Chapter 3
Transport Layer

**A note on the use of these ppt 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.

Thanks and enjoy! JFK/KWR
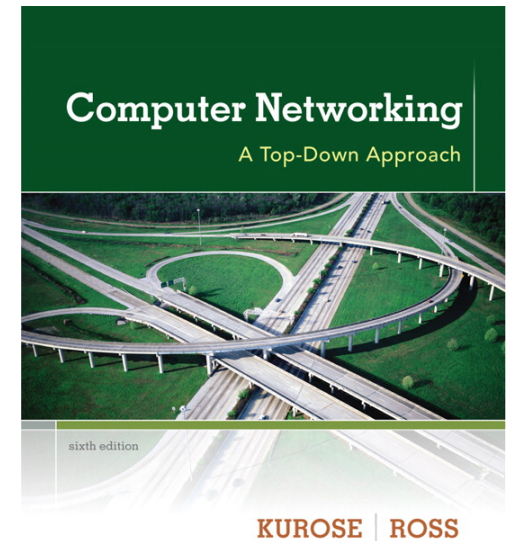
*Computer
Networking: A
Top Down
Approach*
6th edition
Jim Kurose, Keith Ross
Addison-Wesley
March 2012

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

Transport services and protocols

❖ provide *logical communication* between app processes running on different hosts

❖ transport protocols run in end systems

- send side: breaks app messages into *segments*, passes to network layer
- rcv side: reassembles segments into messages, passes to app layer

❖ more than one transport protocol available to apps

application
transport
network
data link
physical

logical end-end transport

application
transport
network
data link
physical
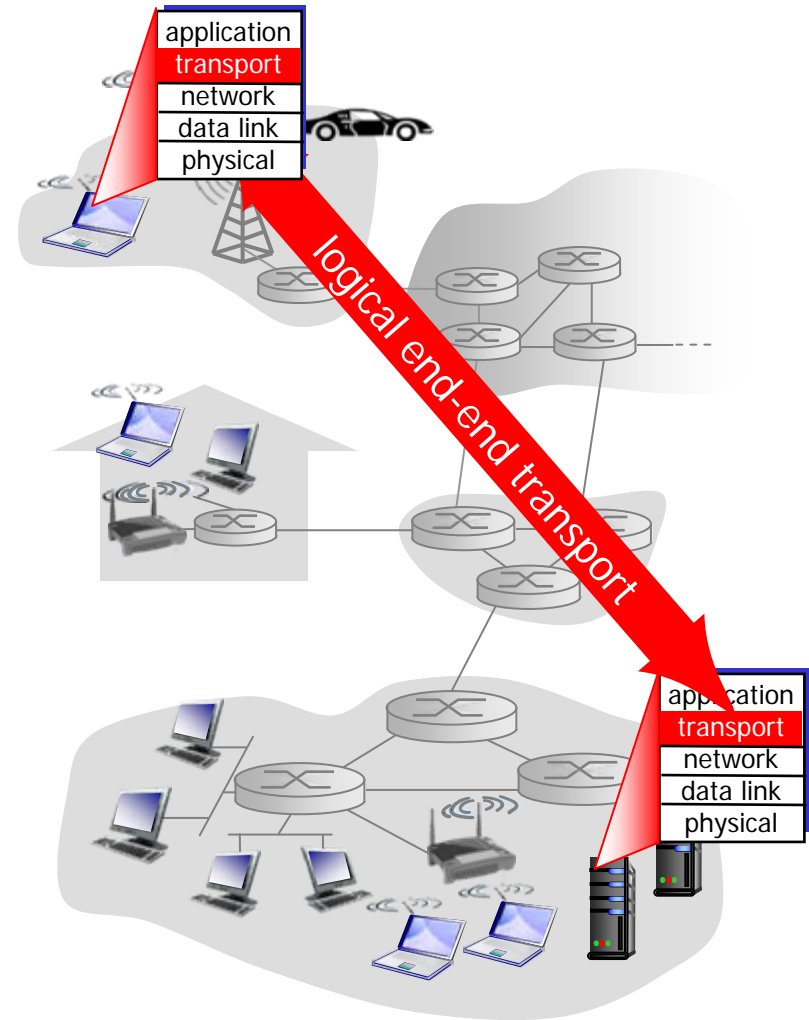
Transport vs. network layer

❖ *network layer:* logical communication between hosts

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

household analogy:

12 kids in Ann's house sending letters to 12 kids in Bill's house:

❖ hosts = houses

❖ processes = kids

❖ app messages = letters in envelopes

❖ transport protocol = Ann and Bill who demux to in-house siblings

❖ network-layer protocol = postal service

Internet transport-layer protocols

❖ reliable, in-order delivery (TCP)
  ▪ congestion control
  ▪ flow control
  ▪ connection setup
❖ unreliable, unordered delivery: UDP
  ▪ no-frills extension of "best-effort" IP
❖ services not available:
  ▪ delay guarantees
  ▪ bandwidth guarantees
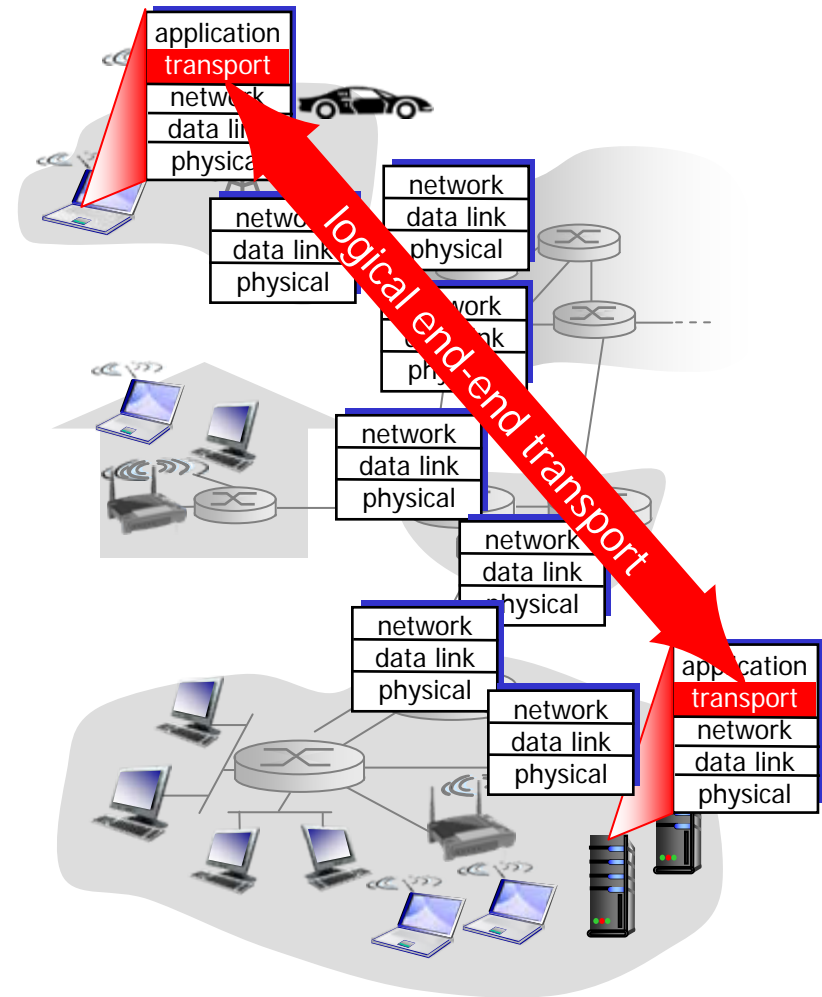
application
transport
network
data link
physical

network
data link
physical

network
data link
physical

network
data link
physical

network
data link
physical

network
data link
physical

network
data link
physical

application
transport
network
data link
physical

logical end-end transport

Chapter 3 outline

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
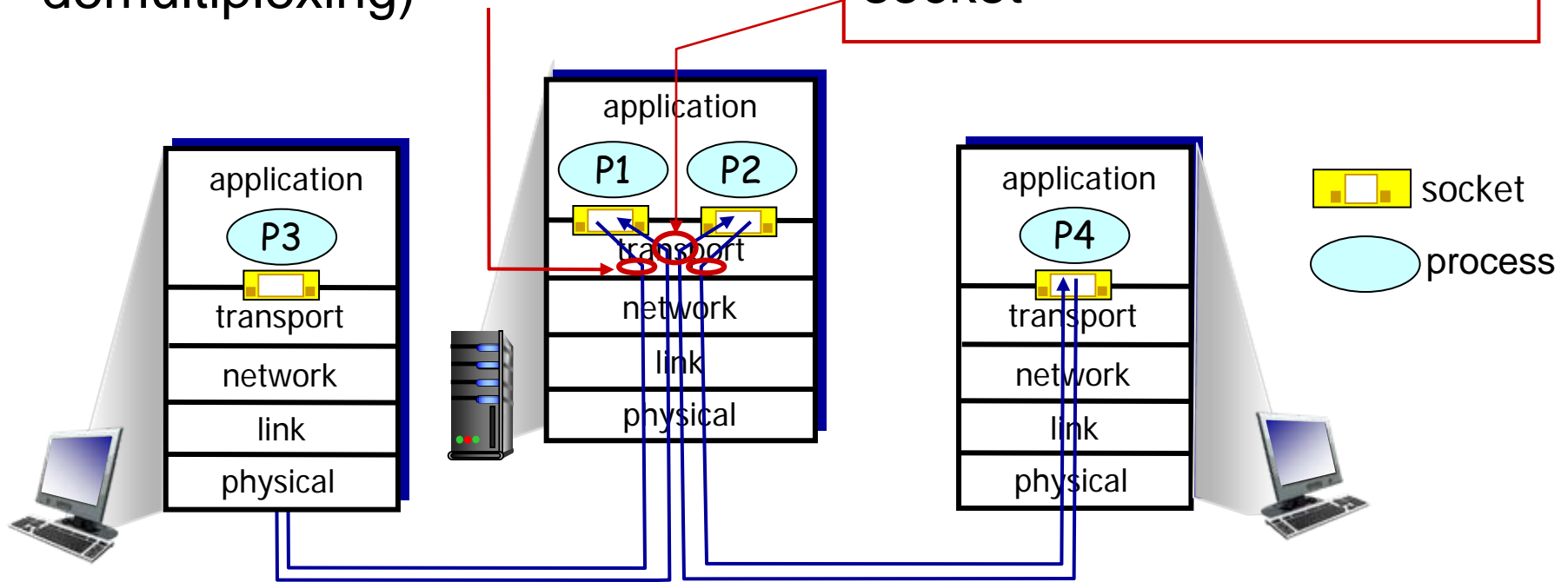
How demultiplexing works

❖ **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**

32 bits

source port #	dest port #
other header fields	
application data (payload)	
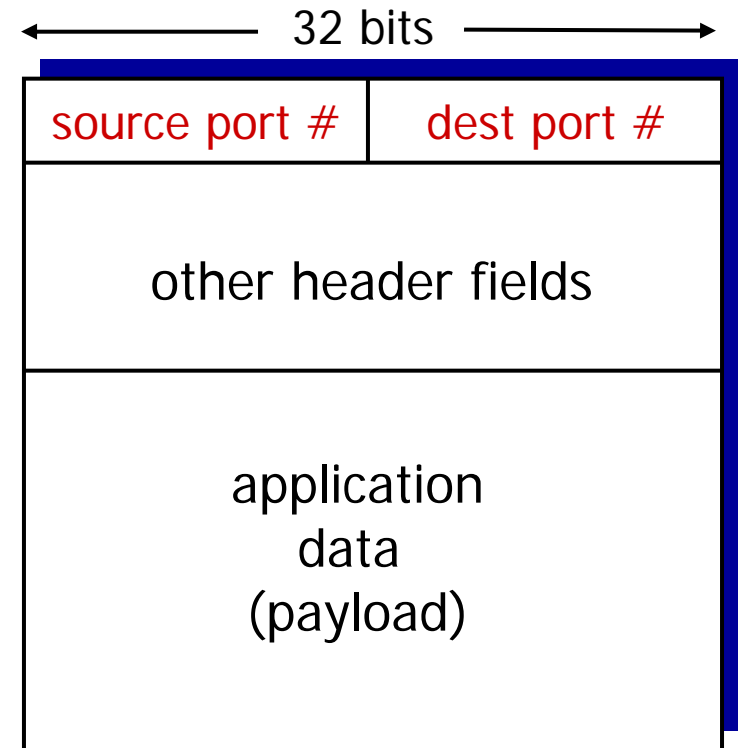
TCP/UDP segment format

Connectionless demultiplexing

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

DatagramSocket mySocket1
= new DatagramSocket(12534);

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

❖ when host receives UDP segment:
  - checks destination port # in segment
  - directs UDP segment to socket with that port #

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

Connectionless demux: example

DatagramSocket mySocket2 = new DatagramSocket (9157);

DatagramSocket serverSocket = new DatagramSocket (6428);

DatagramSocket mySocket1 = new DatagramSocket (5775);

application

P3

transport

network

link

physical

application

P1

transport

network

link

physical

application

P4

transport

network

link

physical

source port: 6428
dest port: 9157

source port: ?
dest port: ?

source port: 9157
dest port: 6428

source port: ?
dest port: ?
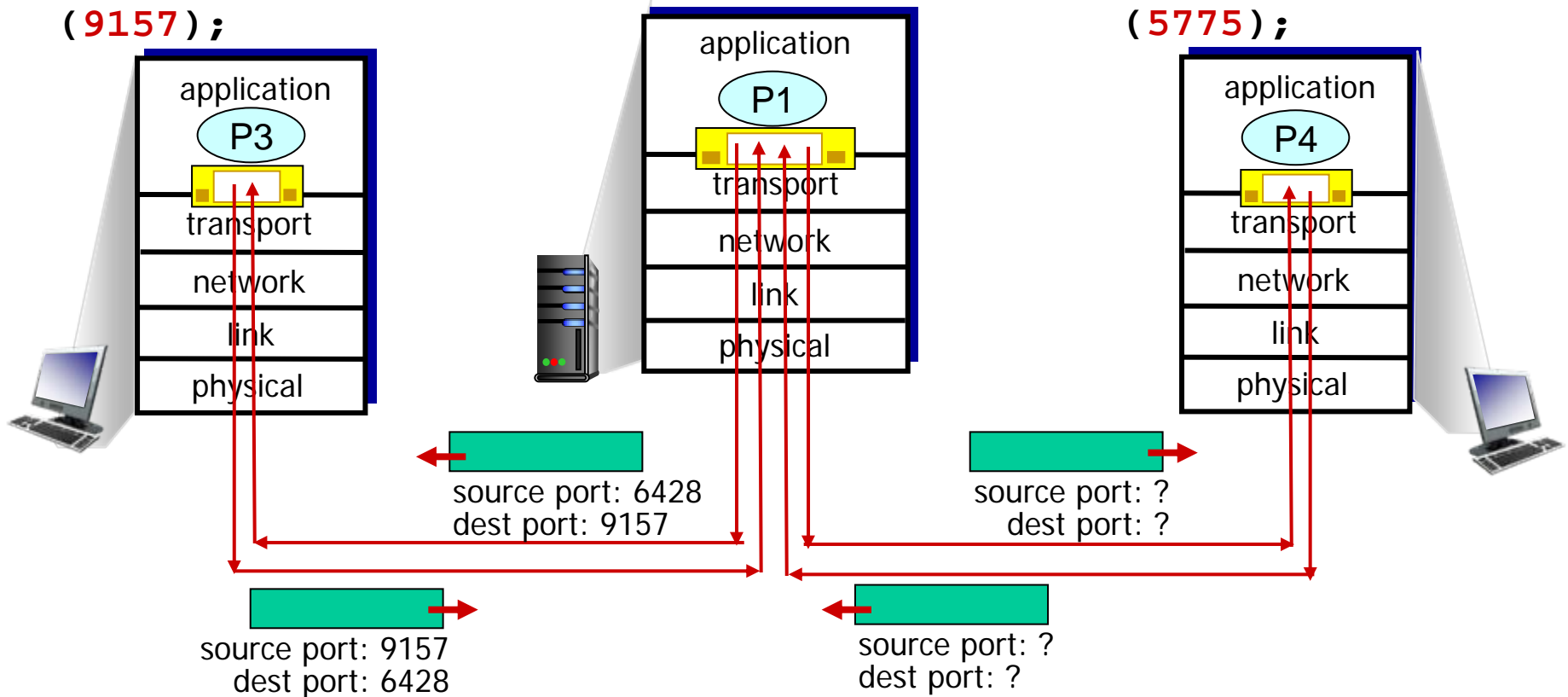
Connection-oriented demux

- ❖ TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- ❖ demux: receiver uses all four values to direct segment to appropriate socket

- ❖ server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- ❖ web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

Connection-oriented demux: example

application

P4 **P5** **P6**

transport

network

link

physical

server:
IP
address
B

application

P3

transport

network

link

physical

host: IP
address
A

application

P2 **P3**

transport

network

link

physical

host: IP
address
C

source IP,port: B,80
dest IP,port: A,9157

source IP,port: A,9157
dest IP, port: B,80

source IP,port: C,5775
dest IP,port: B,80

source IP,port: C,9157
dest IP,port: B,80
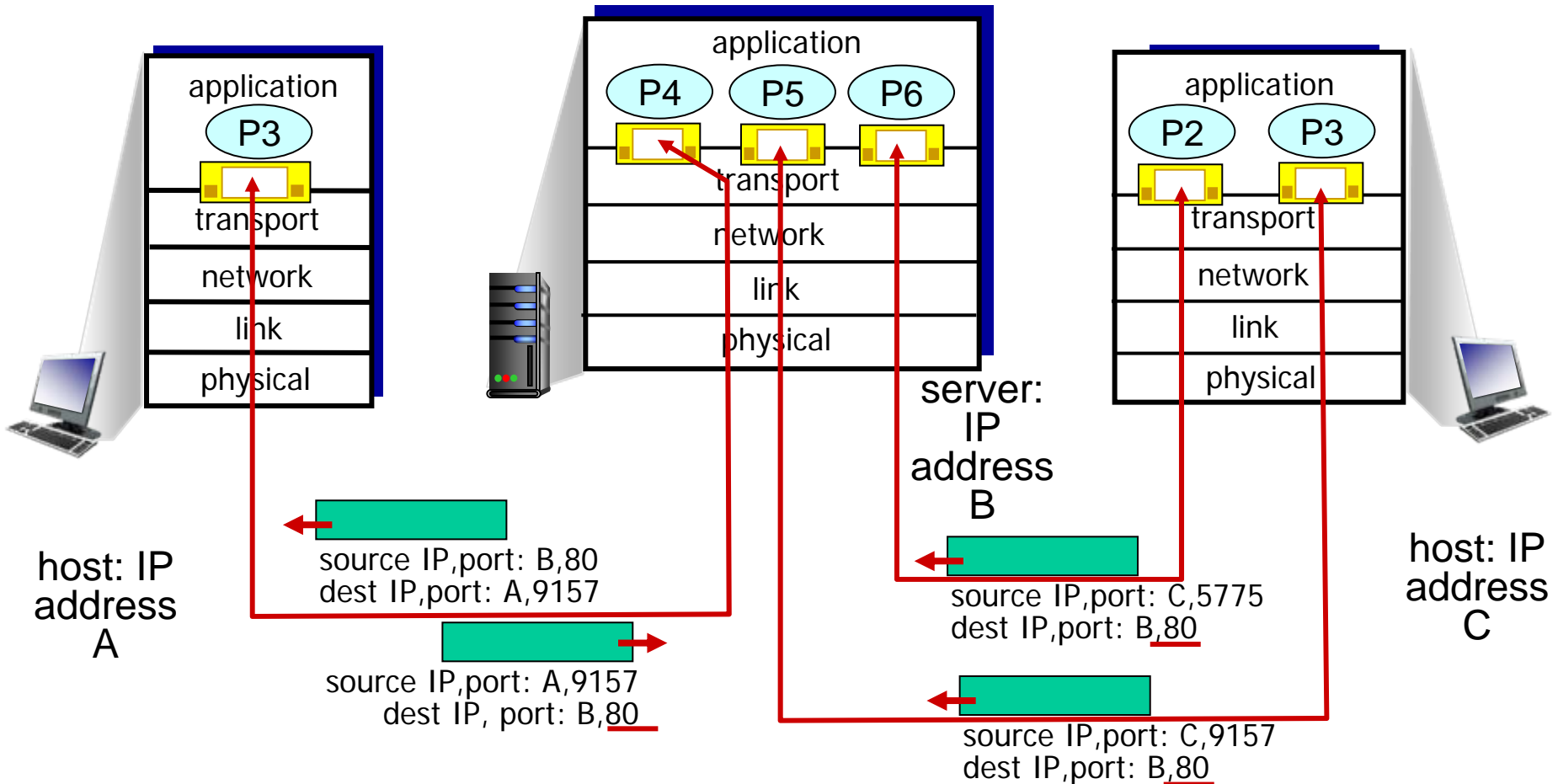
three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different* sockets

Connection-oriented demux: example
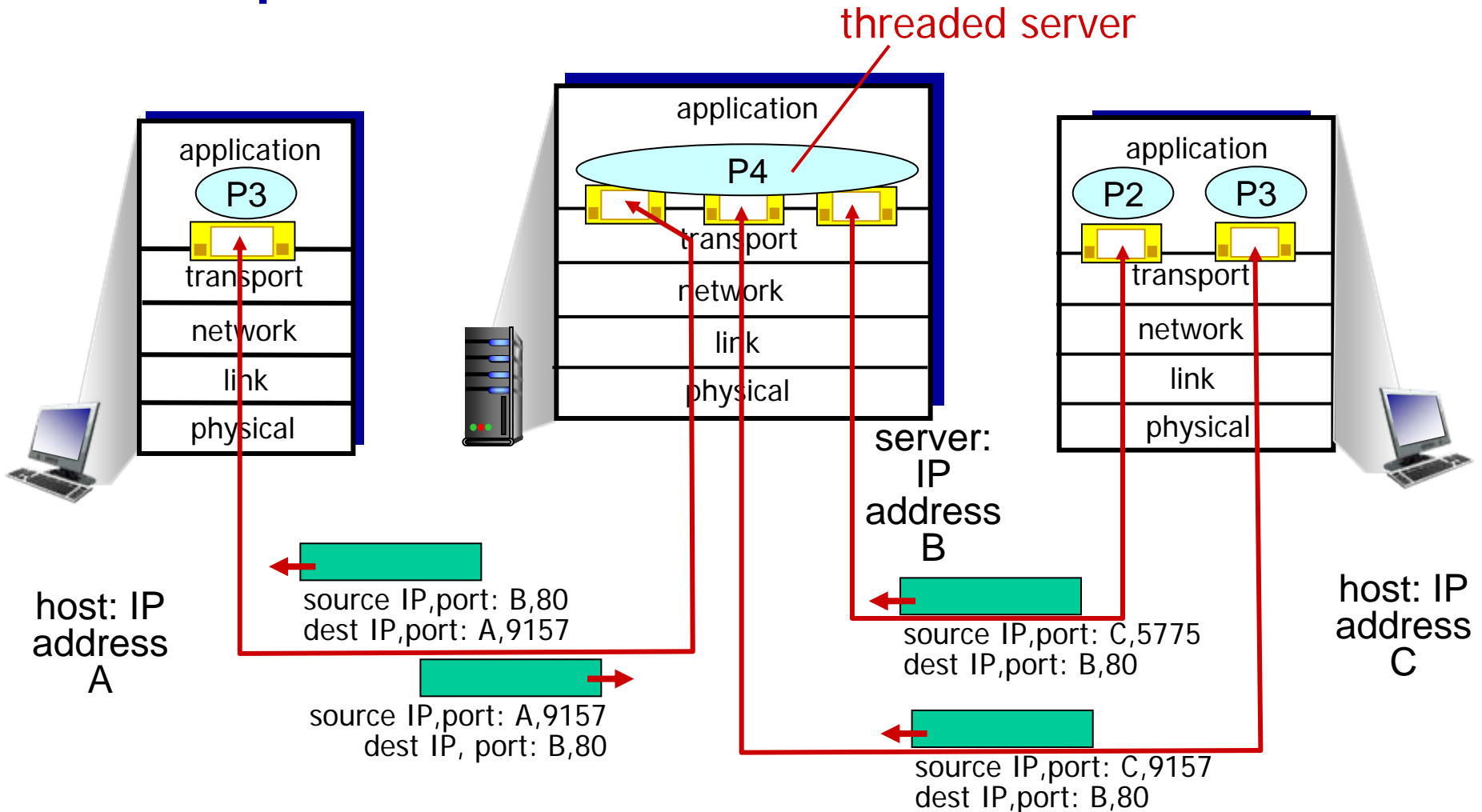
threaded server

host: IP address A

application
P3
transport
network
link
physical

application
P4
transport
network
link
physical

server: IP address B

application
P2   P3
transport
network
link
physical

host: IP address C

source IP,port: B,80
dest IP,port: A,9157

source IP,port: A,9157
dest IP, port: B,80

source IP,port: C,5775
dest IP,port: B,80

source IP,port: C,9157
dest IP,port: B,80

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

UDP: User Datagram Protocol [RFC 768]

- ❖ "no frills," "bare bones" Internet transport protocol
- ❖ "best effort" service, UDP segments may be:
  - lost
  - delivered out-of-order to app
- ❖ *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

- ❖ UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- ❖ reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

UDP: segment header

length, in bytes of
UDP segment,
including header

← 32 bits →

source port #	dest port #
length	checksum
application data (payload)	
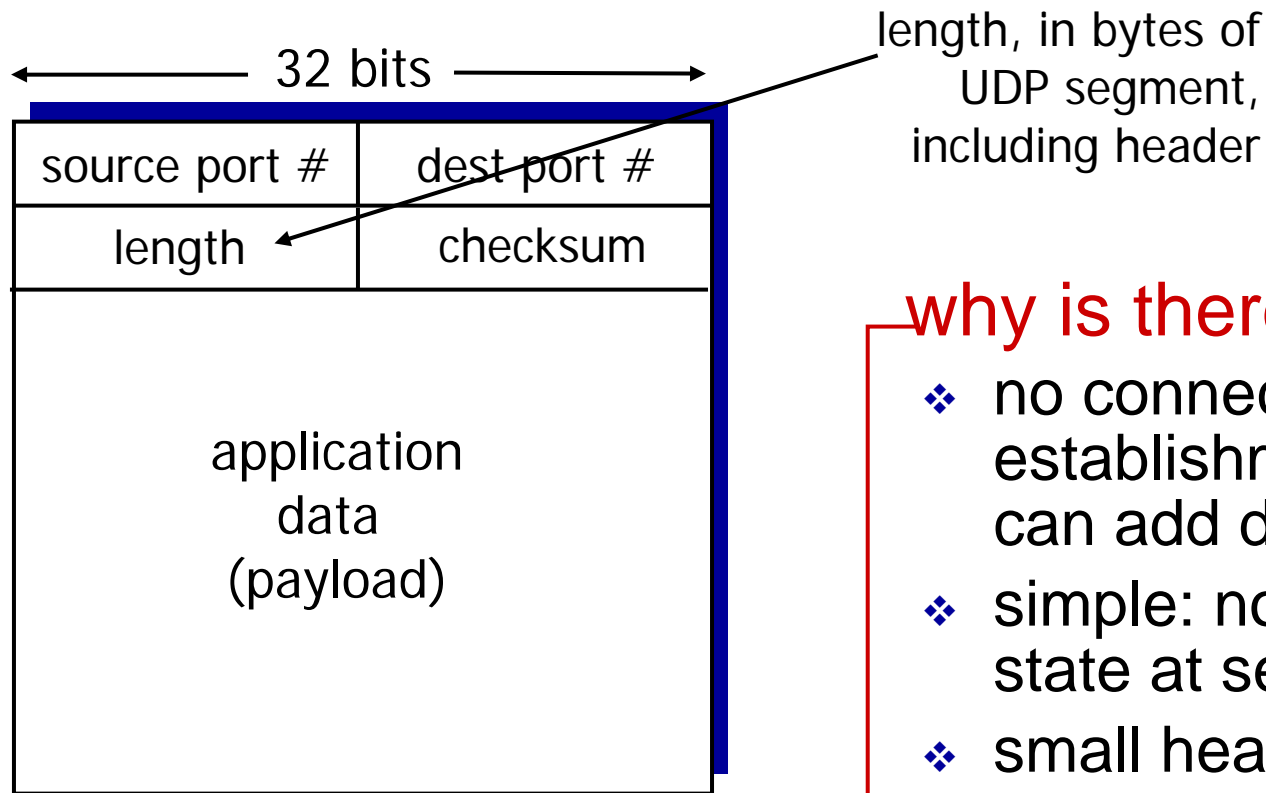
UDP segment format

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
```

wraparound   (1) 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 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

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

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

3.6 principles of congestion control

3.7 TCP congestion control

Principles of reliable data transfer

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

application layer

transport layer

sending process

receiver process

data

data
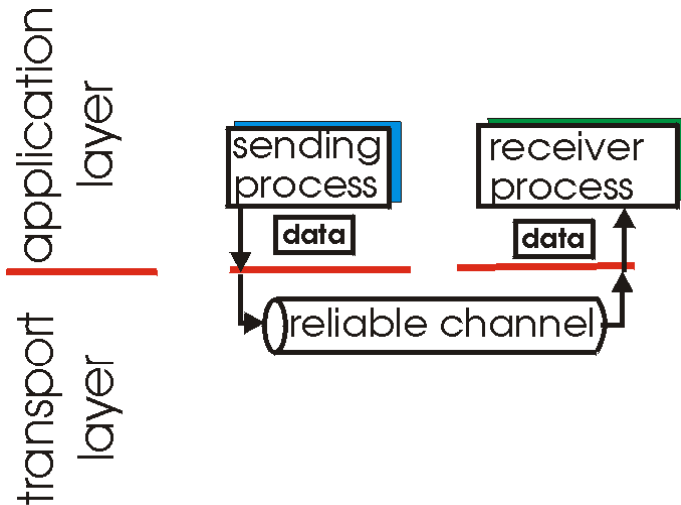
reliable channel

(a) provided service

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

Principles of reliable data transfer

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

application layer

transport layer

sending process

receiver process

data

data

reliable channel

unreliable channel

(a) provided service

(b) service implementation

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

Principles of reliable data transfer

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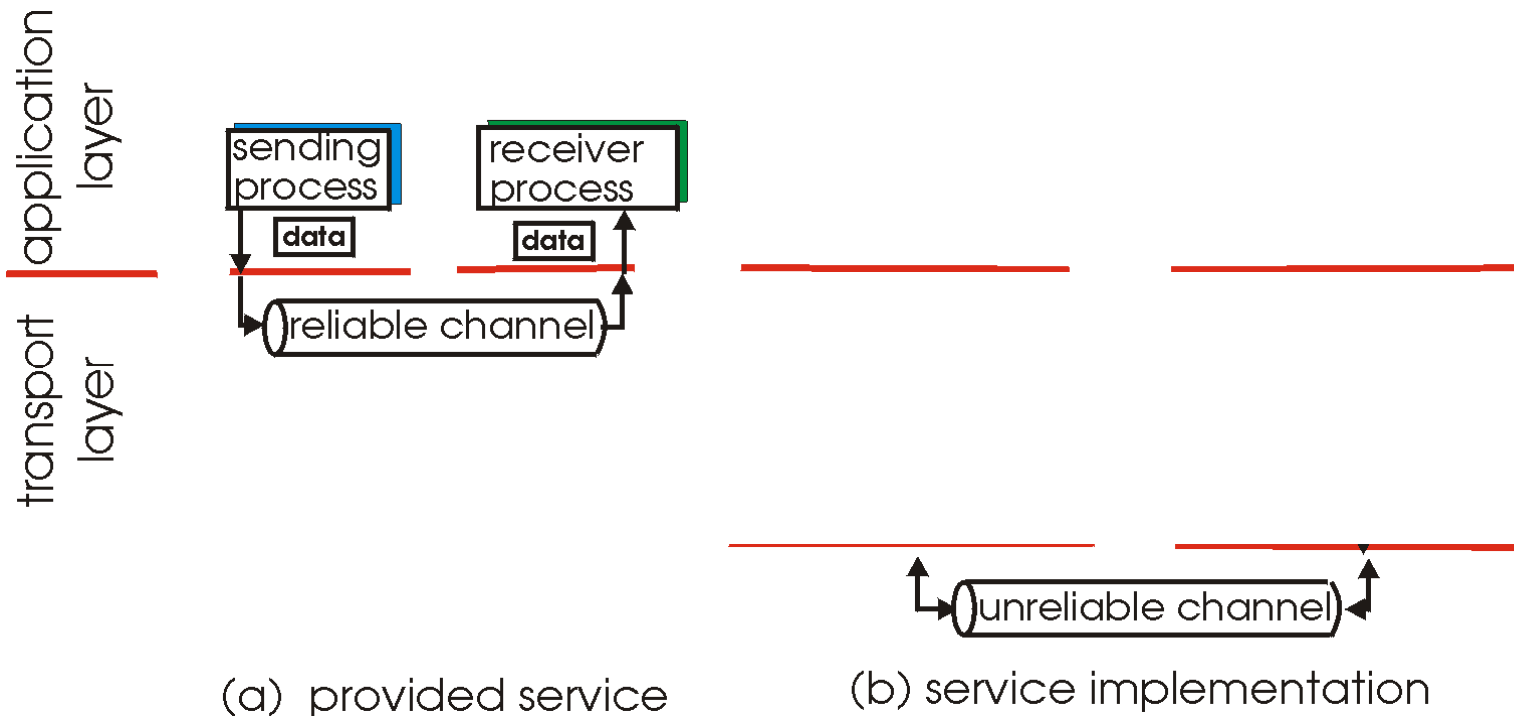 ▪ top-10 list of important networking topics!

application layer

transport layer

sending process → data

receiver process ← data

reliable channel

(a) provided service

rdt_send() ↓ data

reliable data transfer protocol (sending side)

udt_send() ↕ packet

data ↑ deliver_data()

reliable data transfer protocol (receiving side)

packet ↑ rdt_rcv()

unreliable channel

(b) service implementation

❖ characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)
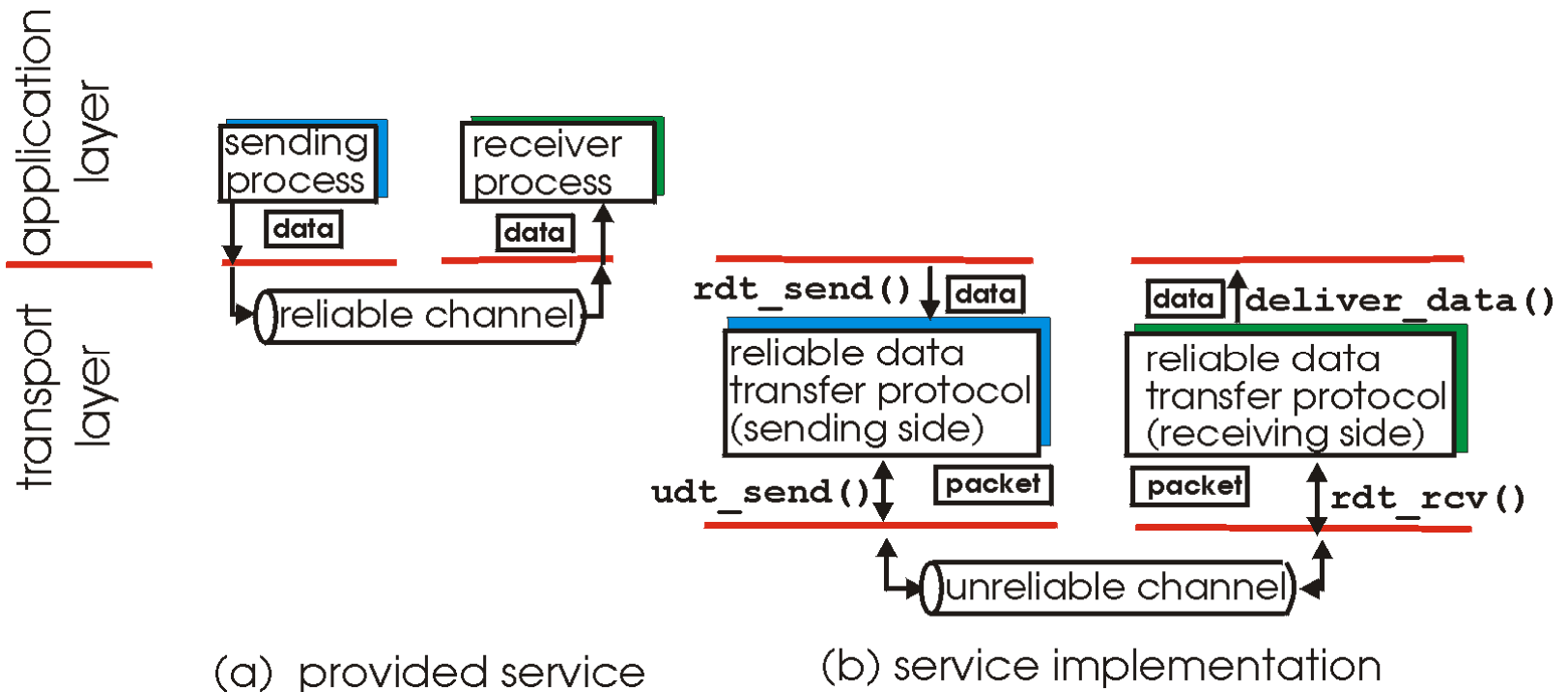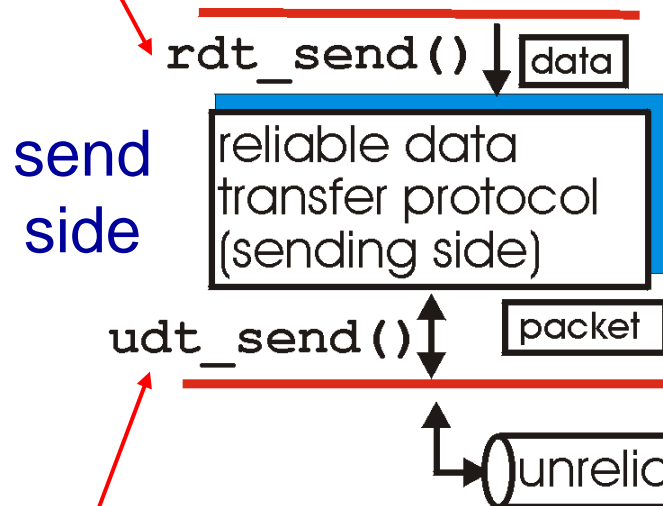
Reliable data transfer: getting started

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

deliver_data(): called by **rdt** to deliver data to upper

rdt_send() ↓ data

data ↑ **deliver_data()**

send side

reliable data transfer protocol (sending side)

reliable data transfer protocol (receiving side)

receive side

udt_send() ↕ packet

packet ↕ **rdt_rcv()**

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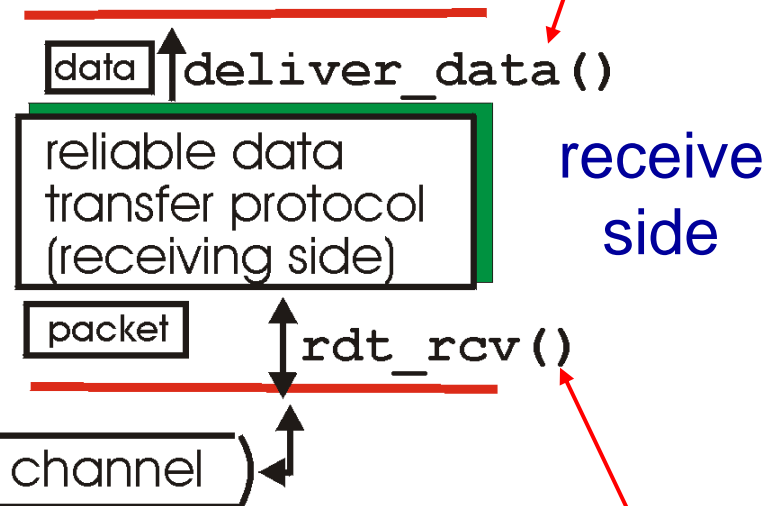
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

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

Reliable data transfer: getting started

we'll:

❖ incrementally develop sender, receiver sides of <u>r</u>eliable <u>d</u>ata <u>t</u>ransfer 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

event causing state transition
actions taken on state transition

state 1

event
actions

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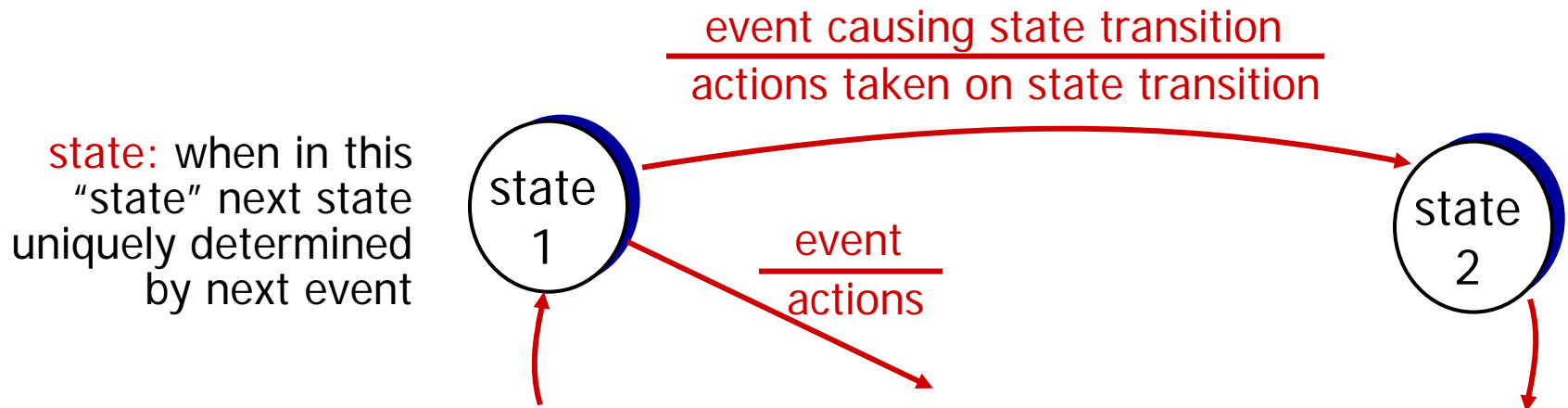
rdt1.0: reliable transfer over a reliable channel

❖ **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

Wait for call from above

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

Wait for call from below

rdt_rcv(packet)
—————————————
extract (packet,data)
deliver_data(data)
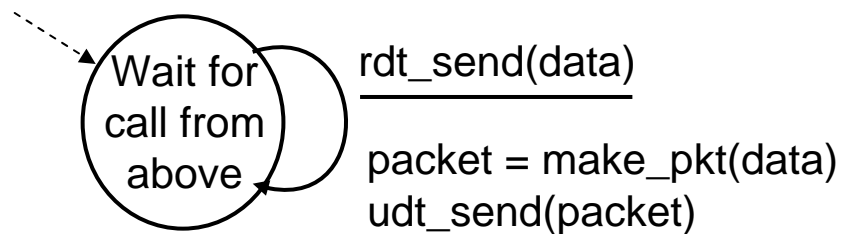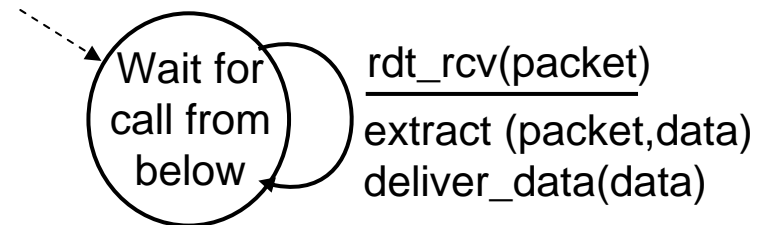
sender   receiver

rdt2.0: channel with bit errors

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

How do humans recover from "errors" during conversation?

rdt2.0: channel with bit errors

❖ **underlying channel may flip bits in packet**
  ▪ checksum to detect bit errors

❖ *the* **question: how to recover from errors:**

  ▪ *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK

  ▪ *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors

  ▪ sender retransmits pkt on receipt of NAK

❖ **new mechanisms in `rdt2.0` (beyond `rdt1.0`):**
  ▪ error detection
  ▪ feedback: control msgs (ACK,NAK) from receiver to sender

rdt2.0: FSM specification

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

receiver

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

Wait for call from above   **Wait for ACK or NAK**

rdt_rcv(rcvpkt) && isACK(rcvpkt)
_____
Λ

sender

rdt_rcv(rcvpkt) &&
   corrupt(rcvpkt)
_____
udt_send(NAK)

Wait for call from below

rdt_rcv(rcvpkt) &&
   notcorrupt(rcvpkt)
_____
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)
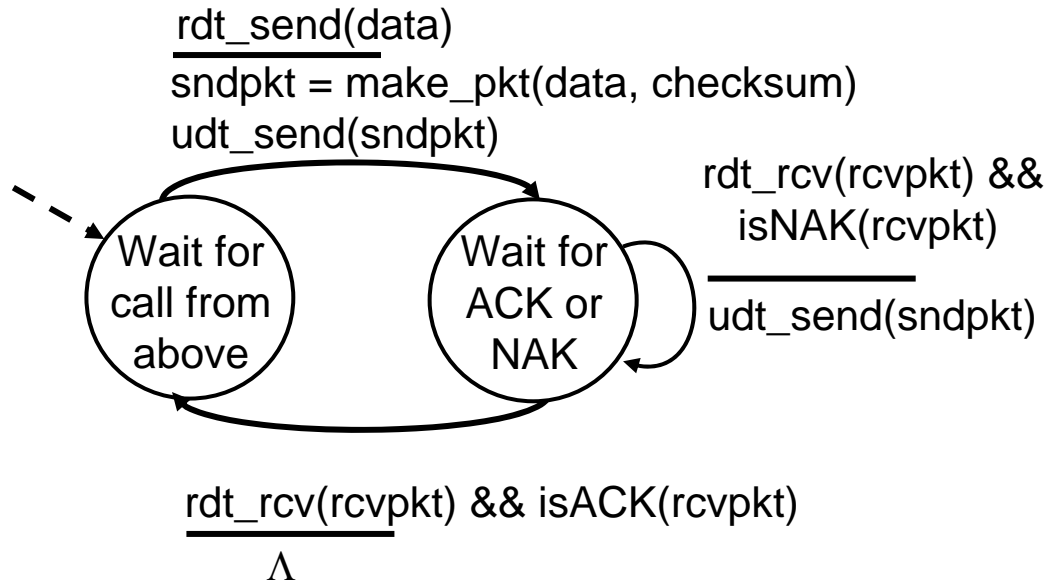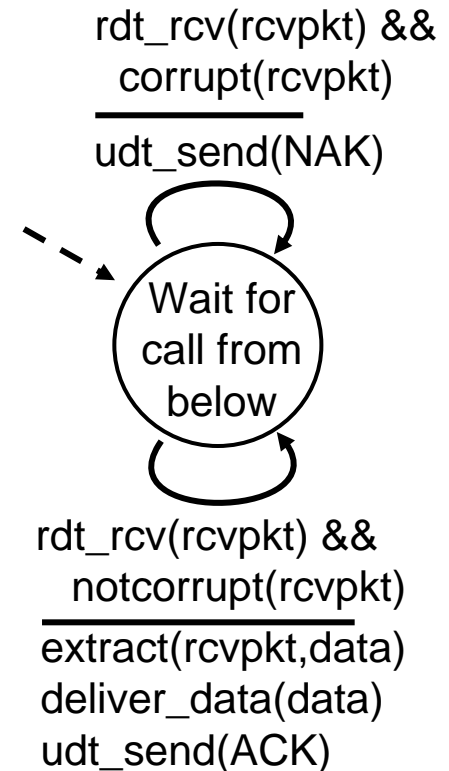
rdt2.0: operation with no errors

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

rdt_rcv(rcvpkt) &&
corrupt(rcvpkt)
_____
udt_send(NAK)

Wait for call from below

rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)
_____
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)
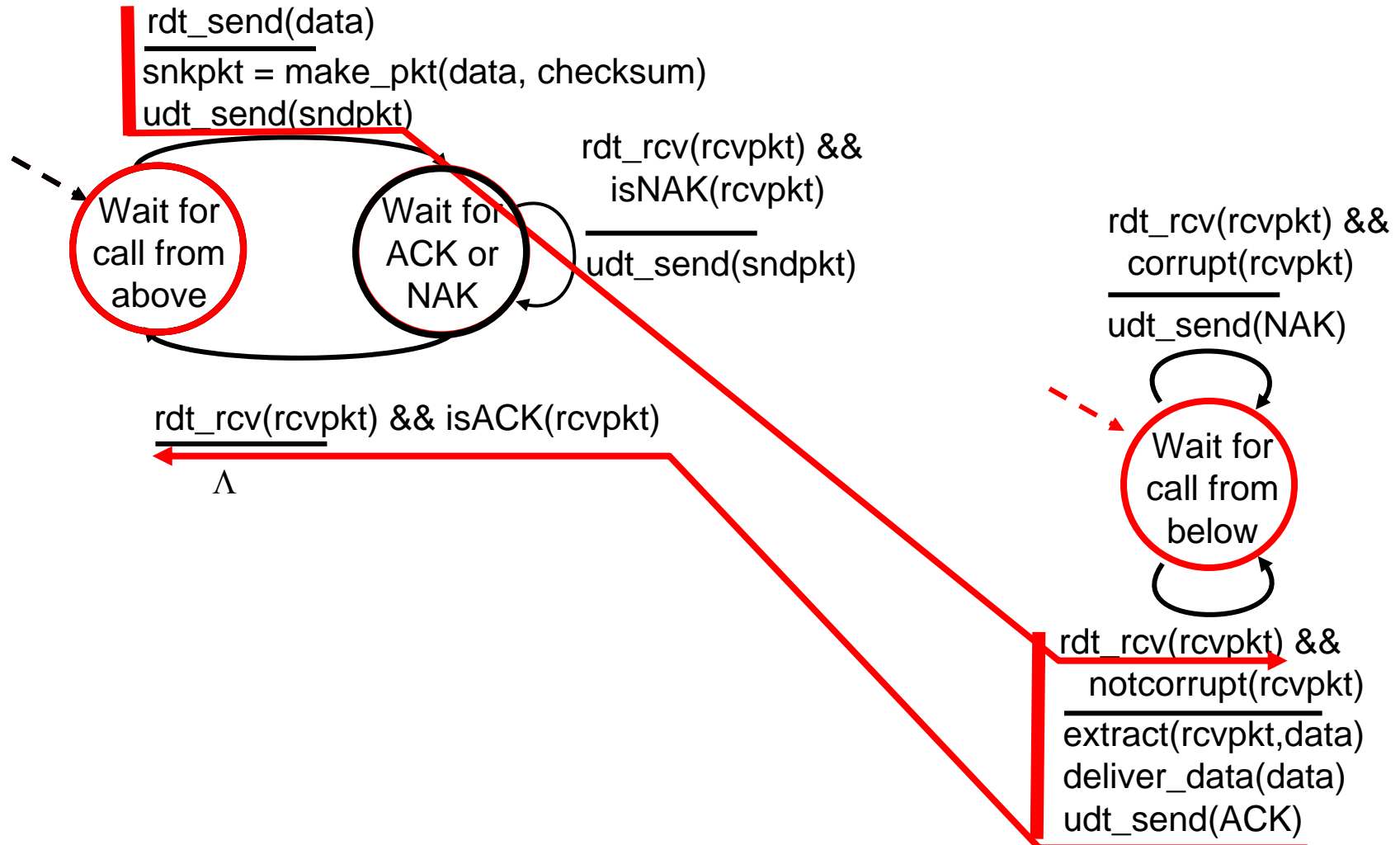
rdt2.0: error scenario

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) && corrupt(rcvpkt)

udt_send(NAK)

rdt_rcv(rcvpkt) && isACK(rcvpkt)

Λ

Wait for call from below

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

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
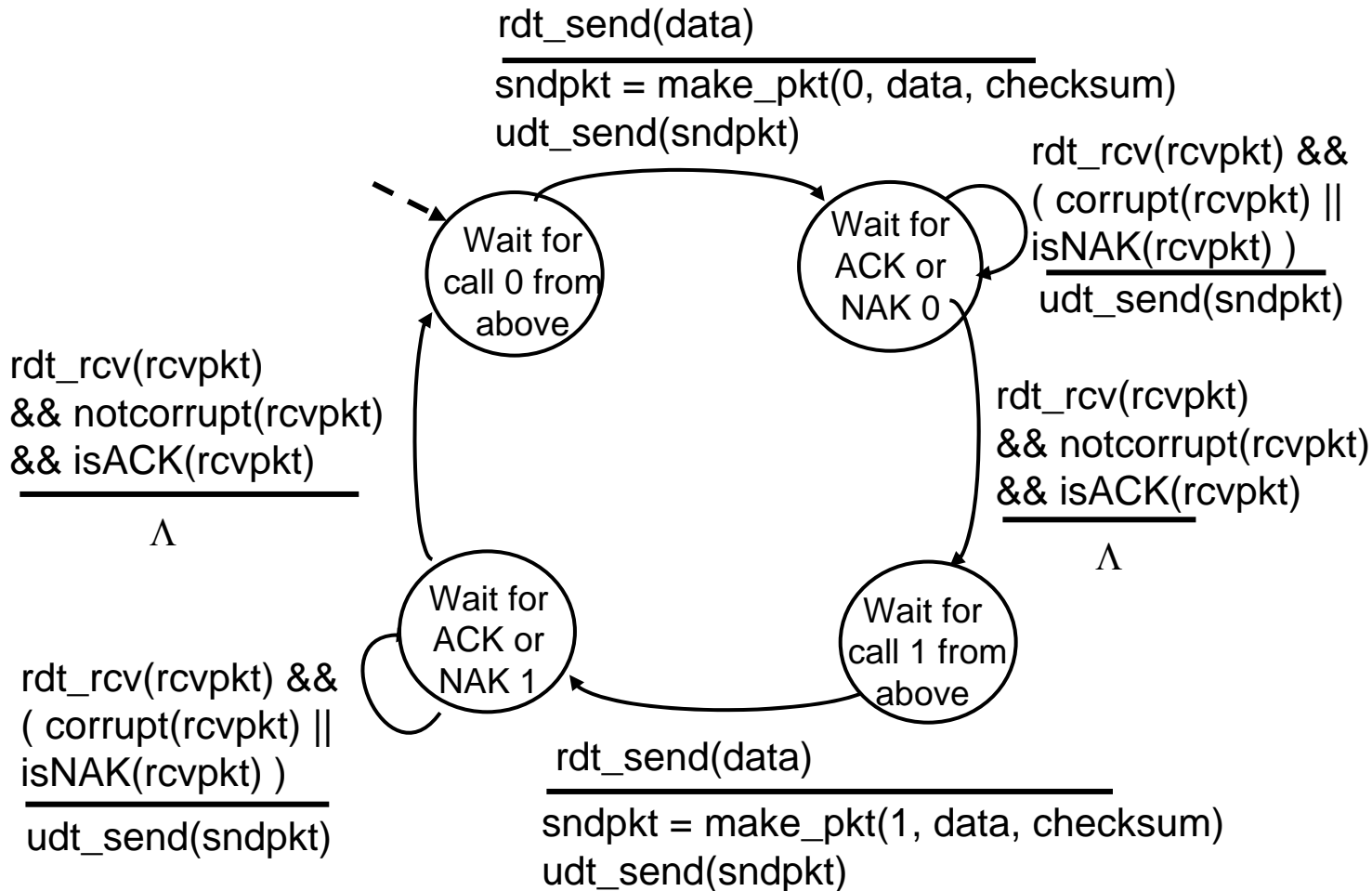
❖ receiver discards (doesn't deliver up) duplicate pkt

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

rdt2.1: sender, handles garbled ACK/NAKs

rdt_send(data)
_____
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)

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

Wait for call 0 from above

Wait for ACK or NAK 0

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)
_____
Λ

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)
_____
Λ

Wait for ACK or NAK 1

Wait for call 1 from above

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

rdt_send(data)
_____
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)

rdt2.1: receiver, handles garbled ACK/NAKs

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)
_____
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
_____
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
  not corrupt(rcvpkt) &&
  has_seq1(rcvpkt)
_____
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

Wait for 0 from below

Wait for 1 from below

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
_____
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
  not corrupt(rcvpkt) &&
  has_seq0(rcvpkt)
_____
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)
_____
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt2.1: discussion

sender:

❖ seq # added to pkt

❖ two seq. #'s (0,1) will suffice. Why?

❖ must check if received ACK/NAK corrupted

❖ twice as many states
- state must "remember" whether "expected" pkt should have seq # of 0 or 1

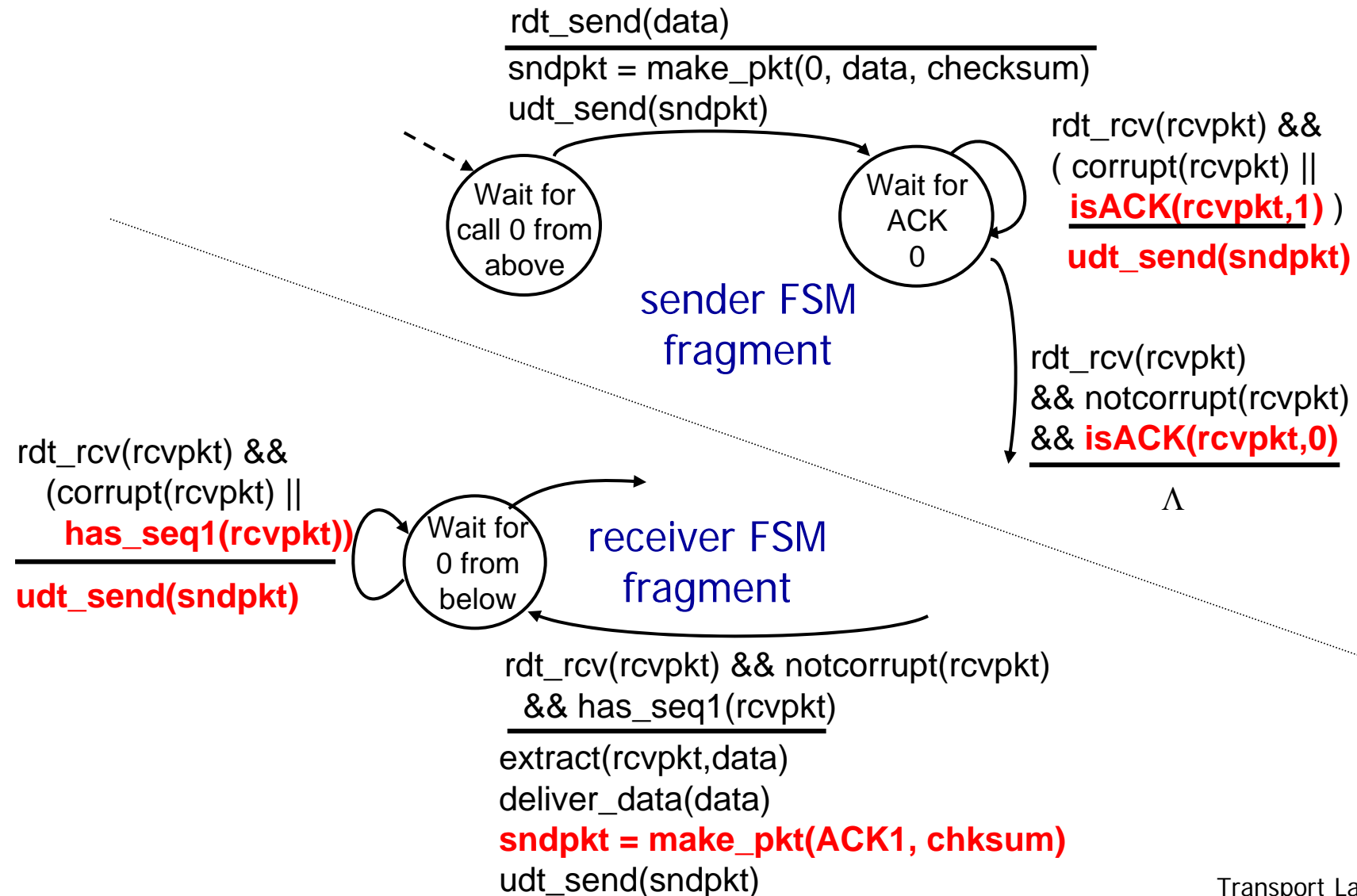**receiver:**

❖ must check if received packet is duplicate
- state indicates whether 0 or 1 is expected pkt seq #

❖ note: receiver can *not* know if its last ACK/NAK received OK at sender

rdt2.2: a NAK-free protocol

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

rdt2.2: sender, receiver fragments

rdt_send(data)
_____
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,1))

udt_send(sndpkt)

(Wait for call 0 from above)

(Wait for ACK 0)

sender FSM fragment

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& **isACK(rcvpkt,0)**
_____
Λ

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
has_seq1(rcvpkt))
_____
udt_send(sndpkt)

(Wait for 0 from below)

receiver FSM fragment

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)
_____
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK1, chksum)
udt_send(sndpkt)

rdt3.0: channels with errors *and* loss

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

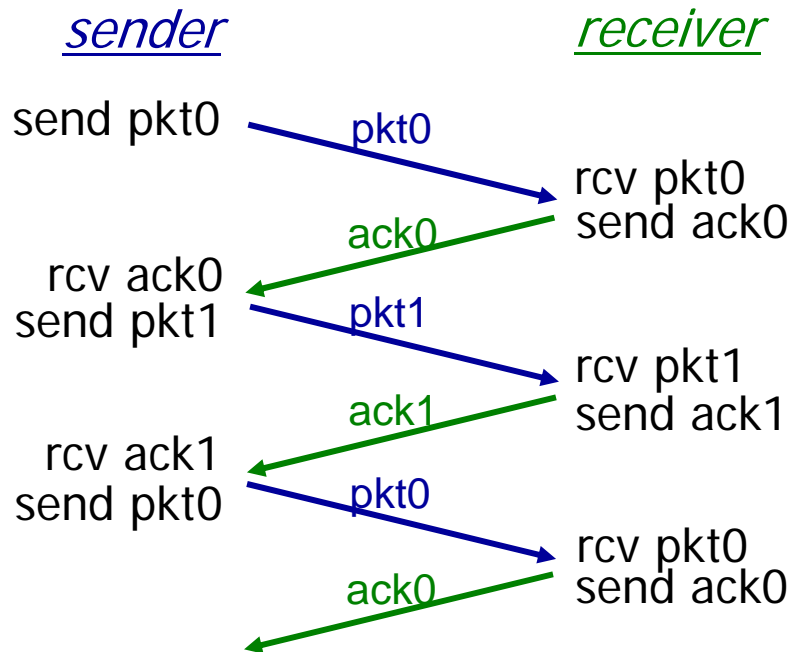- checksum, seq. #, ACKs, retransmissions will be of help … but not enough

approach: sender waits "reasonable" amount of time for ACK

❖ retransmits if no ACK received in this time

❖ if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq. #'s already handles this
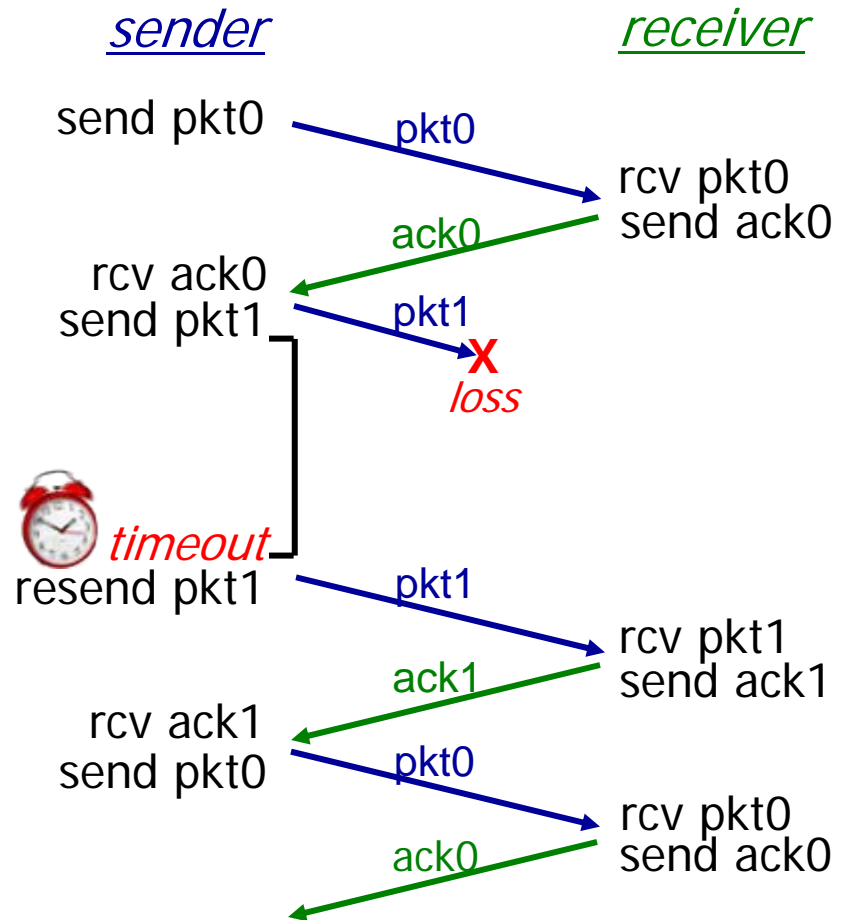 - receiver must specify seq # of pkt being ACKed

❖ requires countdown timer

rdt3.0 sender

rdt_send(data)
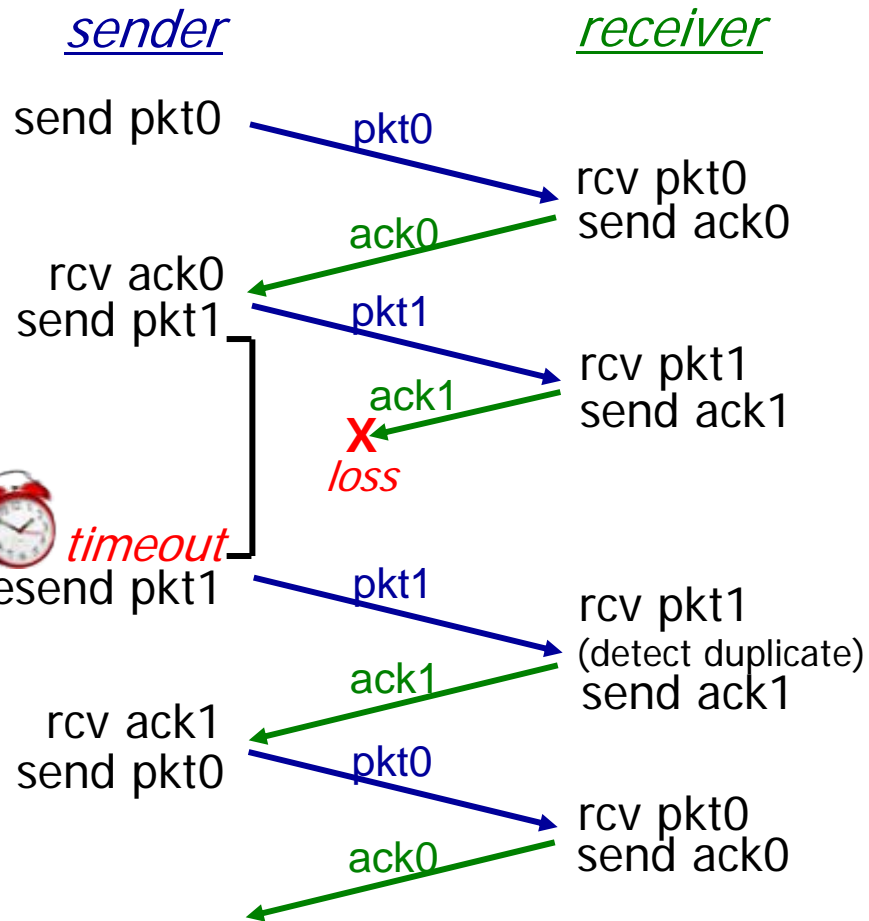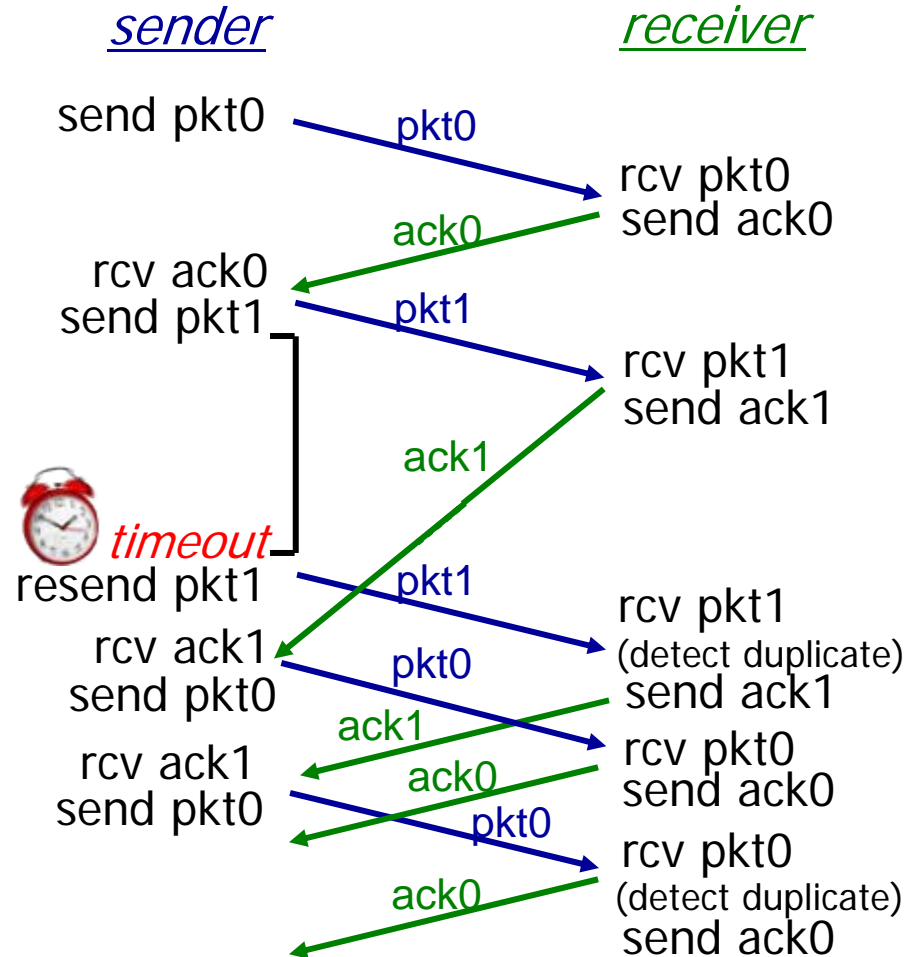_____
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,1))
_____
Λ

rdt_rcv(rcvpkt)
_____
Λ

Wait for call 0 from above

Wait for ACK0

timeout
_____
udt_send(sndpkt)
start_timer

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,1)
_____
stop_timer

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,0)
_____
stop_timer

timeout
_____
udt_send(sndpkt)
start_timer

Wait for ACK1

Wait for call 1 from above

rdt_rcv(rcvpkt)
_____
Λ

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,0))
_____
Λ

rdt_send(data)
_____
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer

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

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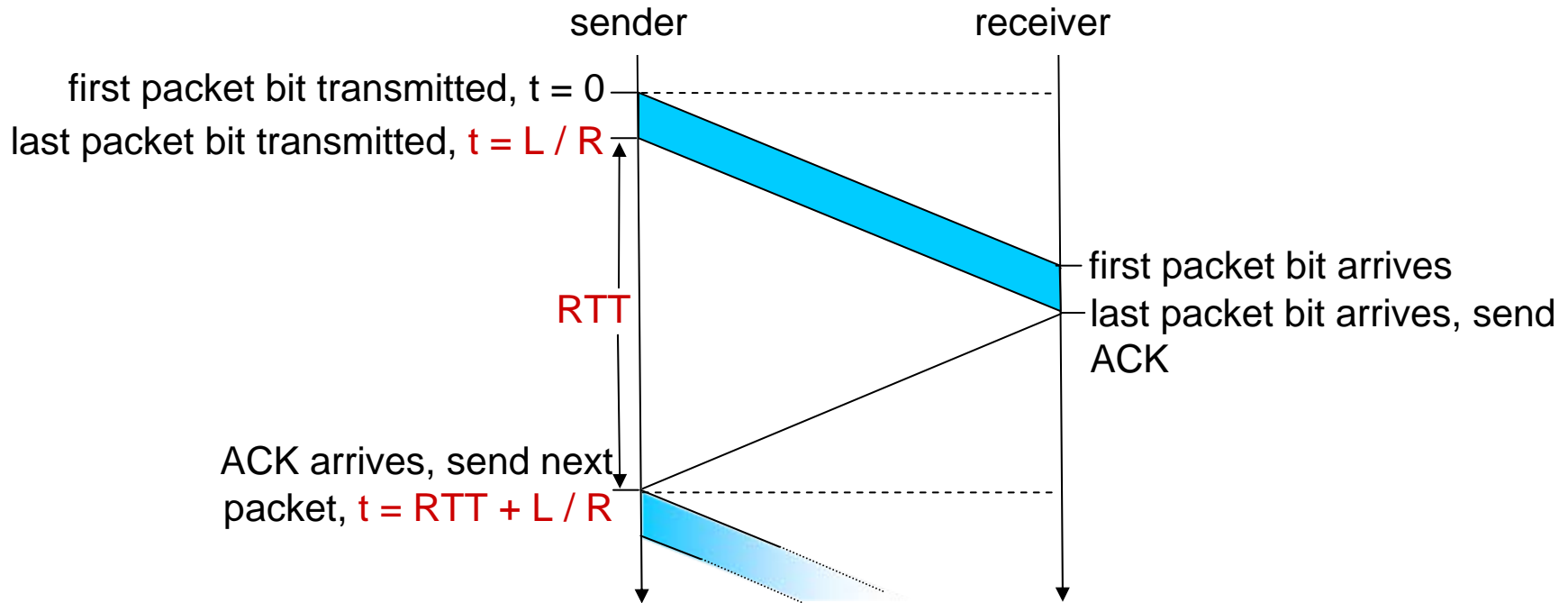❖ e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

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

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

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

■ if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec thruput over 1 Gbps link

❖ network protocol limits use of physical resources!
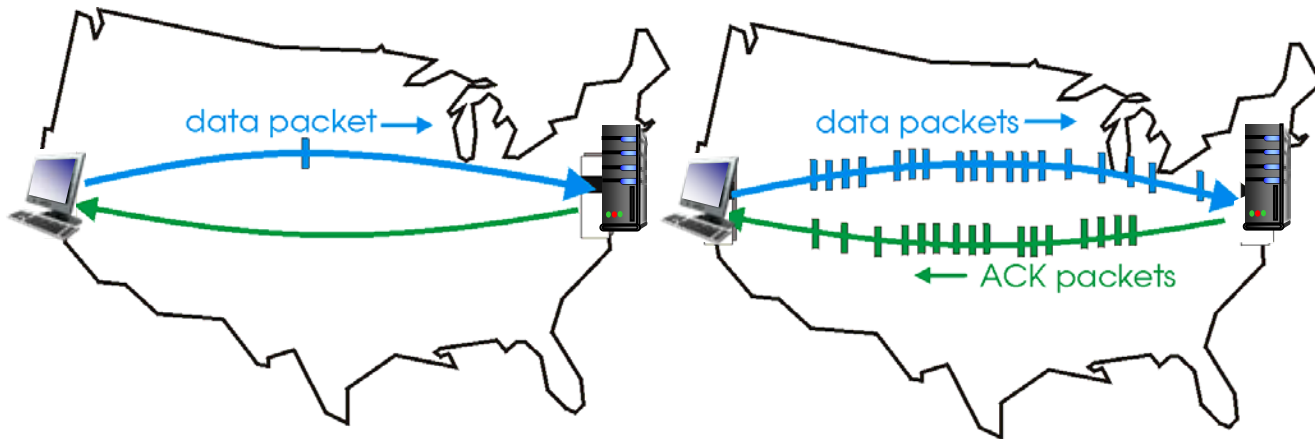
rdt3.0: stop-and-wait operation

sender   receiver

first packet bit transmitted, t = 0
last packet bit transmitted, t = L / R

RTT

first packet bit arrives

last packet bit arrives, send ACK

ACK arrives, send next packet, t = RTT + L / R

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

Pipelined protocols

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

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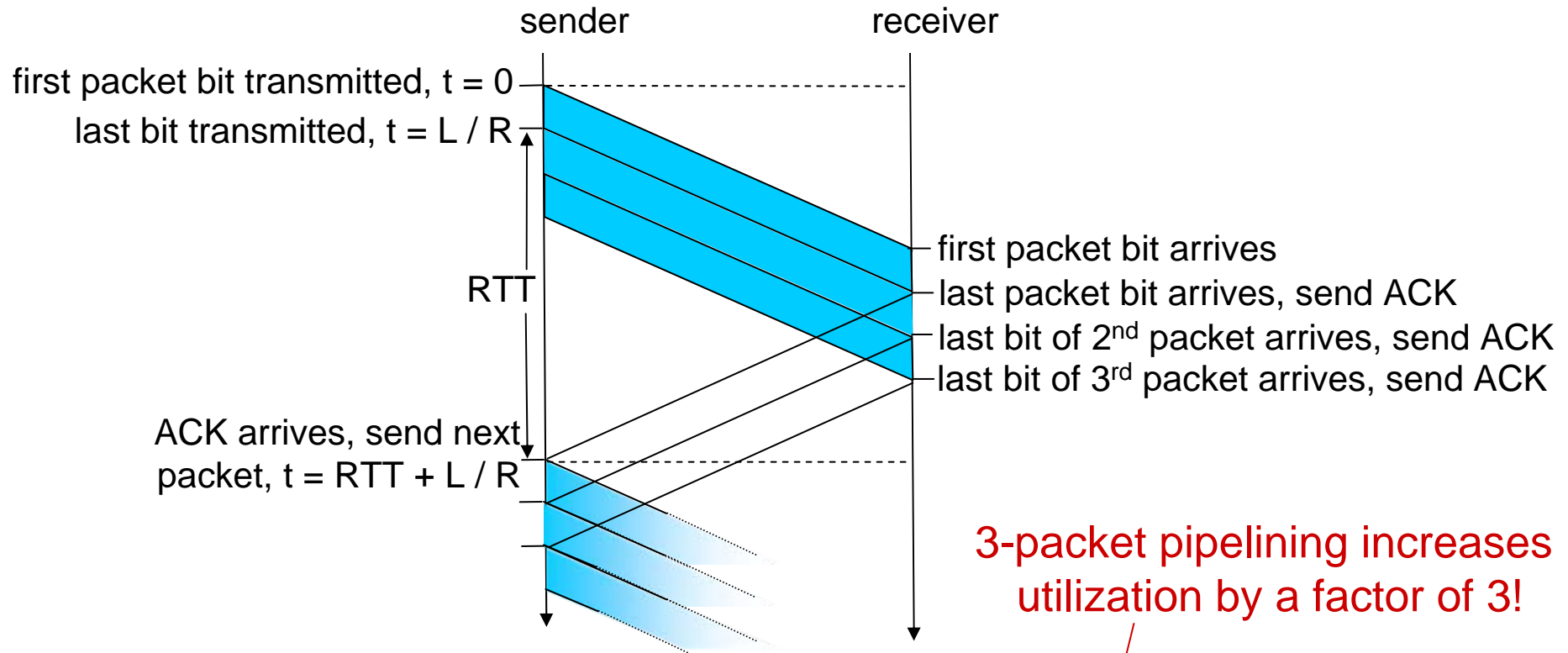- buffering at sender and/or receiver

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

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

Pipelining: increased utilization

sender   receiver

first packet bit transmitted, t = 0

last bit transmitted, t = L / R

first packet bit arrives

RTT

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!

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

Pipelined protocols: overview

Go-back-N:

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

Selective Repeat:

- ❖ sender can have up to N unack'ed packets in pipeline
- ❖ rcvr sends *individual ack* for each packet
- ❖ sender maintains timer for each unacked packet
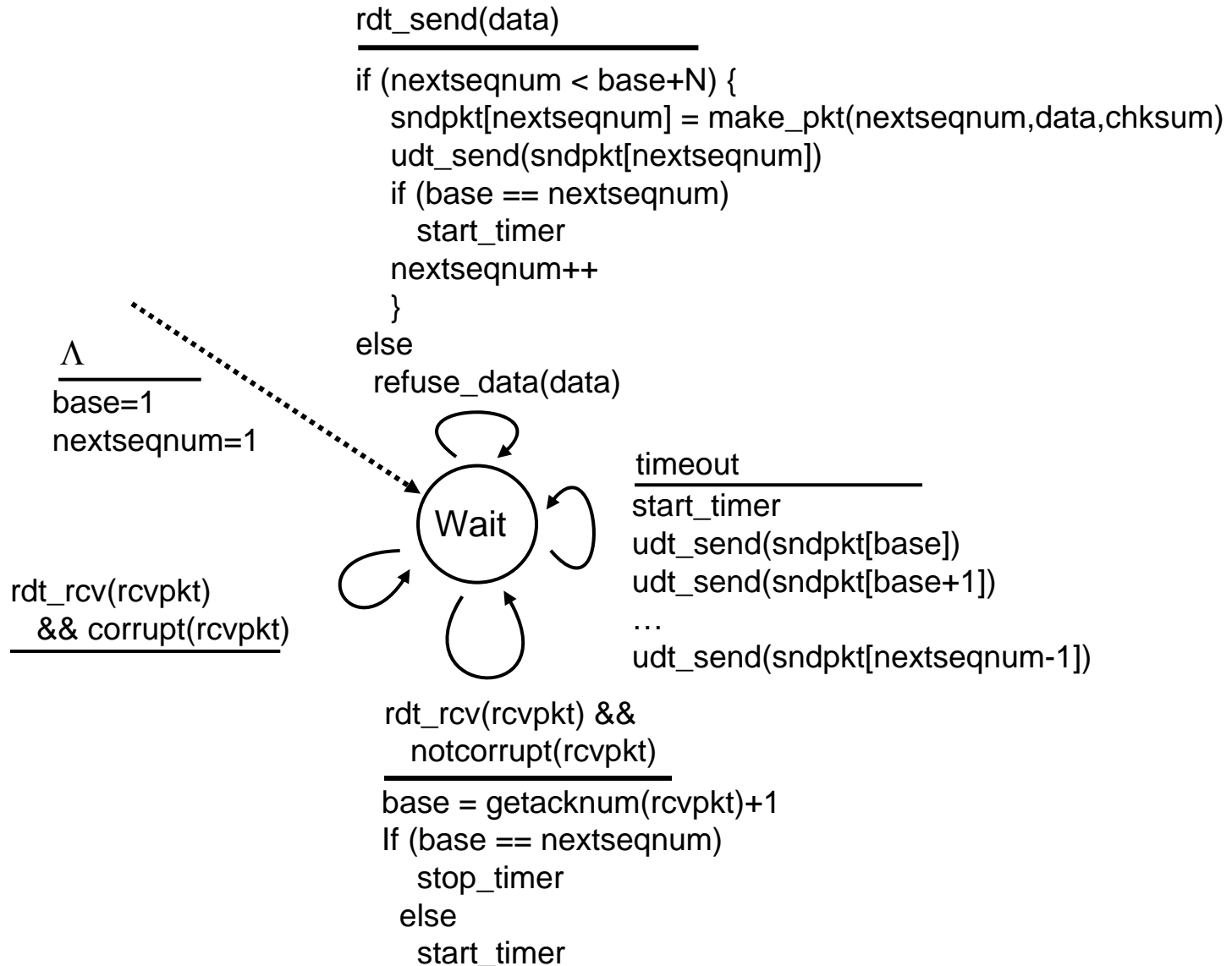 - ▪ when timer expires, retransmit only that unacked packet

Go-Back-N: sender

- ❖ k-bit seq # in pkt header
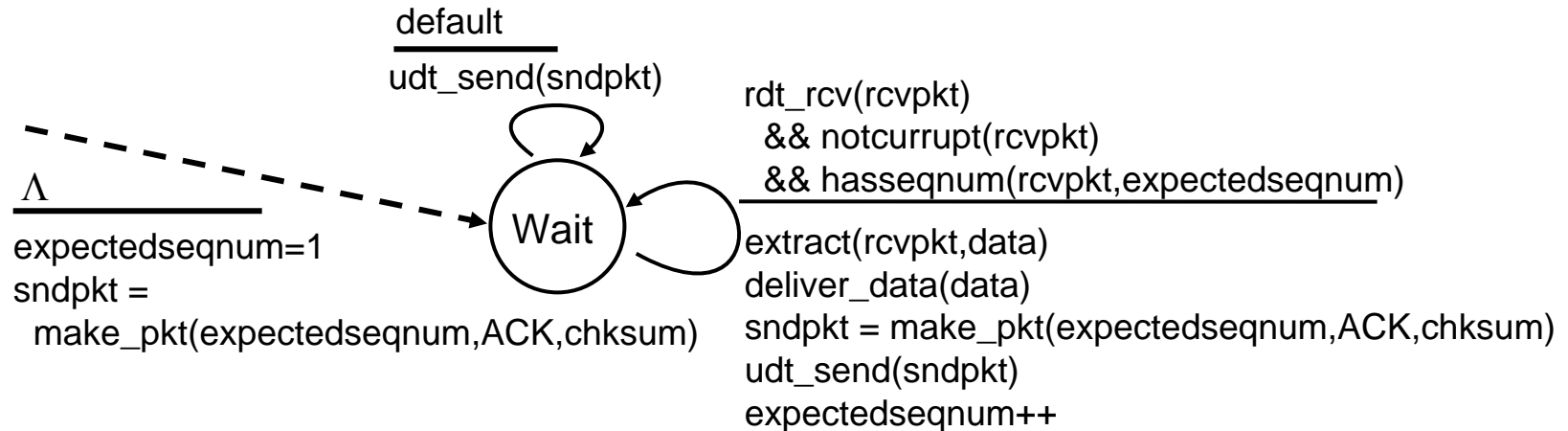- ❖ "window" of up to N, consecutive unack'ed pkts allowed
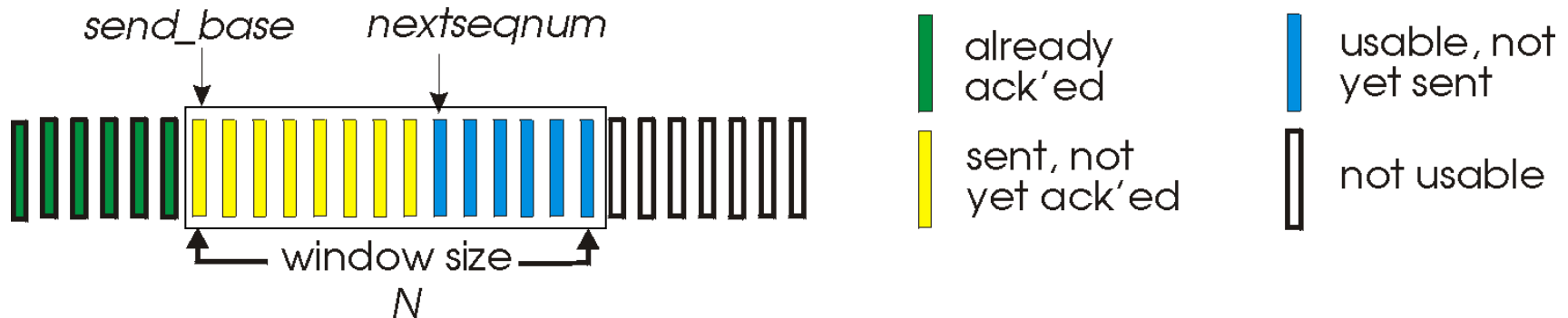
- ❖ ACK(n): ACKs all pkts up to, including seq # n - *"cumulative ACK"*
    - ■ may receive duplicate ACKs (see receiver)
- ❖ timer for oldest in-flight pkt
- ❖ *timeout(n):* retransmit packet n and all higher seq # pkts in window

GBN: sender extended FSM

rdt_send(data)
_____

```
if (nextseqnum < base+N) {
    sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
    udt_send(sndpkt[nextseqnum])
    if (base == nextseqnum)
        start_timer
    nextseqnum++
    }
else
    refuse_data(data)
```

Λ
_____
base=1
nextseqnum=1

Wait

timeout
_____
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])

rdt_rcv(rcvpkt)
&& corrupt(rcvpkt)
_____

rdt_rcv(rcvpkt) &&
    notcorrupt(rcvpkt)
_____
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
    stop_timer
  else
    start_timer

GBN: receiver extended FSM

default
_____
udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& notcurrupt(rcvpkt)
&& hasseqnum(rcvpkt,expectedseqnum)
_____

Λ
_____
expectedseqnum=1
sndpkt =
make_pkt(expectedseqnum,ACK,chksum)

Wait

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(expectedseqnum,ACK,chksum)
udt_send(sndpkt)
expectedseqnum++

ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- may generate duplicate ACKs
- need only remember `expectedseqnum`

❖ out-of-order pkt:

- discard (don't buffer): *no receiver buffering!*
- re-ACK pkt with highest in-order seq #

GBN in action

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 pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer

❖ sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt

❖ sender window
  - *N* consecutive seq #'s
  - limits seq #s of sent, unACKed pkts

Selective repeat: sender, receiver windows

send_base   nextseqnum

| | already ack'ed
| | sent, not yet ack'ed

| | usable, not yet sent
| | not usable

window size N

(a) sender view of sequence numbers

| | out of order (buffered) but already ack'ed
| | Expected, not yet received

| | acceptable (within window)
| | not usable

window size N

rcv_base

(b) receiver view of sequence numbers
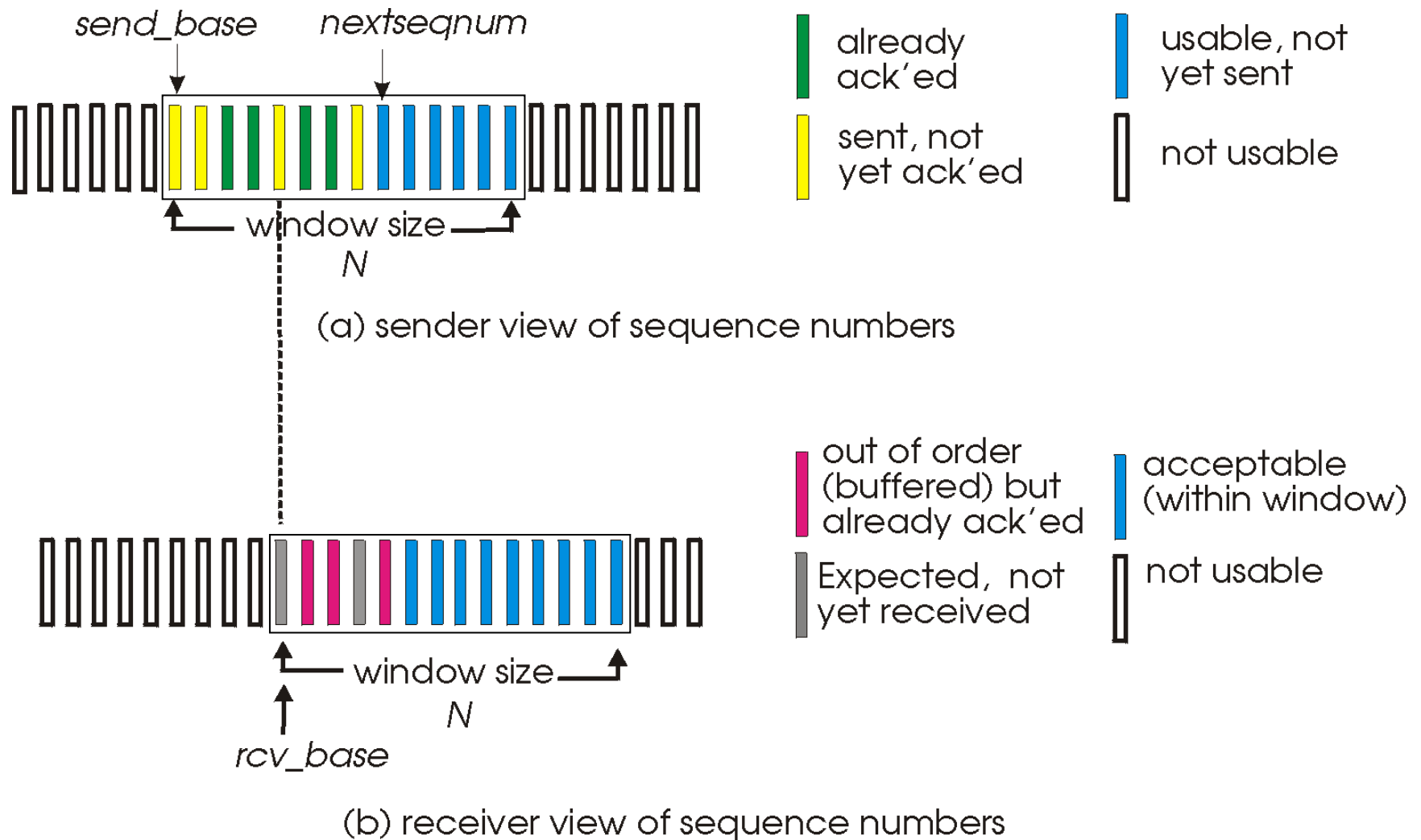
Selective repeat

sender

data from above:
- ❖ if next available seq # in window, send pkt

timeout(n):
- ❖ resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+N]:
- ❖ mark pkt n as received
- ❖ if n smallest unACKed pkt, advance window base to next unACKed seq #

receiver

pkt n in [rcvbase, rcvbase+N-1]
- ❖ send ACK(n)
- ❖ out-of-order: buffer
- ❖ in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

pkt n in [rcvbase-N,rcvbase-1]
- ❖ ACK(n)

otherwise:
- ❖ ignore

Selective repeat in action

sender window (N=4)   sender   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,
0 1 2 3 4 5 6 7 8   rcv ack0, send pkt4   send ack3
0 1 2 3 4 5 6 7 8   rcv ack1, send pkt5
                                                                receive pkt4, buffer,
                       record ack3 arrived   send ack4
                                                                receive pkt5, buffer,
                       pkt 2 timeout   send ack5
0 1 2 3 4 5 6 7 8   send pkt2
0 1 2 3 4 5 6 7 8   record ack4 arrived
0 1 2 3 4 5 6 7 8   record ack5 arrived   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: 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)?

sender window (after receipt)   receiver window (after receipt)
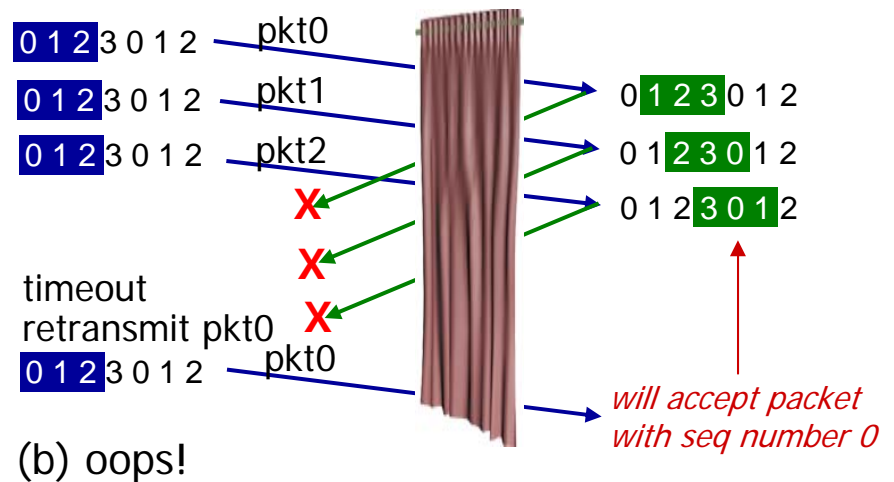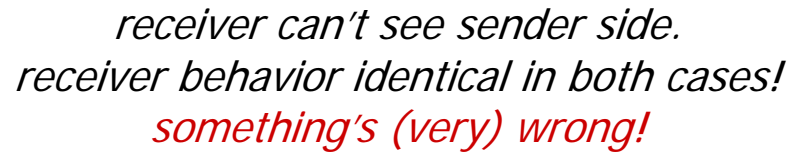
0 1 2 3 0 1 2   pkt0
0 1 2 3 0 1 2   pkt1   0 1 2 3 0 1 2
0 1 2 3 0 1 2   pkt2   0 1 2 3 0 1 2
                           0 1 2 3 0 1 2
0 1 2 3 0 1 2   pkt3
0 1 2 3 0 1 2   X
                pkt0   will accept packet
                           with seq number 0

(a) no problem

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

0 1 2 3 0 1 2   pkt0
0 1 2 3 0 1 2   pkt1   0 1 2 3 0 1 2
0 1 2 3 0 1 2   pkt2   0 1 2 3 0 1 2
                X   0 1 2 3 0 1 2
                X
timeout   X
retransmit pkt0
0 1 2 3 0 1 2   pkt0   will accept packet
                           with seq number 0

(b) oops!

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

TCP: Overview RFCs: 793,1122,1323, 2018, 2581

- ❖ point-to-point:
  - one sender, one receiver
- ❖ reliable, in-order *byte steam:*
  - no "message boundaries"
- ❖ pipelined:
  - TCP congestion and flow control set window size

- ❖ full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- ❖ connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- ❖ flow controlled:
  - sender will not overwhelm receiver

TCP segment structure

URG: urgent data (generally not used)

ACK: ACK # valid

PSH: push data now (generally not used)

RST, SYN, FIN: connection estab (setup, teardown commands)

Internet checksum (as in UDP)

32 bits

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

counting by bytes of data (not segments!)

bytes rcvr willing to accept

TCP seq. numbers, ACKs

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

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

Q: how receiver handles out-of-order segments
- A: TCP spec doesn't say, - up to implementor

outgoing segment from sender

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

window size
N

sender sequence number space

sent ACKed	sent, not-yet ACKed ("in-flight")	usable but not yet sent	not usable
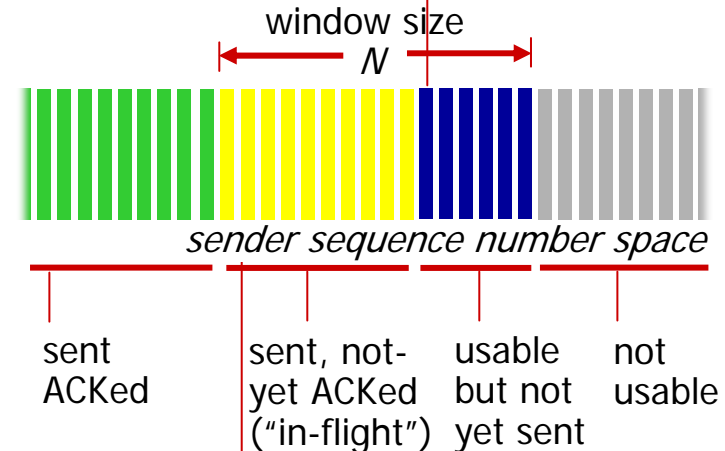
incoming segment to sender

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

TCP seq. numbers, ACKs

Host A   Host B

User
types
'C'
                Seq=42, ACK=79, data = 'C'
                                                host ACKs
                                                receipt of
                                                'C', echoes
                Seq=79, ACK=43, data = 'C'   back 'C'
host ACKs
receipt
of echoed
'C'   Seq=43, ACK=80
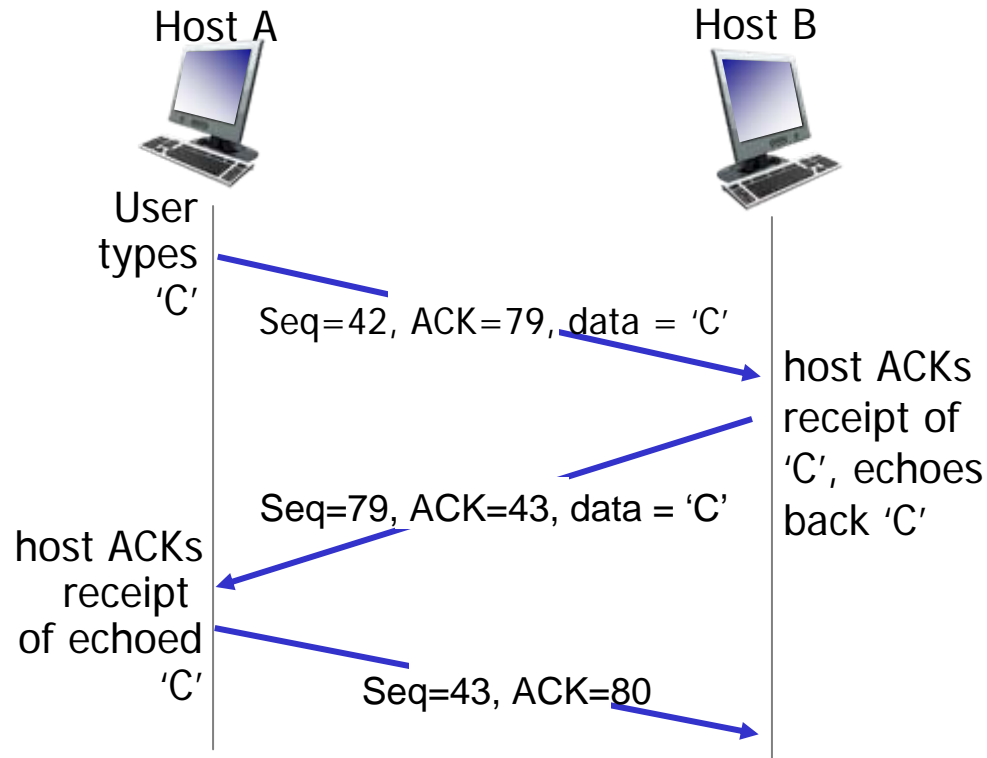
simple telnet scenario

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

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

- ❖ exponential weighted moving average
- ❖ influence of past sample decreases exponentially fast
- ❖ typical value: $\alpha = 0.125$

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr

RTT (milliseconds)

time (seconds)

sampleRTT
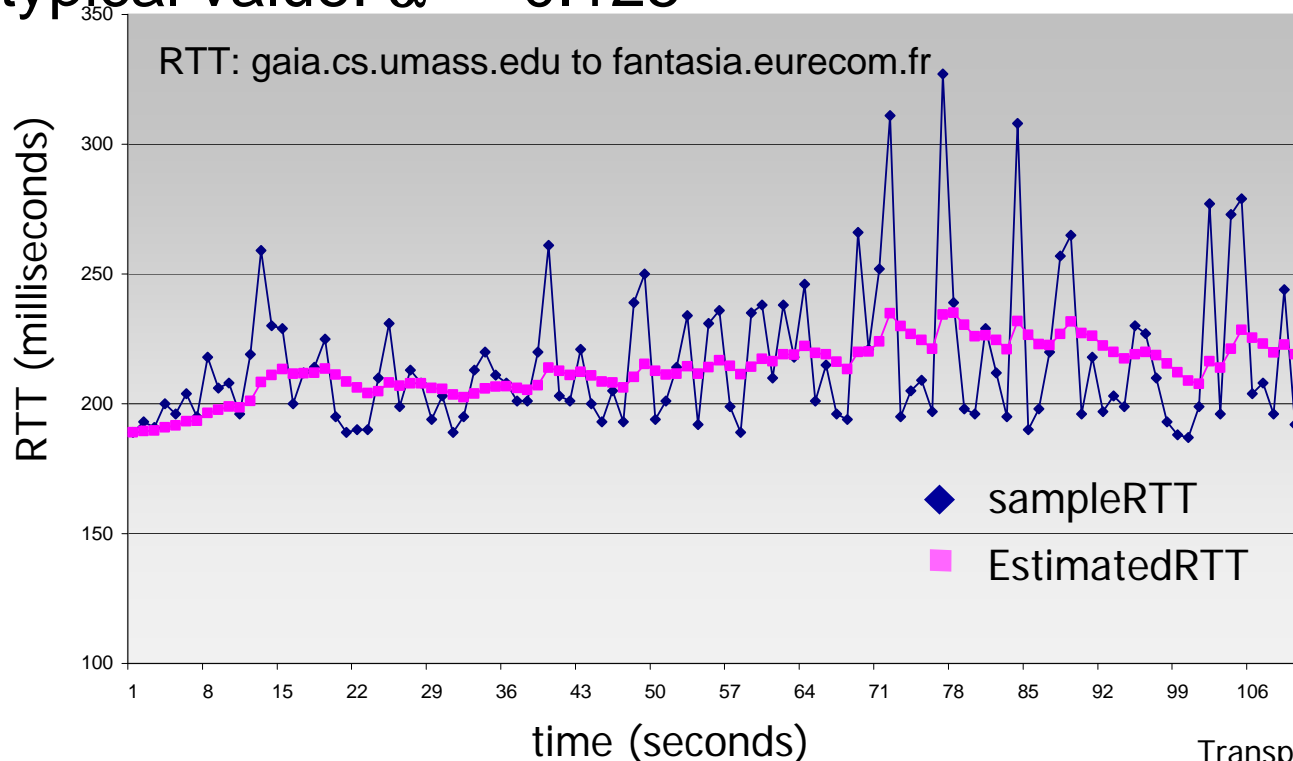
EstimatedRTT

TCP round trip time, timeout

❖ **timeout interval:** `EstimatedRTT` plus "safety margin"
  ▪ large variation in `EstimatedRTT` -> larger safety margin

❖ estimate SampleRTT deviation from EstimatedRTT:

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

$$\texttt{(typically, }\beta\texttt{ = 0.25)}$$

`TimeoutInterval = EstimatedRTT + 4*DevRTT`

estimated RTT   "safety margin"

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

TCP reliable data transfer

❖ **TCP creates rdt service on top of IP's unreliable service**
  - pipelined segments
  - cumulative acks
  - single retransmission timer

❖ **retransmissions triggered by:**
  - timeout events
  - duplicate acks

let's initially consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control

TCP sender events:

data rcvd from app:

- ❖ create segment with seq #
- ❖ seq # is byte-stream number of first data byte in segment
- ❖ start timer if not already running
  - ▪ think of timer as for oldest unacked segment
  - ▪ expiration interval: `TimeOutInterval`

timeout:

- ❖ retransmit segment that caused timeout
- ❖ restart timer

ack rcvd:

- ❖ if ack acknowledges previously unacked segments
  - ▪ update what is known to be ACKed
  - ▪ start timer if there are still unacked segments

TCP sender (simplified)

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

Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

wait
for
event

timeout

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

ACK received, with ACK field value y

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

TCP: retransmission scenarios

lost ACK scenario
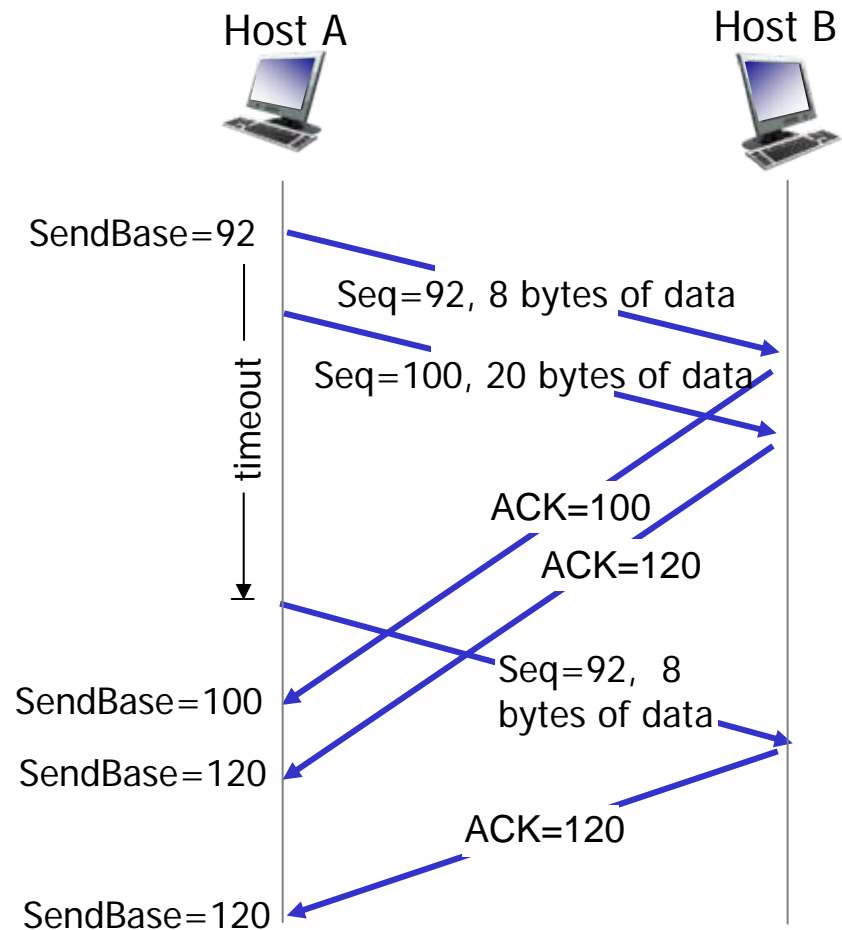
premature timeout

TCP: retransmission scenarios

Host A   Host B

Seq=92, 8 bytes of data

Seq=100, 20 bytes of data

ACK=100

X

ACK=120

timeout

Seq=120, 15 bytes of data
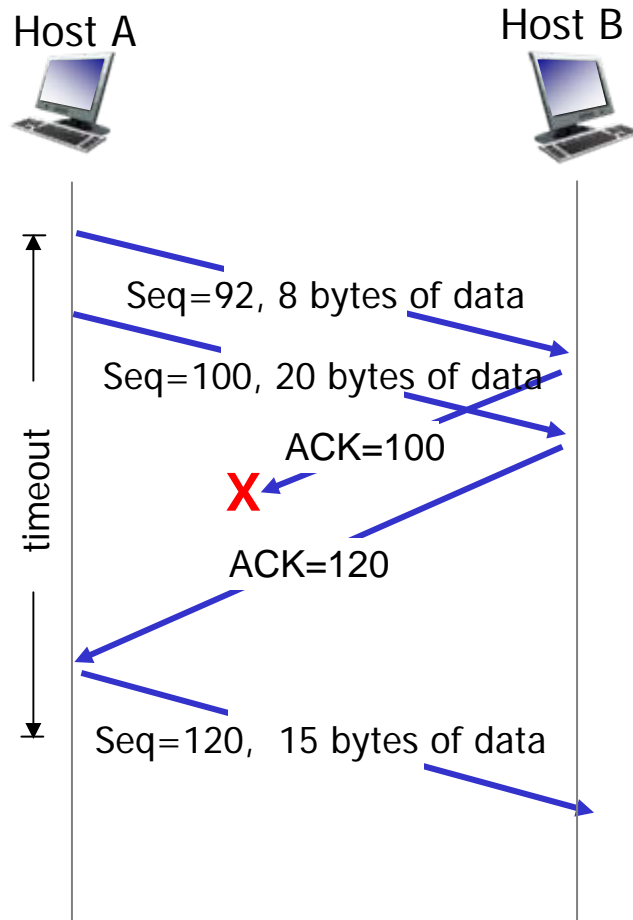
cumulative ACK

TCP ACK generation [RFC 1122, RFC 2581]

event at receiver	TCP receiver action
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send *duplicate ACK*, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

TCP fast retransmit

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

TCP fast retransmit

if sender receives 3 ACKs for same data ("triple duplicate ACKs"), resend unacked segment with smallest seq #
  ▪ likely that unacked segment lost, so don't wait for timeout

TCP fast retransmit
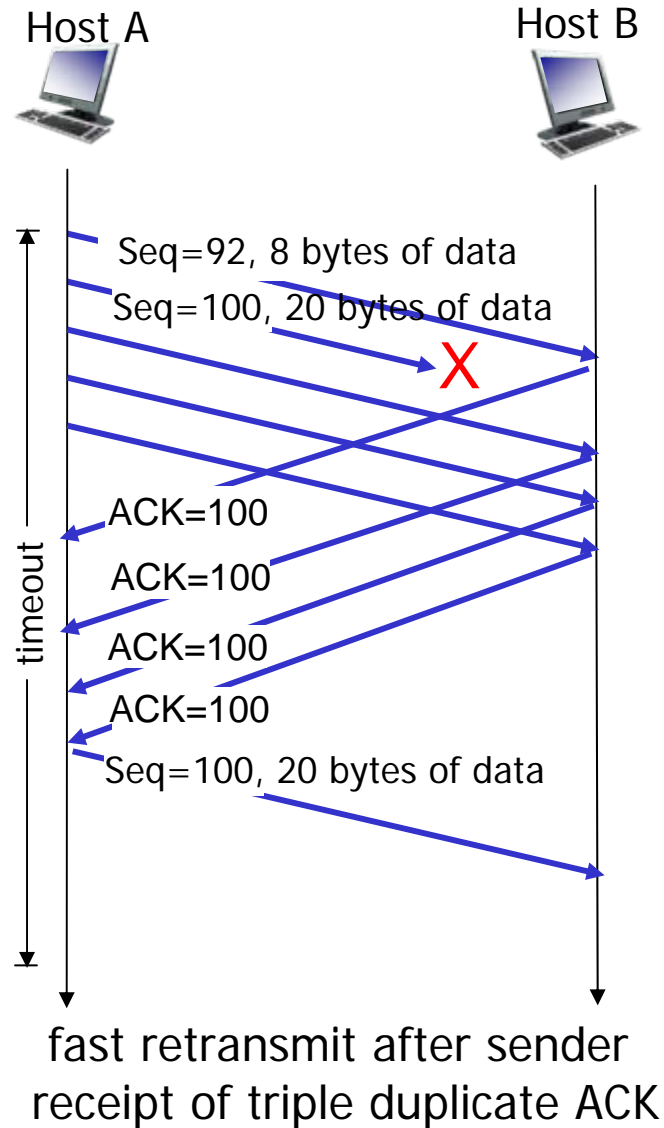
Host A   Host B

Seq=92, 8 bytes of data
Seq=100, 20 bytes of data

X

timeout

ACK=100

ACK=100

ACK=100

ACK=100

Seq=100, 20 bytes of data

fast retransmit after sender
receipt of triple duplicate ACK

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

TCP flow control

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

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

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

application
process

application
OS

TCP socket
receiver buffers

TCP
code

IP
code

from sender

receiver protocol stack
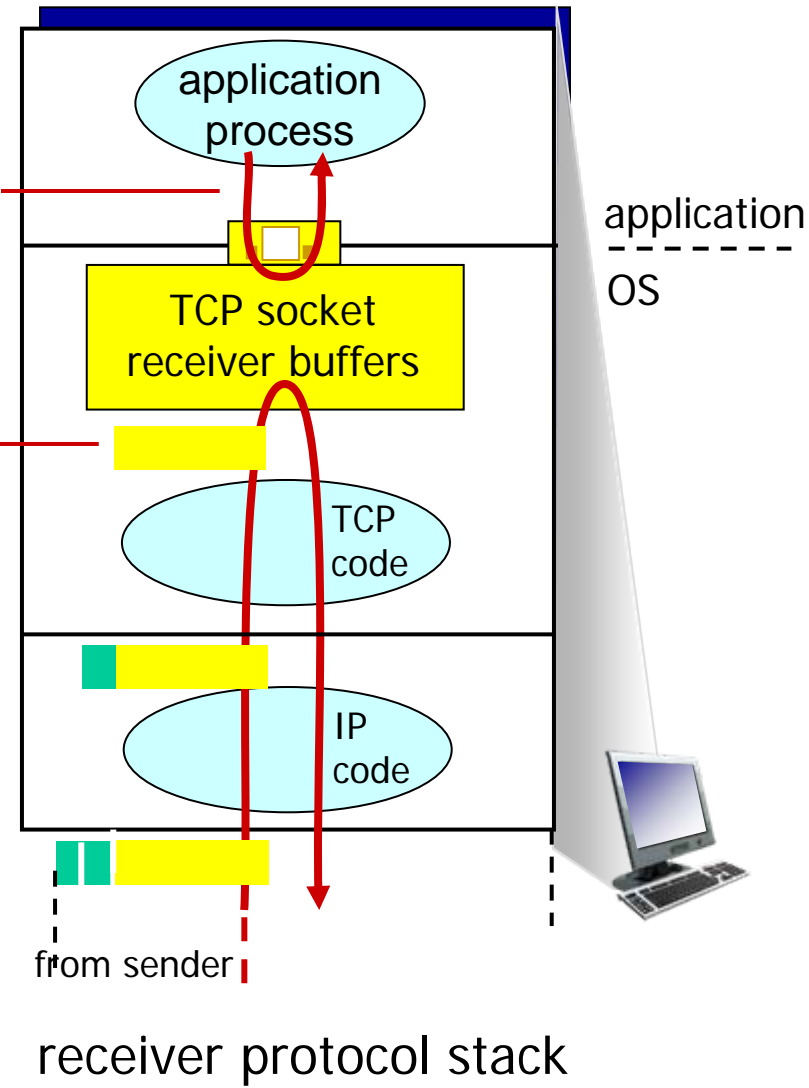
TCP flow control

❖ receiver "advertises" free buffer space by including `rwnd` value in TCP header of receiver-to-sender segments

 ▪ `RcvBuffer` size set via socket options (typical default is 4096 bytes)

 ▪ many operating systems autoadjust `RcvBuffer`

❖ sender limits amount of unacked ("in-flight") data to receiver's `rwnd` value

❖ guarantees receive buffer will not overflow
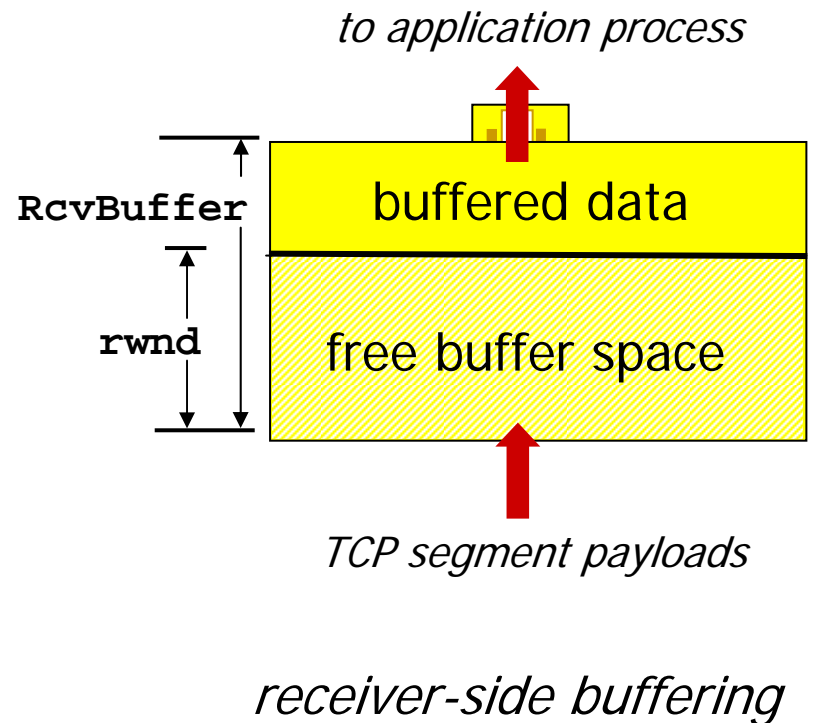
to application process

`RcvBuffer`

buffered data

`rwnd`

free buffer space

TCP segment payloads

receiver-side buffering

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

Connection Management

before exchanging data, sender/receiver "handshake":

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

❖ agree on connection parameters

application		application

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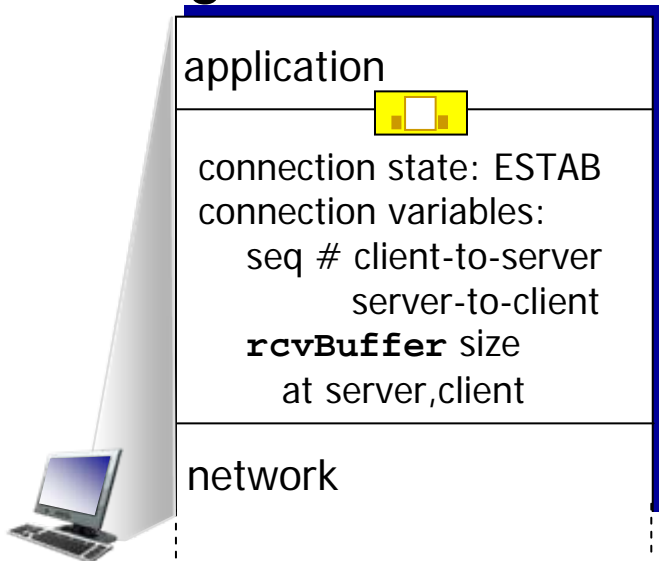
network		network

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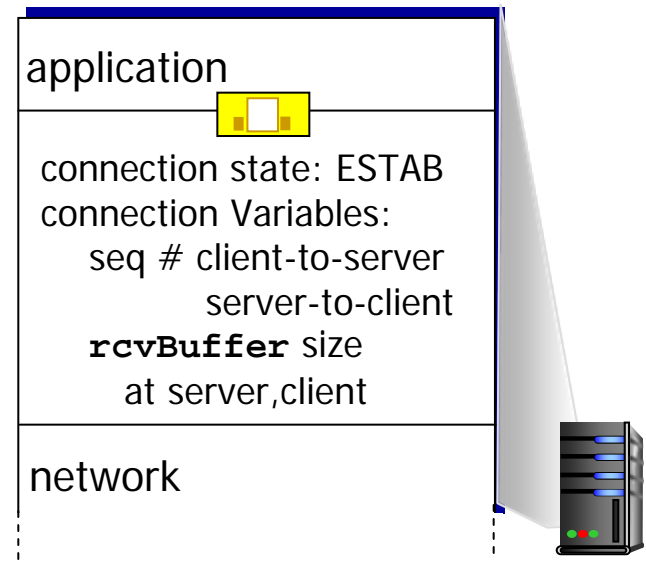
```
Socket connectionSocket =
  welcomeSocket.accept();
```

Agreeing to establish a connection
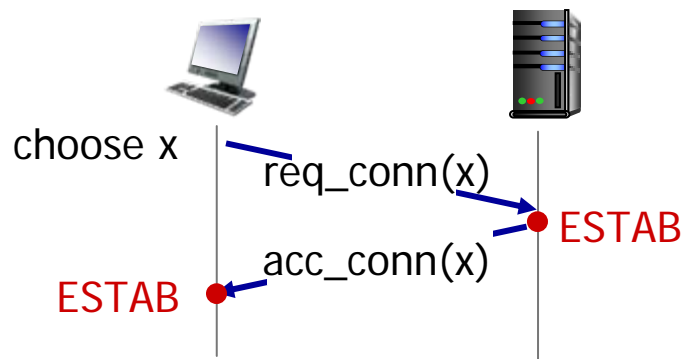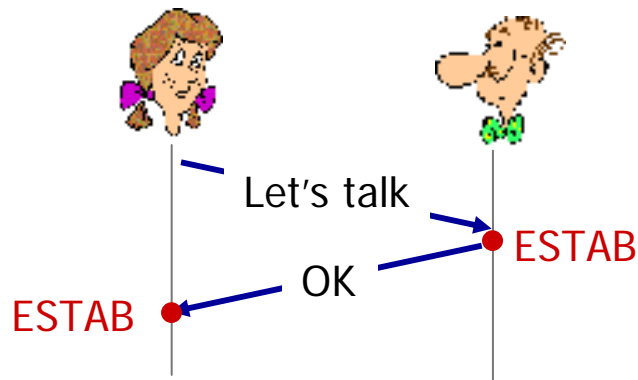
2-way handshake:

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:

Left diagram:

choose x → req_conn(x) → ESTAB

retransmit req_conn(x), acc_conn(x)

ESTAB

req_conn(x)

client terminates — connection x completes — server forgets x

ESTAB

half open connection!
(no client!)

Right diagram:

choose x → req_conn(x) → ESTAB

retransmit req_conn(x), acc_conn(x)

ESTAB

data(x+1) → accept data(x+1)

retransmit data(x+1)

client terminates — connection x completes — server forgets x

req_conn(x)

data(x+1) → ESTAB accept data(x+1)

TCP 3-way handshake

client state

LISTEN

SYNSENT

ESTAB

server state

LISTEN

SYN RCVD

ESTAB

choose init seq num, x
send TCP SYN msg

SYNbit=1, Seq=x

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

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

ACKbit=1, ACKnum=y+1

received ACK(y)
indicates client is live

TCP 3-way handshake: FSM

closed

Socket connectionSocket =
welcomeSocket.accept();
—————————————————
Λ

Socket clientSocket =
newSocket("hostname","port
number");
—————————————————
SYN(seq=x)

SYN(x)
—————————————————
SYNACK(seq=y,ACKnum=x+1)
create new socket for
communication back to client

listen

SYN rcvd

SYN sent

ESTAB

SYNACK(seq=y,ACKnum=x+1)
—————————————————
ACK(ACKnum=y+1)

ACK(ACKnum=y+1)
—————————————————
Λ

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

client state

server state

ESTAB

ESTAB

clientSocket.close()

FINbit=1, seq=x

FIN_WAIT_1

can no longer
send but can
receive data

CLOSE_WAIT

ACKbit=1; ACKnum=x+1

FIN_WAIT_2

wait for server
close

can still
send data

LAST_ACK

FINbit=1, seq=y

TIMED_WAIT

can no longer
send data

ACKbit=1; ACKnum=y+1

timed wait
for 2*max
segment lifetime

CLOSED

CLOSED

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 1

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

original data: λ_{in}

throughput: λ_{out}

Host A

Host B

unlimited shared output link buffers
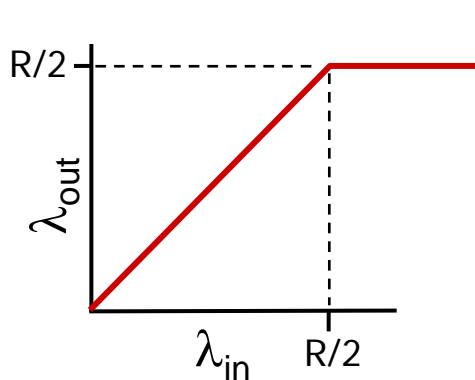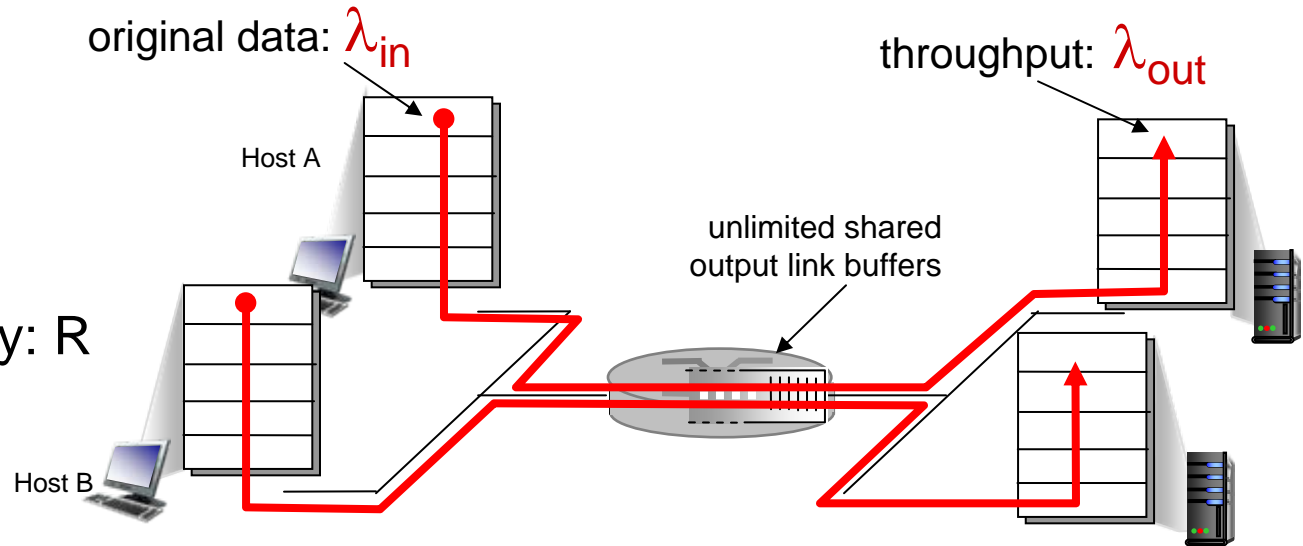
- ❖ maximum per-connection throughput: R/2
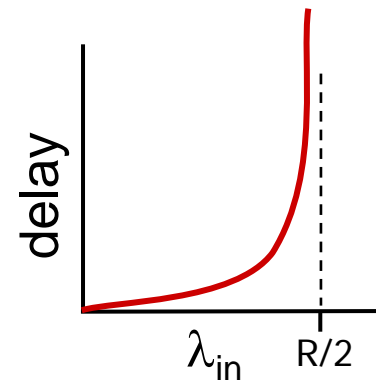
- ❖ large delays as arrival rate, λ_{in}, approaches capacity

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*: $\lambda'_{in} \geq \lambda_{in}$

λ_{in} : original data

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

λ_{out}

Host A

Host B

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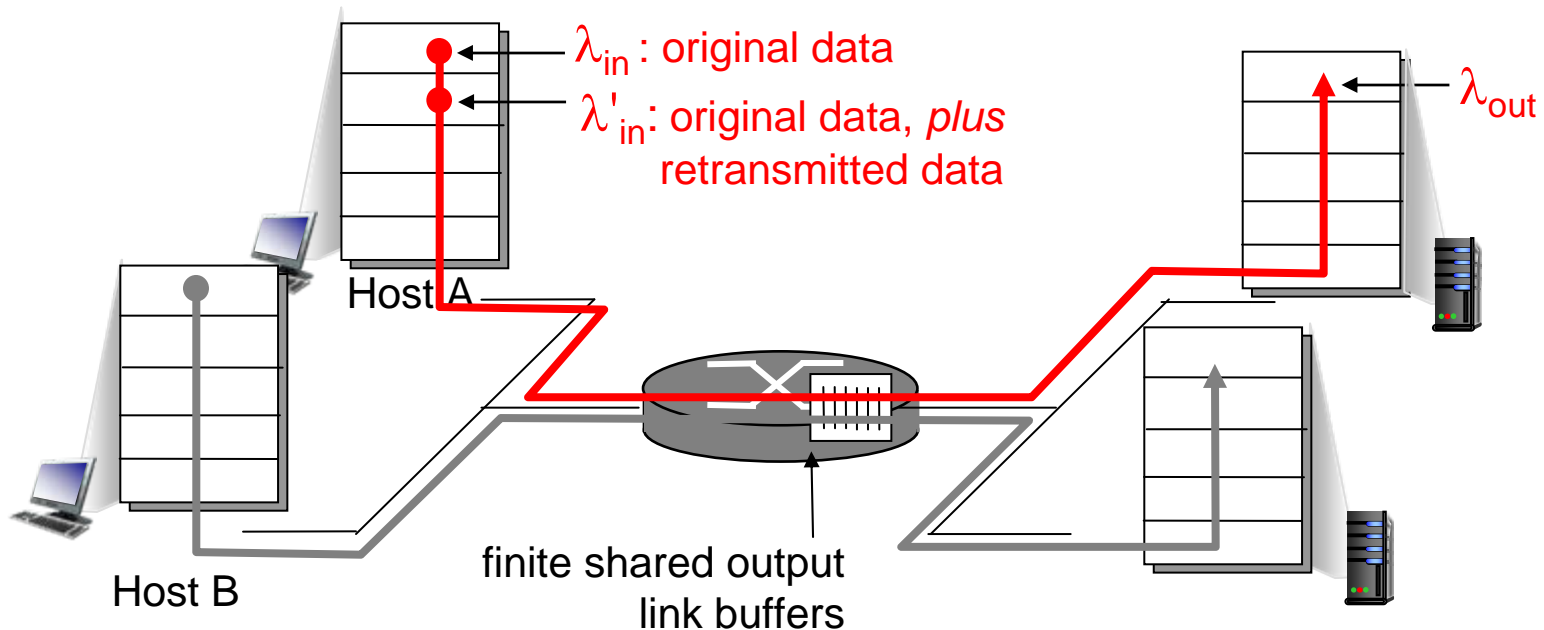
Causes/costs of congestion: scenario 2

idealization: perfect knowledge

❖ sender sends only when router buffers available

λ_{in} : original data

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

copy

λ_{out}

A

free buffer space!

Host B

finite shared output link buffers

Causes/costs of congestion: scenario 2

Idealization: known loss packets can be lost, dropped at router due to full buffers

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

λ_{in} : original data

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

copy

λ_{out}

A

Host B

no buffer space!

Causes/costs of congestion: scenario 2

Idealization: known loss packets can be lost, dropped at router due to full buffers

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

when sending at R/2, some packets are retransmissions but asymptotic goodput is still R/2 (why?)

λ_{in} : original data

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

λ_{out}

A

free buffer space!

Host B

Causes/costs of congestion: scenario 2

Realistic: duplicates

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

* sender times out prematurely, sending *two* copies, both of which are delivered

when sending at R/2, some packets are retransmissions including duplicated that are delivered!

λ_{in}

λ'_{in}

timeout

λ_{out}

A

free buffer space!

Host B

Causes/costs of congestion: scenario 2

Realistic: duplicates

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

❖ sender times out prematurely, sending *two* copies, both of which are delivered

λ_{out}

R/2

λ'_{in} R/2

when sending at R/2, some packets are retransmissions including duplicated that are delivered!

"costs" of congestion:

❖ more work (retrans) for given "goodput"

❖ unneeded retransmissions: link carries multiple copies of pkt

  ▪ decreasing goodput
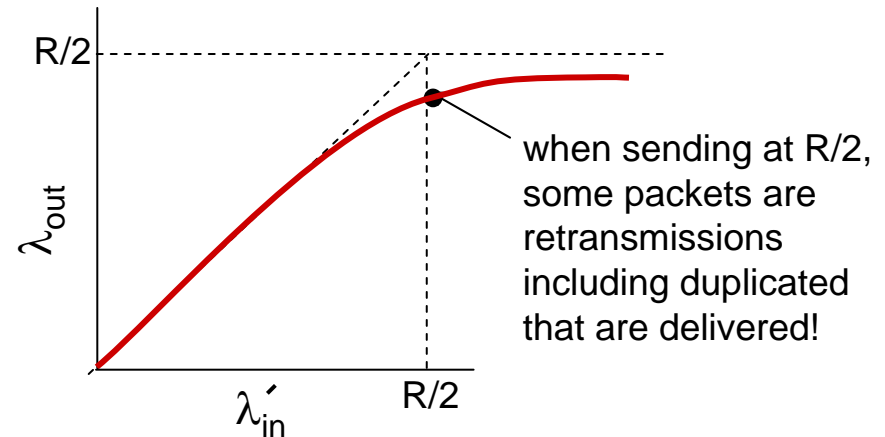
Causes/costs of congestion: scenario 3

* four senders
* multihop paths
* timeout/retransmit

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

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

Host A

λ_{in} : original data

λ_{in}': original data, *plus* retransmitted data

λ_{out}

Host B

finite shared output link buffers

Host D

Host C
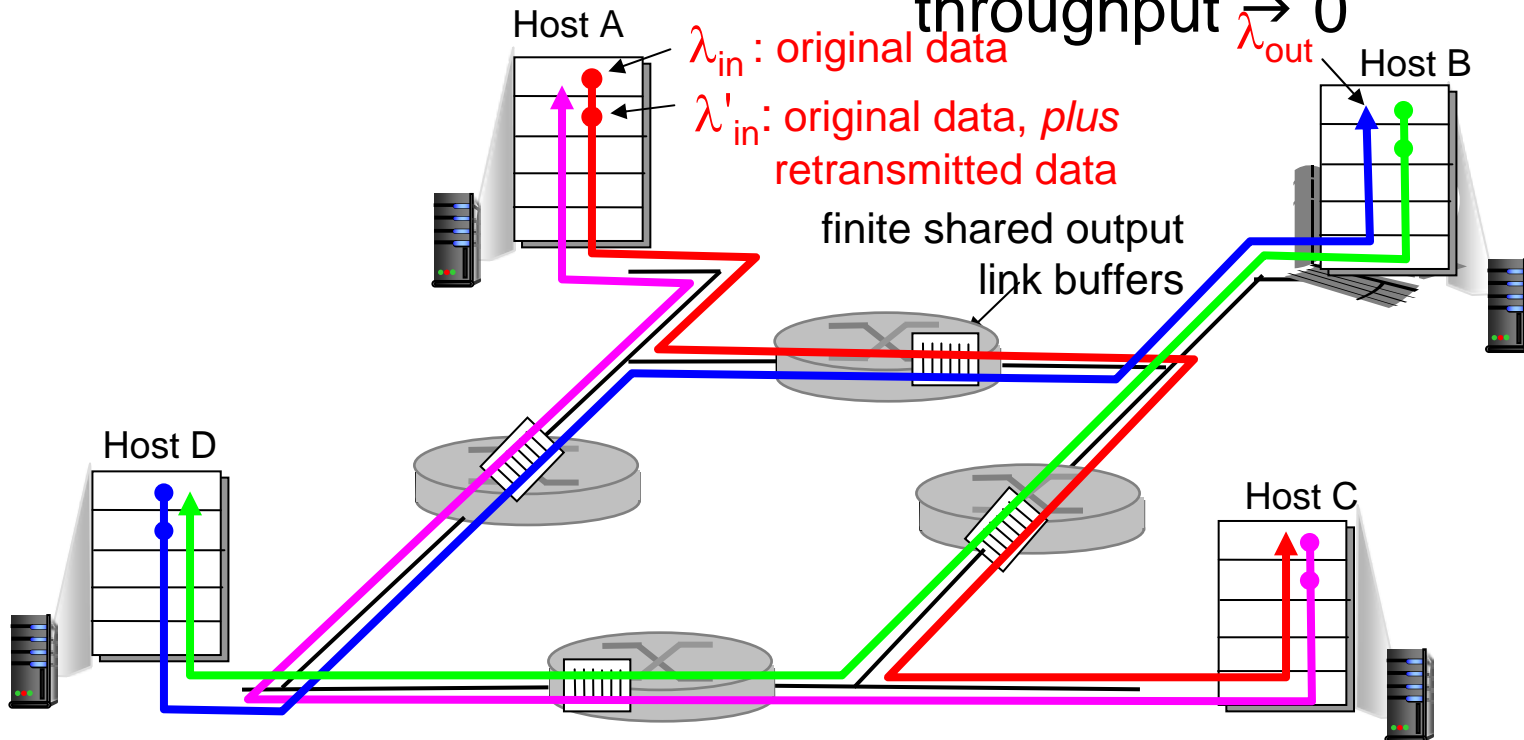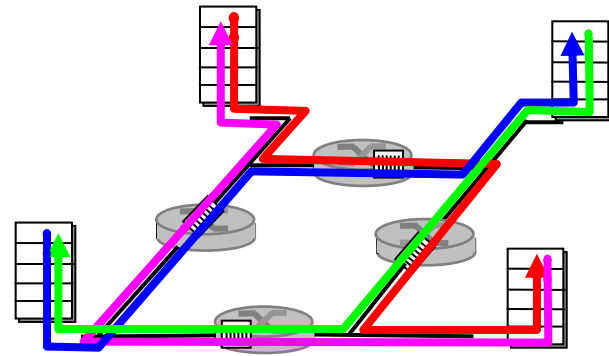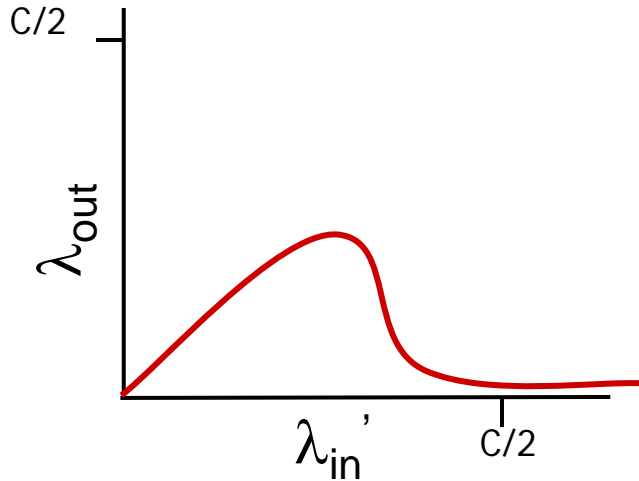
another "cost" of congestion:

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

Approaches towards congestion control

two broad approaches towards congestion control:

end-end congestion control:

- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

network-assisted congestion control:

- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate for sender to send at

Case study: ATM ABR congestion control

ABR: available bit rate:

❖ "elastic service"
❖ if sender's path "underloaded":
  ▪ sender should use available bandwidth
❖ if sender's path congested:
  ▪ sender throttled to minimum guaranteed rate

RM (resource management) cells:

❖ sent by sender, interspersed with data cells
❖ bits in RM cell set by switches ("*network-assisted*")
  ▪ *NI bit:* no increase in rate (mild congestion)
  ▪ *CI bit:* congestion indication
❖ RM cells returned to sender by receiver, with bits intact

Case study: ATM ABR congestion control
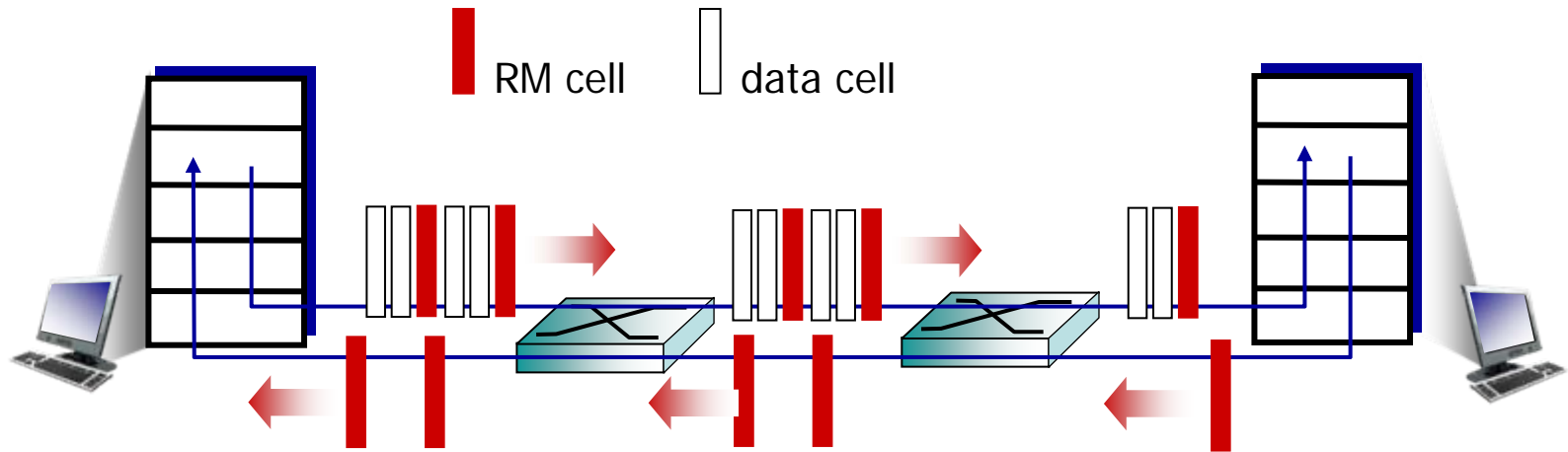
RM cell   data cell

- ❖ **two-byte ER (explicit rate) field in RM cell**
  - ▪ congested switch may lower ER value in cell
  - ▪ senders' send rate thus max supportable rate on path
- ❖ **EFCI bit in data cells: set to 1 in congested switch**
  - ▪ if data cell preceding RM cell has EFCI set, receiver sets CI bit in returned RM cell

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management

3.6 principles of congestion control

3.7 TCP congestion control

TCP congestion control: additive increase multiplicative decrease

❖ *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs

   ▪ *additive increase:* increase `cwnd` by 1 MSS every RTT until loss detected

   ▪ *multiplicative decrease*: cut `cwnd` in half after loss

AIMD saw tooth behavior: probing for bandwidth

additively increase window size ...

.... until loss occurs (then cut window in half)

`cwnd:` TCP sender congestion window size

time
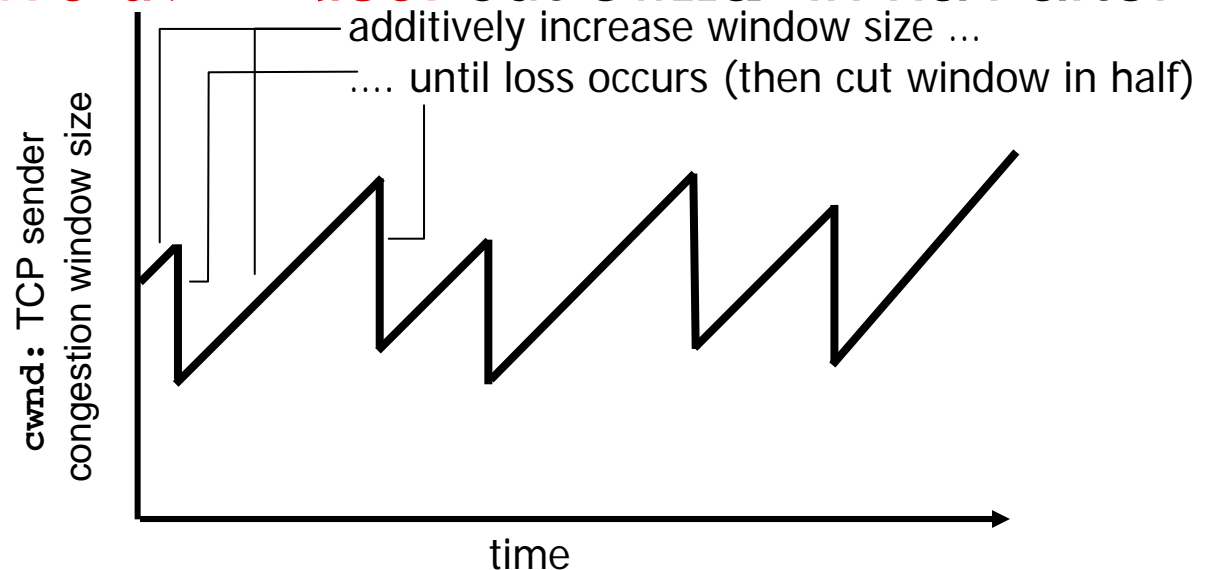
TCP Congestion Control: details

sender sequence number space

last byte ACKed

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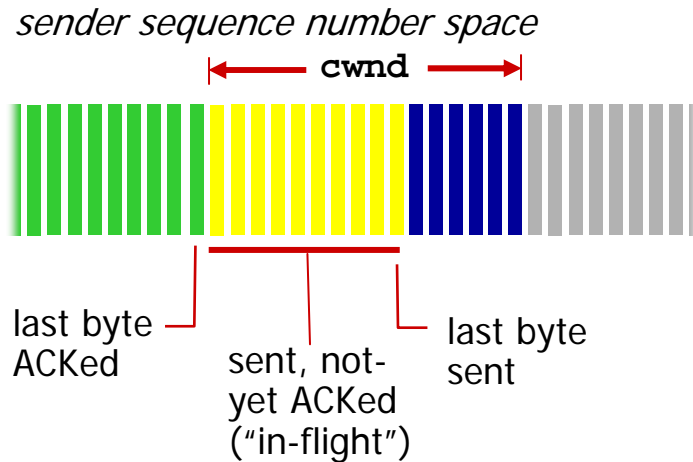
last byte sent

❖ sender limits transmission:

$$LastByteSent - LastByteAcked \leq cwnd$$

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

TCP sending rate:

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

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

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

Host A   Host B

RTT

one segment

two segments

four segments
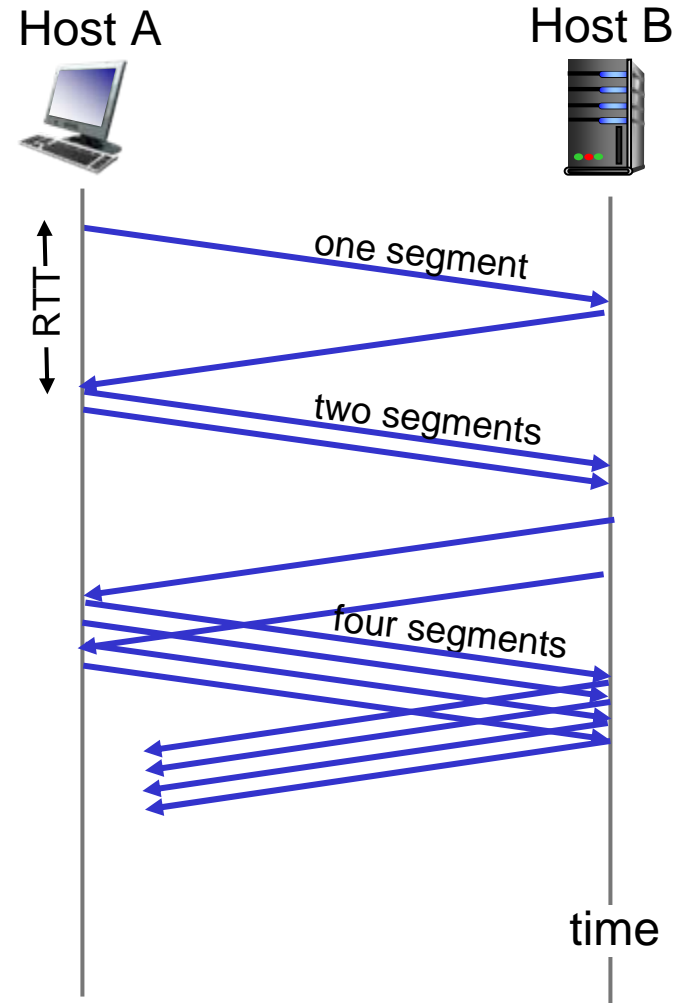
time

TCP: detecting, reacting to loss

❖ loss indicated by timeout:
  ▪ `cwnd` set to 1 MSS;
  ▪ window then grows exponentially (as in slow start) to threshold, then grows linearly
❖ loss indicated by 3 duplicate ACKs: TCP RENO
  ▪ dup ACKs indicate network capable of delivering some segments
  ▪ `cwnd` is cut in half window then grows linearly
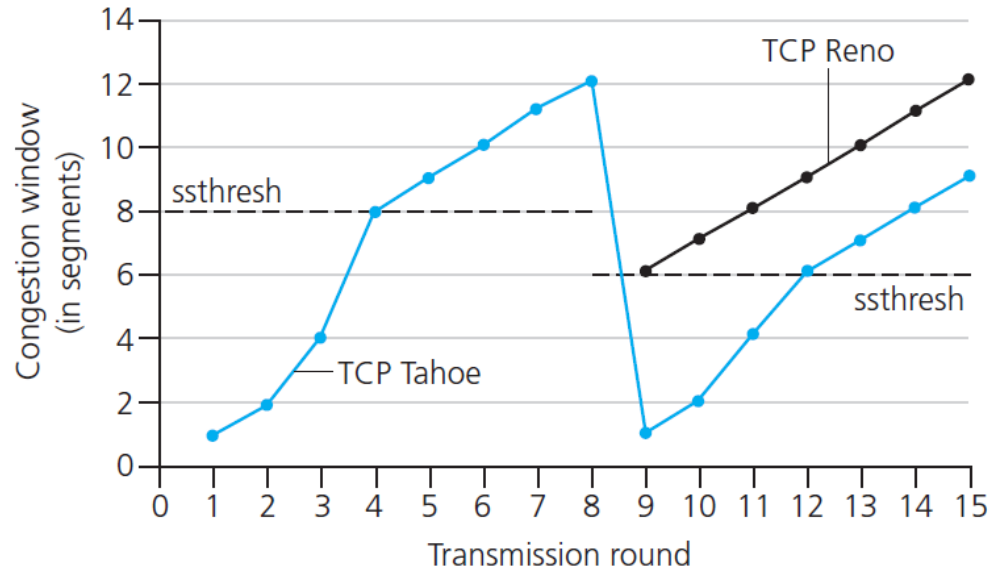❖ TCP Tahoe always sets `cwnd` to 1 (timeout or 3 duplicate acks)

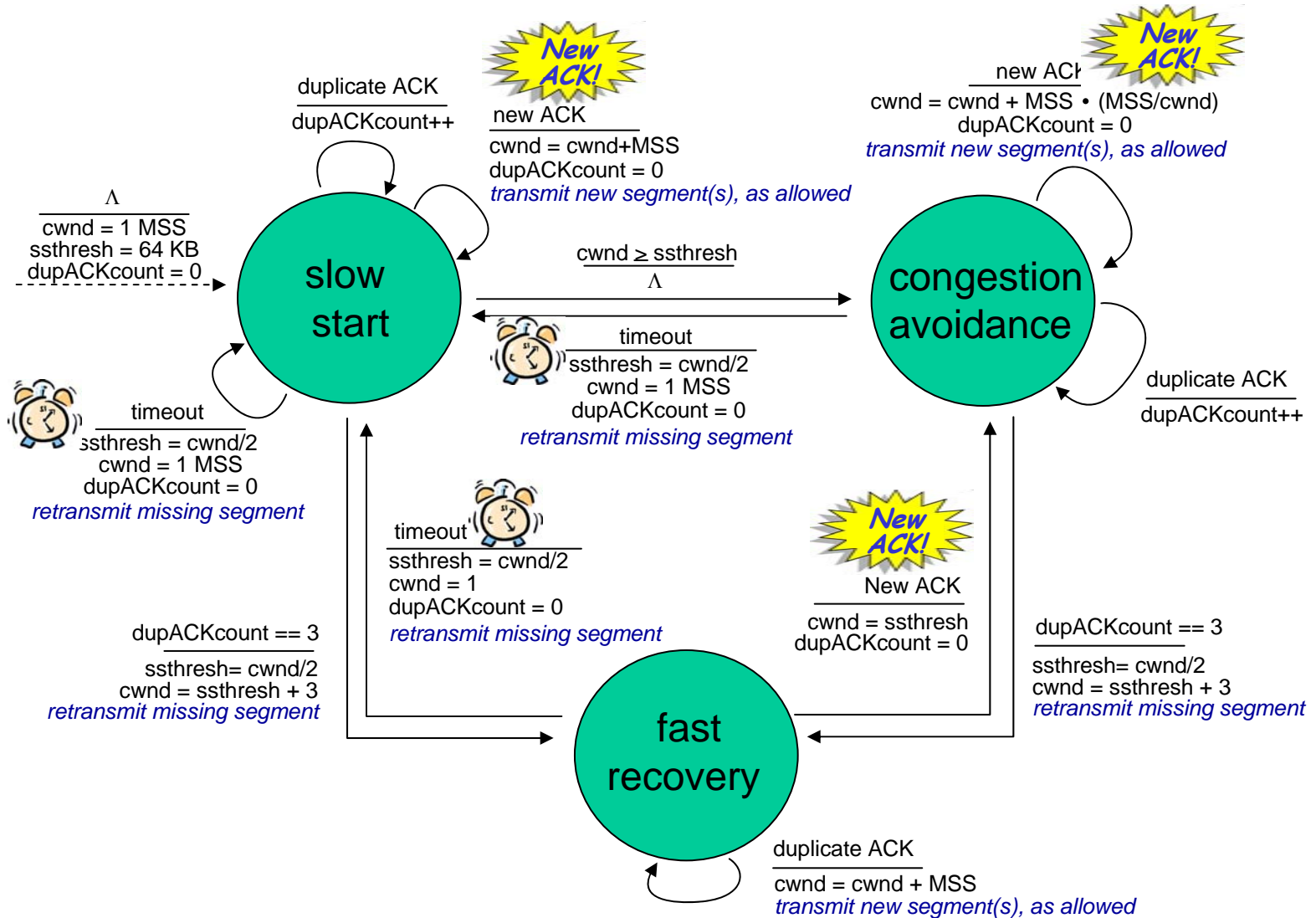# TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

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

Implementation:

❖ variable **ssthresh**

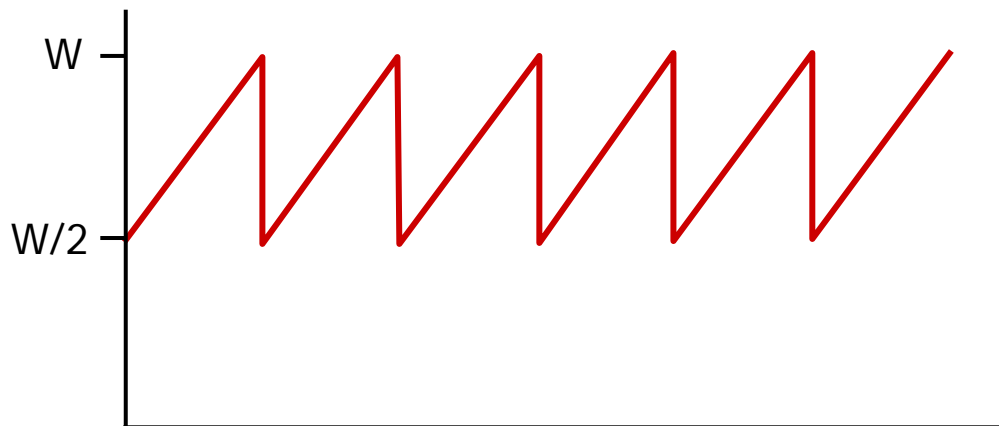❖ on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event

Summary: TCP Congestion Control

TCP throughput

❖ avg. TCP thruput as function of window size, RTT?

  ▪ ignore slow start, assume always data to send

❖ W: window size (measured in bytes) where loss occurs

  ▪ avg. window size (# in-flight bytes) is ¾ W

  ▪ avg. thruput is 3/4W per RTT

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

TCP Futures: TCP over "long, fat pipes"

❖ example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput

❖ requires W = 83,333 in-flight segments

❖ throughput in terms of segment loss probability, L [Mathis 1997]:

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

➜ to achieve 10 Gbps throughput, need a loss rate of L = $2 \cdot 10^{-10}$ *– a very small loss rate!*
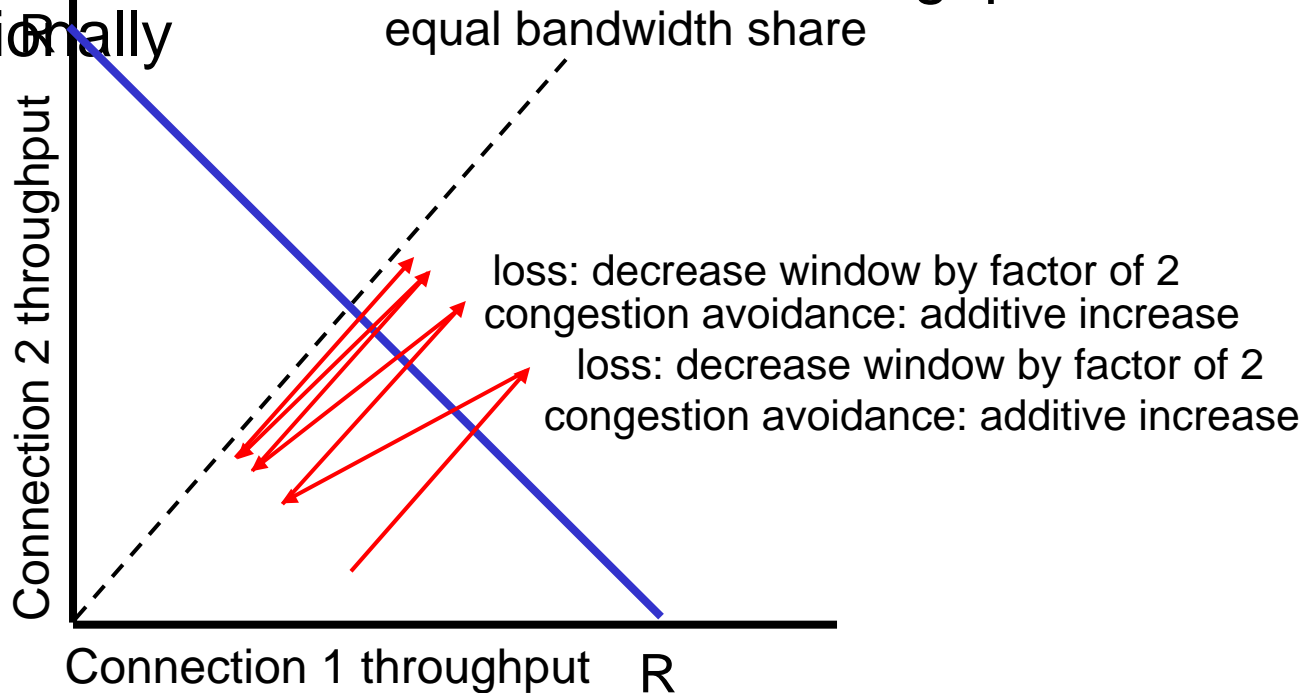
❖ new versions of TCP for high-speed

TCP Fairness

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

TCP connection 1

TCP connection 2

bottleneck
router
capacity R
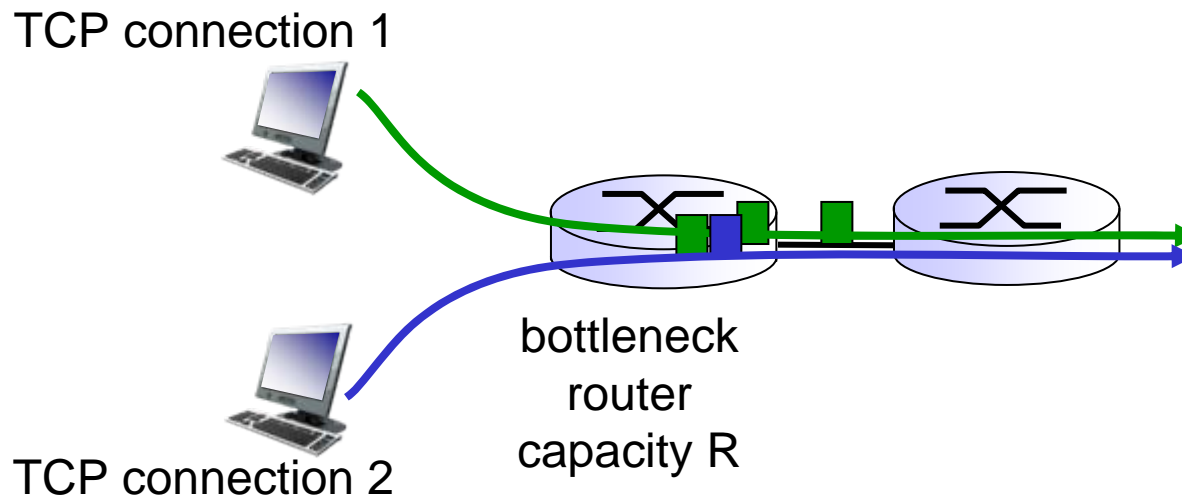
Why is TCP fair?

two competing sessions:

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

equal bandwidth share

loss: decrease window by factor of 2
congestion avoidance: additive increase

loss: decrease window by factor of 2
congestion avoidance: additive increase

Connection 2 throughput

Connection 1 throughput   R

Fairness (more)

Fairness and UDP

❖ multimedia apps often do not use TCP

  ▪ do not want rate throttled by congestion control

❖ instead use UDP:

  ▪ send audio/video at constant rate, tolerate packet loss

Fairness, parallel TCP connections

❖ application can open multiple parallel connections between two hosts

❖ web browsers do this

❖ e.g., link of rate R with 9 existing connections:

  ▪ new app asks for 1 TCP, gets rate R/10

  ▪ new app asks for 11 TCPs, gets R/2

Chapter 3: summary

❖ **principles behind transport layer services:**
- multiplexing, demultiplexing
- reliable data transfer
- flow control
- congestion control

❖ **instantiation, implementation in the Internet**
- UDP
- TCP

next:

❖ leaving the network "edge" (application, transport layers)

❖ into the network "core"