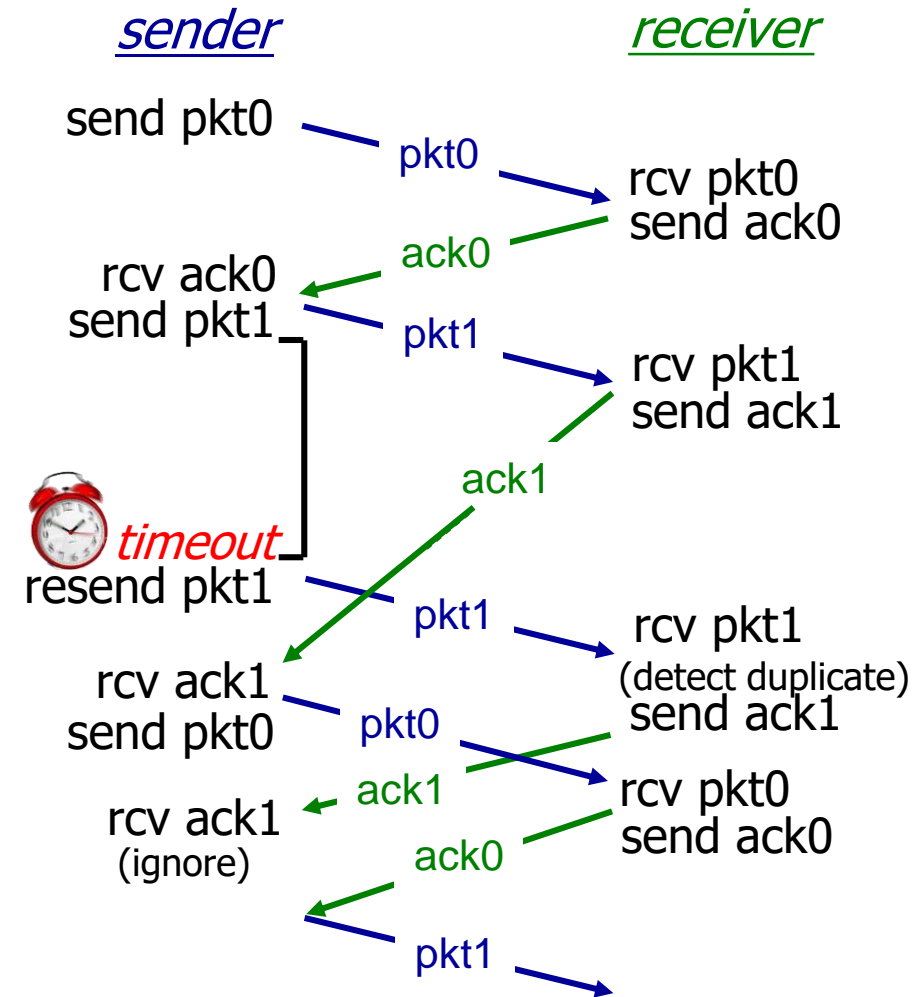(b) packet loss

rdt3.0 in action

(c) ACK loss

(d) premature timeout/ delayed ACK

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

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

- example: 1 Gbps link, 15 ms prop. delay, 8000 bit packet

  - time to transmit packet into channel:

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

rdt3.0: stop-and-wait operation

sender   receiver

first packet bit transmitted, t = 0

RTT

first packet bit arrives

last packet bit arrives, send ACK

ACK arrives, send next
packet, t = RTT + L / R
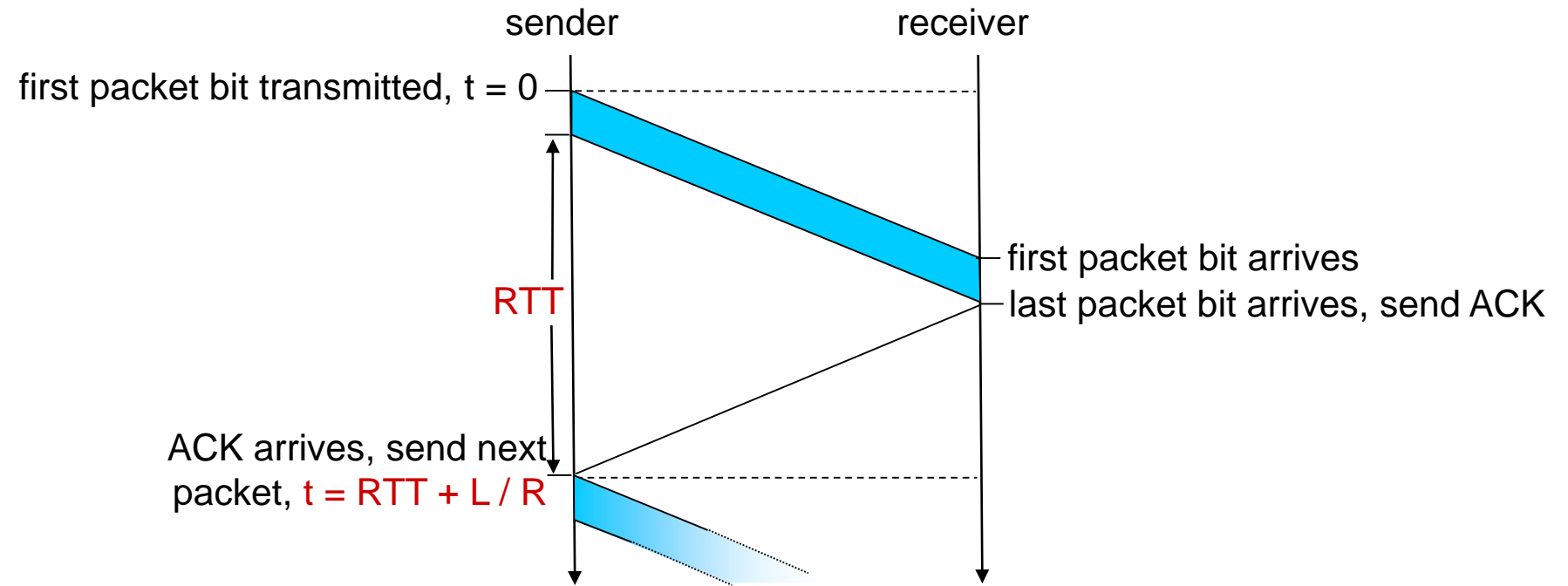
rdt3.0: stop-and-wait operation

$$U_{sender} = \frac{L / R}{RTT + L / R}$$

$$= \frac{.008}{30.008}$$

$$= 0.00027$$

sender   receiver

L/R

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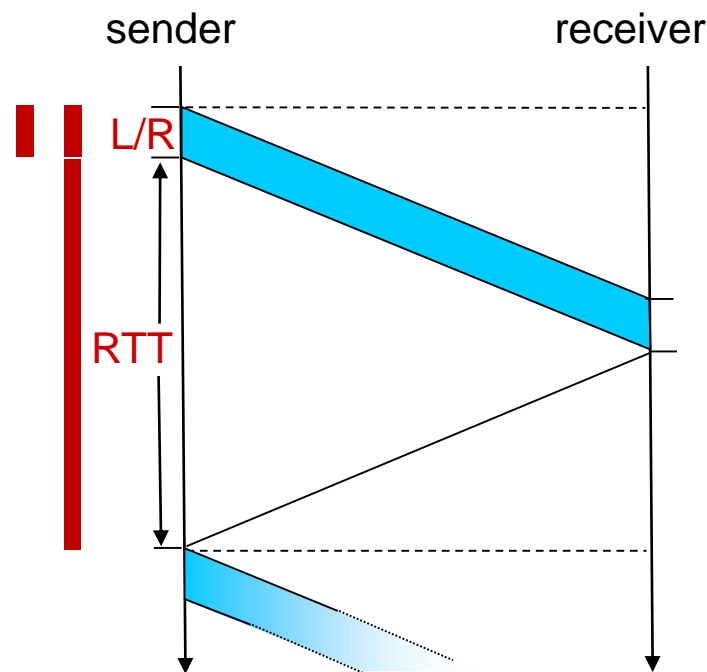
- rdt 3.0 protocol performance stinks!
- Protocol limits performance of underlying infrastructure (channel)

rdt3.0: pipelined protocols operation

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

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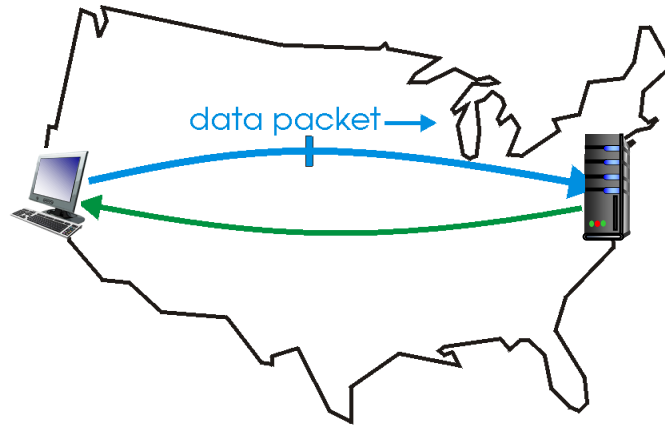- buffering at sender and/or receiver

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

Pipelining: increased utilization

sender   receiver

first packet bit transmitted, t = 0

last bit transmitted, t = L / R

RTT

first packet bit arrives

last packet bit arrives, send ACK

last bit of 2nd packet arrives, send ACK

last bit of 3rd packet arrives, send ACK

ACK arrives, send next
packet, t = RTT + L / R
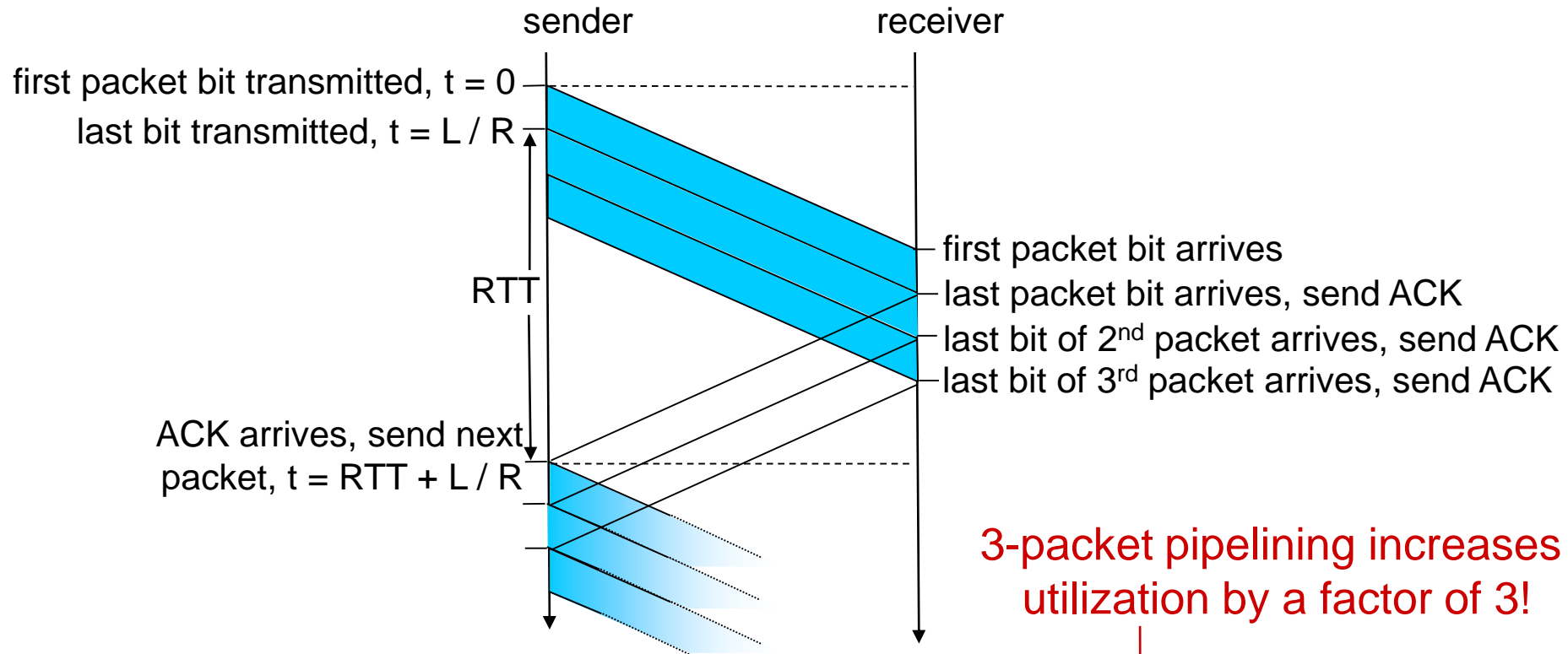
3-packet pipelining increases
utilization by a factor of 3!

$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

Go-Back-N: sender

- sender: "window" of up to N, consecutive transmitted but unACKed pkts
  - k-bit seq # in pkt header
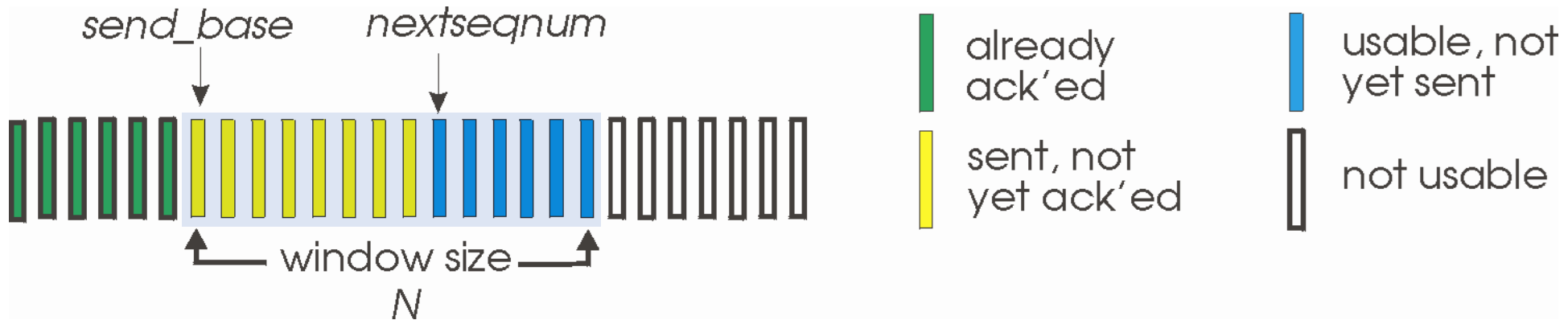
- *cumulative ACK:* ACK(*n*): ACKs all packets up to, including seq # *n*
  - on receiving ACK(*n*): move window forward to begin at *n+1*
- timer for oldest in-flight packet
- *timeout(n):* retransmit packet n and all higher seq # packets in window

Go-Back-N: receiver

- ACK-only: always send ACK for correctly-received packet so far, with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember `rcv_base`

- on receipt of out-of-order packet:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order seq #

Receiver view of sequence number space:

... rcv_base ...

received and ACKed

Out-of-order: received but not ACKed

Not received

Go-Back-N in action

sender window (N=4)   sender   receiver

0 1 2 3 4 5 6 7 8   send pkt0
0 1 2 3 4 5 6 7 8   send pkt1
0 1 2 3 4 5 6 7 8   send pkt2   receive pkt0, send ack0
0 1 2 3 4 5 6 7 8   send pkt3   **X** *loss* receive pkt1, send ack1
                        (wait)
                                                        receive pkt3, discard,
                                                              (re)send ack1
0 1 2 3 4 5 6 7 8   rcv ack0, send pkt4
0 1 2 3 4 5 6 7 8   rcv ack1, send pkt5   receive pkt4, discard,
                                                              (re)send ack1
                   ignore duplicate ACK   receive pkt5, discard,
                                                              (re)send ack1

                    pkt 2 timeout

0 1 2 3 4 5 6 7 8   send pkt2
0 1 2 3 4 5 6 7 8   send pkt3
0 1 2 3 4 5 6 7 8   send pkt4   rcv pkt2, deliver, send ack2
0 1 2 3 4 5 6 7 8   send pkt5   rcv pkt3, deliver, send ack3
                                                  rcv pkt4, deliver, send ack4
                                                  rcv pkt5, deliver, send ack5

Selective repeat

- receiver *individually* acknowledges all correctly received packets
  - buffers packets, as needed, for eventual in-order delivery to upper layer

- sender times-out/retransmits individually for unACKed packets
  - sender maintains timer for each unACKed pkt

- sender window
  - *N* consecutive seq #s
  - limits seq #s of sent, unACKed packets

Selective repeat: sender, receiver windows

(a) sender view of sequence numbers
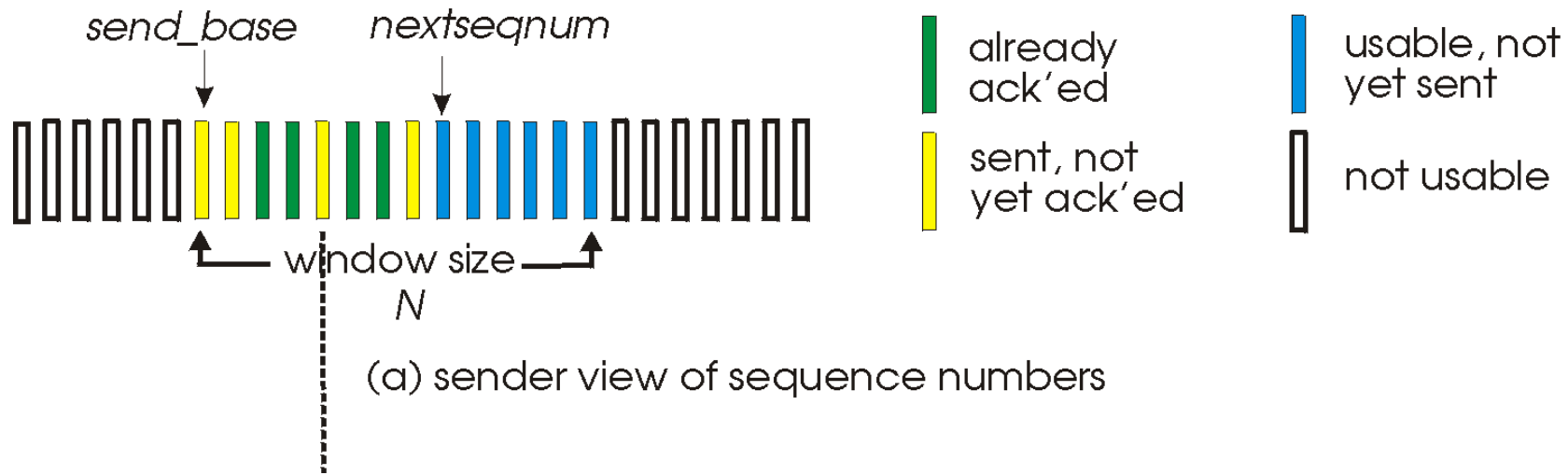
Selective repeat: sender and receiver

sender

data from above:

- if next available seq # in window, send packet

timeout(*n*):

- resend packet *n*, restart timer

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

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

receiver

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

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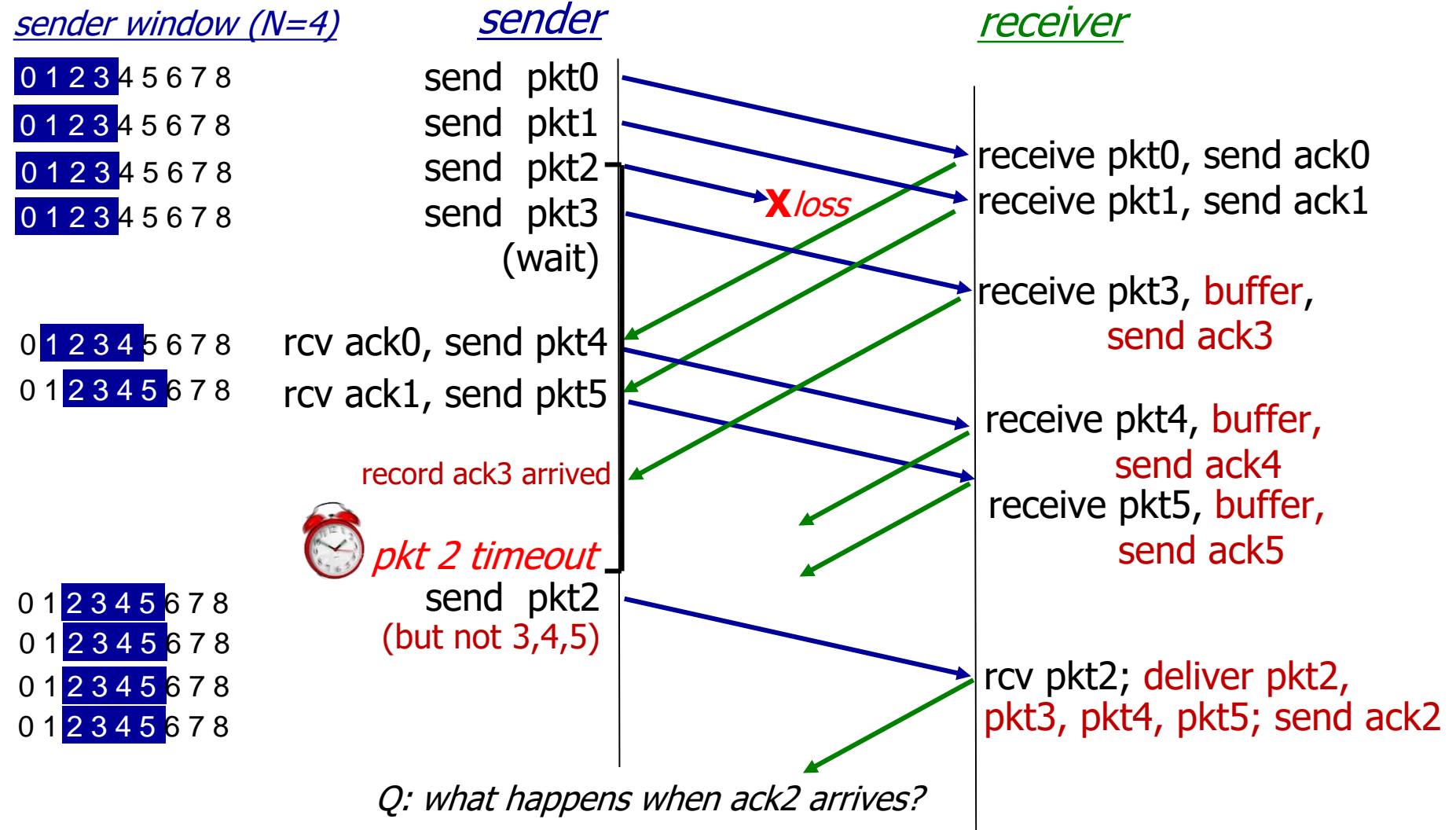- in-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet

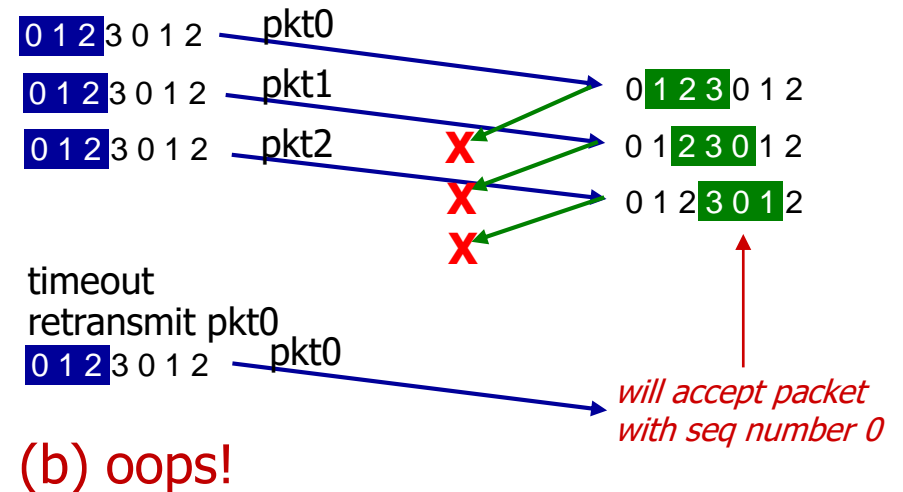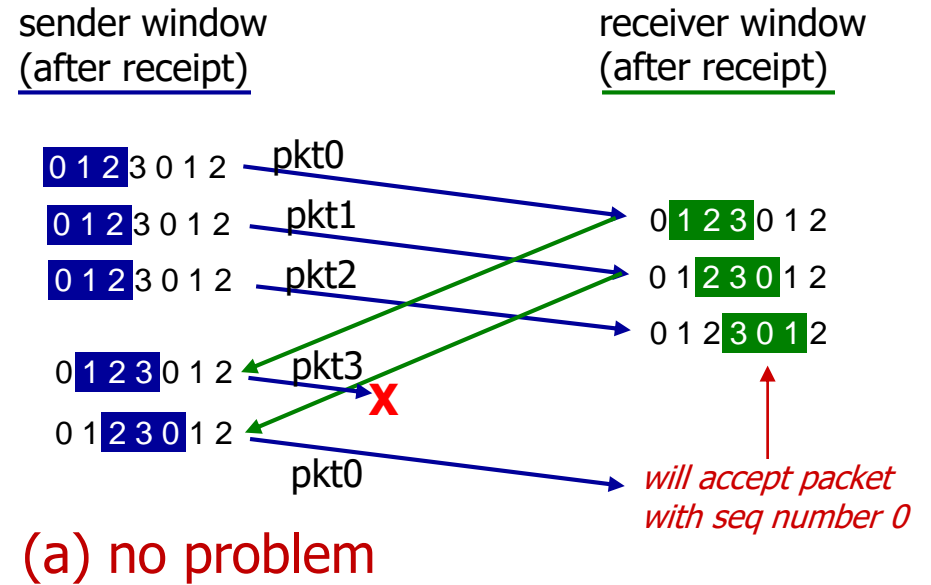### packet *n* in [rcvbase-N,rcvbase-1]

- ACK(*n*)

otherwise:

- ignore

Selective Repeat in action

sender window (N=4)   _sender_   _receiver_

0 1 2 3 4 5 6 7 8   send pkt0

0 1 2 3 4 5 6 7 8   send pkt1

0 1 2 3 4 5 6 7 8   send pkt2   receive pkt0, send ack0

0 1 2 3 4 5 6 7 8   send pkt3   **X** _loss_   receive pkt1, send ack1

                 (wait)   receive pkt3, buffer,
                                      send ack3

0 1 2 3 4 5 6 7 8   rcv ack0, send pkt4

0 1 2 3 4 5 6 7 8   rcv ack1, send pkt5   receive pkt4, buffer,
                                        send ack4

      record ack3 arrived   receive pkt5, buffer,
                                        send ack5

     pkt 2 timeout

0 1 2 3 4 5 6 7 8   send pkt2

0 1 2 3 4 5 6 7 8   (but not 3,4,5)

0 1 2 3 4 5 6 7 8   rcv pkt2; deliver pkt2,

0 1 2 3 4 5 6 7 8   pkt3, pkt4, pkt5; send ack2

Q: what happens when ack2 arrives?

Selective repeat:
a dilemma!

example:

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

sender window
(after receipt)

receiver window
(after receipt)

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

0 1 2 3 0 1 2   pkt3
0 1 2 3 0 1 2   X

pkt0

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

will accept packet
with seq number 0

(a) no problem

0 1 2 3 0 1 2   pkt0
0 1 2 3 0 1 2   pkt1
0 1 2 3 0 1 2   pkt2   X
                        X
                        X

timeout
retransmit pkt0
0 1 2 3 0 1 2   pkt0

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

will accept packet
with seq number 0

(b) oops!

Selective repeat: a dilemma!

example:

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

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

sender window
(after receipt)

receiver window
(after receipt)

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

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

will accept packet
with seq number 0
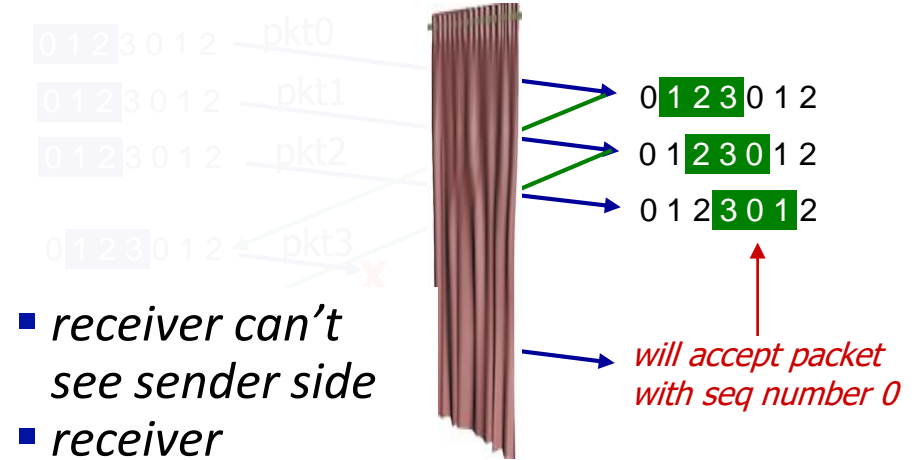
- *receiver can't see sender side*
- *receiver behavior identical in both cases!*
- *something's (very) wrong!*

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

0 1 2 3 0 1 2   pkt2

timeout
retransmit pkt0
0 1 2 3 0 1 2   pkt0

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

will accept packet
with seq number 0
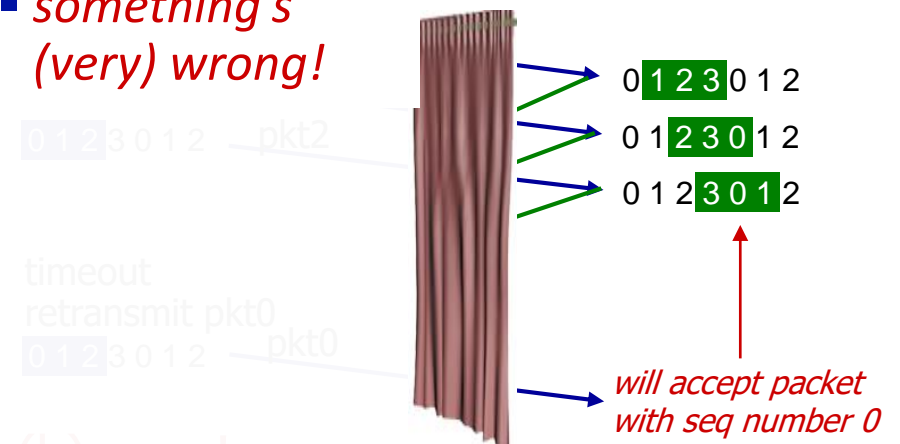
(b) oops!

Protecting against Wraparound

- Sequence Num: **32 bits long**

- Advertised Window: **16 bits long**

- Sliding window algorithm

  - Sequence number be twice as big as the window size

    - $2^{32} = 2 \times 2^{16}$

Chapter 3: roadmap

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

TCP: overview   RFCs: 793,1122, 2018, 5681, 7323

- **point-to-point:**
  - one sender, one receiver
- **reliable, in-order *byte steam:***
  - no "message boundaries"
- **full duplex data:**
  - bi-directional data flow in same connection
  - MSS: maximum segment size

- **cumulative ACKs**
- **pipelining:**
  - TCP congestion and flow control set window size
- **connection-oriented:**
  - handshaking (exchange of control messages) initializes sender, receiver state before data exchange
- **flow controlled:**
  - sender will not overwhelm receiver

TCP segment structure

32 bits

source port #	dest port #
sequence number	
acknowledgement number	
head len / not used / C E U A P R S F	receive window
checksum	Urg data pointer
options (variable length)	
application data (variable length)	

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

length (of TCP header)

Internet **checksum**

C, E: congestion notification

TCP **options**

RST, SYN, FIN: connection management

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

flow control: # bytes receiver willing to accept

data sent by application into TCP socket

TCP sequence numbers, ACKs

Sequence numbers:

- byte stream "number" of first byte in segment's data

Acknowledgements:

- seq # of next byte expected from other side

- cumulative ACK

Q: how receiver handles out-of-order segments

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

outgoing segment from sender

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

window size
N

sender sequence number space

sent
ACKed

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

usable
but not
yet sent

not
usable
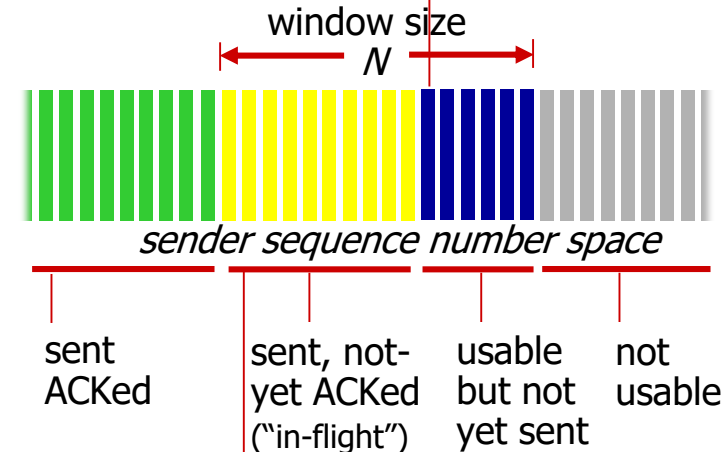
outgoing segment from receiver

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

TCP sequence numbers, ACKs

Host A

Host B

User types 'C'

Seq=42, ACK=79, data = 'C'

host ACKs receipt
of 'C', echoes back 'C'

Seq=79, ACK=43, data = 'C'

host ACKs receipt
of echoed 'C'

Seq=43, ACK=80
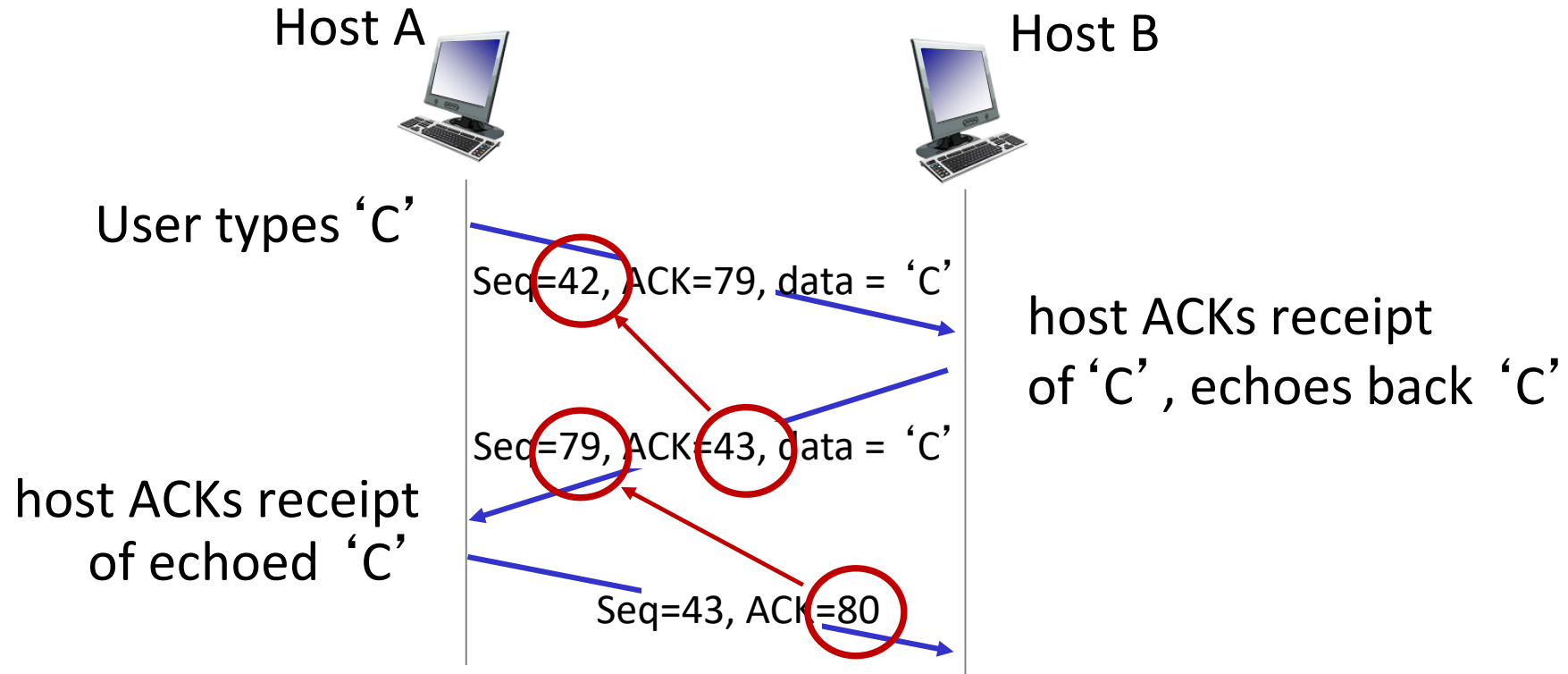
simple telnet scenario

TCP round trip time, timeout

Q: how to set TCP timeout value?

- longer than RTT, but RTT varies!
- *too short:* premature timeout, unnecessary retransmissions
- *too long:* slow reaction to segment loss

Q: how to estimate RTT?

- `SampleRTT:` measured time from segment transmission until ACK receipt
  - ignore retransmissions
- `SampleRTT` will vary, want estimated RTT "smoother"
  - average several *recent* measurements, not just current `SampleRTT`

TCP round trip time, timeout

$$\texttt{EstimatedRTT}_{(T)} = (1- \alpha)\texttt{*EstimatedRTT}_{(T-1)} + \alpha\texttt{*SampleRTT}_{(T)}$$

typical value: α = 0.125

- timeout interval: **EstimatedRTT** plus "safety margin"
  - large variation in **EstimatedRTT**: want a larger safety margin

- **DevRTT**: EWMA of **SampleRTT** deviation from **EstimatedRTT**:

$$\texttt{DevRTT = (1-}\beta\texttt{)*DevRTT + }\beta\texttt{*|SampleRTT-EstimatedRTT|}$$

(typically, β = 0.25)

Chapter 3: roadmap

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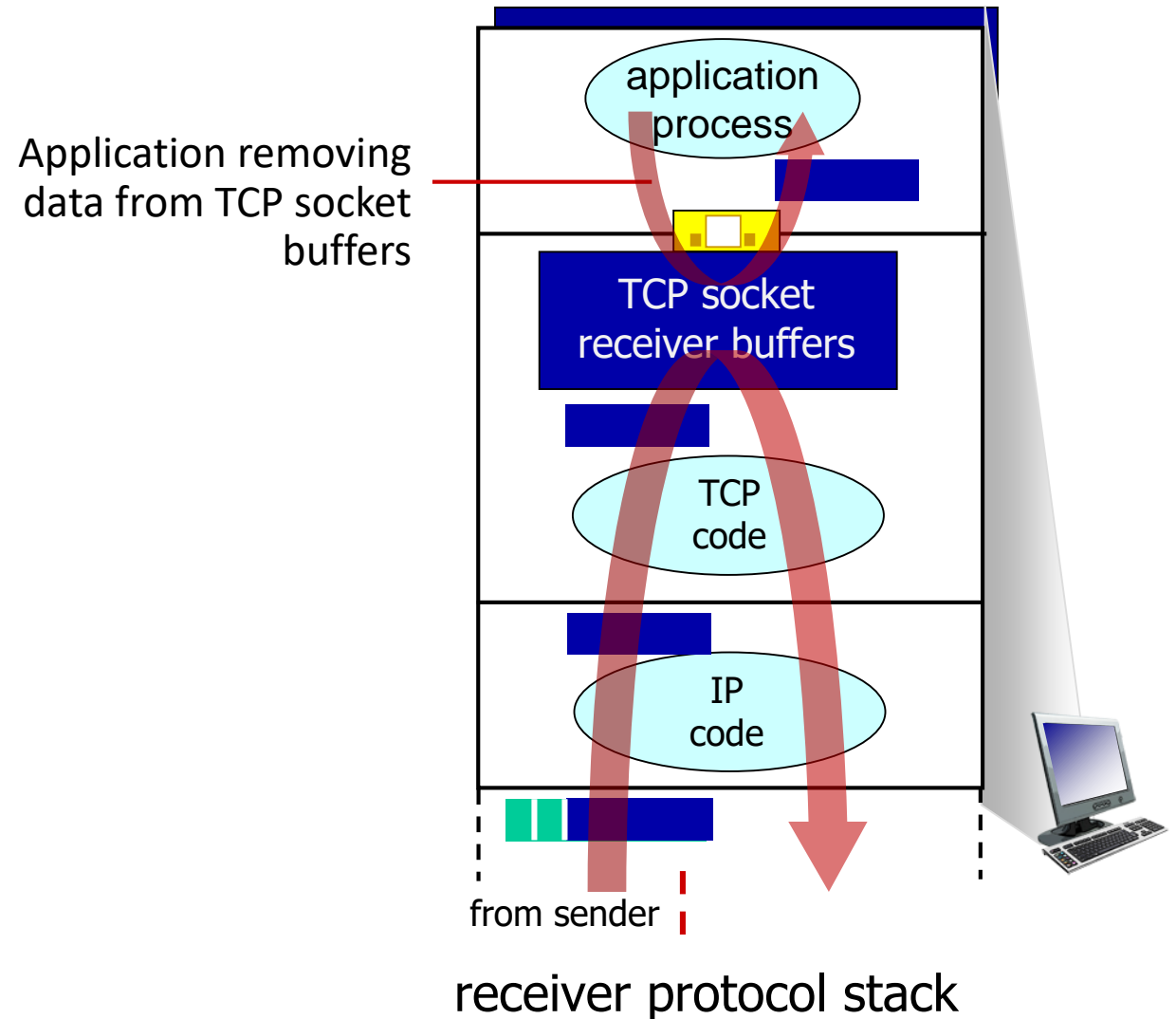- Principles of congestion control
- TCP congestion control

TCP flow control

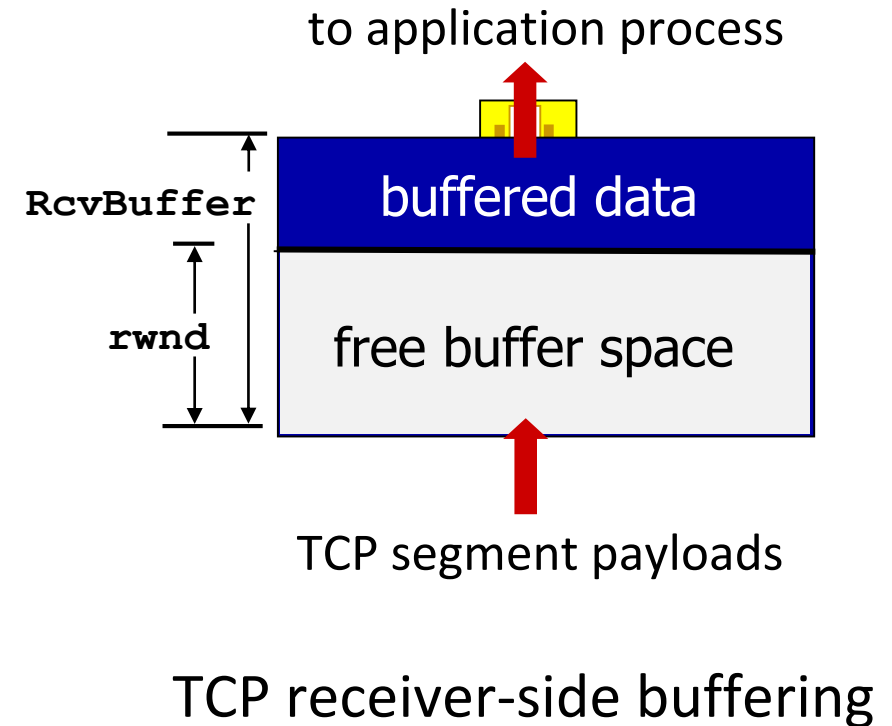*Q:* What happens if network layer delivers data faster than application layer removes data from socket buffers?

Application removing data from TCP socket buffers

application process

TCP socket receiver buffers

TCP code

Network layer delivering IP datagram payload into TCP socket buffers

IP code

from sender

receiver protocol stack

TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

Application removing data from TCP socket buffers

application process

TCP socket receiver buffers

TCP code

Network layer delivering IP datagram payload into TCP socket buffers

IP code

from sender

receiver protocol stack

TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

Application removing data from TCP socket buffers

application process

TCP socket receiver buffers

TCP code

IP code

receive window

flow control: # bytes receiver willing to accept

from sender

receiver protocol stack

TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

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

Application removing data from TCP socket buffers

application process

TCP socket receiver buffers

TCP code

IP code

from sender

receiver protocol stack

TCP flow control

- TCP receiver "advertises" free buffer space in `rwnd` field in TCP header
  - `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 received `rwnd`

- guarantees receive buffer will not overflow

to application process

RcvBuffer

rwnd

buffered data

free buffer space

TCP segment payloads

TCP receiver-side buffering

TCP flow control

- TCP receiver "advertises" free buffer space in **rwnd** field in TCP header
  - **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 received **rwnd**

- guarantees receive buffer will not overflow

flow control: # bytes receiver willing to accept

receive window
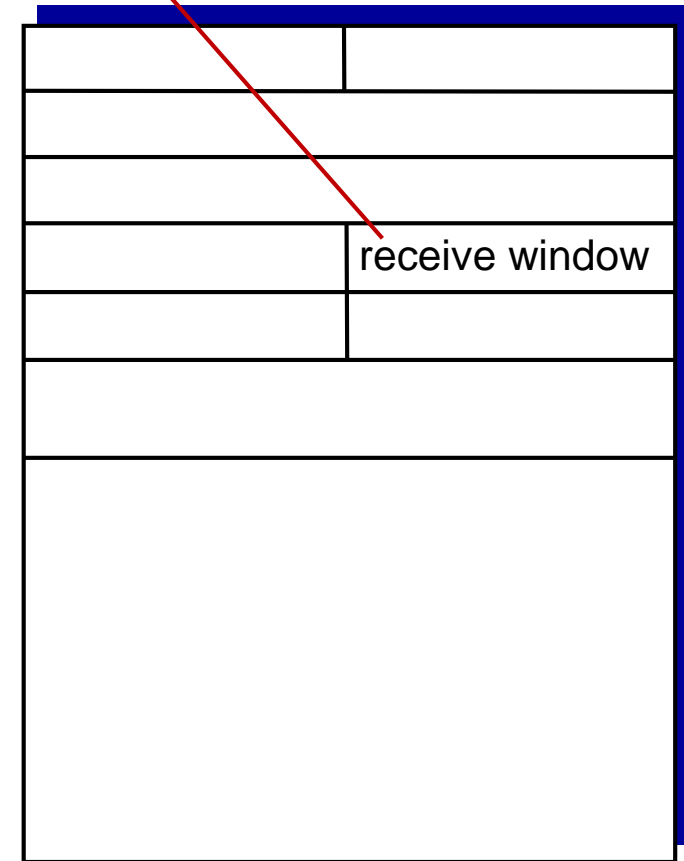
TCP segment format

TCP connection management

before exchanging data, sender/receiver "handshake":
- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters (e.g., starting seq #s)

application

connection state: ESTAB
connection variables:
   seq # client-to-server
       server-to-client
   rcvBuffer size
   at server,client

network

application

connection state: ESTAB
connection Variables:
   seq # client-to-server
       server-to-client
   rcvBuffer size
   at server,client

network
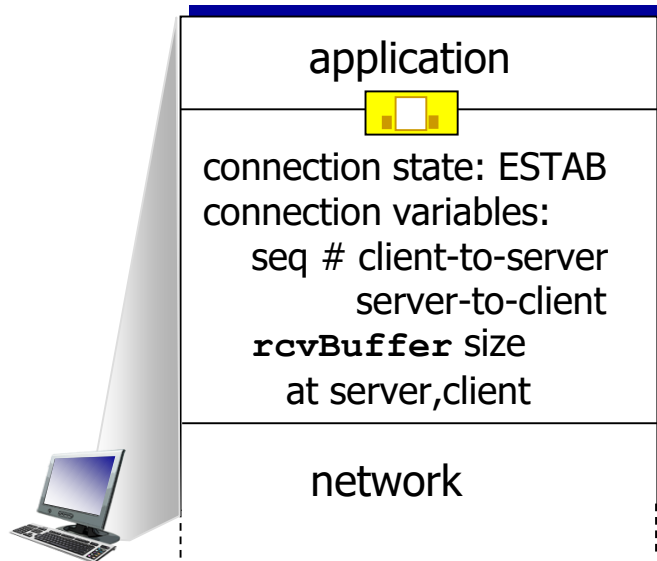
```
Socket clientSocket =
   newSocket("hostname","port number");
```
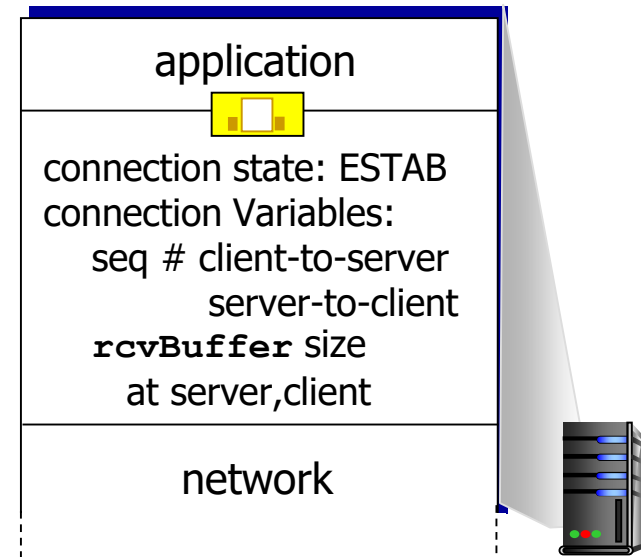
```
Socket connectionSocket =
   welcomeSocket.accept();
```

TCP 3-way handshake

Client state

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

LISTEN

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

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

LISTEN

choose init seq num, x
send TCP SYN msg

SYNSENT

SYNbit=1, Seq=x

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

SYN RCVD

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

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

ESTAB

ACKbit=1, ACKnum=y+1

received ACK(y)
indicates client is live

ESTAB

A human 3-way handshake protocol

Closing a TCP 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

Chapter 3: roadmap

Principles of congestion control

Congestion:

- informally: "too many sources sending too much data too fast for *network* to handle"

- manifestations:
  - long delays (queueing in router buffers)
  - packet loss (buffer overflow at routers)

- different from flow control!

- a top-10 problem!

congestion control: too many senders, sending too fast

flow control: one sender too fast for one receiver

Causes/costs of congestion: scenario 1

Simplest scenario:
- one router, infinite buffers
- input, output link capacity: R
- two flows
- no retransmissions needed

original data: λ_{in}

Host A

throughput: λ_{out}

infinite shared output link buffers

R   R

Host B
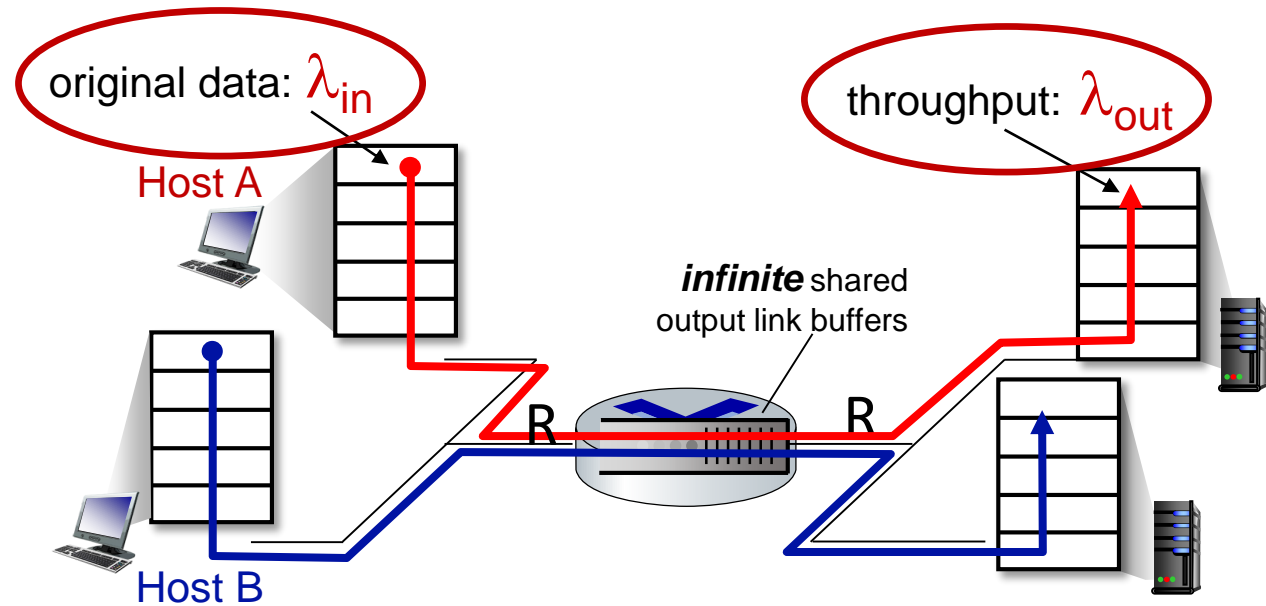
Q: What happens as arrival rate λ_{in} approaches R/2?
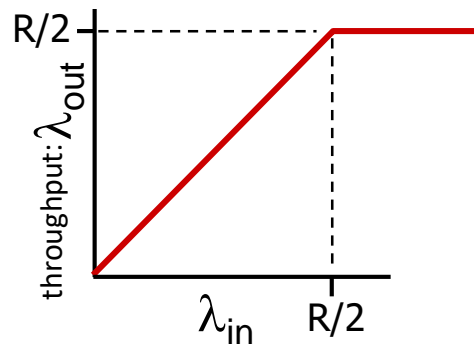
maximum per-connection throughput: R/2
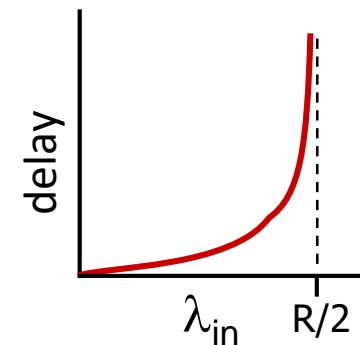
large delays as arrival rate λ_{in} approaches capacity

Causes/costs of congestion: scenario 2

- one router, *finite* buffers

- sender retransmits lost, timed-out packet
  - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$

Host A

λ_{in} : original data

λ'_{in}: original data, *plus* retransmitted data

λ_{out}

Host B

finite shared output link buffers
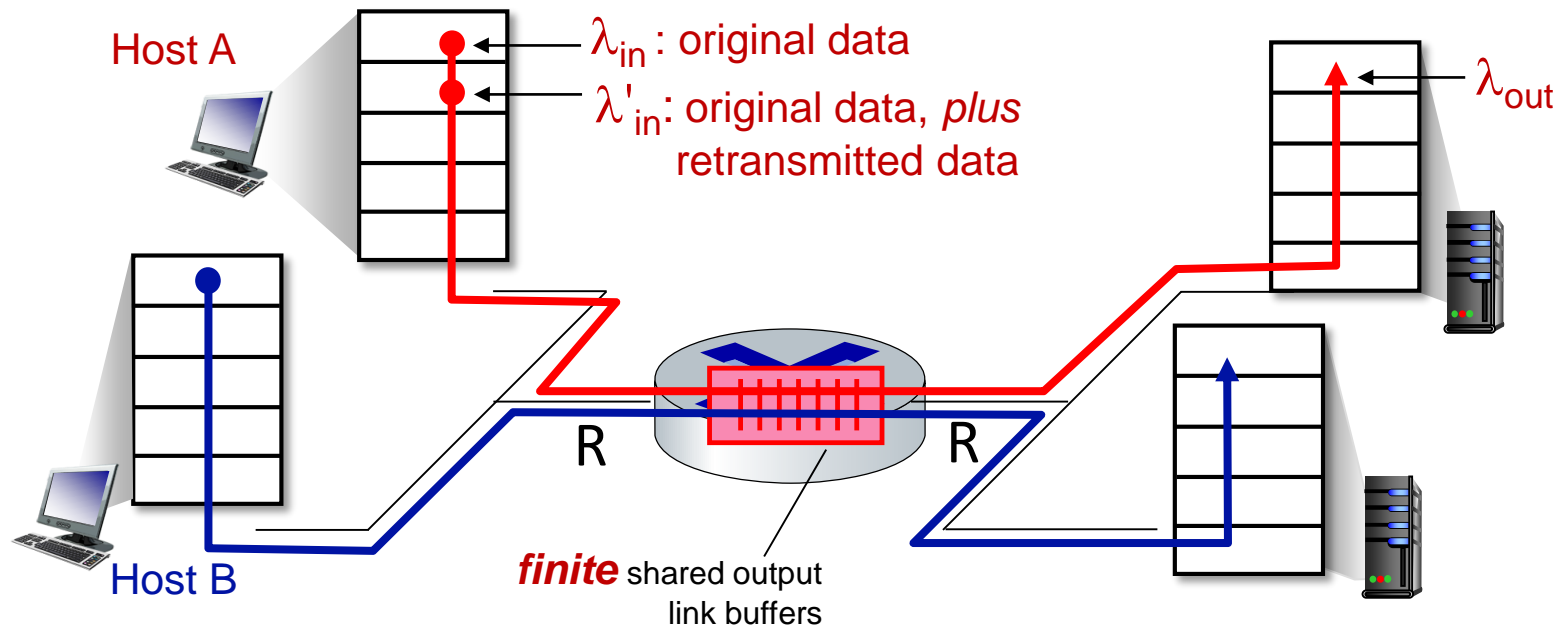
Causes/costs of congestion: scenario 2

Idealization: perfect knowledge

- sender sends only when router buffers available

Host A

copy

λ_{in} : original data

λ'_{in}: original data, *plus* retransmitted data

free buffer space!

Host B

finite shared output link buffers

λ_{out}

R   R

Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

- packets can be lost (dropped at router) due to full buffers

- sender knows when packet has been dropped: only resends if packet *known* to be lost

Host A

copy

λ_{in} : original data

λ'_{in}: original data, *plus* retransmitted data

no buffer space!

R   R

Host B

finite shared output link buffers

Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

- packets can be lost (dropped at router) due to full buffers

- sender knows when packet has been dropped: only resends if packet *known* to be lost

Host A

λ_{in} : original data

λ'_{in}: original data, *plus* retransmitted data

free buffer space!

R   R

Host B

finite shared output link buffers

Causes/costs of congestion: scenario 2

Realistic scenario: *un-needed duplicates*

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

λ_{in} : original data

λ'_{in}: original data, *plus* retransmitted data

"wasted" capacity due to un-needed retransmissions

when sending at R/2, some packets are retransmissions, including needed and *un-needed* duplicates, that are delivered!

timeout

free buffer space!

Host B

R   R

finite shared output link buffers

Causes/costs of congestion: scenario 2

Realistic scenario: *un-needed duplicates*

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

R/2

throughput: λ_{out}

λ'_{in} R/2

"wasted" capacity due to un-needed retransmissions

when sending at R/2, some packets are retransmissions, including needed and *un-needed* duplicates, that are delivered!

"costs" of congestion:

- more work (retransmission) for given receiver throughput
- unneeded retransmissions: link carries multiple copies of a packet
  - decreasing maximum achievable throughput

Causes/costs of congestion: scenario 3
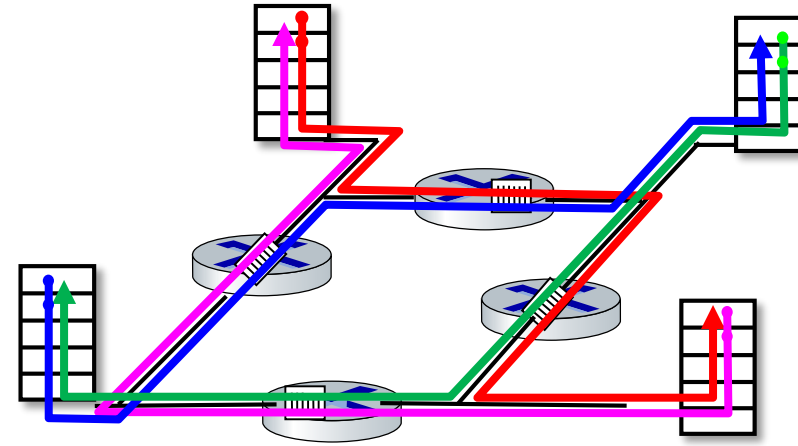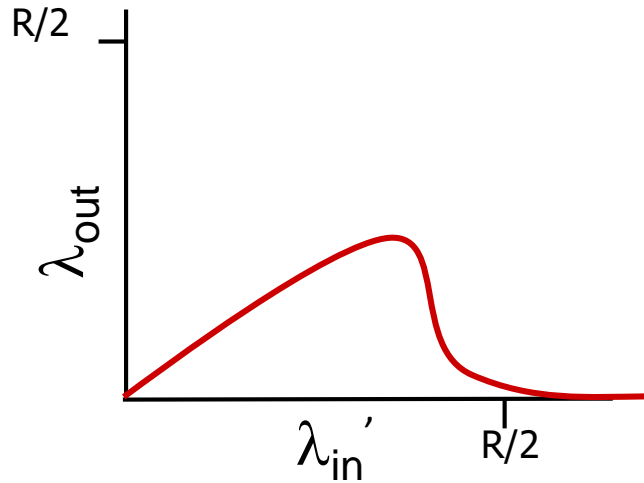
another "cost" of congestion:

- when packet dropped, any upstream transmission capacity and buffering used for that packet was wasted!

Causes/costs of congestion: insights

- throughput can never exceed capacity

- delay increases as capacity approached

- loss/retransmission decreases effective throughput

- un-needed duplicates further decreases effective throughput
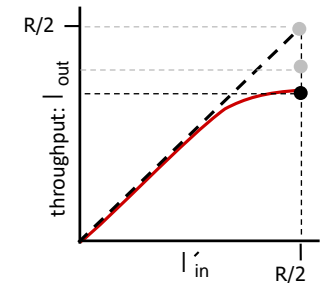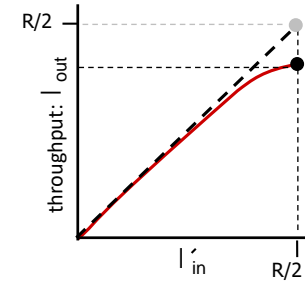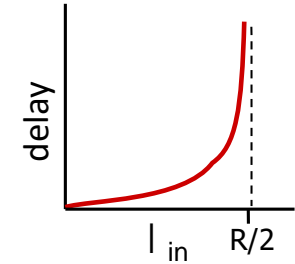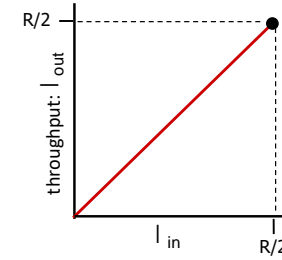
Approaches towards congestion control

End-end congestion control:

- no explicit feedback from network

- congestion *inferred* from observed loss, delay

- approach taken by TCP
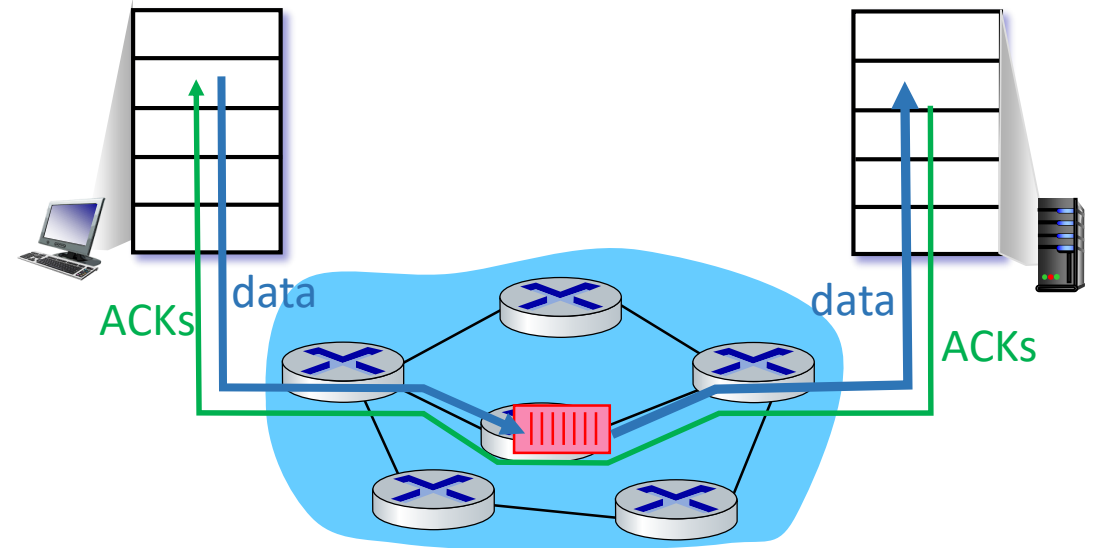
Approaches towards congestion control

Network-assisted congestion control:

- routers provide *direct* feedback to sending/receiving hosts with flows passing through congested router

- may indicate congestion level or explicitly set sending rate

- TCP ECN, ATM, DECbit protocols
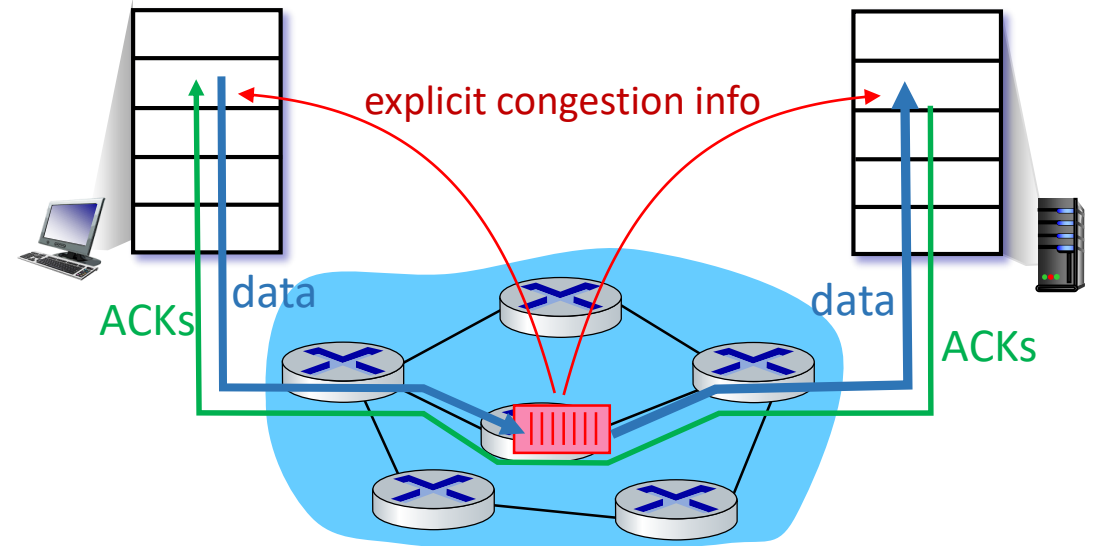
Chapter 3: roadmap

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

TCP congestion control: AIMD

- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

Additive Increase

increase sending rate by 1 maximum segment size every RTT until loss detected

Multiplicative Decrease
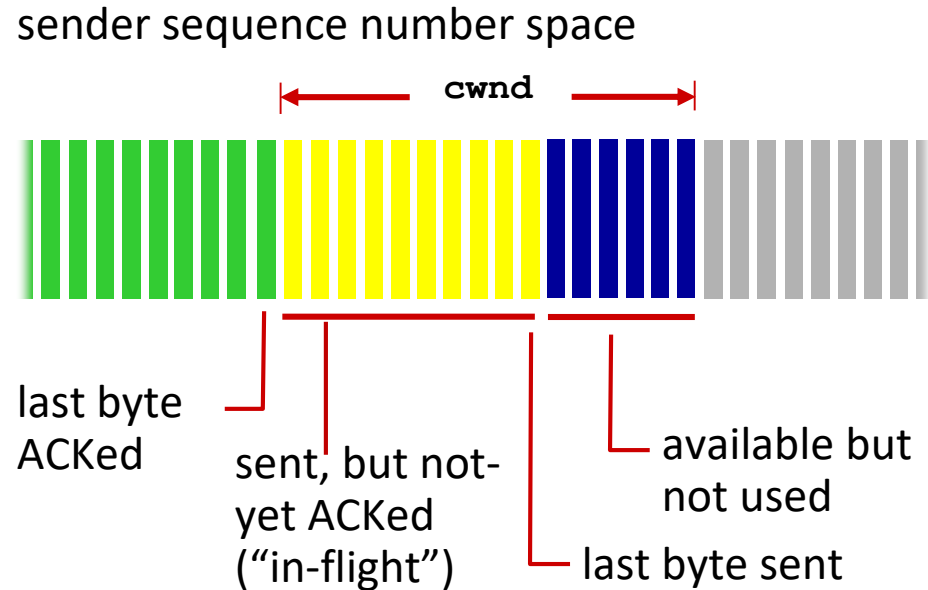
cut sending rate in half at each loss event
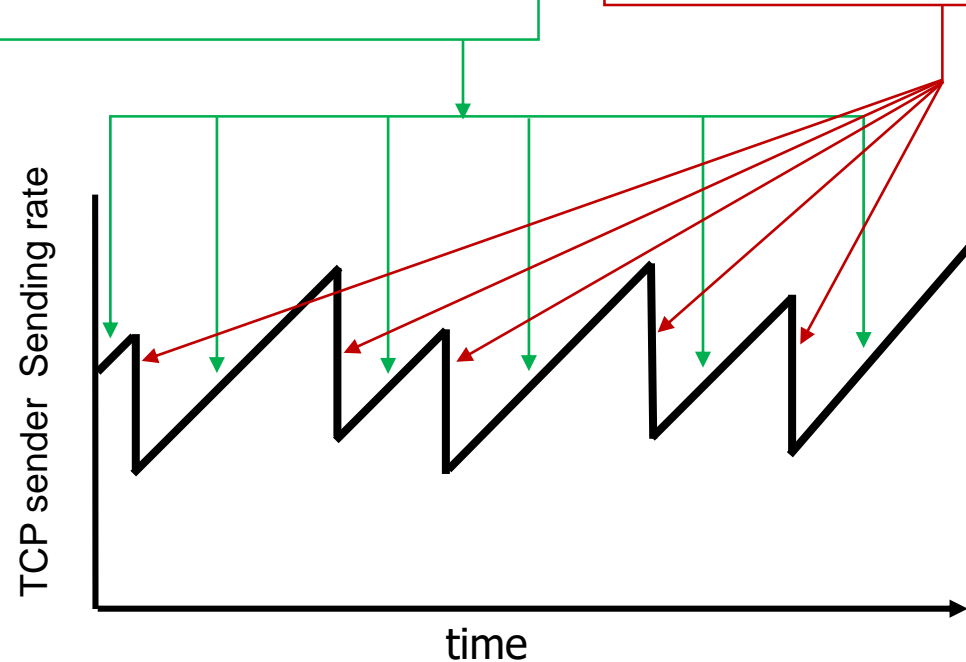
AIMD sawtooth behavior: *probing* for bandwidth

TCP sender Sending rate

time

TCP AIMD: more

Multiplicative decrease detail: sending rate is

- Cut in half on loss detected by triple duplicate ACK (TCP Reno)
- Cut to 1 MSS (maximum segment size) when loss detected by timeout (TCP Tahoe)

TCP congestion control: details

sender sequence number space

cwnd

last byte ACKed

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

available but not used

last byte sent

TCP sending behavior:

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

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

- TCP sender limits transmission: `LastByteSent- LastByteAcked` $\leq$ `cwnd`

- `cwnd` is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)

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
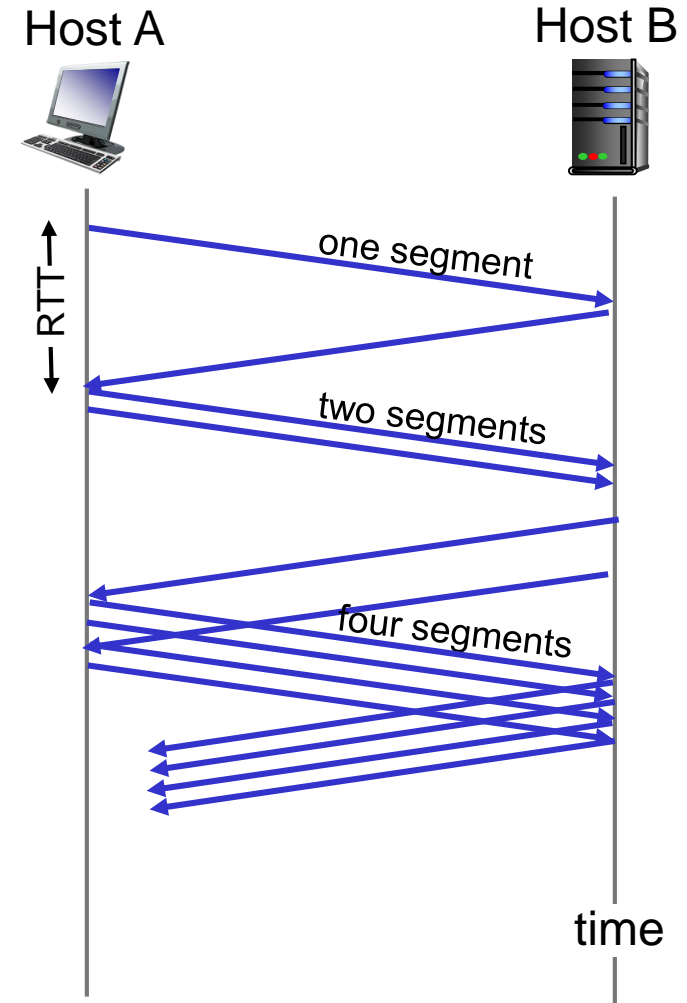
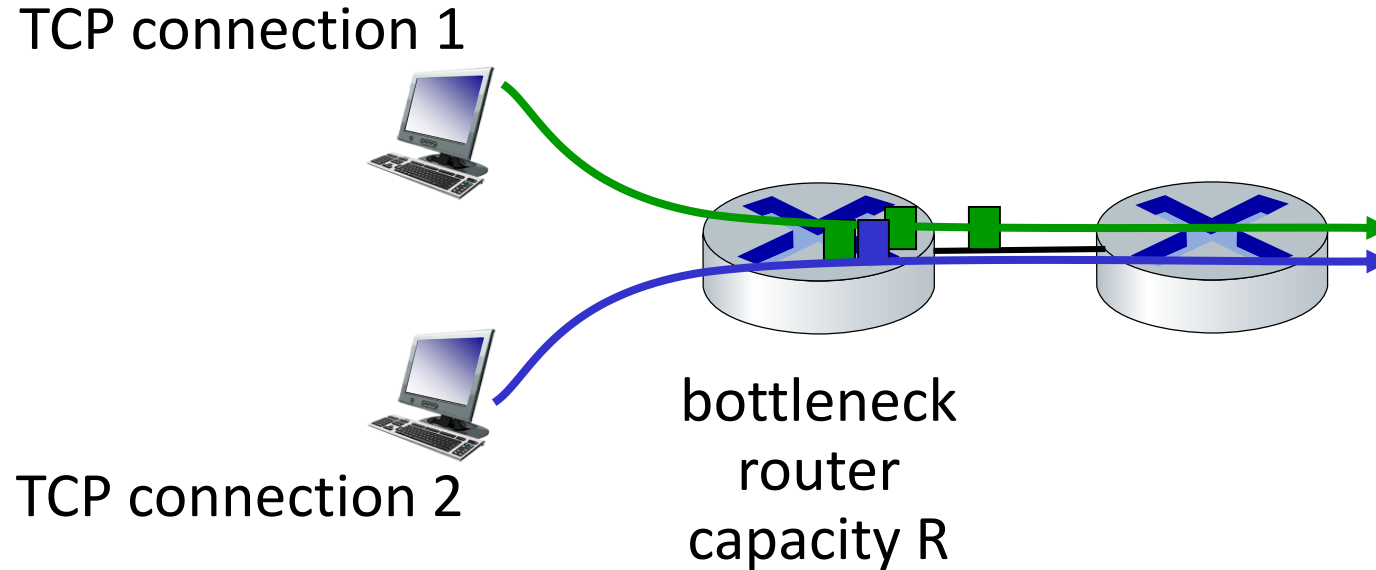# TCP: from slow start to congestion avoidance

Q: when should the exponential increase switch to linear?

A: when `cwnd` gets to 1/2 of its value before timeout.
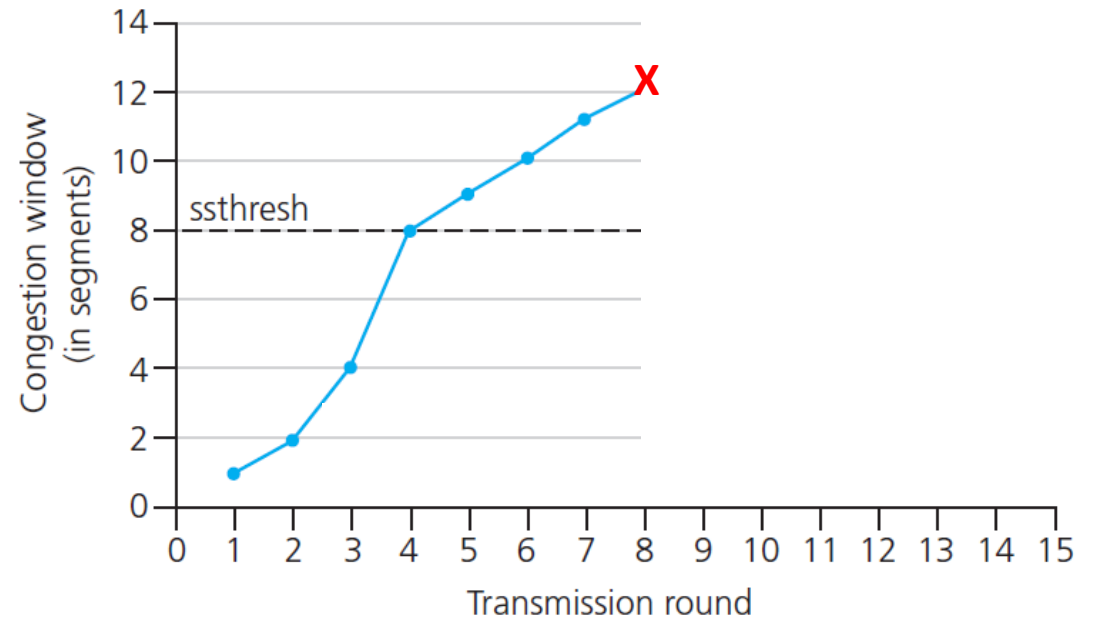
TCP fairness

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

TCP connection 1

TCP connection 2

bottleneck
router
capacity R

Chapter 3
Transport Layer

A note on the use of these PowerPoint slides:

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

- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

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

Thanks and enjoy! JFK/KWR
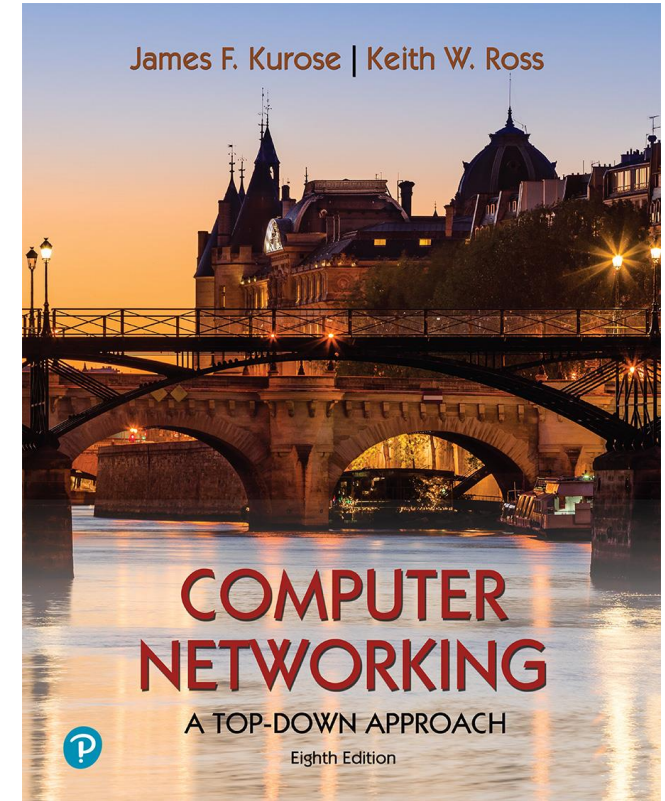
Computer Networking: A Top-Down Approach
8th edition
Jim Kurose, Keith Ross
Pearson, 2020

Thank you